PSG INSTITUTE OF TECHNOLOGY AND APPLIED RESEARCH

NEELAMBUR, COIMBATORE-641 062

EE3512 - Control and Instrumentation Laboratory

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

ACADEMIC YEAR 2023-2024

NAME	
REGISTER No.	
BRANCH / YEAR	**EEE / III year**
SEMESTER	**V**

PSG INSTITUTE OF TECHNOLOGY AND APPLIED RESEARCH

NEELAMBUR, COIMBATORE-641 062

BONAFIDE CERTIFICATE

Certified that this is a Bonafide Record of work done by ..... ……..…….... of III-year B.E (Electrical and Electronics Engineering) in the **EE3512 - Control and Instrumentation Laboratory** conducted in this institution, as prescribed by Anna University, Chennai, for the fifth semester, during the academic year 2023-2024.

Faculty in-charge **Head of the Department**

Date:

University Register Number: _____

Submitted on **:** _____

Internal Examiner **External Examiner**

List of Experiments

S. No.	Title
1	Analog (op amp based) simulation of linear differential equation
2	Numerical Simulation of given nonlinear differential equations
3	Real time simulation of differential equations
4	Mathematical modelling and simulation of physical systems in at least two fields. Mechanical Electrical Chemical process
5	System Identification through process reaction curve.
6	Stability analysis using Pole zero maps and Routh Hurwitz Criterion in simulation platform.
7	Root Locus based analysis in simulation platform
8	Determination of transfer function of a physical system using frequency response and Bode's asymptotes
9	Design of Lag, lead compensators and evaluation of closed loop performance.
10	Design of PID controllers and evaluation of closed loop performance.
11	Discretization of continuous system and effect of sampling
12	Test of controllability and observability in continuous and discrete domain in simulation platform
13	State feedback and state observer design and evaluation of closed loop performance
14	Mini Project1: Simulation of complete closed loop control system including sensor and actuator dynamics
15	Mini Project 2: Demonstration of closed loop system in hardware.

Ex. No.: 1	**Analog (OP-amp based) Simulation of Linear Differential Equation**
Date:	

Aim

To perform analog simulation of linear differential equation using opamp.

Introduction

There are two kinds of computers. A digital computer is most likely what comes to mind when you hear the word "computer." Numbers are represented by sets of 1's and 0's, where 1 and 0 are two separate voltages (usually +5V and 0V). Operations are either simple logical operations (such as AND, OR, and so on) or mathematical operations (such as addition or subtraction). Calculus-type operations are extremely difficult to perform. The analogue computer is the opposite form, in which numbers are represented by continually altering values. Voltage is the most commonly used quantity. The OPAMP circuits can be used for performing various mathematical operations with varying voltages. These circuits can be used in conjunction to solve a variety of differential equations.

Procedure

Step 1: Derive differential equation for the given system in figure 1

Step 2: Identify Suitable Op-Amp Circuit Configuration

Step 3: Implement the Circuit in MATLAB simscape

Step 4: Apply the Input and analyse the Output

Step 5: Consider Initial Conditions (Optional)

Mechanical Translational System:

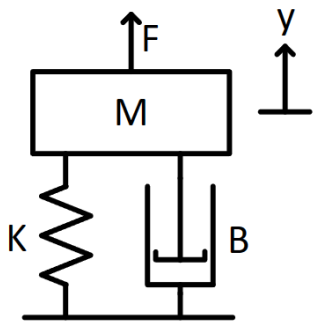
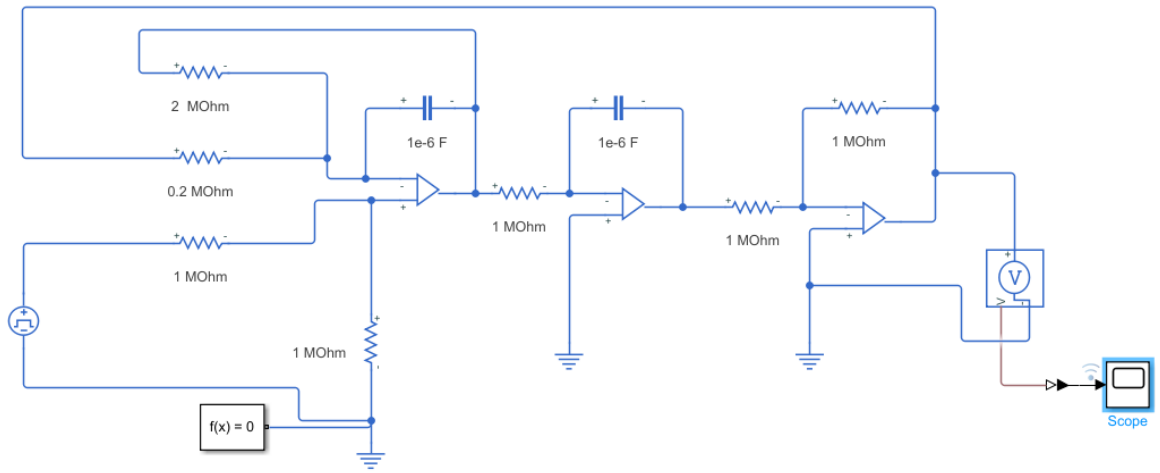


Figure 1 Mass Spring Damper System

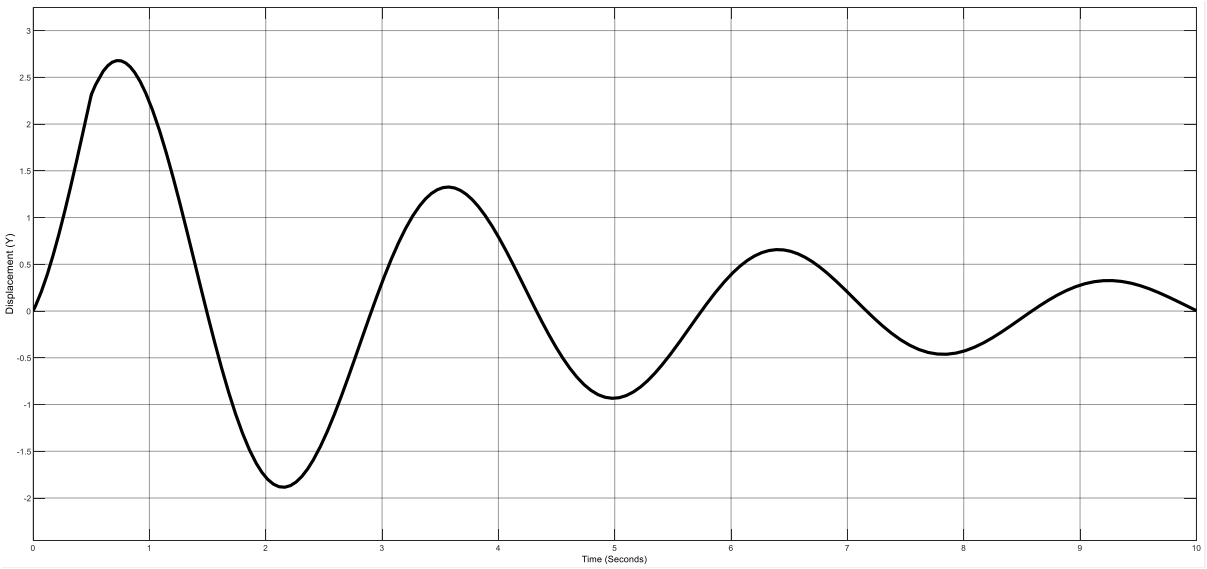
Differential Equation for Mass Spring Damper System:

$$M\frac{d^2y}{dt^2} + B\frac{dy}{dt} + Ky = f(t)$$

MATLAB Circuit



Output Waveform



Inference

Result

The linear differential equation of a mass spring damper system is simulated using opamp based circuits using MATLAB simscape, and the system is tested for different inputs.

Ex. No.: 2	**Numerical Simulation of given Nonlinear Differential Equations**
Date:	

Aim

The objective of this lab experiment is to understand the process of numerically simulating nonlinear differential equations

Introduction

Differential equations play a crucial role in modeling real-world phenomena in physics, engineering, biology, and many other disciplines. Nonlinear differential equations are particularly challenging to solve analytically, so numerical methods like ode45 become invaluable in simulating their behavior.

ode45 is a powerful function in MATLAB used for solving ordinary differential equations (ODEs) numerically. It employs a Runge-Kutta method (specifically, the Dormand-Prince method) to approximate the solution of the ODE. The general syntax of ode45 is as follows:

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

Where:

t is the vector of time points at which the solution is computed.

y is the matrix of solutions, where each row corresponds to the state of the system at the corresponding time in t.

odefun is a function handle that defines the ODEs you want to solve.

tspan is the time interval over which you want to solve the ODE. It is specified as [t0, tf], where t0 is the initial time and tf is the final time.

y0 is the initial condition of the system at time t0.

options (optional) is a structure that allows you to specify additional settings for ode45

Mass spring damper system with non linear stiffness

$$M\frac{d^2x}{dt^2} + M\frac{dx}{dt} + kx + \propto x^3 = 0 \text{ ----------(1)}$$

Procedure

Step 1: Choose the non-linear differential equation representing the real-world physical system.

Step 2:

Open MATLAB and create a script file

Step3: Define the ODE function which represent the non-linear system

Step4: Set initial conditions and parameters

Step5: Solve the ODE using ode45 function

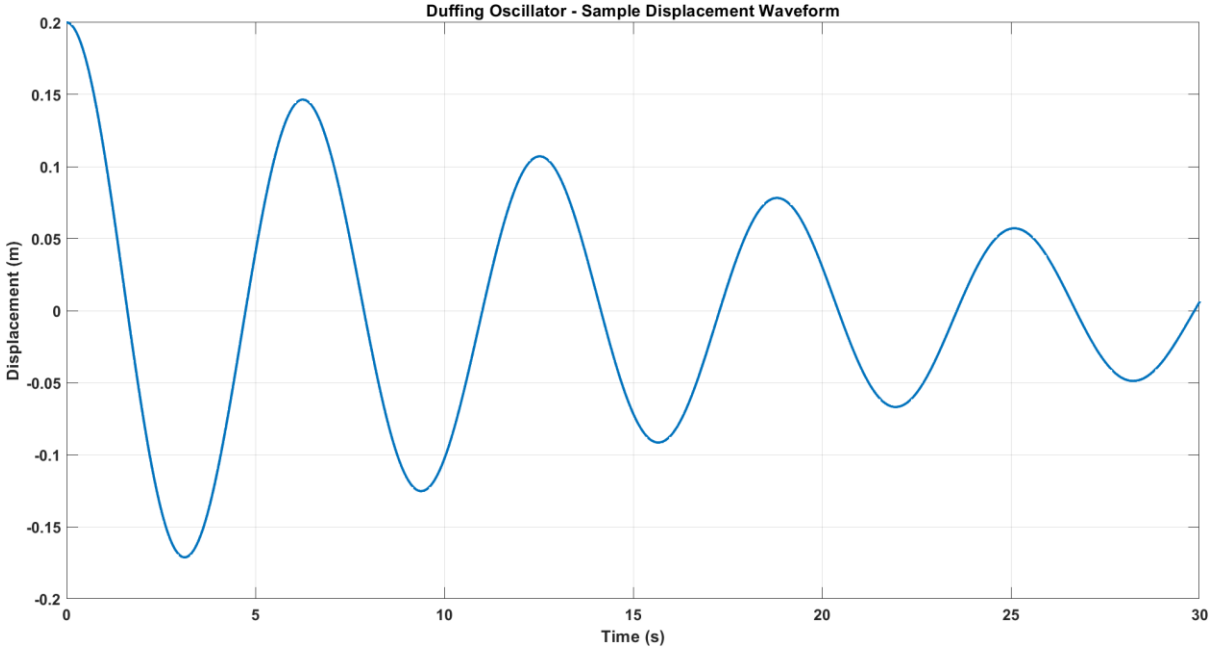
Step6: Plot the result displacement vs time

Step7: Experiment with different data and write the inference

Matlab Code

```
% Constants
m = 1;       % Mass (kg)
c = 0.1;     % Damping coefficient (Ns/m)
k = 1;       % Linear stiffness coefficient (N/m)
alpha = 0.5; % Nonlinear stiffness coefficient (N/m^3)

% Initial conditions
x0 = 0.2;    % initial displacement
v0 = 0.0;    % initial velocity

% Time points for integration
t_start = 0;
t_end = 30;
dt = 0.01;
t_points = t_start:dt:t_end;

% Numerical integration using MATLAB's ode45 solver
[t, X] = ode45(@(t, x) duffing_oscillator(t, x, m, c, k, alpha), t_points, [x0;
v0]);

% Extract displacement values
x_values = X(:, 1);

% Plotting the displacement waveform
figure;
plot(t, x_values);
xlabel('Time (s)');
ylabel('Displacement (m)');
title('Duffing Oscillator - Sample Displacement Waveform');
grid on;

function dxdt = duffing_oscillator(~, x, m, c, k, alpha)
    dxdt = [x(2); -(c*x(2) + k*x(1) + alpha*x(1)^3) / m];
end
```

Output Waveform:



Inference:

Result:

Thus, a non-linear differential equation is simulated numerically using MATLAB 'ode45' solver and the behaviour of oscillator is studied under different conditions.

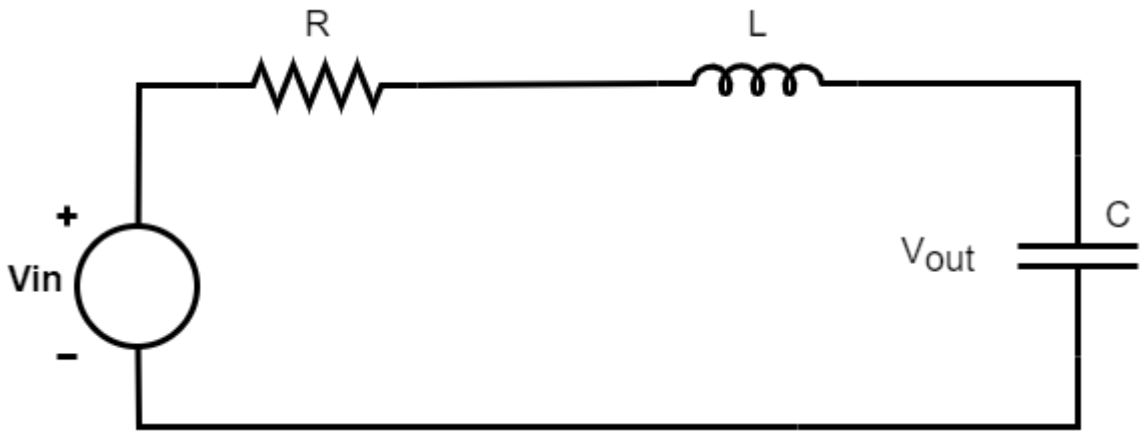
Ex. No.: 3	**Realtime Simulation of Differential Equation**
Date:	

Aim

To simulate the differential equation of series RLC circuit using Matlab

Introduction

The series RLC circuit above has a single loop with the instantaneous current flowing through the loop being the same for each circuit element. Since the inductive and capacitive reactance's XL and XC are a function of the supply frequency, the sinusoidal response of a series RLC circuit will therefore vary with frequency, f. Then the individual voltage drops across each circuit element of R, L and C element will be "out-of-phase"



The differential equation for the above system for input and output voltage is given by equations 1 and 2

$$V_{in} = iR + L\frac{di}{dt} + \frac{1}{C}\int_0^t idt \text{----------(1)}$$

$$V_{out} = \frac{1}{C}\int_0^t idt \text{----------(2)}$$

The above equation can be represented in S domain using the equation 3.

$$H = \frac{V_{out}}{V_{in}} = \frac{1}{(CL)S^2+(CR)S+1} \text{----------(3)}$$

Procedure

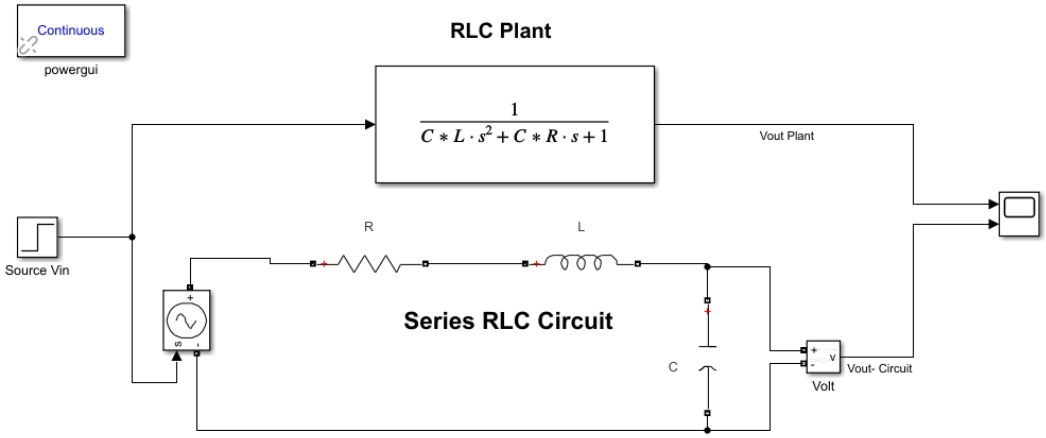
Step1: Input the transfer function obtained from differential equations in transfer function block

Step2: Apply a step input to the RLC plant

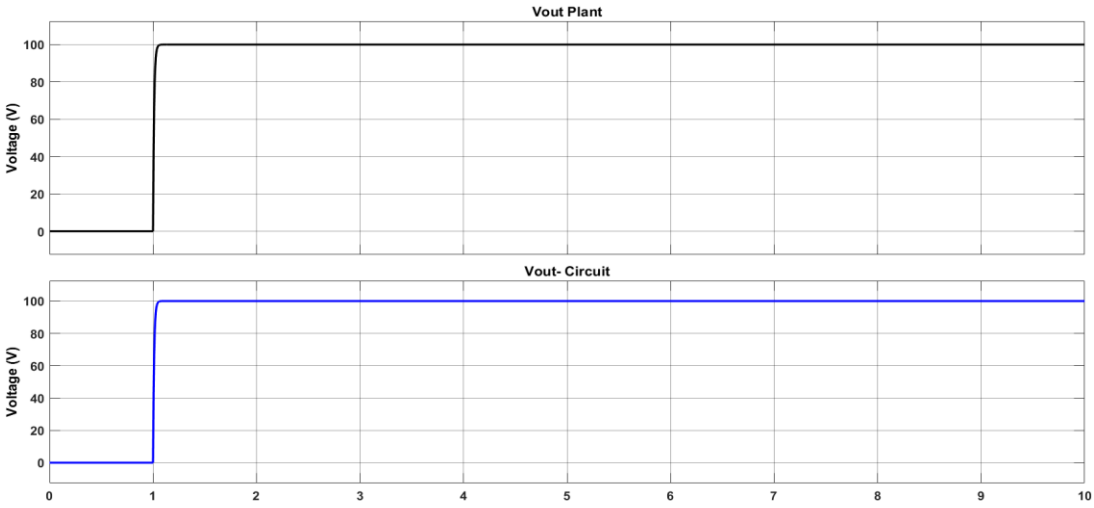
Step3: Plot the V_{in} and V_{out} waveform in scope.

Step4: Compare and verify the results simulated using RLC circuit in Simulink.

Matlab Circuit



Output Waveform



Result

Thus, the realtime differential equation circuit has been simulated using Simulink

Ex. No.: 4	**Mathematical Modelling and Simulation of Physical Systems in at Least Two Fields. Mechanical and Electrical**
Date:	

Aim

Mathematical modelling and simulation of physical systems (DC motor) in at least two fields: Mechanical and Electrical system.

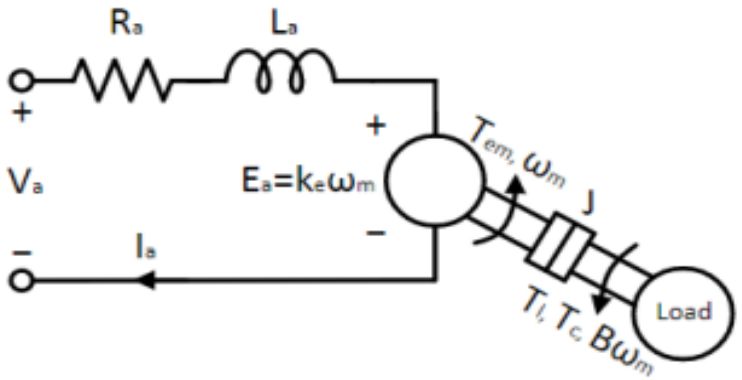
Introduction

A DC motor can be mathematically modelled as an electrical system and a mechanical system, tied together by relationship between back-emf and speed, and relationship between current and torque. In this experiment, the electrical and mechanical parameters of the DC motor are determined. These values are necessary to design a proper control system to control motor speed, torque, or position as will be seen in later experiments.

Theoretical background:

Motor model

The DC motor, like all real-world systems, exhibits non-linear behaviour. In the model shown below, a linearized DC motor model is considered for the sake of design simplicity. It will be verified in later experiments that the results from the simulated linearized model closely matches the real-world model thus validating this approximation.



The torque T_{em} is proportional to the armature current I_a as given in Eqn. 1 where kt is the torque constant.

$$T_{em} = k_t \times I_a \text{---------- (1)}$$

As the motor rotates, the armature winding cuts through the magnetic field which induces the back emf, E_a on the armature winding. When the motor rotates faster, the rate of change of magnetic flux $d\phi/dt$ increases, proportional to the rotor speed. This is given by Eqn. 2 where ω_m is the motor speed and k_e is the back-emf constant.

$$E_a = k_e \times \omega_m \text{---------- (2)}$$

R_a and L_a in the above model refer to the armature resistance and inductance respectively. The voltage applied at the terminal V_a equals the sum of voltage across the passive element - R_a and the back-emf E_a. This represents the electrical model of the DC motor and is given in Eqn. 3, where E_a has been replaced with $k_e \times \omega_m$ from Eqn. 2.

$$V_a = R_a I_a + k_e \omega_m \text{----------(3)}$$

The mechanical model of the DC motor is given in Eqn. 4. Under steady state, the torque generated by the motor T_{em} equals the sum of load torque T_l and the torque necessary to compensate the frictional losses, T_{fric}. In transient condition when $T_{em} \neq Tl + T_{fric}$, the motor accelerates if the former is greater than the latter, storing the excess energy as inertial energy. It decelerates if the former is less than the latter, loosing previously stored inertial energy. The motor inertia is identified as J in Eqn. 4.

$T_{em} = Tl + T_{fric} + Jd\omega mdt$---------- (4)

The frictional component is due to various factors and is always against the direction of rotation. o The torque T_{em} is proportional to the armature current I_a as given in Eqn. 1 where kt is the torque constant.

$$T_{em} = k_t \times I_a \text{---------- (4)}$$

As the motor rotates, the armature winding cuts through the magnetic field which induces the back emf, E_a on the armature winding. When the motor rotates faster, the rate of change of magnetic flux $d\phi/dt$ increases, proportional to the rotor speed. This is given by Eqn. 2 where ω_m is the motor speed and k_e is the back-emf constant.

$E_a = k_e \times \omega_m$---------- *(5)*

R_a and L_a in the above model refer to the armature resistance and inductance respectively. The voltage applied at the terminal V_a equals the sum of voltage across the passive element - R_a and the back-emf E_a. This represents the electrical model of the DC motor and is given in Eqn. 3, where E_a has been replaced with $k_e \times \omega_m$ from Eqn. 2.

$V_a = R_a I_a + k_e \omega_m$----------*(6)*

The mechanical model of the DC motor is given in Eqn. 4. Under steady state, the torque generated by the motor Tem equals the sum of load torque T_l and the torque necessary to compensate the frictional losses, T_{fric}. In transient condition when Tem ≠ Tl + Tfric, the motor accelerates if the former is greater than the latter, storing the excess energy as inertial energy. It decelerates if the former is less than the latter, loosing previously stored inertial energy. The motor inertia is identified as J in Eqn. 4.

Tem = Tl + Tfric + Jdωmdt---------- (4)

The frictional component is due to various factors and is always against the direction of rotation. of the many causes, only Coulomb friction which is a constant and viscous friction which varies proportionately with rotational speed are considered, since these are the ones that have significant effect on the steady-state. The torque associated with Coulomb friction is given by Tc and that of viscous friction is given by $B \times \omega_m$, where B is the coefficient of viscous friction. Substituting these into Eqn. 4 yields the final mechanical model as given in Eqn. 5.

Tem = Tl + Bωm + Jdωmdt---------- (5)

Parameter estimation

In this experiment, the electrical model parameters, R_a, L_a, k_e, kt and, the mechanical model parameters, B and J, are determined for the DC motor as follows:

R_a: Use a Multimeter and measure resistance of armature winding.

L_a: – Use LCR meter measure inductance of armature winding

ke Apply a constant voltage Vo at the motor terminal. Measure the motor current Ia and motor speed ωm at steady-state. Compute ke from Eqn. 6, which is obtained from Eqn. 3 .

k_e: = (Va – RaIa)/ωm---------- (6)

k_t: Calculate kt from name plate details

Note the rated output power P, Rated speed N in radian/sec and rated current I_a

By using the above calculated the rated torque

T = 60*P/(2*π*N) ---------- (7)

Calculate k_t using Eqn. 1

k_t = T/Ia---------- (8)

B: If the motor is not accelerating i.e. motor speed is a constant, and there is no load connected, the inertial torque component and load torque component in Eqn. 5 becomes zero, leading to the following equation:

Tem = Bωm(11) ---------- (9)

Apply a constant voltage V, at the motor terminal. Measure the motor speed ω_m, and current Ia. The electromagnetic torque T_{em} can be computed from the measured current I_a using Eqn. 1. Repeat the same for different voltages and plot the result, T_{em} (y-axis) vs. ω_m (x-axis). The resulting plot will be linear. The slope of the plot equals the coefficient of viscous friction B.

J: Apply a constant voltage V, at the motor terminal. Once the speed settles down, measure the speed ωm, and motor current Ia. At steady state, the electromagnetic torque T_{em} (Eqn. 1), solely compensates for drag due to friction. Disable the inverter so that the motor current dies-down to zero rapidly. After this, there is no electromagnetic torque and all the frictional losses must be supplied from the inertial energy. This leads to the motor to stop gradually. This is summarized in Eqn. 10, which is obtained by equating T_{em} and load torque T_l in Eqn. 4 to zero.

0 = Tfric + Jdωm/dt---------- (10)

At the moment the inverter is disabled, time t = T_{dis}, the frictional torque component T_{fric} is same as, that of electromagnetic T_{em} just before disabling the

inverter, which can be obtained from the steady-state current I_a before disabling the inverter using Eqn. 1. Substituting this into Eqn. 10 yields:

$$J = ke \times Ia(Tdis) / d\omega m/dt(Tdis) \quad \text{---------- (11)}$$

Thus, the motor inertia can be obtained by dividing, the electromagnetic torque just before disabling the inverter, with the motor deceleration just after disabling the inverter.

Procedure:

Step 1: Derive differential equation for the DC motor.

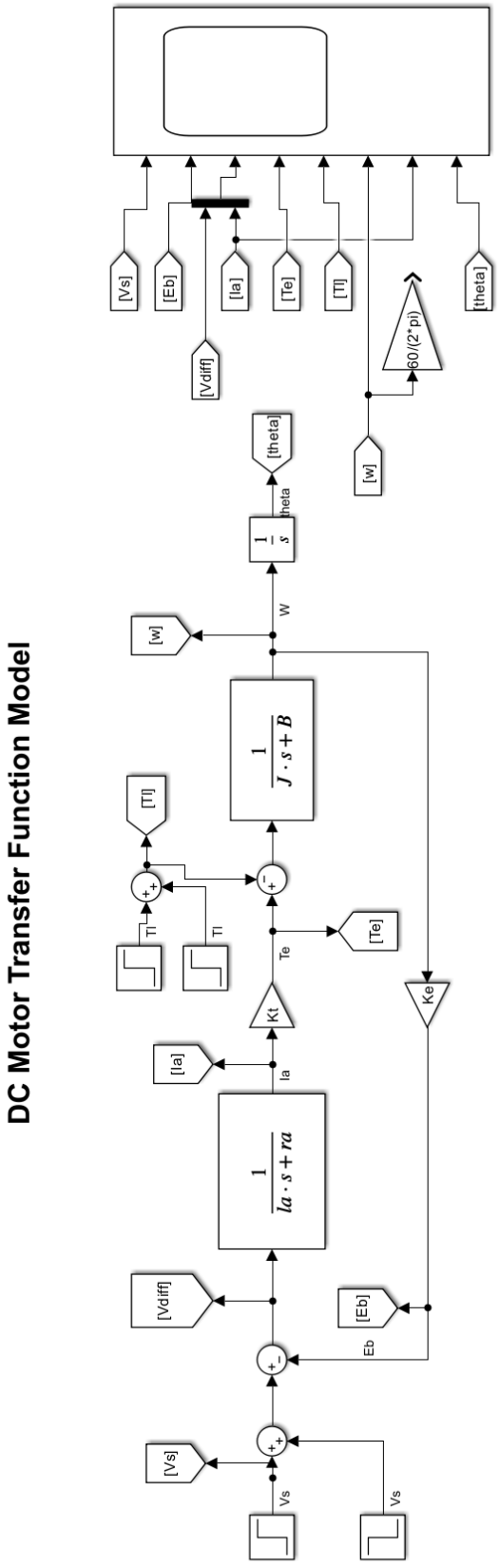
Step 2: Add the respective blocks in MATLAB – simulink for the derived transfer function.

Step 3: By applying the rated speed and note the value of resistance, inductance, current.

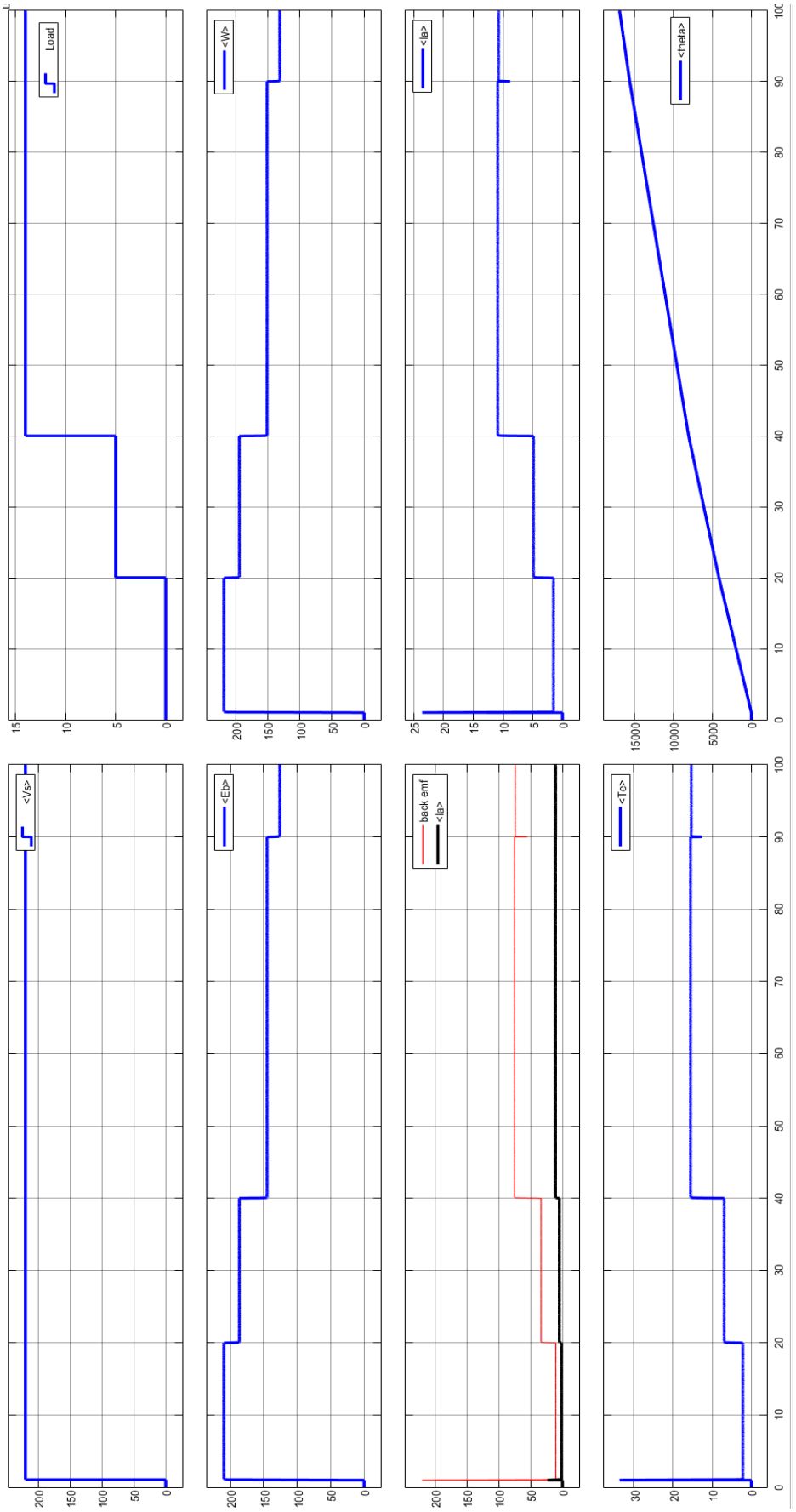
Step 4: Calculate the value of inertia (J) and friction (B)

Step 5: Use the calculated value for the simulation and analyse the graph.

MATLAB Circuit:



Output Waveform



Inference

Result

Thus, mathematical modelling and simulation of physical systems (DC motor) of mechanical and electrical systems is studied.

Ex. No.: 5	**System Identification through Process Reaction Curve**
Date:	

Aim

To determine the process steady state gain (K), dead time(τ_D) and effective process time constant τ from process reaction curve and also the determine controller gain (Kc), integral time constant (T_i) and derivative time constant (T_D) for P, PI and PID controller.

Introduction

Ziegler and Nichols recognized that the open-loop step responses of a large number of processes exhibit a process reaction curve like that shown in Fig. 1 The S-shape of the curve can be approximated by the step response of

$$\frac{Y(s)}{U(s)} = \frac{Ke^{-s\tau_D}}{\tau s + 1}$$

K = The process steady-state gain,

τ_D = The effective process dead time; and

τ = The effective process time constant

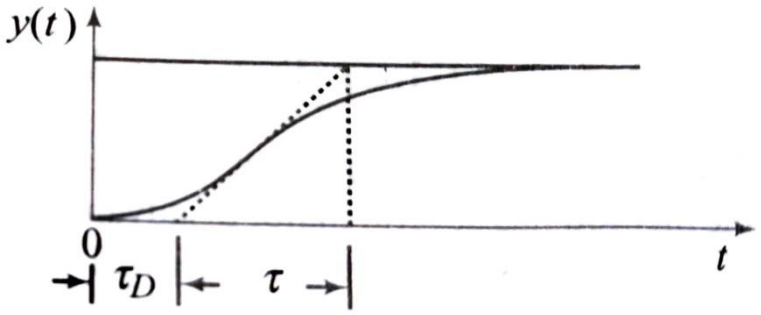


Fig.1 Process Reaction Curve

The constants in Equation can be determined from the experimentally obtained step response y(t) to step input u(t) = Aμ(t). The steady-state value of the response y_{ss} = KA. This gives the parameter K. If line is KA/τ, and the intersection of the tangent line with the time axis identifies the dead-time τ_D.

The formulas in Table were developed empirically for the most common range of τ_D/τ, which is between 0.1 and 0.3.

Table QDR tuning formulas based on process reaction curve

Controller	Gain	Integral time	Derivative time
P	$K_c = \tau / K\tau_D$	-	-
PI	$K_c = 0.9\tau / K\tau_D$	$T_I = 3.33\tau_D$	-
PID	$K_c = 1.5\tau / K\tau_D$	$T_I = 2.5\tau_D$	$T_D = 0.4\tau_D$

Procedure

Step 1: Identify the transfer function of system.

Step 2: Get the open loop step response of the system.

Step 3: Calculate the Gain, time constant and dead time from the process reaction curve.

Step4: Calculate the controller gain, integral time constant and derivative time constant

Step5: Display the result.

Matlab Code

```
% Reaction curve based P I D Controller Tuning
clc;
clearvars;
% Transfer function of system
G1 = tf(1, [1 4 6 4 1]);
du=1;
t0=0;
u=1;
[y,t]=step(G1);

% Calculations of system constants
K=(y(end)-y(1))/du; % gain K
dy=diff(y);
dt=diff(t);
[mdy,I]=max(abs(dy)./dt);
t_c=abs(y(end)-y(1))/mdy; % time constant
t_d=t(I)-abs(y(I)-y(1))/mdy-t0;% dead time

fprintf("System Gain K: %0.2f\n",K);
fprintf("System time constant T: %0.2f\n",t_c);
fprintf("System dead time: %0.2f\n",t_d);

% Plot
figure;
plot(t,y,[t0+t_d t0+t_d+t_c],[y(1) y(end)]);
title('System Response ')
grid on;
```

```
%% Controller Parameters

Controller_P_Kc = t_c/(K*t_d);
Controller_PI_Kc= 0.9*t_c/(K*t_d);
Controller_PID_Kc= 1.5*t_c/(K*t_d);

ti_PI = 3.33 * t_d;
ti_PID = 2.5*t_d;
td_PID = 0.5*t_d;

fprintf("P Controller Gain (Kc) : %0.2f \n",Controller_P_Kc);
fprintf("PI Controller gain (Kc): %0.2f and Integral Time Constant
(Ti): %0.2f \n",Controller_PI_Kc,ti_PI);
fprintf("PID Controller gain (Kc): %0.2f and Integral Time Constant
(Ti): %0.2f ..." + ...
    "Derivative Time Constant (Td) %0.2f\n",Controller_PID_Kc,ti_PID
,td_PID);
```

Output Waveform

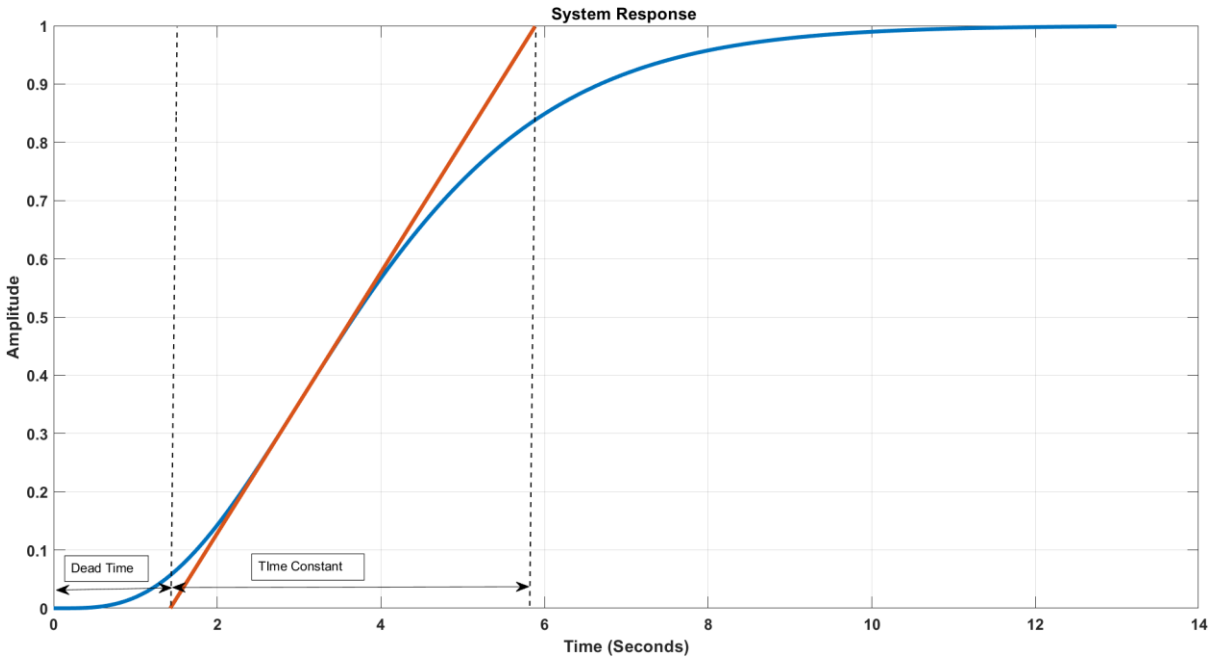


Fig. System Response and Constant Identification

System Constants and Controller Parameters

System Gain K: 1.00

System time constant T: 4.46

System dead time: 1.43

P Controller Gain (Kc): 3.13

PI Controller gain (Kc): 2.82 and Integral Time Constant (Ti): 4.75

PID Controller gain (Kc): 4.70

Integral Time Constant (Ti): 3.56

Derivative Time Constant (Td): 0.71

Result

Thus, the system identification and controller parameters identification through process reaction curve was verified using Matlab code

Ex. No:6	**Stability Analysis using Pole Zero Maps and Routh Hurwitz Criterion in Simulation Platform**
Date:	

Aim:

To perform stability analysis using Pole zero maps and Routh Hurwitz Criterion in simulation platform.

Introduction to Routh Hurwitz Criterion:

1. The Routh-Hurwitz criterion is a mathematical technique used in control theory and engineering to determine the stability of a linear time-invariant system. It is named after its developers, Edward Routh and Adolf Hurwitz. The criterion provides a systematic way to analyze the roots (or poles) of the characteristic equation of a system, and based on the locations of these roots, it can determine whether the system is stable, marginally stable, or unstable.

2. Pole Locations: The roots of the characteristic equation are known as the system's poles. The stability of the system is determined by the locations of these poles in the complex plane. Specifically, for continuous-time systems, a system is considered stable if all its poles have negative real parts (located in the left-half of the complex plane). For discrete-time systems, stability is determined by poles within the unit circle.

3. Routh-Hurwitz Criterion: The Routh-Hurwitz criterion is a systematic procedure for analyzing the stability of a system based on the coefficients of the characteristic equation. It involves constructing a special table known as the Routh array or Routh-Hurwitz table.

4. Routh-Hurwitz Table: The Routh-Hurwitz table is created using the coefficients of the characteristic equation. The table consists of rows and columns, where each entry is a coefficient value or a calculated value. The first column of the table contains the coefficients of the polynomial, and subsequent columns contain values calculated from these coefficients.

5. Stability Analysis: The Routh-Hurwitz criterion is applied to the Routh array to determine the number of sign changes in the first column. Based on these sign changes, the system's stability is determined as follows:
   - If there are no sign changes in the first column, all system poles have negative real parts, and the system is stable.
   - If there are sign changes in the first column but no sign changes in the subsequent columns, the system has poles with zero real parts, indicating marginally stable behavior.
   - If there are sign changes in any column, excluding the case of marginally stable poles, the system is unstable.

Procedure:

Step 1. Obtain the Characteristic Equation
Step 2. Write Down the Coefficients
Step 3. Create the First Two Rows of the Routh Table
Step 4. Calculate the Remaining Rows of the Routh Table
Step 5. Check for Sign Changes in the First Column
Step 6. Determine System Stability
Step 7. Interpret the Results

Matlab code:

```
%% Routh-Hurwitz stability criterion
%
% The Routh-Hurwitz stability criterion is a necessary (and frequently
% sufficient) method to establish the stability of a single-input,
% single-output(SISO), linear time invariant (LTI) control system.
% More generally, given a polynomial, some calculations using only the
% coefficients of that polynomial can lead us to the conclusion that it
% is not stable.
% Instructions
% ------------
%
% in this program you must give your system coefficients and the
% Routh-Hurwitz table would be shown
%% Initialization
clear ; close all; clc
% Taking coefficients vector and organizing the first two rows
syms e s
syms k Integer
coeffVector = input('input vector of your system coefficients: \n i.e. [an an-1 an-2 ... a0] = ');
ceoffLength = length(coeffVector);
rhTableColumn = round(ceoffLength/2);
K=isnumeric(coeffVector);
z_c= ceoffLength-2;
% Initialize Routh-Hurwitz table with empty zero array
rhTable = zeros(ceoffLength,rhTableColumn);
rt_tab=sym(rhTable);
% Compute first row of the table
rt_tab(1,:) = coeffVector(1,1:2:ceoffLength);
% Check if length of coefficients vector is even or odd
if (rem(ceoffLength,2) ~= 0)
% if odd, second row of table will be
rt_tab(2,1:rhTableColumn - 1) = coeffVector(1,2:2:ceoffLength);
else
% if even, second row of table will be
rt_tab(2,:) = coeffVector(1,2:2:ceoffLength);
end
%% Calculate Routh-Hurwitz table's rows
% Set epss as a small value
epss = 0.01;
% Calculate other elements of the table
for i = 3:ceoffLength
% special case: row of all zeros
```

```
if rt_tab(i-1,:) == 0
order = (ceoffLength - i);
cnt1 = 0;
cnt2 = 1;
for j = 1:rhTableColumn - 1
rt_tab(i-1,j) = (order - cnt1) * rhTable(i-2,cnt2);
cnt2 = cnt2 + 1;
cnt1 = cnt1 + 2;
end
end
for j = 1:rhTableColumn - 1
% first element of upper row
firstElemUpperRow = rt_tab(i-1,1);
% compute each element of the table
rt_tab(i,j) = ((rt_tab(i-1,1) * rt_tab(i-2,j+1)) - ....
(rt_tab(i-2,1) * rt_tab(i-1,j+1))) / firstElemUpperRow;
end
% special case: zero in the first column
if rt_tab(i,1) == 0
rt_tab(i,1) = epss;
end
end
if(K==0)
assume(k>0);
x = solve(rt_tab(z_c+1,1)>0,k);
% x = real(x);
fprintf("Routh-Hurwitz Table for k = %f\n ",x(1,1));
disp(subs(rt_tab,[e k],[epss x(1,1)]));
rhTable = subs(rt_tab,[e k],[epss x(1,1)]);
routh_t = double(rhTable);
coeffVector(ceoffLength) = x(1,1);
else
routh_t = rhTable;
end
%% Compute number of right hand side poles(unstable poles)
% Initialize unstable poles with zero
unstablePoles = 0;
% Check change in signs
for i = 1:ceoffLength - 1
if sign(routh_t(i,1)) * sign(routh_t(i+1,1)) == -1
unstablePoles = unstablePoles + 1;
end
end
% Print calculated data on screen
fprintf('\n Routh-Hurwitz Table:\n')
routh_t;
% Print the stability result on screen
if unstablePoles == 0
fprintf('~~~~~> it is a stable system! <~~~~~\n')
else
fprintf('~~~~~> it is an unstable system! <~~~~~\n')
end
fprintf('\n Number of right hand side poles =%2.0f\n',unstablePoles)
```

```
reply = input('Do you want roots of system be shown? Y/N ', 's');
if reply == 'y' || reply == 'Y'
sysRoots = roots(coeffVector);
fprintf('\n Given polynomial coefficients roots :\n')
sysRoots
end
% Plot the roots on a graph
num_roots = size(sysRoots,1);
figure;
% Separate real and imaginary parts
real_parts = real(sysRoots);
imaginary_parts = imag(sysRoots);
% Create a scatter plot with different colors for each root
scatter(real_parts, imaginary_parts, 10,"yellow" , "filled");
colormap((parula(num_roots)))
% Set equal axis scaling for both real and imaginary parts
axis equal;
% Label the axes
xlabel('Real Part');
ylabel('Imaginary Part');
title('Complex Roots Plot');
% Annotate each point with its index
for i = 1:num_roots
str = strcat('\leftarrow ',string(sysRoots(i)));
text(real_parts(i), imaginary_parts(i),str,"FontSize",8,"Color",'r');
end
% Insert vertical and horizontal lines at x-axis and y-axis
xline(0, 'k-');
yline(0, 'k-');
grid on;
box on;
```

Output:

Result 1: Stable system

input vector of your system coefficients:
i.e. [an an-1 an-2 ... a0] = [1 2 3 1]

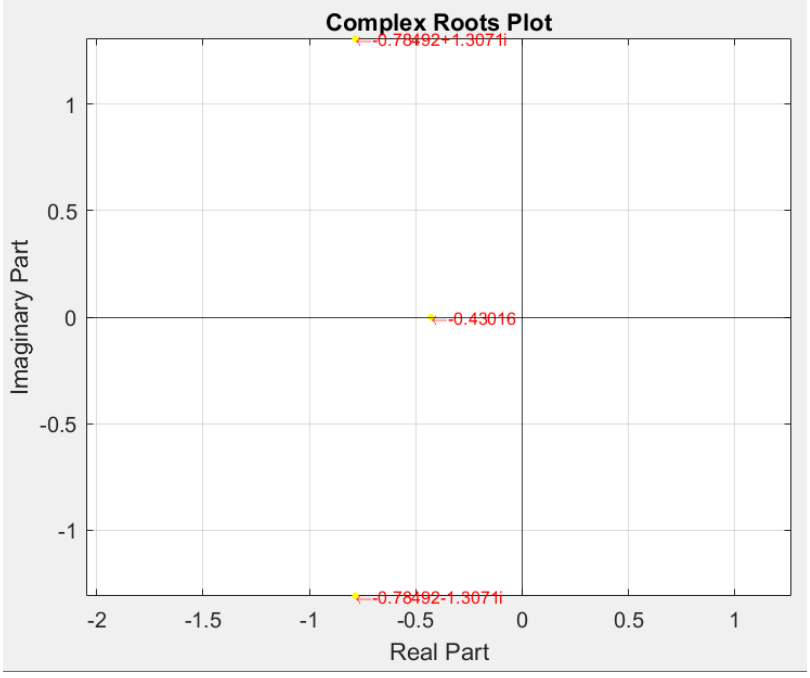
Routh-Hurwitz Table:
~~~~~> it is a stable system! <~~~~~

Number of right hand side poles = 0
Do you want roots of system be shown? Y/N Y

Given polynomial coefficients roots :

sysRoots =

-0.7849 + 1.3071i
-0.7849 - 1.3071i
-0.4302 + 0.0000i


~~~~~

Result 2: Zero in first column

input vector of your system coefficients:
 i.e. [an an-1 an-2 ... a0] = [1 1 2 2 3 5]

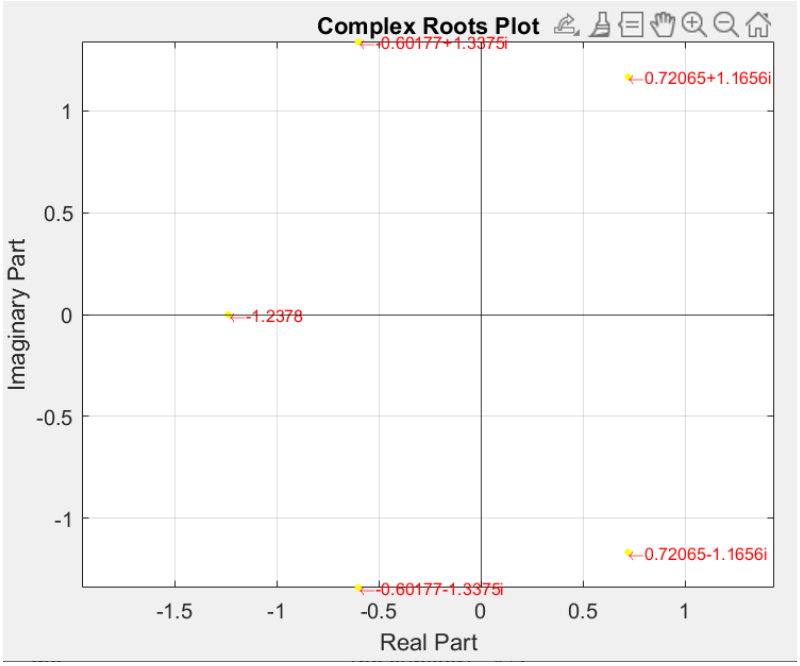
 Routh-Hurwitz Table:
~~~~~> it is a stable system! <~~~~~

 Number of right hand side poles = 0
Do you want roots of system be shown? Y/N Y

 Given polynomial coefficients roots :

sysRoots =

  0.7207 + 1.1656i
  0.7207 - 1.1656i
 -0.6018 + 1.3375i
 -0.6018 - 1.3375i
 -1.2378 + 0.0000i



Result 3: Row with all zeros

Input vector of your system coefficients:
 i.e. [an an-1 an-2 ... a0] = [1 9 24 24 24 24 23 15]

 Routh-Hurwitz Table:

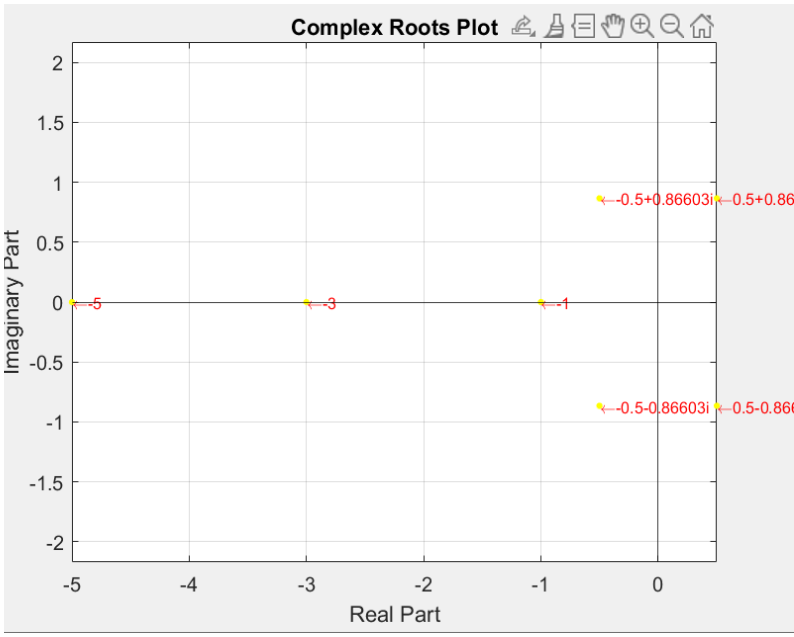
rhTable =

1.0000	24.0000	24.0000	23.0000
9.0000	24.0000	24.0000	15.0000
21.3333	21.3333	21.3333	0
15.0000	15.0000	15.0000	0
0.0100	0	0	0
15.0000	15.0000	0	0
-0.0100	0	0	0
15.0000	0	0	0

~~~~~> it is an unstable system! <~~~~~

 Number of right hand side poles = 2
Do you want roots of system be shown? Y/N Y

 Given polynomial coefficients roots :



```
sysRoots =

  -3.0000 + 0.0000i
  -5.0000 + 0.0000i
   0.5000 + 0.8660i
   0.5000 - 0.8660i
  -0.5000 + 0.8660i
  -0.5000 - 0.8660i
  -1.0000 + 0.0000i
```

Result 4: Finding k value

```
input vector of your system coefficients:
 i.e. [an an-1 an-2 ... a0] = [1 k 2 1]
Routh-Hurwitz Table for k = 1.500000
 [  1, 2]
[3/2, 1]
[4/3, 0]
[  1, 0]


 Routh-Hurwitz Table:
~~~~~> it is a stable system! <~~~~~

 Number of right hand side poles = 0
```

RESULT:
Thus stability analysis using Pole zero maps and Routh Hurwitz Criterion in the simulation platform is performed.

Ex. No:7	**Root Locus based Analysis in Simulation Platform**
Date:	

Aim

To perform Root Locus based analysis in a simulation platform.

Introduction to Root Locus Technique

The Root Locus Technique is a powerful and graphical method used in control system engineering to analyze the behavior of a closed-loop control system as a parameter, typically a proportional gain or a complex variable, is varied. It provides valuable insights into how changing system parameters affect the stability and transient response of a control system.

The fundamental concept behind the Root Locus Technique is to plot the trajectories of the closed-loop poles of the system as a parameter is adjusted. These poles represent the characteristic roots of the system's transfer function, and their locations in the complex plane directly influence system behavior. By visualizing the movement of these poles, engineers can make informed decisions to design a control system that meets desired performance specifications.

1. Open-Loop Transfer Function:To begin, you must have the open-loop transfer function of the system. This transfer function captures the relationship between the input and the output of the system before feedback is applied.

2. Closed-Loop Poles: The technique focuses on determining the locations of the closed-loop poles. The stability and transient response of the system depend on the positions of these poles.

3. Parameter Variation: Typically, the parameter being varied is a proportional gain (K), but it can also be other system parameters, such as time constants or damping ratios.

4. Complex Plane: The complex plane is used to represent the possible locations of the closed-loop poles. The real part of the poles positions corresponds to the system's damping and speed of response, while the imaginary part affects the system's oscillatory behavior.

5. Root Locus Plot: By varying the parameter over a range, you can create a Root Locus Plot, which is a graphical representation of the pole trajectories as the parameter changes. This plot helps you visualize how changes in the parameter affect system stability.

6. Stability Analysis: The Root Locus Plot can quickly indicate whether the system is stable, marginally stable, or unstable for different parameter values. Stable systems have poles with negative real parts.

7. Design Insights: Engineers can use the Root Locus Technique to design control systems that meet specific performance criteria, such as desired damping ratios, settling times, or overshoot limits. By adjusting the parameter value, they can shape the locus to meet these goals.

8. Iterative Process:Root locus analysis often involves an iterative process of adjusting the parameter and observing the pole movement until the desired system behavior is achieved.

Procedure:
Step 1. To locate the open loop poles and zeros
Step 2. Locate the root locus on real axis
Step 3. Find the angle of asymptote and centroid
Step 4. To find break away or break in points
Step 5. Determine angle of departure or angle of arrival if conjugate pole pairs are present.
Step 6. Find the point of intersection at the imaginary axis and its corresponding gain.

Problem 1:

Matlab code:

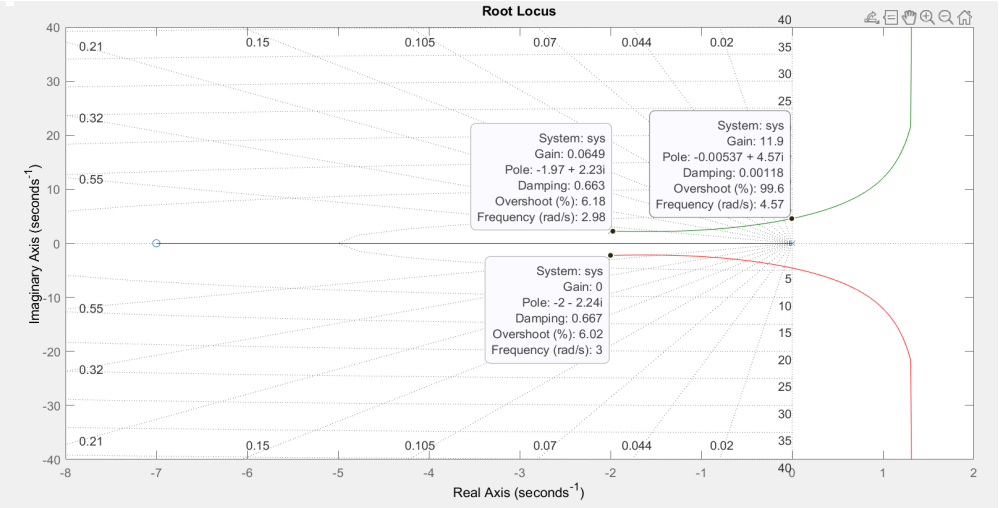
```
clearvars;
close all;
%% Transfer function
sys = tf([1 7],[1 4 9 0]);
P = pole(sys);
disp(P)
rlocus(sys);
[r,k] = rlocus(sys,k)
% returns poles at different gain values
%k= 1:1:100; % user defined gain values
whos
% Get crossover gain from the plot and use the value in simulation
```

Output:

Root locus:

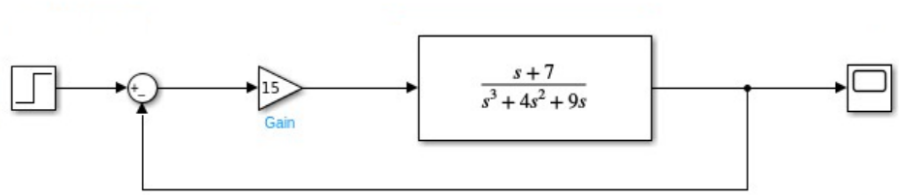
```
   0.0000 + 0.0000i
  -2.0000 + 2.2361i
  -2.0000 - 2.2361i
```

Plot:



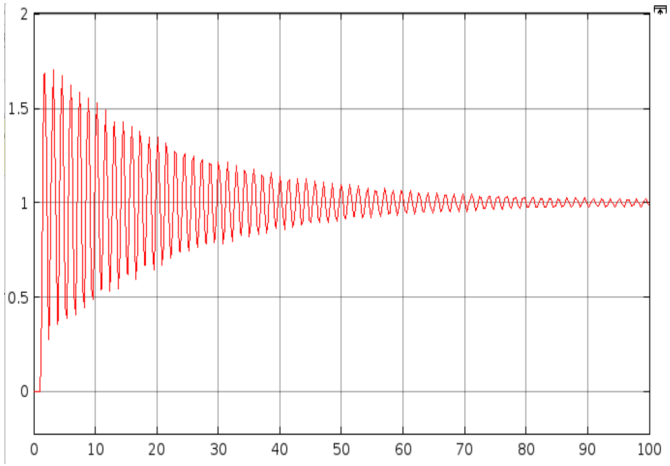
Inference:

Simulation:

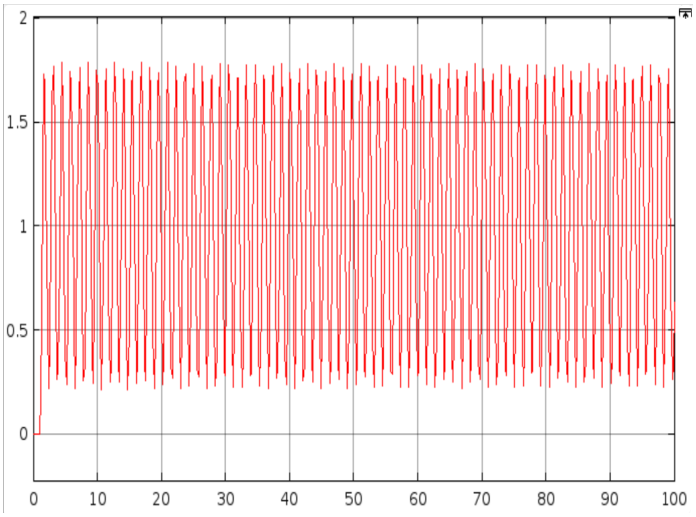


Simulation Results:

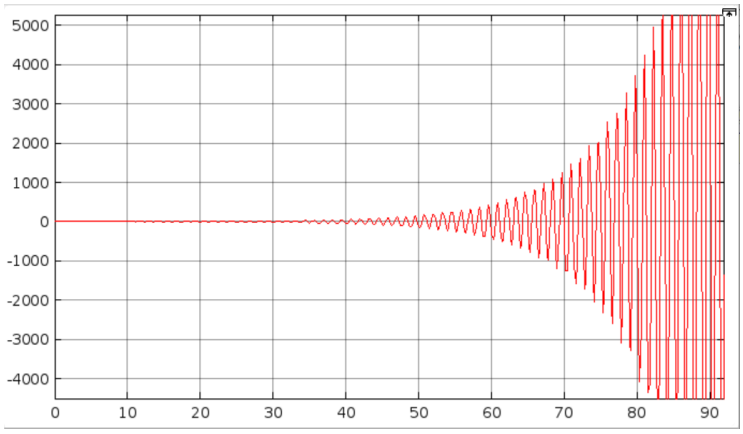
For k = 11 **Inference:**



For k = 12 **Inference:**



For k = 15



Inference:

Problem:2

Matlab code:

```
clearvars;
close all;
%% Transfer function
sys = tf([1],[1 6 8 0]);
P = pole(sys);
disp(P)
rlocus(sys);
%%
% returns poles at different gain values
k= 1:1:100; % user defined gain values
[r,k] = rlocus(sys,k);% k can be default also ( positive to inf)
whos
% Get cross over gain from the plot and use the value in simulation
```

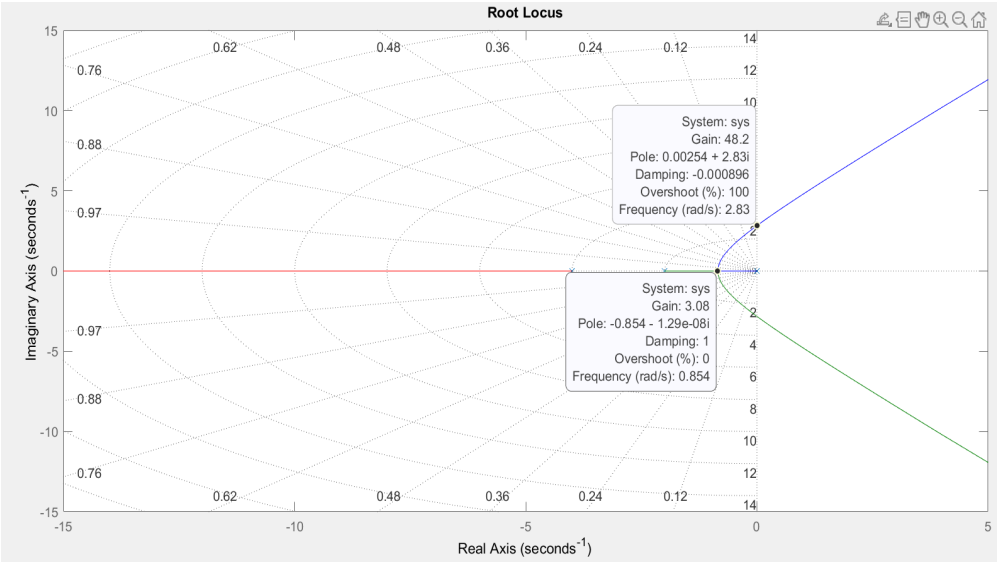
Output:

Root locus:

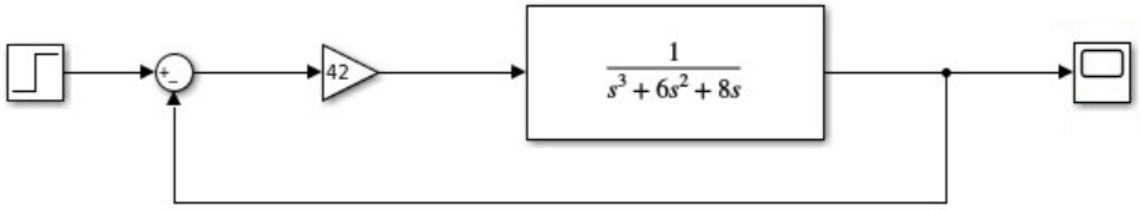
0
-4
-2

Inference:

Plot:

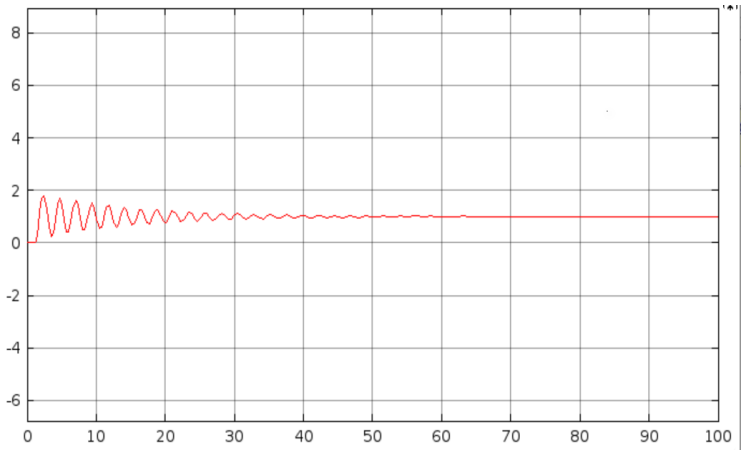


Simulation:

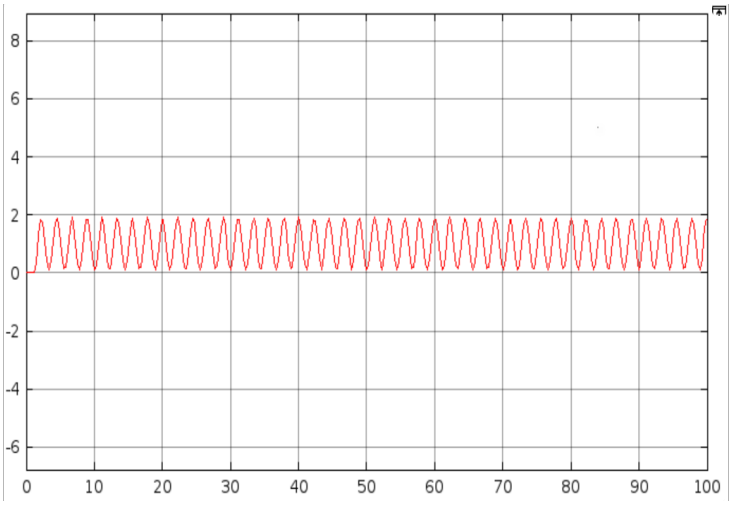


Simulation result:

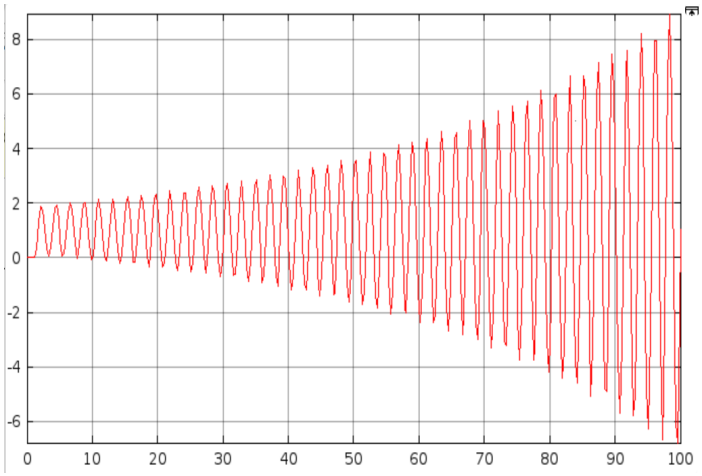
For k = 42 **Inference:**



For k = 48 **Inference:**



For k = 50 **Inference:**



Result:

Thus Root Locus based analysis in a simulation platform is performed and verified using Matlab code.

Exp. No.:8	**Determination of Transfer Function of a Physical System using Frequency Response and Bode's Asymptotes**
Date:	

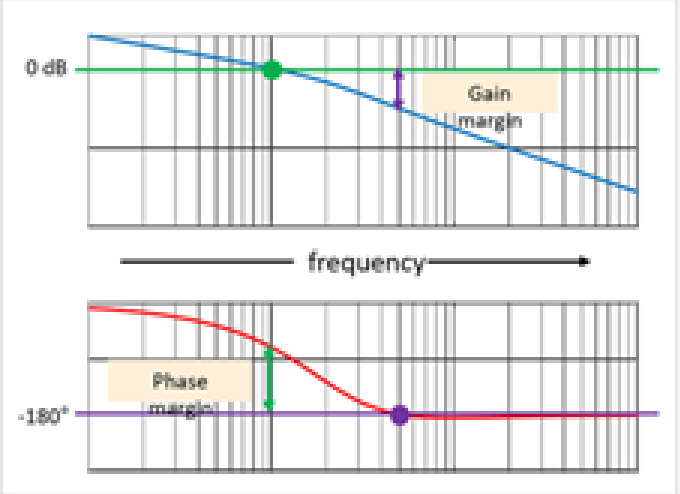
Aim

To determine the transfer function of a physical system using frequency response and Bode's asymptotes

Introduction

Bode plots were originally devised by Dr. Henrik Wayne Bode while he was working for Bell Labs in the 1930s. They are most used to analyze the stability of control systems, for example when designing and analyzing power supply feedback loops. The advantage of using Bode plot is that they provide a straightforward and common way of describing the frequency response of a linear time invariant system.

Phase and gain margins



Phase margin is measured at the frequency where gain equals 0 dB. This is commonly referred to as the "crossover frequency". Phase margin is a measure of the distance from the measured phase to a phase shift of -180°. In other words, how many degrees the phase must be decreased in order to reach -180°.

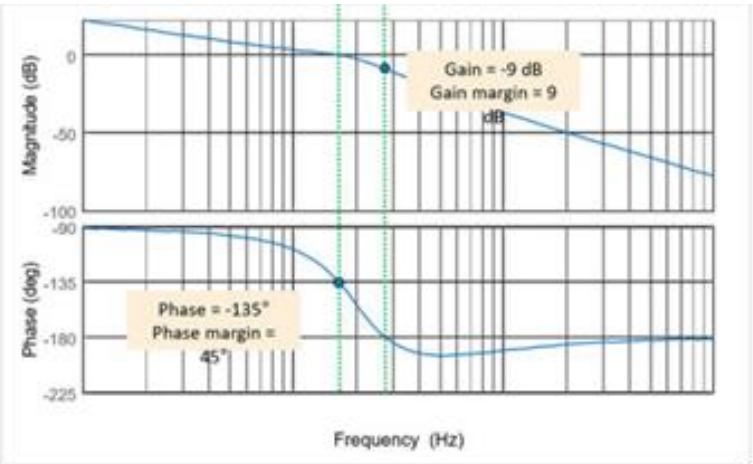
Gain margin, on the other hand, is measured at the frequency where the phase shift equals -180°. Gain margin indicates the distance, in dB, from the measured

gain to a gain of 0 dB. These values, 0 dB and -180° are important because system instability occurs if these two values meet.

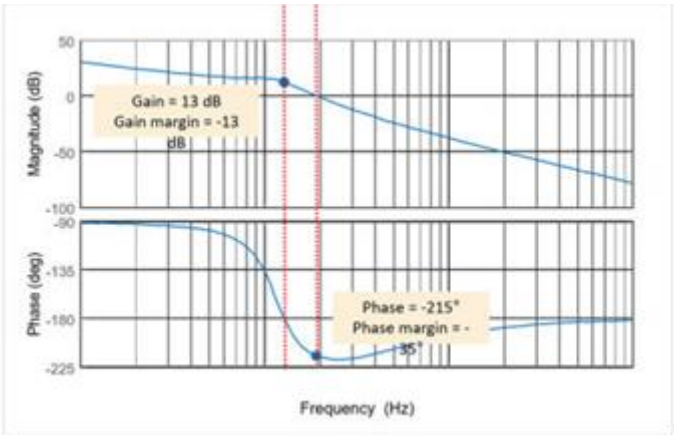
Gain and phase margins represent the distance from the points at which instability could occur. The greater the distance or margin the better, because higher gain and phase margins mean greater stability. A loop with

a gain margin of zero or even less would only be conditionally stable and could easily become unstable if gain changed. A typical goal for phase margin is to have at least 45 degrees, and even higher values might be desirable in more critical applications.

Stable and unstable closed loop systems



The measured phase at 0 dB is -135°, so the phase margin is 45°. The gain at -180° degrees is -9 dB, so the gain margin is 9 dB. Since phase margin is positive, this system is stable.



The measured gain is +13 dB when phase is -180°, so the gain margin is minus 13 dB. At a gain of 0 dB, the measured phase is minus 215°, so the phase margin is minus 35° at the gain crossover point. This system is unstable

Procedure:

Step1: Get the transfer function

Step2: Declare an array of gain values

Step3: Plot the Bode plot and find the phase margin and gain margin for all the gain.

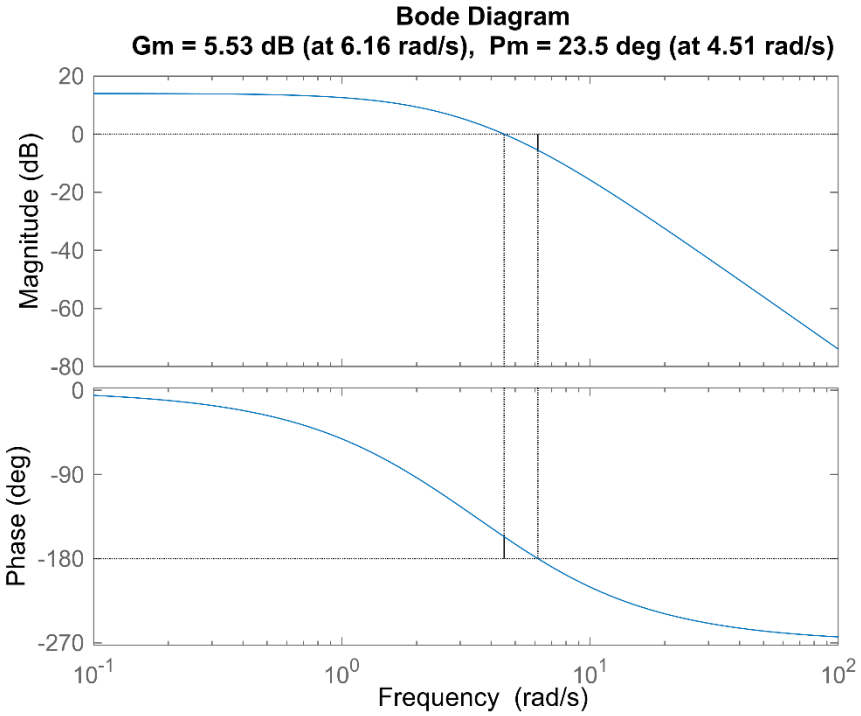
Matlab code

Initialize Gain Values and Frequency Range

```
clear vars;
k=[5 10 15]';
gain_len = length(k);
w = logspace(-
1,2,200); [r,c]
= size(w);
mag_db = zeros(c,gain_len);
phase_a11 = zeros(c,gain_len);
```

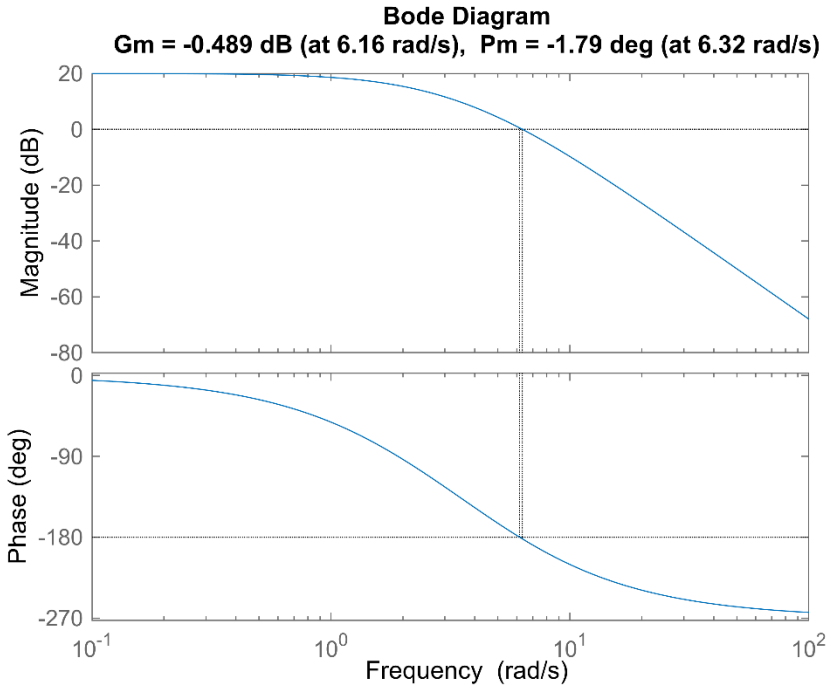
Calculate Magnitude, Phase , Gain Margin and Phase Margin

```
for i=1:gain_len
    [mag,phase,w]=bode([k(i)],[0.025 0.275
    0.95 1],w); [Gm,Pm,Wcg,Wcp] =
    margin(mag,phase,w);
    figure;
    margin(mag,phase,w);
    fprintf(['For Gain K = %d Gain Margin Gm is : %0.2f and Phase Cross freq.
    Wcp :
%0.2f \n' ...
        'Phase Margin is :%0.2f and Gain crossover freq Wgc: %0.2f
\n'],k(i), Gm, Wcp, Pm ,Wcg);
    mag_db(:,i) = 20*log10(mag);
    phase_a11(:,i) = phase;
end
```



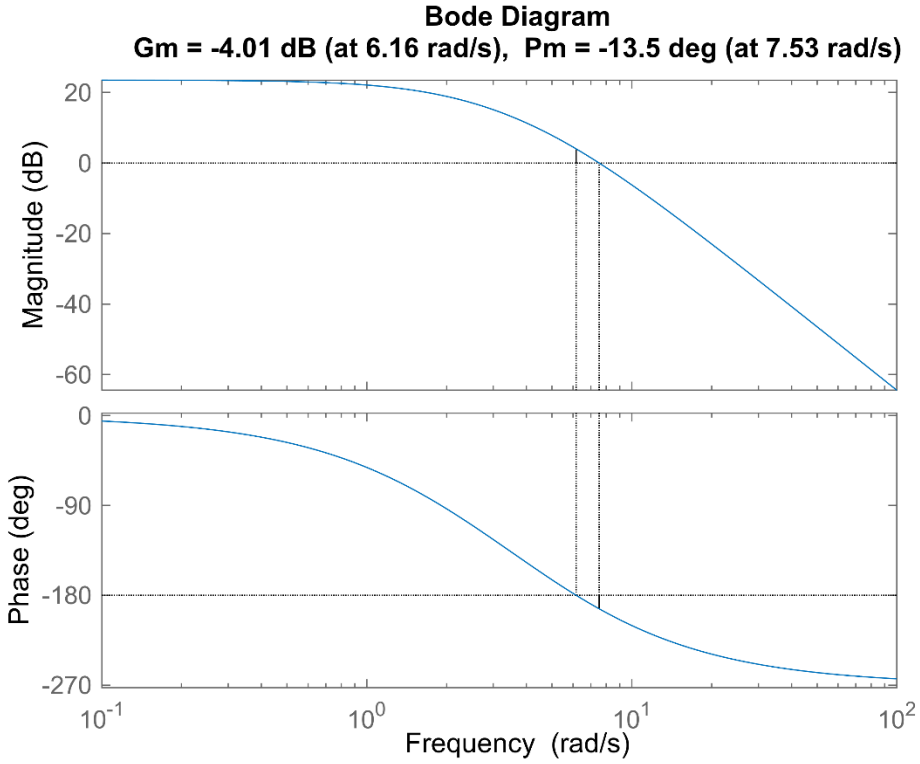
For Gain K = 5 Gain Margin Gm is : 1.89 and Phase Cross freq. Wcp : 4.51

Phase Margin is :23.50 and Gain crossover freq. Wgc: 6.16



For Gain K = 10 Gain Margin Gm is : 0.95 and Phase Cross freq. Wcp : 6.32

Phase Margin is :-1.79 and Gain crossover freq Wgc: 6.16



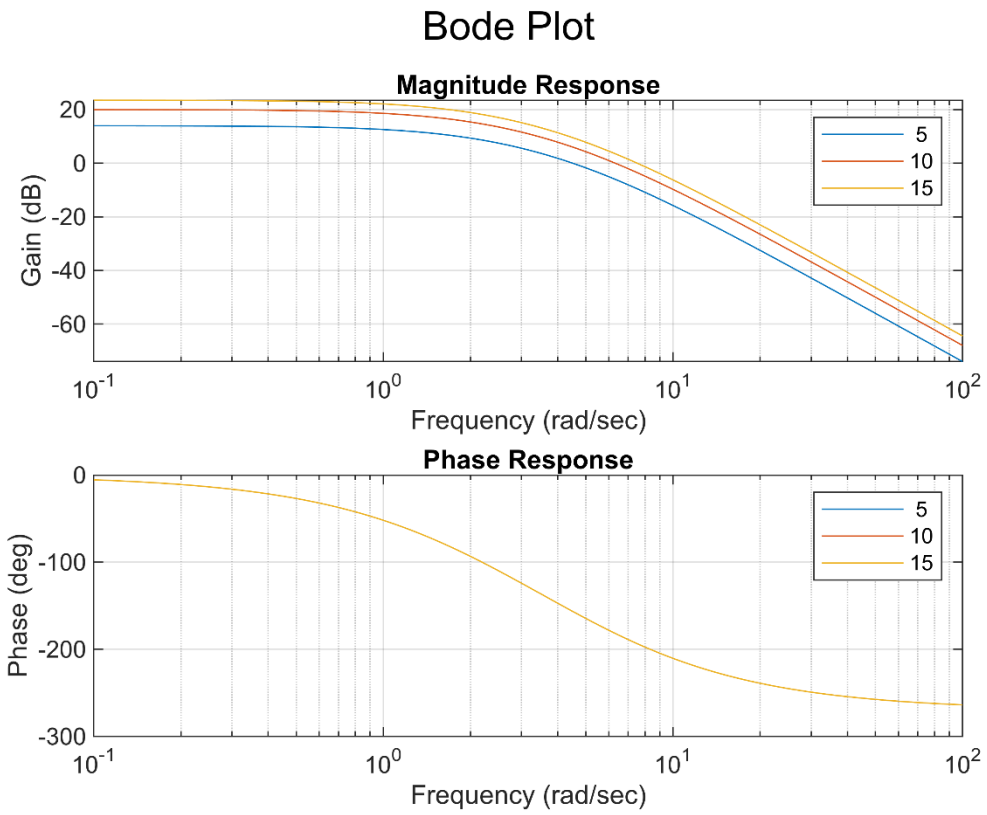
For Gain K = 15 Gain Margin Gm is : 0.63 and Phase Cross freq. Wcp : 7.53

Phase Margin is :-13.53 and Gain crossover freq Wgc: 6.16

Bode Plot for Different Gain Values in Single Plot

Bode Plot

```
  subplot(2,1,1);
  semilogx(w,mag_db);
  title('Magnitude Response')
  xlabel('Frequency
  (rad/sec)'); ylabel('Gain
  (dB)'); legend(num2str(k));
  grid subplot(2,1,2);
  phase_all =
  phase_a11';
  semilogx(w,phase_a11)
  ; title('Phase
  Response')
  xlabel('Frequency
  (rad/sec)'); ylabel('Phase
  (deg)'); legend(num2str(k));
  grid
sgtitle('Bode Plot')
```



Inference

Result

Thus, the transferred function has been determined from frequency response and Bode's asymptotes

Exp. No.9 Date:	**Design of Lag, Lead Compensators and Evaluation of Closed Loop Performance**

Aim:

The design lag lead compensator and evaluate the closed loop performance.

Introduction:

Lag and lead compensators play a crucial role in control systems, addressing challenges in system response, transient behavior, and steady-state performance. This topic explores the nuanced design and evaluation of these compensators, aiming to minimize system error and optimize functionality. Through parameter tuning and rigorous assessment of closed-loop performance, valuable insights into control system design principles are gained, applicable across various engineering contexts.

Procedure:

Step 1: Analyze the dynamic characteristics of the given system and derive its transfer function.

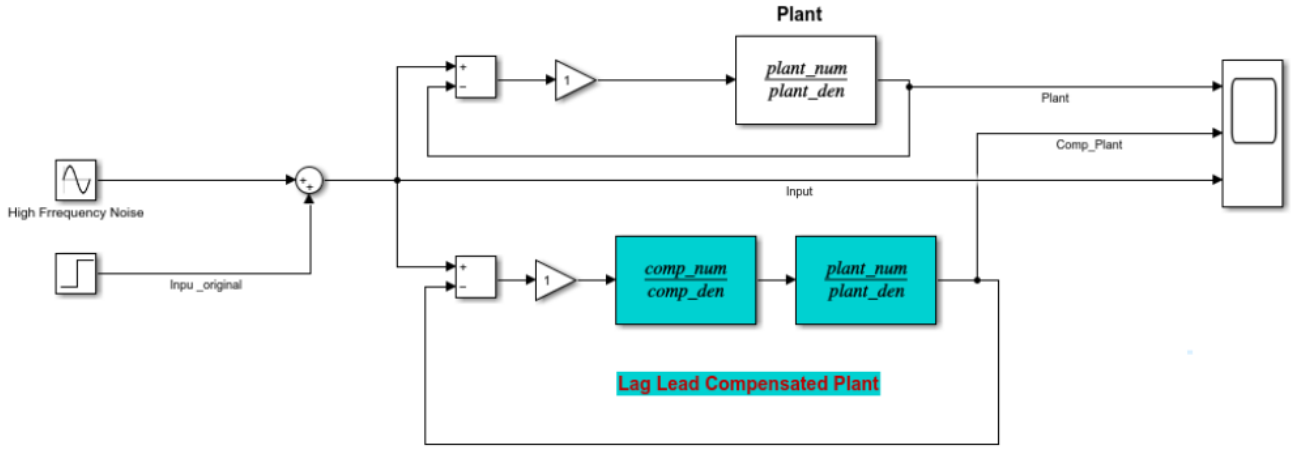
Step 2: Design the lag compensator by selecting appropriate parameters like time constant and gain.

Step 3: Design the lead compensator by choosing suitable parameters such as time constant and gain.

Step 4: Implement the designed lag and lead compensators in MATLAB using the Control System Toolbox.

Step 5: Analyze the closed-loop response, evaluating transient behavior, steady-state error, and overall stability.

MATLAB Circuit:

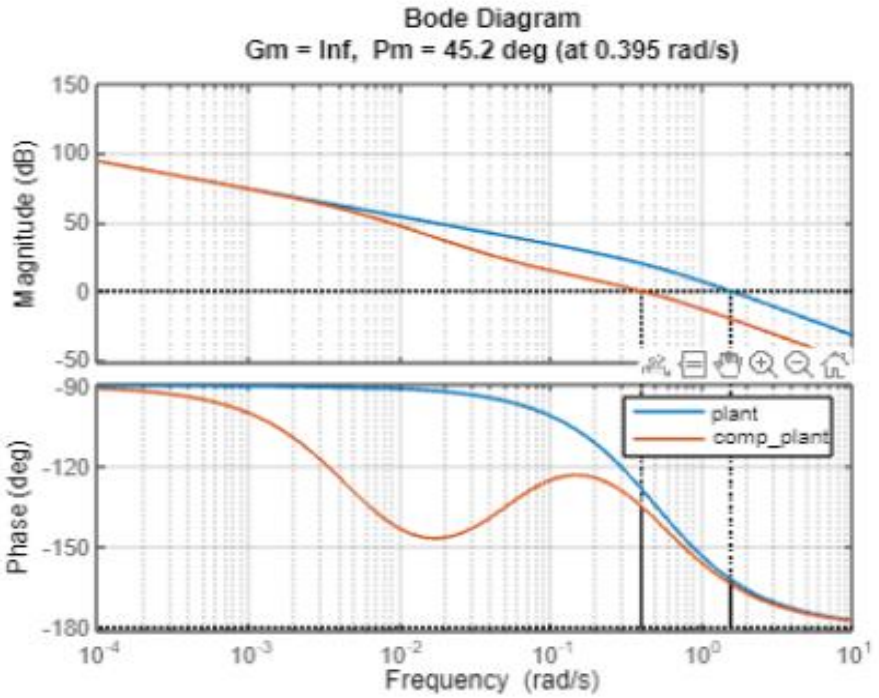


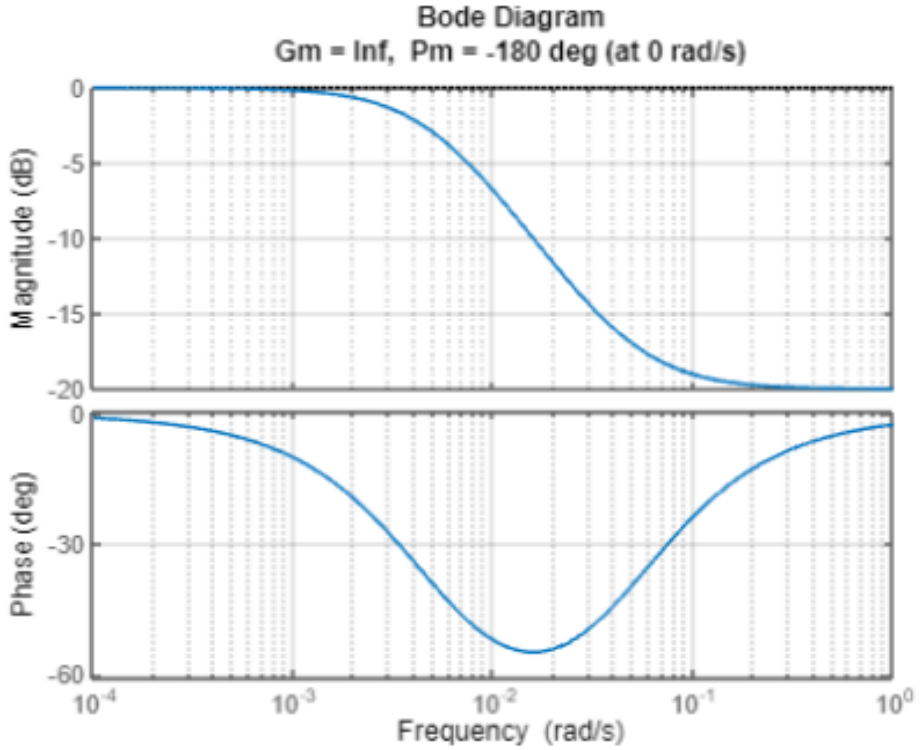
MATLAB Code : Lag Compensator

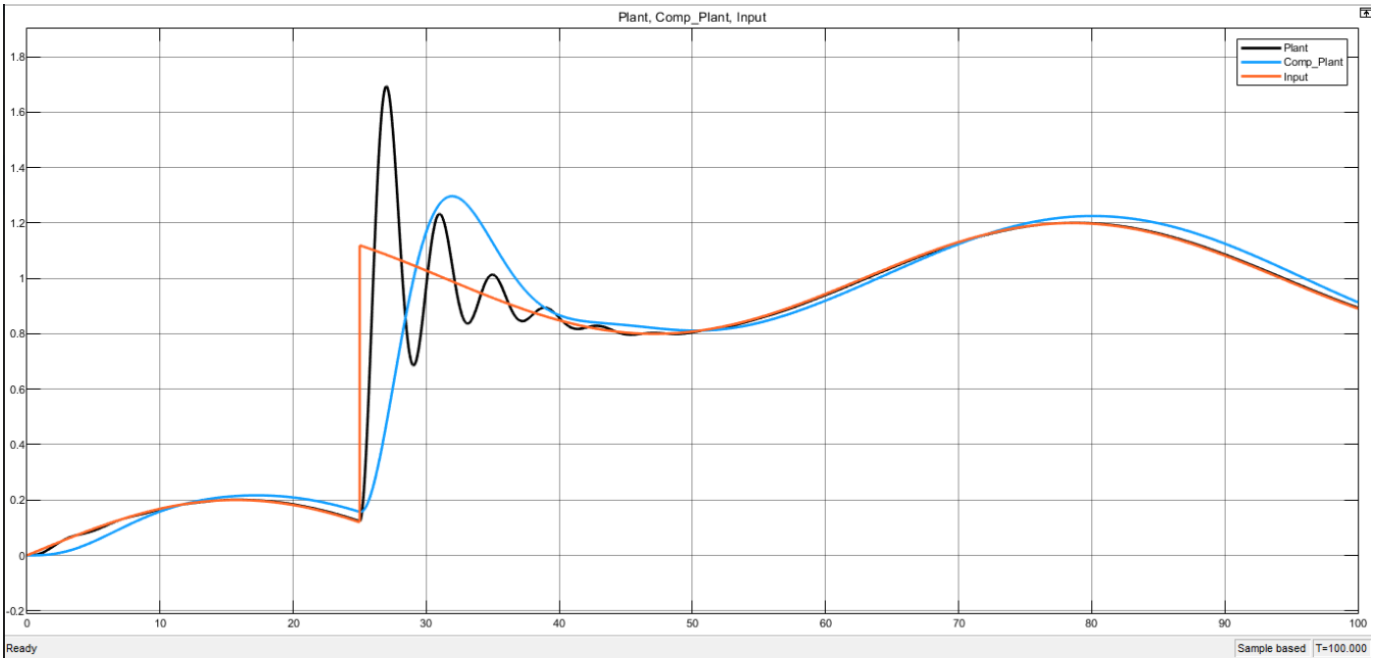
clc;

```
close all;
clearvars;
% Lag Compensator Design
plant_num = [5];
plant_den = [2 1 0];
plant_tf = tf(plant_num,plant_den);

comp_num = [20 1];
comp_den = [200 1];
comp_tf = tf(comp_num,comp_den);

% Plotting Gain and Phase margin for plant
figure;
margin(plant_tf);
grid on;
% Compensated System
lag_comp_sys = plant_tf*comp_tf;
disp(lag_comp_sys);
hold on;
margin(lag_comp_sys);
legend ('plant', 'comp_plant');
% Plotting Gain and Phase margin for compensator
figure;
margin(comp_tf);
grid on;
```

Output Waveform



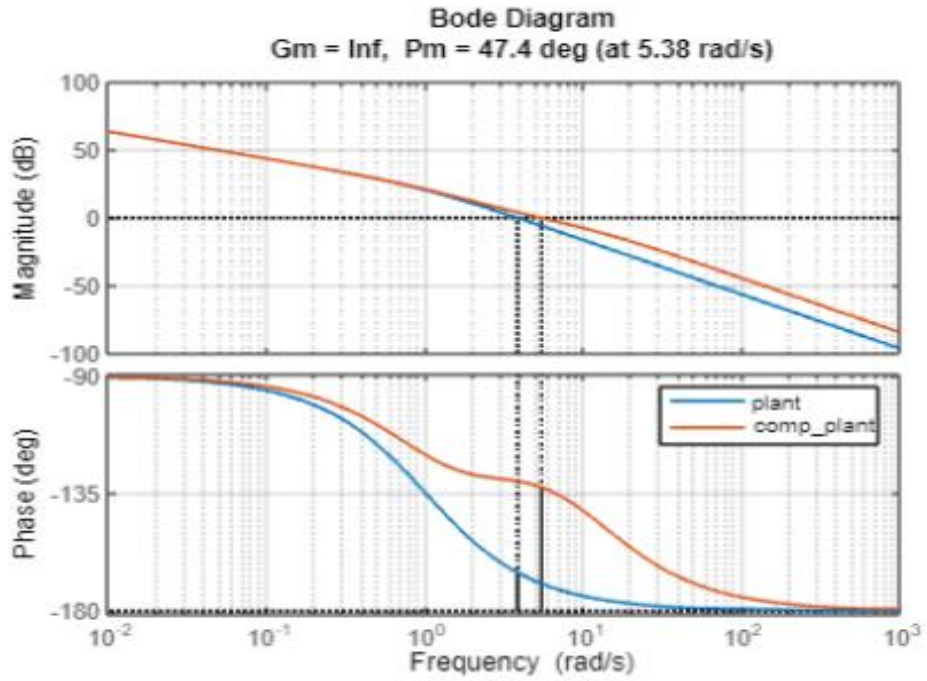


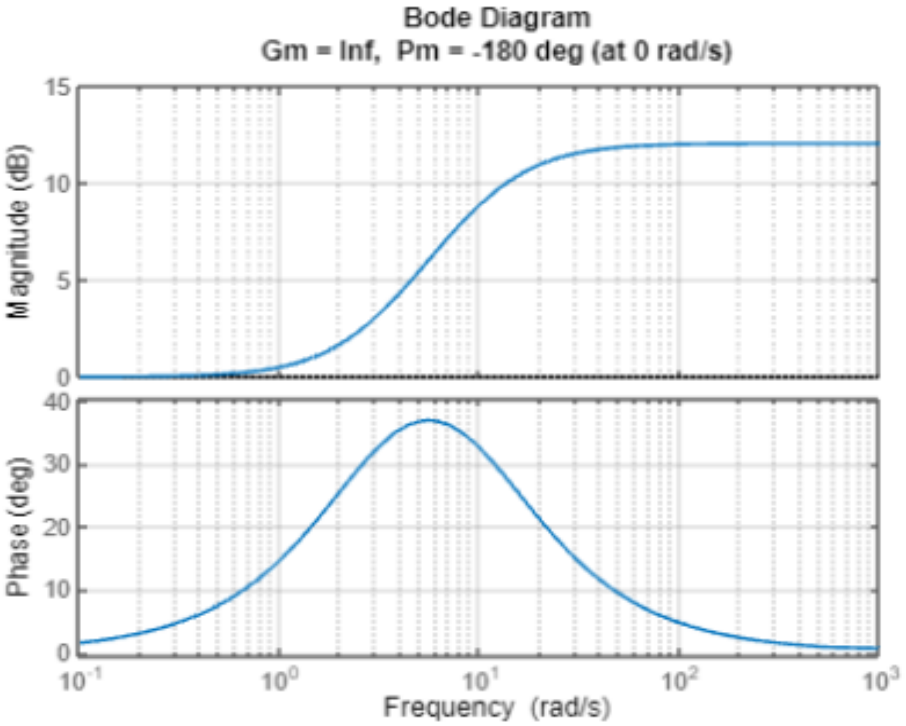


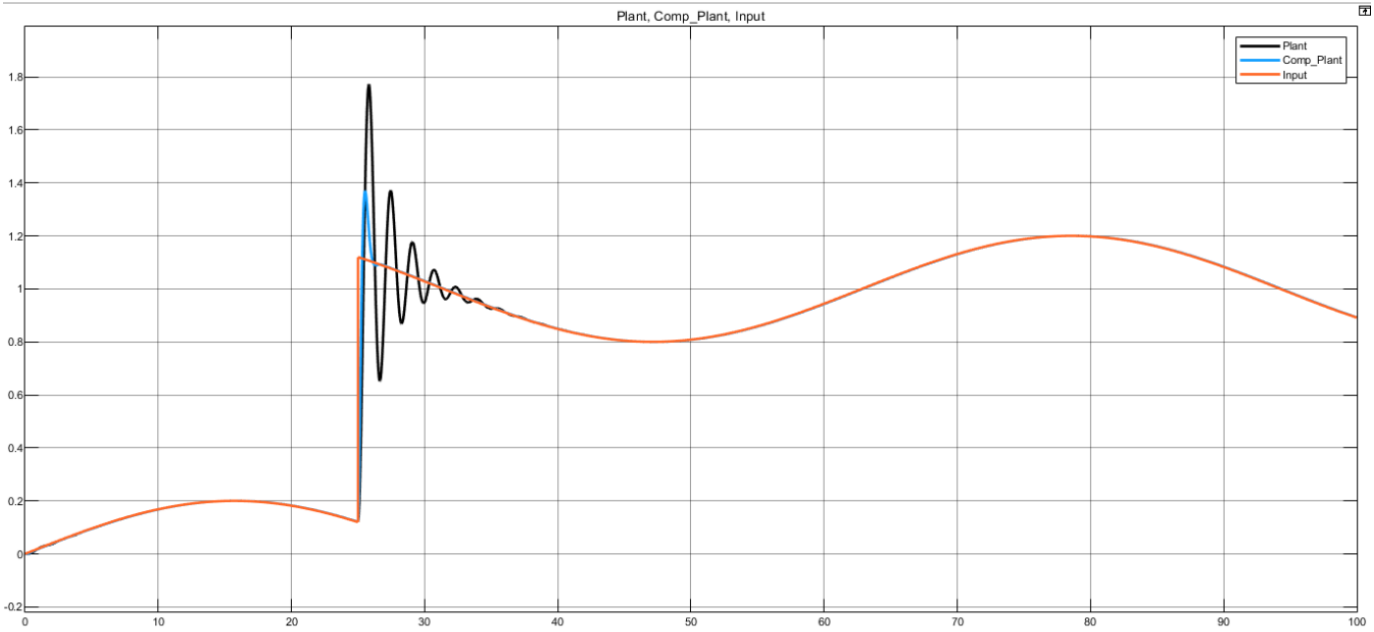
MATLAB Code: Lead Compensator

```
clc;
close all;
clearvars;
% Lead Compensator Design
plant_num = [15];
plant_den = [1 1 0];
plant_tf = tf(plant_num,plant_den);

comp_num = [0.36 1];
comp_den = [0.09 1];
comp_tf = tf(comp_num,comp_den);

% Plotting Gain and Phase margin for plant
figure;
margin(plant_tf);
grid on;
hold on;
% Compensated System
lead_comp_sys = plant_tf*comp_tf;
disp(lead_comp_sys);
margin(lead_comp_sys);
hold off;
legend ('plant', 'comp_plant');
% Plotting Gain and Phase margin for compensator
figure;
margin(comp_tf);
grid on;
```

Output Waveform







Inference

Result

Thus, the lag, lead compensator has been simulated and the output waveform has been analyzed.

Ex. No:10	**Design of PID Controllers and Evaluation of Closed Loop Performance**
Date:	

Aim:

To design PID controllers and evaluate the closed loop performance.

Introduction

PID Controller

A PID controller (Proportional, Integral, Derivative) is a feedback control system used in engineering. It has three components:

1. Proportional (P): Responds to the current error, reducing steady-state error and providing a quick response.

2. Integral (I): Considers accumulated past errors to eliminate steady-state error over time.

3. Derivative (D): Anticipates future errors by analyzing the rate of change of the error, contributing to stability.

PID Controller Design

The design of a PID controller involves tuning the three components (P, I, and D) to achieve the desired system performance. The tuning process is often an iterative one and involves adjusting the parameters to balance the trade-offs between responsiveness, stability, and minimizing overshoot.

1. Proportional Gain (K_p):
   - Adjusting the proportional gain determines how much the controller responds to the current error.
   - Increasing K_p makes the system more responsive but may lead to overshooting and oscillations.
   - Decreasing K_p may result in sluggish response and increased steady-state error.

2. Integral Gain (K_i):
- The integral gain determines the impact of accumulated past errors on the control action.
- Increasing K_i helps eliminate steady-state error but may introduce overshoot and oscillations.
- Decreasing K_i can lead to a faster response but may result in a larger steady-state error.

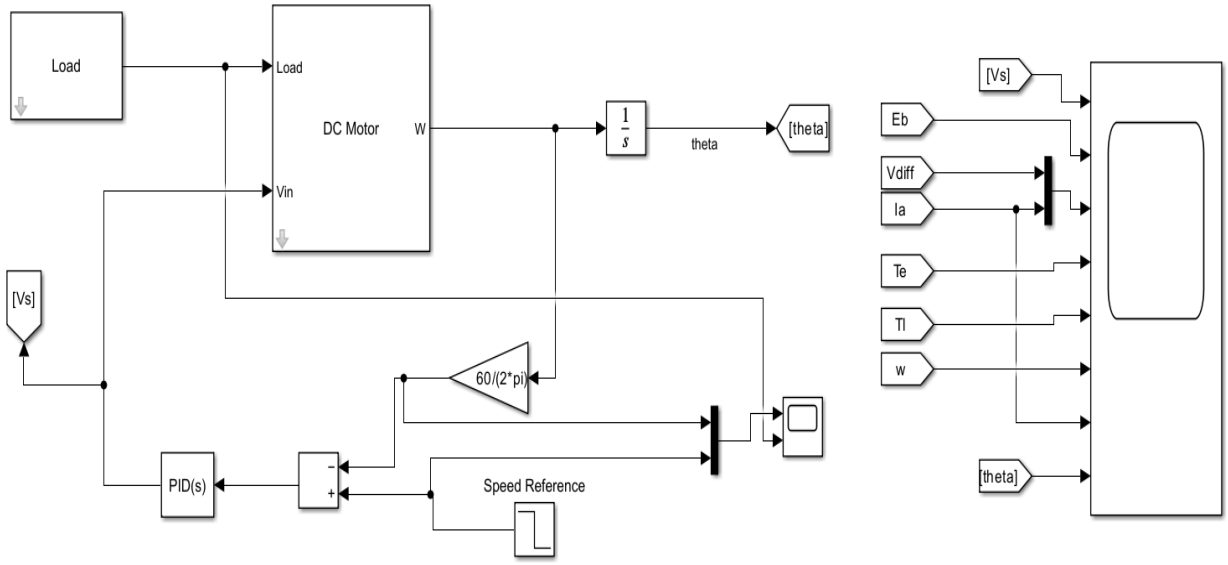
3. Derivative Gain (K_d):
- The derivative gain influences the controller's response to the rate of change of the error.
- Increasing K_d improves stability and reduces overshoot but may slow down the system's response.
- Decreasing K_d can lead to increased overshoot and oscillations.

Tuning:
- PID controller tuning is often done experimentally, adjusting the gains until the desired system behavior is achieved.
- Various methods, such as trial-and-error, Ziegler-Nichols, or optimization algorithms, can be used for tuning.

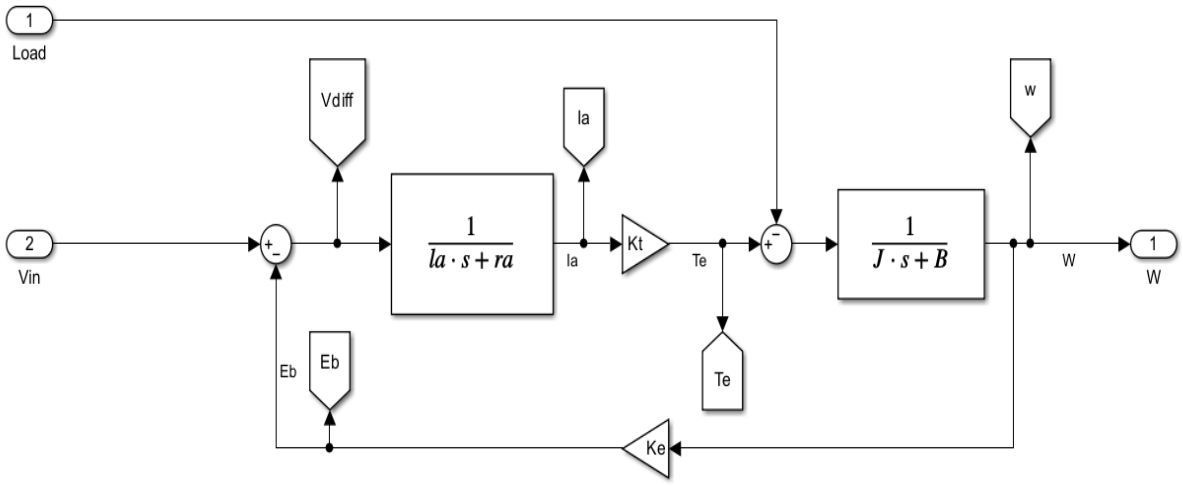
Procedure

Step 1. Obtain an open-loop response and determine what needs to be improved
Step 2. Add a proportional control to increase the rise time.
Step 3. Add a derivative control to reduce the overshoot.
Step 4. Add an integral control to reduce the steady-state error
Step 5. Adjust each of the gains Kp, Ki, and Kd until you obtain a desired overall response. You can always refer to the table shown in this "PID Tutorial" page to find out which controller controls which characteristics.

Matlab Circuit:

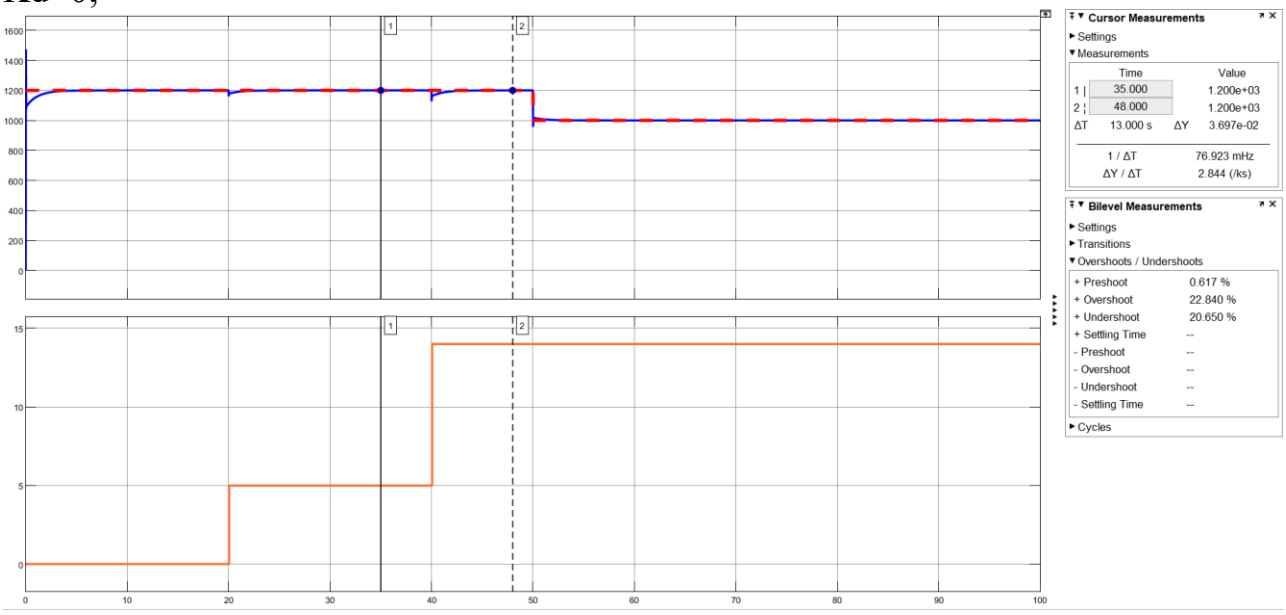


DC Motor Subsystem

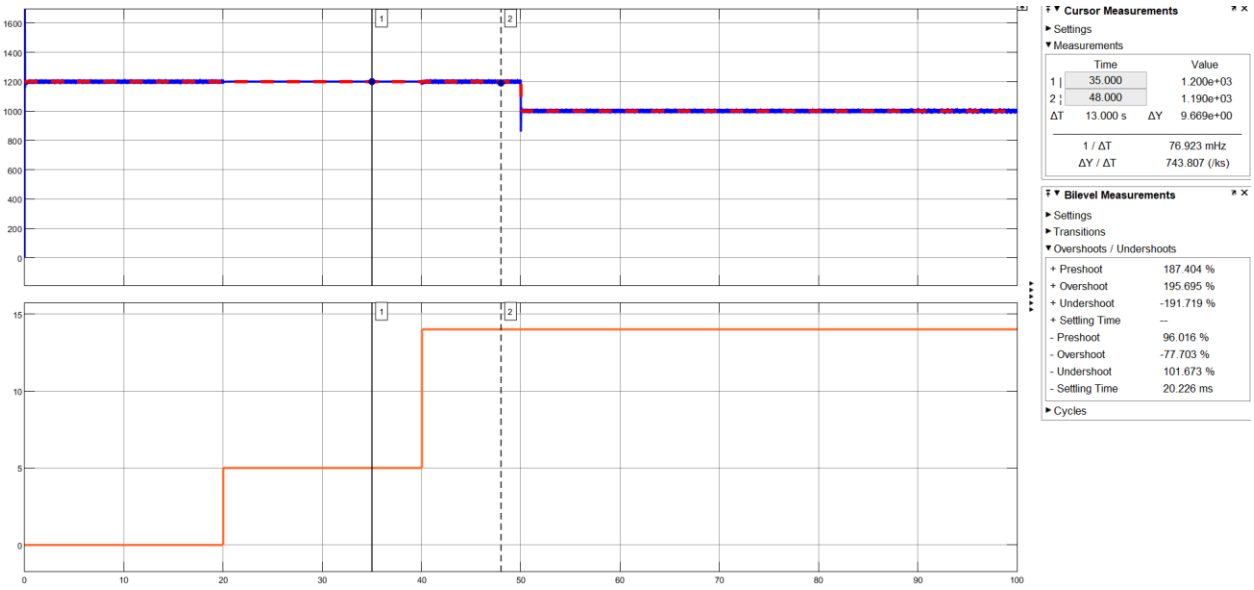


Trail and Error Method

Kp=1;
Ki=1;
Kd=0;



Kp = 10
Ki = 50
Kd = 0

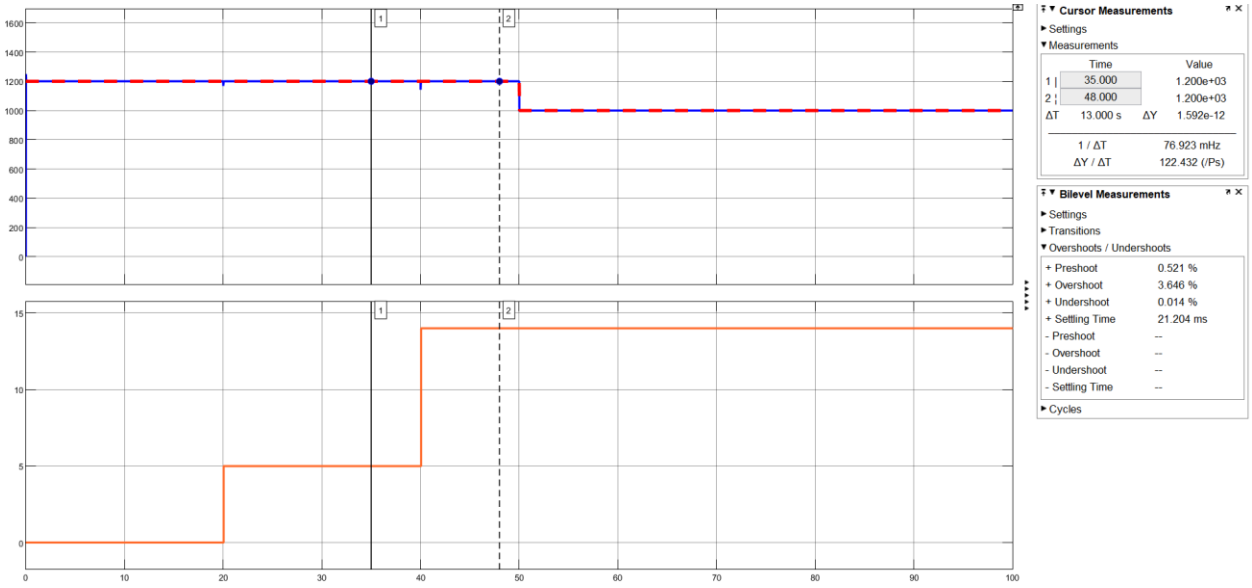


After Tuning

Kp = 0.68

Ki = 31.87

Kd = 0.0036



Inference

Result

Thus, design of PID controllers and evaluation of closed loop assessment performed using Matlab Simulink.

Ex. No.: 11	**Discretization of Continuous System and Effect of Sampling**
Date:	

Aim

To write a Matlab code for discretization of continuous system and effect of sampling and understand the results.

Procedure

Step 1: Open blank script file in Matlab.

Step 2: Input the transfer function with delay and define the sampling interval.

Step 3: Use c2d function to convert continuous to discrete system and plot it.

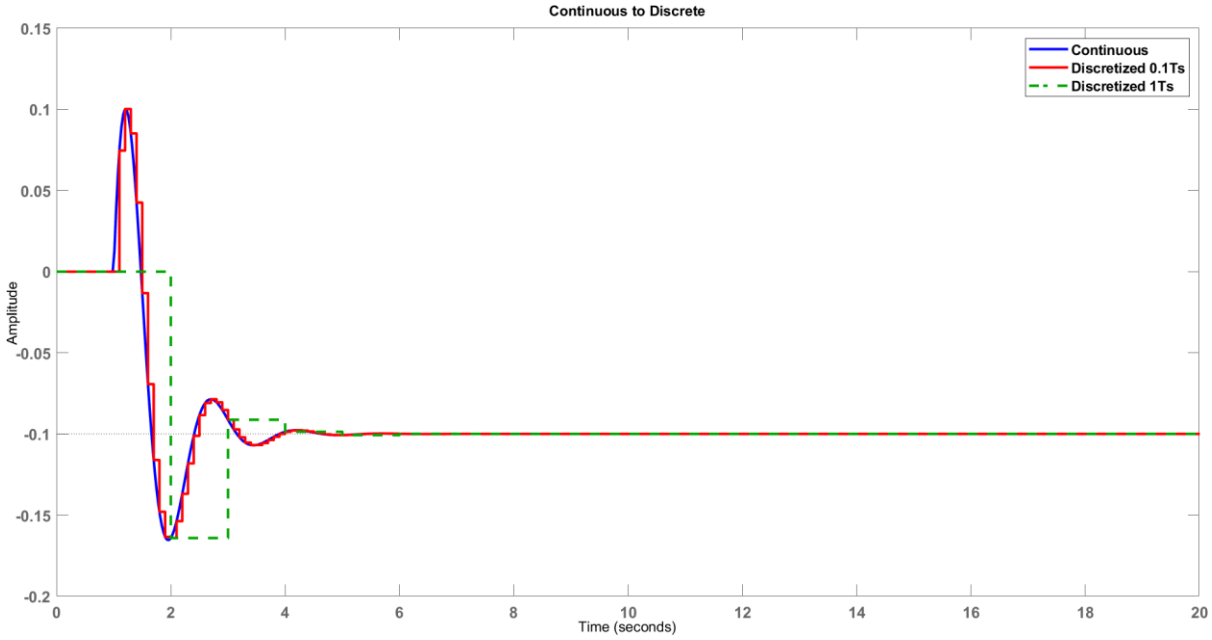
Step 4: Use d2c function to convert back to continuous system and observe the reproducibility with respect to sampling time.

Program:

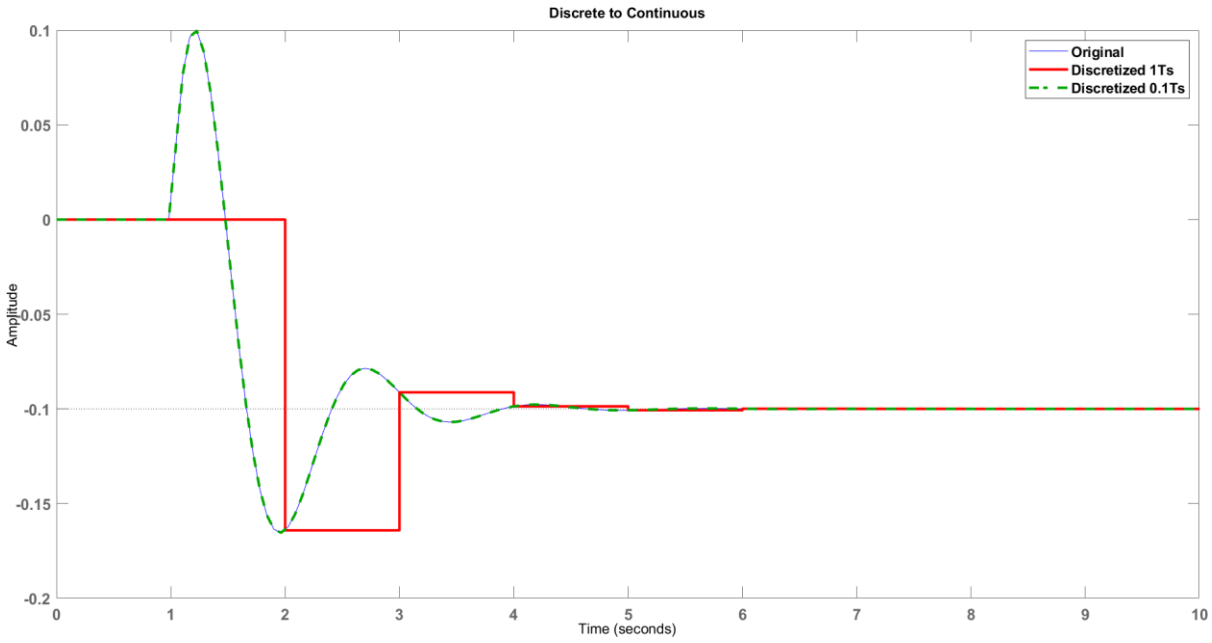
```
close all;
clearvars;
% Transfer Function
G = tf([1 -2],[1 3 20],'inputdelay',1);

Ts = 0.1;   % sampling interval
Gd = c2d(G,Ts);
Ts = 1;  % 10 times larger than previously
Hd = c2d(G,Ts);
% Compare the continuous and discrete step responses:
figure;
step(G,'b',Gd,'r',Hd,'g--')
legend('Continuous','Discretized 0.1Ts','Discretized 1Ts')


figure;
Hc = d2c(Hd);
Gc = d2c(Gd);
step(G,'b',Hd,'r',Gc,'g--',10)
legend('Original','Discretized 1Ts','Discretized 0.1Ts')
```

Output Waveform:

Continuous to Discrete:



Discrete to Continuous:



Result:

Thus, the discretization of continuous system and effect of sampling is simulated in Matlab and the reproducibility is been observed.

Ex. No.:12	**Test of Controllability and Observability in Continuous and Discrete Domain in Simulation Platform.**
Date:	

Aim:

To Test Controllability and Observability for the given state model representation of system in simulation platform.

Introduction:

Controllability:

A system is completely controllable if the initial state of the system is transferred to any particular state, in a finite time duration, when a controlled input is provided to it.

There are several ways to test for controllability, including the Kalman test and the Hautus test. The Kalman test checks if the controllability matrix is of full rank, while the Hautus test checks if the eigenvalues of the matrix are all nonzero. In general, these tests are performed in the frequency domain and are used to generate a controllability Gramian, which is a measure of the effectiveness of the control inputs in steering the system towards a desired output.

It is important to note that while these tests are useful in determining controllability, they do not guarantee that a system is controllable. Other factors, such as system noise and disturbances, can also affect the controllability of a system. The rank of the quadratic form is equal to the number of non-zero Eigen values of the matrix of quadratic form.

Observability:

Observability of a control system is the ability of the system to determine the internal states of the system by observing the output in a finite time interval when input is provided to the system.

There are several ways to test for observability, including the Kalman test and the Hautus test. The Kalman test checks if the observability matrix is of full rank, while the Hautus test checks if the eigenvalues of the matrix are all nonzero. These tests are important because they allow us to determine whether a system is observable before we try to control it. The rank of the quadratic form is equal to

the number of non-zero Eigen values of the matrix of quadratic form.

Procedure:

Step1: Obtain the state space model matrix of the system.

Step2: Determine Qc using formula

$$Qc = [B \;\; AB \;\; A^2 B \ldots\ldots A^{n-1} B]$$

Step3: Check for |Qc| is not equal to zero.

Step4: Determine Qo using formula

$$Qo = [C^T \; A^T C^T \;\; (A^T)^2 C^T \ldots\ldots\ldots (A^T)^{n-1} C^T]$$

Step5: Check for |Qo| is not equal to zero.

Step6: Find Rank for both Qc and Qo.

Matlab code:

```
%% Check for observability and controllability

clear
close all
clc

%% Consider the system representation
A=[ 1 0;0 0 1;-6 -11 -6];
B=[0; 0; 2];
C=[1 0 0];
D=0;
% eigenvaues (just for checking stability)
E=eig(A);

%% Controllability Test
Qc1=[B A*B A^2*B];
det_Qc1 = det(Qc1);
R1=rank(Qc1);
disp('Controllability Matrix');
disp(Qc1);
fprintf('Determinant Value %d\n',det_Qc1);
```

```
fprintf('Rank %d\n',R1);

%% Observability Test
At = A';
Ct=C';
Qo1=[Ct, At*Ct, (At)^2*Ct];
R2=rank(Qo1);
det_Qo1=det(Qo1);
disp('Observability Matrix');
disp(Qo1);
fprintf('Determinant Value %d\n',det_Qo1);
fprintf('Rank %d \n',R2);
```

Output:

```
Controllability Matrix
     0     0     2
     0     2   -12
     2   -12    50

Determinant Value -8

Rank 3


Observability Matrix
     1     0     0
     0     1     0
     0     0     1

Determinant Value 1

Rank 3
```

Inference:

Result:

Thus, the Controllability and Observability test in simulation platform is performed and verified.

Ex. No.:13	**State Feedback and State Observer Design and Evaluation of Closed Loop Performance**
Date:	

Aim:

To design State feedback and state observer model and to evaluate closed loop performance of DC motor.

Introduction:

State feedback control systems open up a different landscape to control system design for complex systems that have a higher order or have many input and output variables. In control theory, a state observer or state estimator is a system that provides an estimate of the internal state of a given real system, from measurements of the input and output of the real system. It is typically computer-implemented, and provides the basis of many practical applications.

Knowing the system state is necessary to solve many control theory problems; for example, stabilizing a system using state feedback. In most practical cases, the physical state of the system cannot be determined by direct observation. Instead, indirect effects of the internal state are observed by way of the system outputs.. By choosing a set of desired closed-loop eigenvalues, a state feedback controller is designed

Procedure

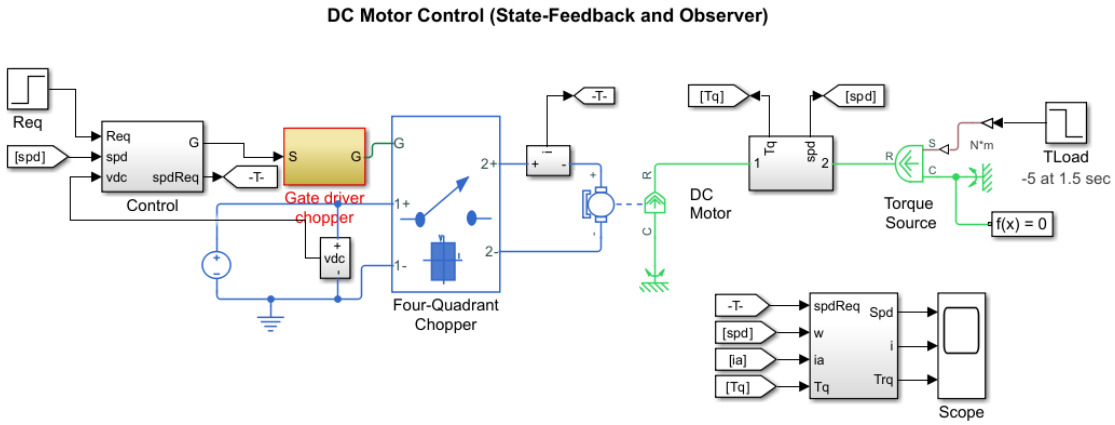
Step1: Calculate the state space model for the given DC motor

Step2: Check for controllability and observability

Step3: Build the state feedback and observer model in simscape using the calculated state variables

Step4: Simulate and observe the results.

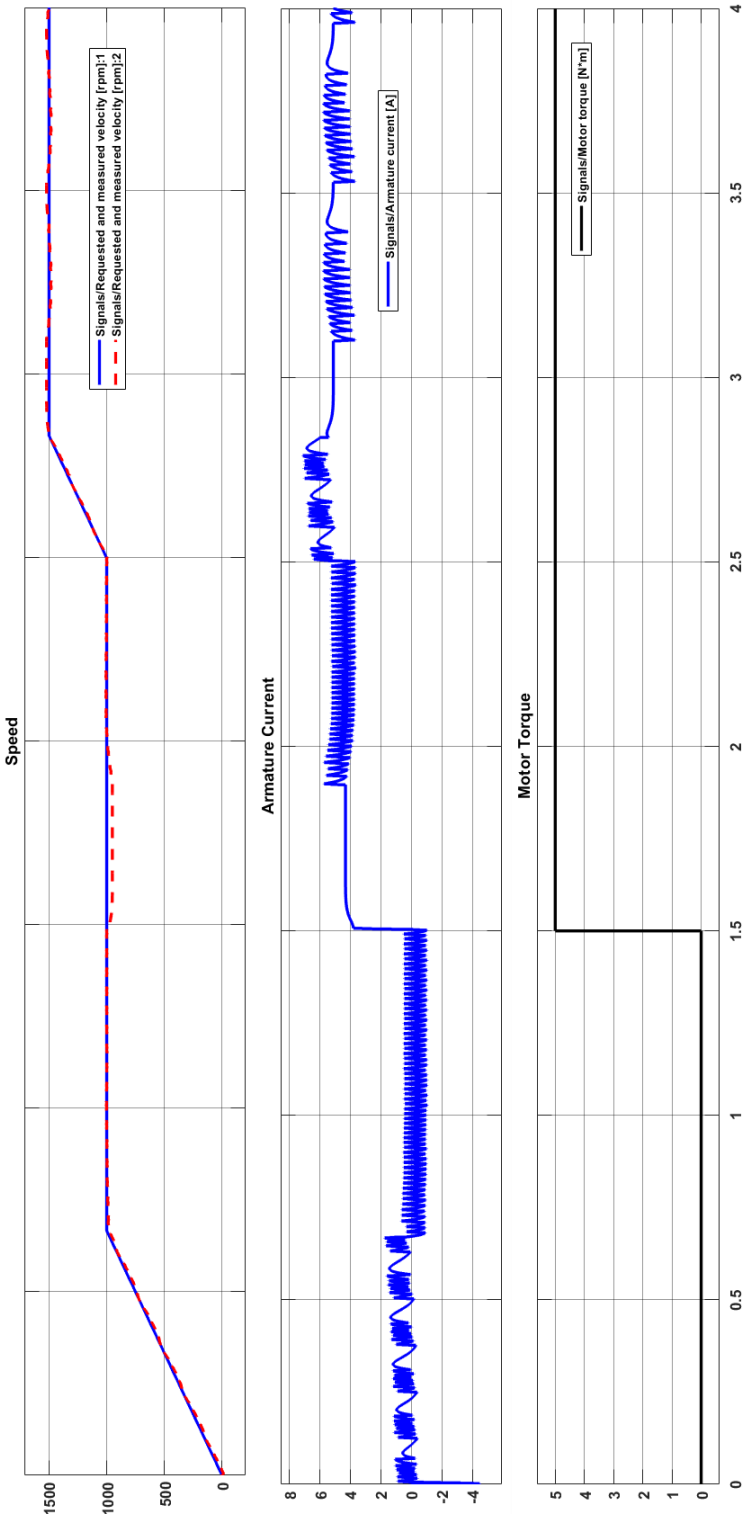
Matlab Circuit



Matlab Code:

```
%% Parameters for DC Motor Control Example (State-Feedback and
Observer)

%% DC Motor Parameters
Ra=4.4;      %[Ohm]
La=41.2e-3;    %[H]
Bm=0.005;   %[N*m/(rad/s)]
Jm=0.009;   %[Kg*m^2]
Kb=1.085;   %[V/(rad/s)]
Kt=1.26;

%% Control Parameters
Ts     = 5e-5;       % Fundamental sample time  [s]
fsw    = 1000;       % Switching frequency      [Hz]
Tsc    = 1/fsw;      % Control sample time      [s]
deadTime = 5;        % Sensor delay in control samples
eig = [0.3,0.3,0.2].*exp(1i*[pi/6,-pi/6,0]); % Desired eigenvalues

va_max = 220;        % Maximum armature voltage [V]
ia_max = 12;         % Maximum armature current [A]
ia_min = 0;        % Minimum armature current [A]

%% Continuous state-space model
A=[-Bm/Jm Kt/Jm; -Kb/La -Ra/La ]
B=[0; 1/La]
C=[1 0]
D=[0];
```

Output Waveform



Result

Thus, the controllability and observability test in simulation platform is performed and verified.

Ex. No.: 14	**Speed Control of Induction Motor Using PI Controller and SVPWM**
Date:	

Aim

To design and simulate a speed controller for induction motors in EV using PI Controller and Space Vector Pulse Width Modulation (SVPWM)

Introduction

In electric vehicle (EV) applications, the speed control of induction motors is crucial for efficient and responsive operation. A PI (Proportional-Integral) controller is commonly employed to achieve precise speed regulation. The PI controller adjusts the motor's input voltage based on the error signal, which is the difference between the desired and actual speeds.

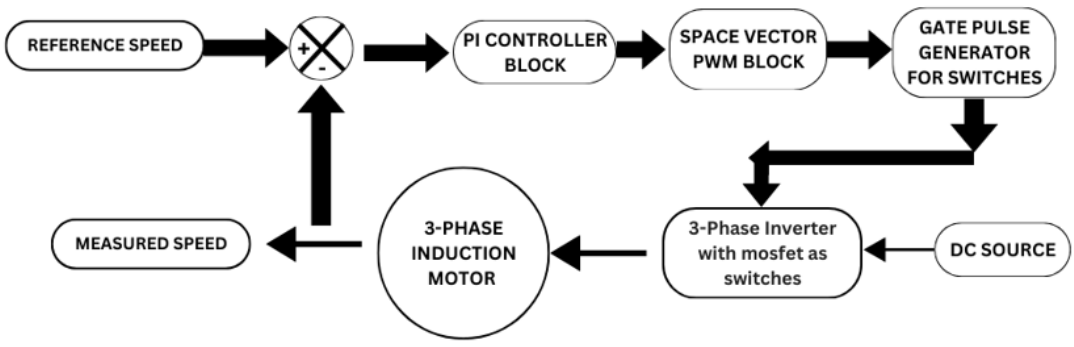
The Proportional component responds to the present error, providing a control action proportional to the speed deviation. The Integral component adds a corrective action based on the accumulated past errors, minimizing any steady-state speed errors.

The PI controller helps maintain a stable and accurate speed, enhancing overall performance and energy efficiency. This control strategy ensures that the induction motor operates at the desired speed under varying load and driving conditions, contributing to a smoother and more reliable electric vehicle experience.

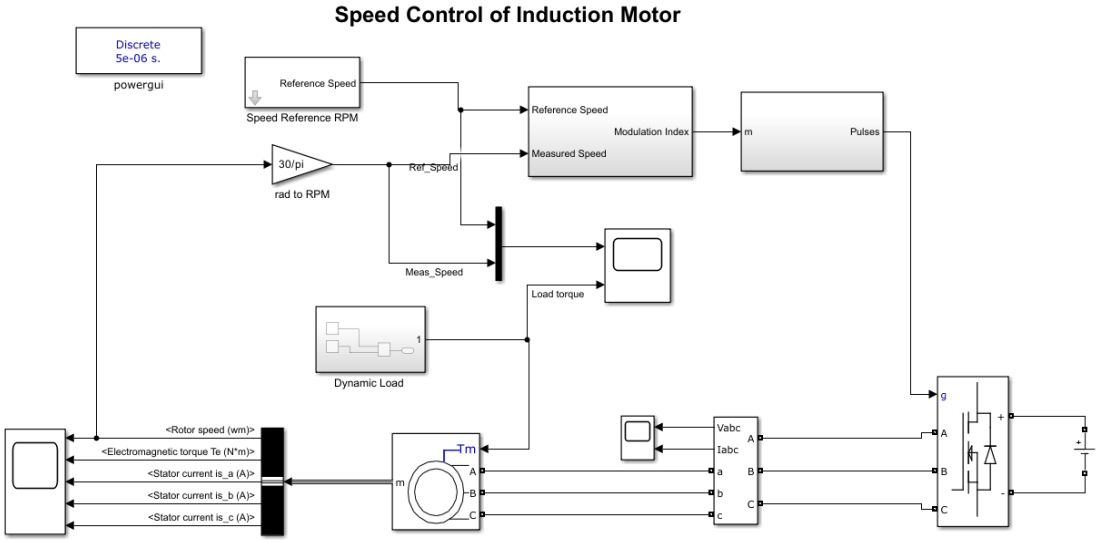
Algorithm

The algorithm for speed control of an induction motor involves setting a reference speed, measuring the actual speed using a sensor, and calculating the speed error. The PI controller adjusts the modulation index based on the speed error. This modulation index is then used in the SVM (Space Vector Modulation) block, which generates three sine waves 120 degrees out of phase to represent reference voltages for the motor phases. A repeating sequence block ensures the cyclical and repetitive nature of the SVM output. Relational operators compare SVM output with a carrier signal to determine gate pulses for a three-phase inverter. These gate pulses control the inverter switches, modulating the voltage applied to the induction motor. The closed-loop system continuously adjusts the modulation index via the PI controller to minimize speed error, resulting in precise speed control of the induction motor.

Block Diagram



Matlab Circuit



Output Waveform

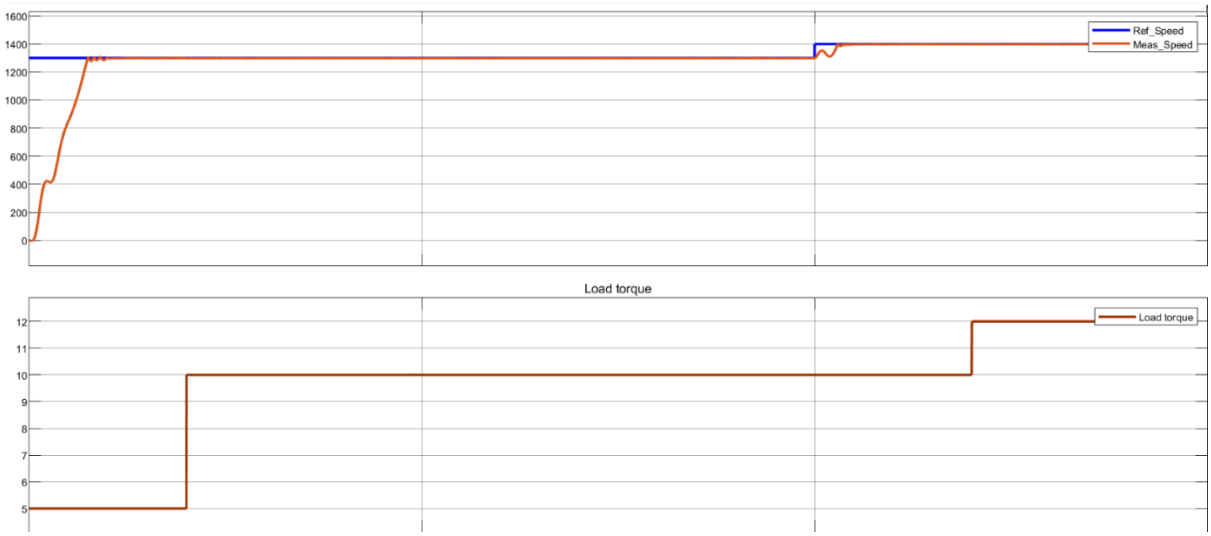


Fig.1: Plot of reference speed and measured speed for different load torques.

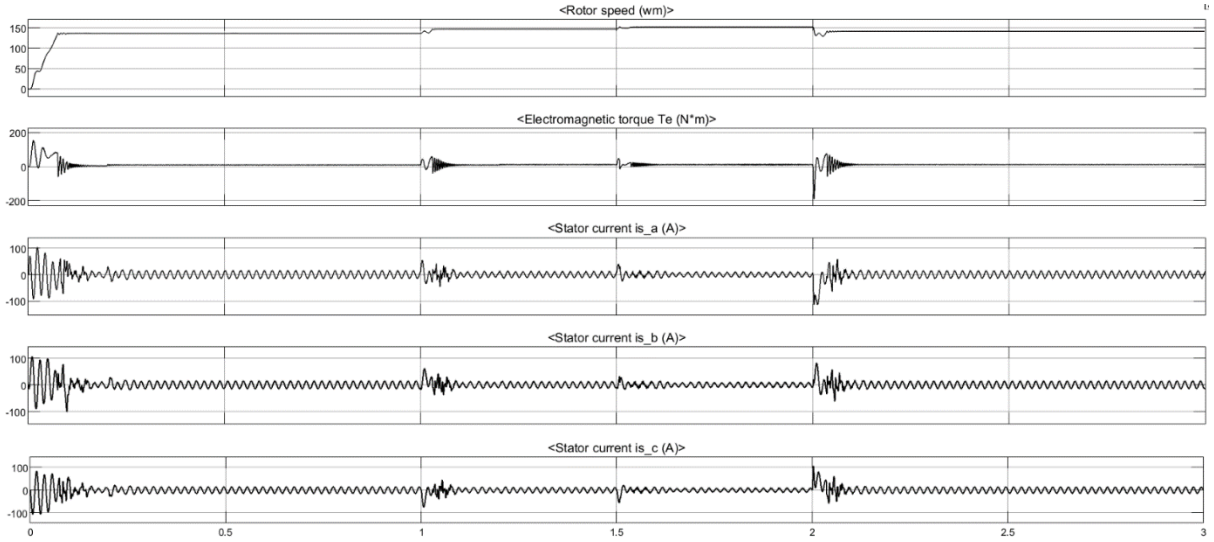


Fig.2: Plot of stator currents and Electromagnetic Torque

Inference

The simulation shows that the motor responds well to changes. The actual speed closely follows the desired speed, indicating effective speed control. The load torque graph illustrates the motor's ability to handle different loads smoothly. In summary, the simulation confirms the reliability of the control system for maintaining accurate motor speed, even with dynamic load variations.

Result

Thus, the design and simulation of the speed controller for induction motors for EV application is simulated using Simulink.

Ex. No:15	**Demonstration of Closed Loop Speed Control of DC Motor**
Date:	

Aim

The objective of this Mini project is to Demonstrate closed loop speed control of DC motor in hardware.

Introduction

The Proportional Controller, commonly known as a P controller, is a fundamental component in control systems engineering that plays a crucial role in regulating and maintaining desired outputs in various processes. Control systems are employed in diverse fields, ranging from industrial processes and manufacturing to robotics and environmental systems.

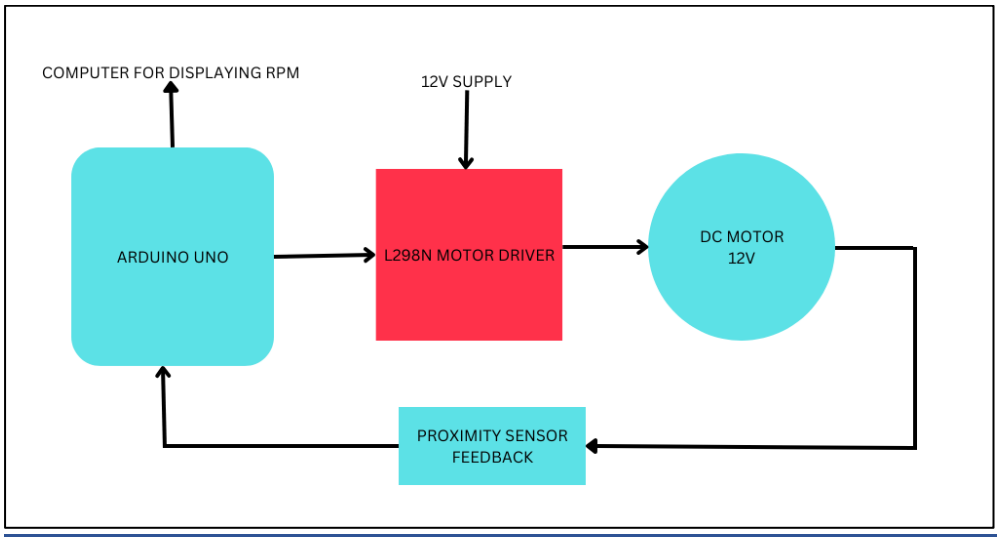
At its core, the P controller operates on the principle of proportionality, adjusting the system's output in direct proportion to the difference between the desired setpoint and the current state or output of the system. This control strategy is based on the premise that the larger the error between the setpoint and the actual output, the greater the corrective action applied by the controller.

The simplicity and effectiveness of the P controller make it a widely used element in control systems. By modulating the system output in proportion to the error signal, the P controller provides a rapid response to deviations from the desired setpoint. However, it has limitations, such as the potential for steady-state error in certain situations

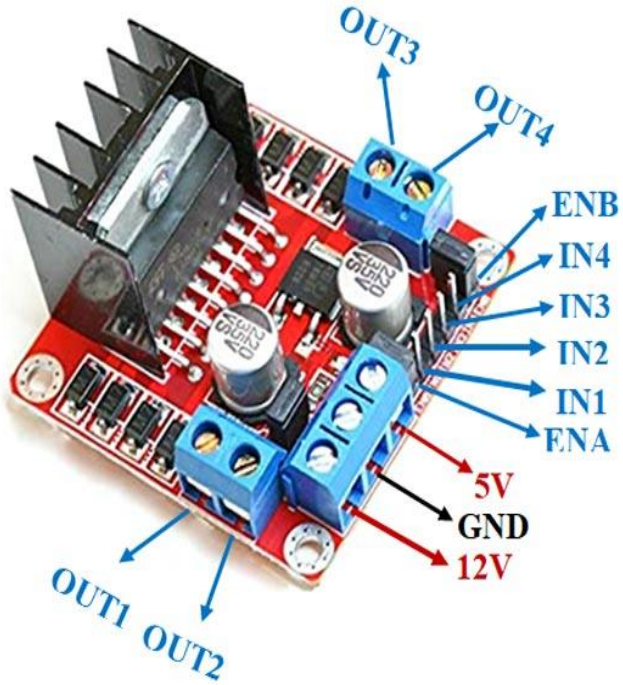
Required Components:

1. Arduino Uno(1N).
2. L298N Motor driver(1N).
3. Proximity Sensor(1N).
4. 12 Dc motor(1N).
5. Jumper wire(As required).
6. Regulated power supply(0-36V).
7. Optical Tachometer.

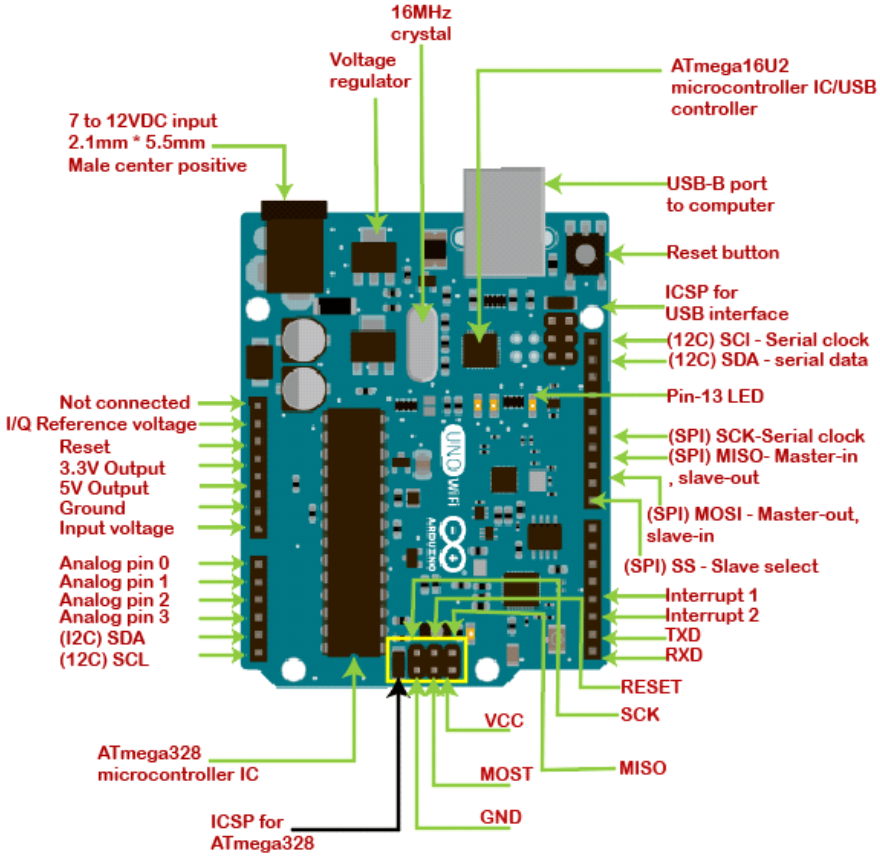
Block Diagram



Pin Diagram



L298N motor driver



Arduino Uno

Procedure

Step 1: Connections:

- Connect the Vcc of the sensor to 5V of Arduino.
- Connect the GND of the sensor to GND of Arduino.
- Connect the OUT pin of the sensor to digital pin 2 of the Arduino.
- Connect the GND of Arduino and the GND of L298N motor driver.
- Connect the +V of the RPS to the 12V of the motor driver.
- Common Ground the GND of sensor,Arduino and RPS.
- Connect the ENA pin of the L298N motor driver to the ~3 pin of the Arduino.
- Connect the OUT1 of the motor driver to the 12V Dc Motor.
- Connect Arduino pin 4 to the IN1 pin in L298N motor driver.
- Connect Arduino pin 5 to the IN2 pin in L298N motor driver.

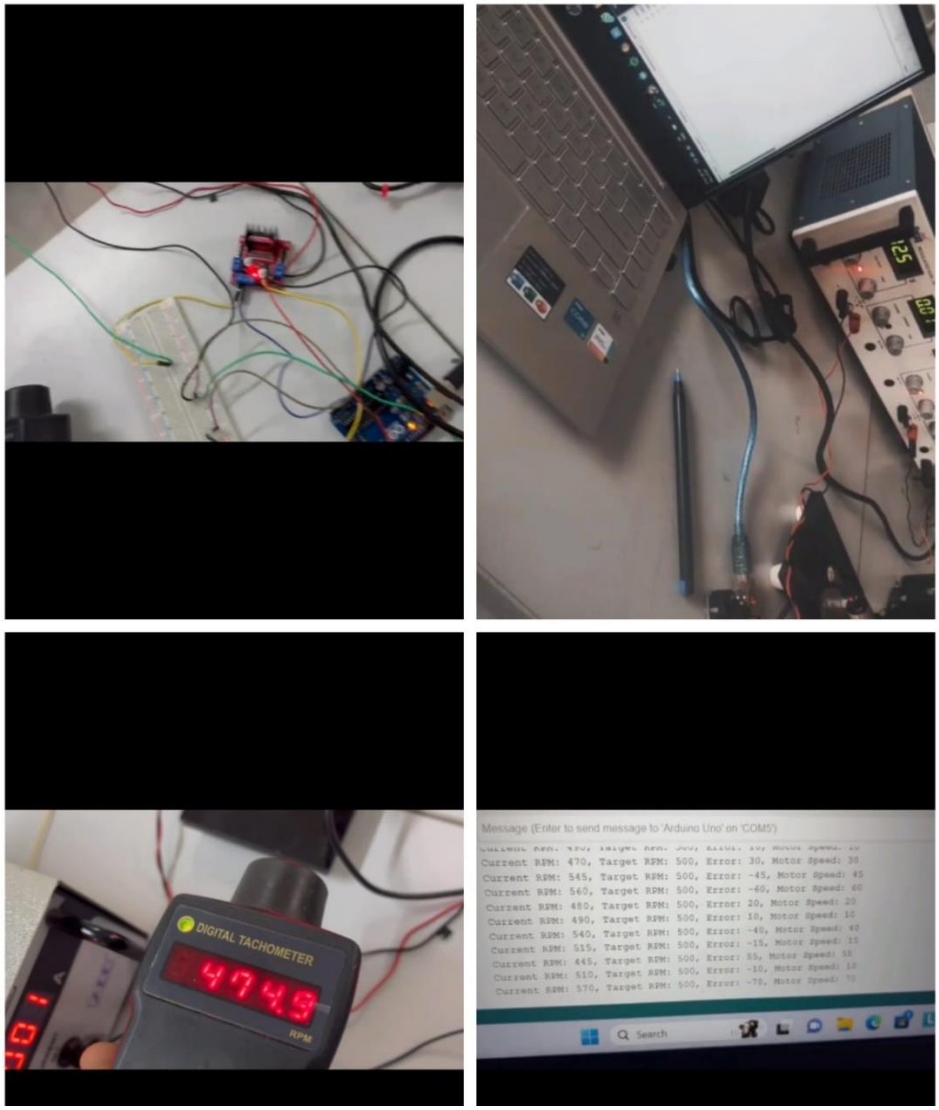
Step 2:Write the Arduino code for implementing P control using Arduino Uno using Arduino IDE.

Step 3: Verify the speed of the motor using a optical tachometer.

Arduino Code:

```
const int sensorPin = 2; // The digital input pin to which the proximity sensor is connected
unsigned long prevTime = 0;
unsigned long pulseCount = 0;
unsigned long rpm = 0;
int currentRPM = 0;
const int motorSpeedPin = 3; // PWM pin for motor speed control
const int motorDirectionPin1 = 4; // Motor driver input 1
const int motorDirectionPin2 = 5;
const int targetRPM=1000;// Target RPM (adjust as needed)
const float Kp = 0.71;
void setup() {
  pinMode(motorSpeedPin, OUTPUT);
  pinMode(motorDirectionPin1, OUTPUT);
  pinMode(motorDirectionPin2, OUTPUT);
  Serial.begin(9600); // Initialize serial communication
  attachInterrupt(digitalPinToInterrupt(sensorPin), countPulse, RISING); // Attach interrupt on rising edge
}

void loop() {
  unsigned long currentTime = millis();

  // Calculate RPM every 1 second
  if (currentTime - prevTime >= 1000) {
    rpm = (pulseCount * 60) / 12; // RPM calculation


    pulseCount = 0; // Reset pulse count
    prevTime = currentTime; // Update previous time
    currentRPM = rpm;
```

```
    int error = targetRPM - currentRPM;
    int motorSpeed = Kp * error;

    motorSpeed = constrain(motorSpeed, -225, 225);

    if (motorSpeed > 0) {
      digitalWrite(motorDirectionPin1, HIGH);
      digitalWrite(motorDirectionPin2, LOW);
    } else {
      digitalWrite(motorDirectionPin1, LOW);
      digitalWrite(motorDirectionPin2, HIGH);
      motorSpeed = -motorSpeed;
    }
    analogWrite(motorSpeedPin, motorSpeed);

    Serial.print("Current RPM: ");
    Serial.print(currentRPM);
    Serial.print(", Target RPM: ");
    Serial.print(targetRPM);
    Serial.print(", Error: ");
    Serial.print(error);
    Serial.println();
  }
}

void countPulse() {
  // This function is called when the proximity sensor detects a pulse
  pulseCount++;
}
```



Hardware result

Result

Hence the demonstration of closed loop system in hardware is successfully implemented.