University of Global Village (UGV), Barishal

Dept. of Electrical and Electronic Engineering (EEE)

Lab Manual
Control Systems Sessional
EEE 0714-3104

Noor Md Shahriar
BSc in EEE, RUET
Senior Lecturer
Co-chairman, Dept. of EEE
University of Global Village (UGV)
874/322, C&B Road, Barishal, Bangladesh.
Contact: +8801743500587
Facebook | LinkedIn | Twitter

Contents

Course Title:	Control Systems Sessional	Total Class Hour	37
Course Code:	EEE 0713-3104	Total Practice Hour	37
Supervised by	Noor Md Shahriar	Total Hour	85

Course Rationale

The Control Systems Sessional course serves as a practical complement to theoretical knowledge in control systems. It provides students with hands-on experience in modeling and analyzing physical systems using MATLAB Simulink, helping them understand the dynamic behavior of systems. Through simulations and experiments, students develop skills in designing controllers using techniques like root locus and frequency domain methods. This course emphasizes critical thinking and problem-solving, enabling students to apply control theories to real-world scenarios, preparing them for advanced studies or industry roles in automation and system control.

Course Objectives

- Apply control system theories through hands-on experiments and simulations.
- Develop proficiency in using MATLAB Simulink for modeling and analyzing dynamic systems.
- Analyze open-loop and closed-loop responses of physical systems to understand their behavior.
- Design controllers using root locus and frequency domain methods.
- Gain technical skills and an analytical mindset for solving control engineering problems.

Course Learning Outcome

CLO1	Understand control systems' basics and modeling techniques.
CLO2	Analyze open and closed-loop responses using MATLAB Simulink.
CLO3	Interpret system characteristics and dynamic behaviors.
CLO4	Design controllers using root locus and frequency domain methods.
CLO5	Apply theoretical knowledge to real-world control system problems.

Assessment Pattern

- **Continuous Assessment**

Bloom's Category	**Tests**
Imitation	12
Manipulation	8
Precision	6
Articulation	2
Naturalization	2

- **Semester End Examination: (SEE):**

Bloom's Category Marks (out of 30)	Tests (20)	Quiz (10)	External Participation in Curricular/Co-Curricular Activities (20)
Imitation	06	06	Bloom's Affective Domain: (Attitude or will) • Attendance: 10 • Viva-Voca: 5 • Report Submission: 5
Manipulation	04	04	
Precision	06		
Articulation	02		
Naturalization	02		

Course Outline

Sl. No.	Topic & Details	Class Hours	CLO Mapping
1	Introduction to Control Systems and MATLAB SIMULINK: Familiarization with tools and software	3	CLO1
2	Open-Loop Response Analysis: Modeling physical systems such as Mass-Spring-Damper and RLC circuits	6	CLO1, CLO2
3	Closed-Loop Response Analysis: Feedback system modeling and performance evaluation	6	CLO2, CLO3
4	DC Motor Analysis and Simulation: Characteristics and control strategies	4	CLO1, CLO2
5	Root Locus Method: Controller design and stability analysis	5	CLO3, CLO4
6	Frequency Domain Methods: Bode plot, Nyquist criteria, and controller design	6	CLO3, CLO4
7	Practical Applications and Case Studies: Real-world control systems and project work	4	CLO4, CLO5

Course Schedule

Week	Topic & Details	Teaching & Learning Strategy	Assessment Strategy	CLO Mapping
1	Introduction to Control Systems and MATLAB SIMULINK: Overview and tools	Lecture, Hands-on MATLAB tutorials	Attendance, Class Participation	CLO1
2	Open-Loop Response Analysis: Modeling Mass-Spring-Damper and RLC Circuits	Lecture, Simulation Activities	Quiz, Assignment	CLO1, CLO2
3	Open-Loop Response Analysis: Continued	Group Discussions, MATLAB Exercises	Class Test 1	CLO1, CLO2
4	DC Motor Characteristics: Modeling and Simulation in MATLAB	Hands-on MATLAB Exercises	Quiz, Assignment	CLO1, CLO2
5	Closed-Loop System Analysis: Feedback and Stability Concepts	Lecture, Simulation Case Studies	Attendance, Class Participation	CLO2, CLO3
6	Closed-Loop Response Analysis: Modeling Real-World Systems	Practical Exercises	Quiz	CLO2, CLO3
7	Root Locus Method: Introduction and Controller Design	Lecture, Worked Examples	Class Test 2	CLO3, CLO4
8	Root Locus Method: Continued	Hands-on MATLAB Activities	Assignment	CLO3, CLO4
9	Frequency Domain Methods: Bode Plots and Nyquist Criteria	Lecture, Problem-Solving Sessions	Attendance, Class Participation	CLO3, CLO4

10	Frequency Domain Methods: Controller Design Using Frequency Response	Simulation Activities	Quiz	CLO3, CLO4
11	Practical Controller Design: Case Studies in Root Locus and Frequency Domain Techniques	Group Projects	Class Test 3	CLO4, CLO5
12	Stability Analysis: Techniques and Applications	Lecture, Discussions	Assignment	CLO4, CLO5
13	Case Studies in Control Systems: Real-World Applications	Project Work	Attendance, Class Participation	CLO4, CLO5
14	Review of System Designs: Open-Loop vs Closed-Loop Performance	Peer Discussions	Quiz	CLO4, CLO5
15	Capstone Project: Finalizing Models and Preparing Presentations	Group Work, Instructor Guidance	Class Test 4	CLO4, CLO5
16	Capstone Project Presentation: Real-World System Demonstrations	Presentation, Q&A	Project Evaluation	CLO5
17	Final Review and Assessment: Comprehensive Analysis of Course Outcomes	Interactive Discussions, Reflection Activities	Final Evaluation	CLO1, CLO2, CLO3, CLO4, CLO5

References

1. **Ogata, K.** (2010). *Modern Control Engineering* (5th ed.). Pearson Education.
2. **Dorf, R. C., & Bishop, R. H.** (2017). *Modern Control Systems* (13th ed.). Pearson.
3. **Franklin, G. F., Powell, J. D., & Emami-Naeini, A.** (2015). *Feedback Control of Dynamic Systems* (7th ed.). Pearson.
4. **MATLAB Documentation** – MathWorks. Available at: https://www.mathworks.com/help/matlab/
5. **Simulink Documentation** – MathWorks. Available at: https://www.mathworks.com/help/simulink/
6. **Nise, N. S.** (2019). *Control Systems Engineering* (8th ed.). Wiley.
7. Research Articles from IEEE Xplore: *Control Systems and Automation Topics*. Available at: https://ieeexplore.ieee.org/

List of Experiments
Modeling of Physical systems and study of their open loop response
To get familiar with Matlab SIMULINK
Analysis of a Mass-Spring-Damper System
Analysis of the Characteristics of an RLC Series Circuit in MATLAB Simulink
Analysis of the Characteristics of a DC Motor and DC Motor Simulation in MATLAB.
Modeling of Physical systems and study of their closed loop response
Root Locus Controller Design of Physical systems
Frequency Domain Methods for Controller Design of Physical systems Key MATLAB

Report Writing:

The report must be arranged according to the following fashion.

(1) Objectives (must be hand written)
(2) Introduction / Theoretical background / Mathematical Expression (hand written) of the assigned works of the experiments
(3) Circuit diagram / Flow diagram (hand written or print)
(4) C/C++ code or Matlab code (in print form from where results will be taken)
(5) Experimental Data, graph, analysis (print form)
(6) Application of the experiment (hand written)
(7) Discussions (hand written)
(8) Conclusions
(9) References

The following information should be available on the top page of the report

1. Experiment Number:
2. Name of the Experiment:
3. Date of Experiment:
4. Date of Submission:
5. Submitted to
6. Name of the student
7. Student ID
8. Subject Group (if any)

Experiment No: 01

Name of the experiment: Modeling of Physical systems and study of their open loop response

Objective:

(i) The objective of this experiment is the modeling of physical systems and study of their open loop response.

(ii) Study open loop system with the variation of parameters, such as damping coefficient and mass

(iii) Simulation of a cruise control system / a dc motor

Introduction:

Physical Interpretation & system equations:

Let us assume a car that travels only in one direction. Control to the car was applied in such a way that it has a smooth start up, along with a constant-speed ride. The force applied is 'f [newton]', velocity at any time is 'v [m/sec]' and frictional constant=b [n.sec/m]. The frictional force is linearly proportional to velocity [f_b *proportional to* v]. So the applied force, 'u' accelerates the mass while overcoming the frictional force.

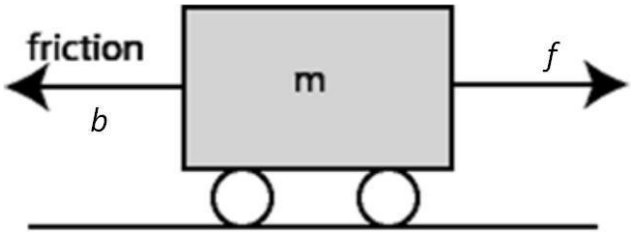


Fig. 1: Car/Rickshaw/Van control system

If it is assumed that friction is opposing the motion of the car, then the modeling equations become

$$f_m + f_b = f$$

$$ma + bv = f$$

$$m\frac{d^2x}{dt} + b\frac{dx}{dt} = f$$

$$m\frac{dv}{dt} + bv = f \text{ output, } Y = v$$

Taking the Laplace transform of the equations, we find

Mathematical analysis

$$msV(s) + bV(s) = F(s)$$

$$Y(s) = V(s)$$

Substituting V(s) in terms of Y(s), $msY(s) + bY(s) = F(s)$ The transfer function of the system becomes

$$\frac{V(s)}{F(s)} = \frac{1}{ms + b}$$

Sample Matlab Code: expt1.m

```
clear;
clc;
b=50;
u=602;
figure;
for i=1:5
    m=200*i;
    num=[1];
    den=[m b];
    sys=tf(num,den);
    step(u*sys); hold
    on;
    title('Step response of the Cruise Control when
Friction is fixed');
    xlabel('time')
    ylabel('velocity(m/s)');
end
```

```
figure;
m=1000;
for j=1:5
    b=j*10;
    num=[1];
    den=[m b];
    sys=tf(num,den);
    step(u*sys); hold
    on;
    title('Step response of the Cruise Control when
Mass is fixed');
    xlabel('time')
    ylabel('velocity(m/s)');
end
```

Output Curve:

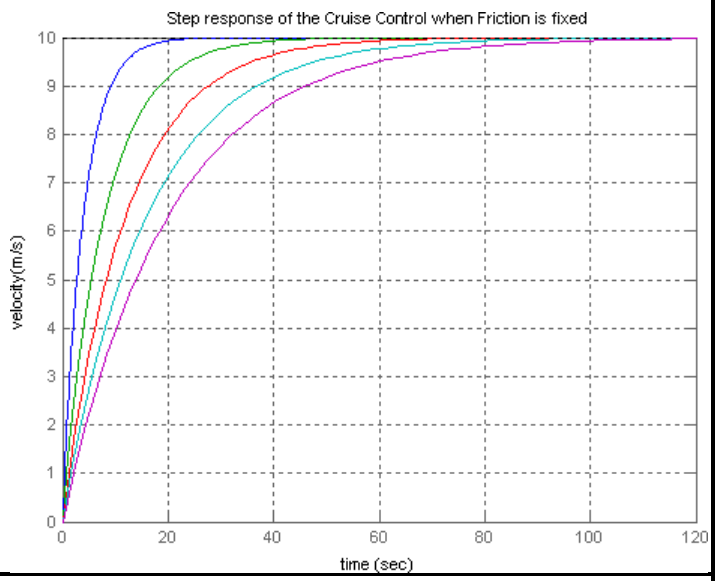


Figure 1:

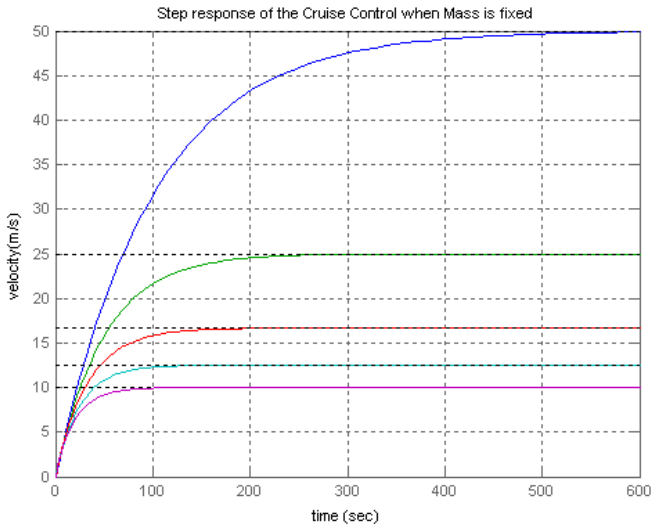


Figure: 2

MATLab Data: For Figure 1:

Table 1

Mass (m)	Friction (b)	Rise Time	Settling Time	Overshoot Percentage	Steady State Speed	Steady State Error
200	50	8.79	15.6	0	9.94	0
400	50	17.6	31.6	0	9.94	0
600	50	26.4	46.9	0	9.94	0
800	50	35.2	62.6	0	9.94	0
1000	50	44	78.2	0	9.94	0

For Figure 2:

Table 2

Mass (m)	Friction (b)	Rise Time	Settling Time	Overshoot Percentage	Steady State Speed	Steady State Error
1000	10	220	391	0	49.7	0
1000	20	110	196	0	24.9	0
1000	30	73.3	130	0	16.6	0
1000	40	55	97.8	0	12.4	0
1000	50	44	78.2	0	9.94	0

Study materials:

2. Define rise time, settling time, percentage overshoot and steady state error of a system for step input.

Answer: **Rise time** (*Tr*) :

Time required for the response to rise from 0% to 100% of its final value.

Settling time (*Ts*) :

Time required for the response to reach and stay within the range about the final value of size specified z

Overshoot percentage (*P.O.*) :

The amount the system output response proceeds beyond the desired response.

$$Percent\ Overshoot(P.O.) = \frac{M_{Pt} - f_v}{f_v} \times 100\%$$

Where,

M_{Pt} = The peak value of the time response.

f_v = The final value of the response.

Steady State Error:

The error when the time is large and the transient response has decayed, leaving the continuous response.

3. How steady state speed, rise time and overshoot of the output response vary with the variation of m and b?

Answer: In Figure 1, Friction (b) is fixed Mass (m) has been varied. We take 5 different values for Mass (m) respectively 200, 400, 600, 800, and 1000. If we observe the graph, we can find that Rise time is increasing as Mass (m) increases. If we calculate the overshoot percentage for a given value we find it Zero.

In Figure 2, Mass (m) is kept fixed and Friction (b) varied. We measure the rise time for five different values of Friction (b) and the values are 10, 20, 30, 40, and 50. From the graph, we can see that Rise time and steady state speed is decreasing as Friction (b) increases. Overshoot percentage value is Zero for measured value.

4. Justify that the steady state speed of the vehicle depends only on the friction coefficient b.

Answer: We Know that, $u = \frac{dv}{d} + bv$

Here v is fixed, so $\frac{dv}{d} = 0$,

Then $u = bv$

$\Rightarrow v = \frac{u}{b}$

Here u is fixed for figure 1 and Figure 2.

So, v is only depends on the friction Coefficient b

Your own Experiments:

(i) Prepare the following results for the following car control system

Develop system equations, find the transfer function of the system and Prepare the Matlab data (similar to Table 1 and Table 2, Figure 1 and Figure 2) by using the following information and Fig. 2.

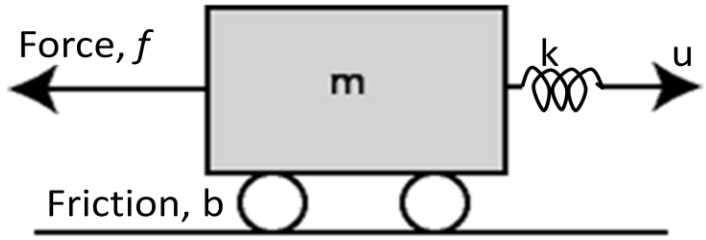


Fig. 2: Car control system

Assume, Friction, b=50 + last two digit of your Roll Number and m=fixed (ii) Mass, m=500 – last two digit of your Roll Number and b=fixed. (iii) Assume any **k** value

Experiment No: 02

Name of the experiment: To get familiar with Matlab SIMULINK

Objectives:

Objectives of this lab are to:

1. Use SIMULINK graphical user interface (GUI) to design control systems in forms of block diagrams.

2. Analyze designs or models mathematically.

3. Plot the response of various parts of control systems.

Required software:

Following software package is required.

- MATLAB/SIMULINK

Introduction:

Simulink is a simulation and model-based design environment for dynamic and embedded systems, integrated with MATLAB. Simulink, also developed by MathWorks, is a data flow graphical programming language tool for modeling, simulating and analyzing multi-domain dynamic systems. It is basically a graphical block diagramming tool with a customizable set of block libraries.
It allows you to incorporate MATLAB algorithms into models as well as export the simulation results into MATLAB for further analysis.
Simulink supports:

- system-level design
- simulation
- automatic code generation
- testing and verification of embedded systems.

SIMULINK:

The block diagram based GUI Simulink provides a wide range of libraries with an extensive set of blocks. Those blocks are required for design, simulation and analysis of physical dynamic systems. According to the classes of functions, the blocks are grouped into libraries:

- Mathematical functions such as summers and gains in the Math library.
- Integrators, derivatives, transfer function, state space etc. in the Continuous library.
- Signal generator, input functions such as- sine wave, clock, step signal, ramp signal, constants etc. in the Sources library.
- Scope, To workspace blocks are in the Sinks library.

The libraries are found in the library browser.

In the graphical user interface Simulink, users actually create programs in the form of block diagrams. Arrays of variables, defined in Simulink, are created when the programs are run. For getting these variables available in the MATLAB platform, an interfacing is required. This is done by identifying these variables by Simulink using "To Workspace" block found in the Sinks library. The block "Scope" is used for displaying the output of a dynamic system designed in Simulink in graphical form.

Some blocks from different libraries used in the experiment are shown below:

Sine wave :

Fig. 1.1: A sine wave block.

The amplitude of the generated sine wave from block parameters option. It's found in the Sources library.

Sum:

Fig. 1.2: A Sum block.

The block provides addition of two or more signals. Number of input ports and signs(+ or -) can be changed by the Block parameter option. It's found in a commonly used blocks library.

Product:

Fig. 1.3: A product block.

This block is used for multiplying two or more signals. Number of input ports can be changed from the Block parameter option. It's found in a commonly used blocks library.

Mux:

Fig. 1.4: A Mux block.

This block is used for multiplexing scalar or vector signals. It is found in Signal Routing library.

To workspace:

Fig. 1.5: To the workspace block.

This block is used for getting the variables created in Simulink available into MATLAB. It's found in the Sinks library.

Scope:

Fig. 1.6: A Scope block.

This block is used for displaying the plotted signal of the output of the designed dynamic system. It's found in the Sinks library.

Displaying output signal in Simulink:

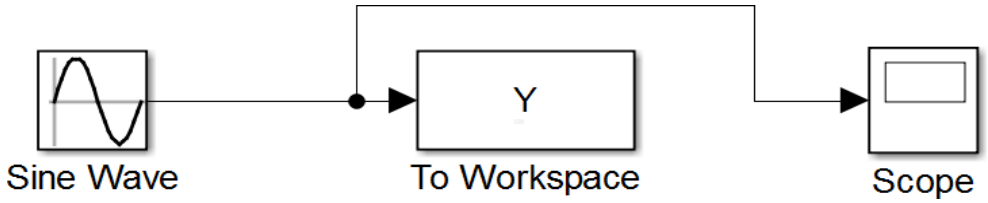


Fig. 1.7: Block diagram for displaying sine waves in Simulink.

Sine wave block is connected with both To workspace and Scope block. Scope acts as a medium for displaying the plotted signal of sine wave block. The workspace block links the variable generated in Simulink with MATLAB. Plot(Y) function is used for plotting the output.

Displayed plot is as below:

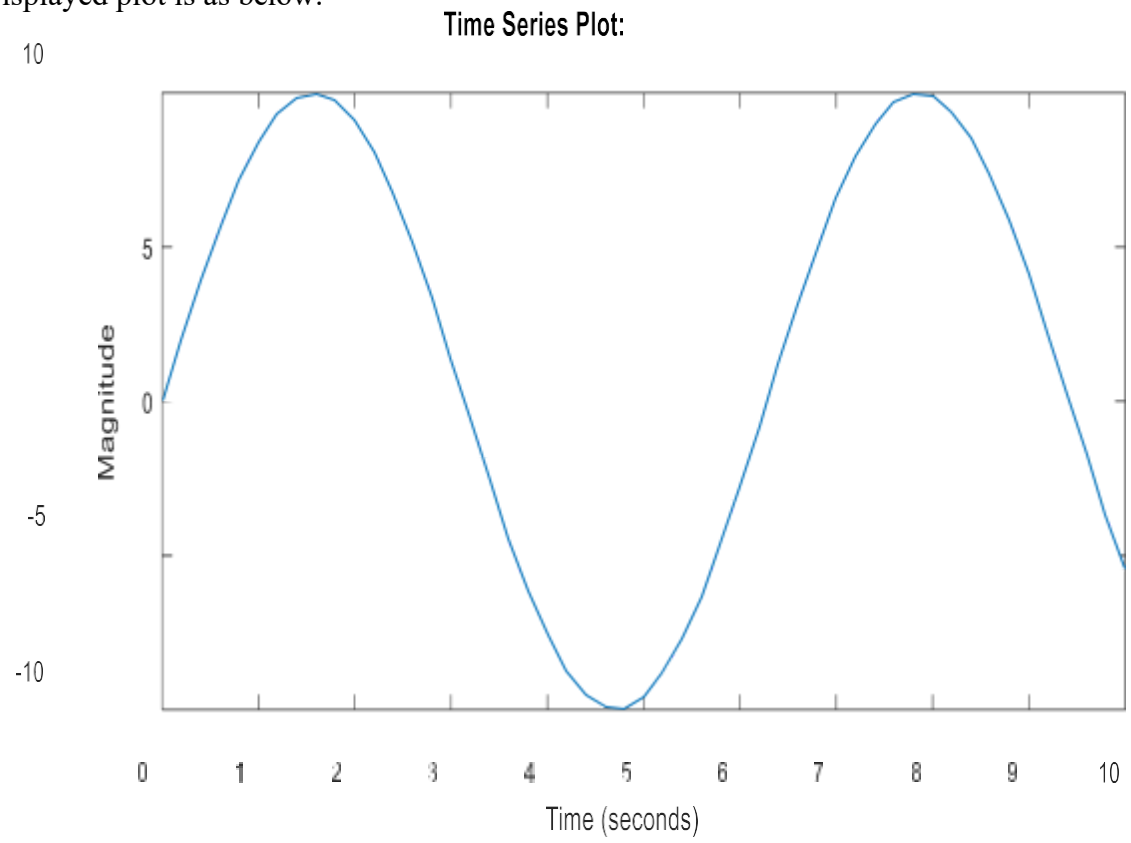


Fig.1.8: Output plot for block diagram of fig.1.6.

Here, it is noticed that the output signal is a sine wave having an amplitude of 10V which is equal to the input signals. Hence, the output is displayed correctly.

Addition of two signals:

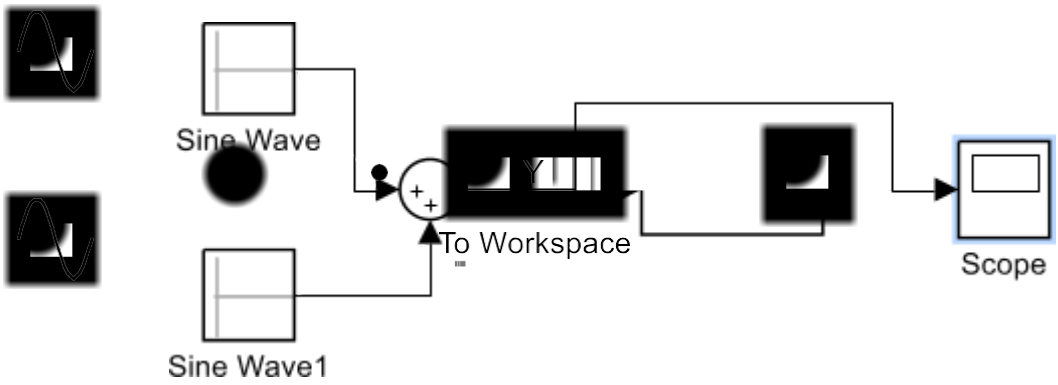


Fig. 1.9: Block diagram for adding two sine wave signals in Simulink.

Two sine waves have amplitudes of 10V and 5V respectively. These two sine wave blocks are connected to the sum block. The output is plotted in MATLAB using plot() function. The workspace block links the variables of Simulink with the MATLAB platform.

Plotted output is as below:

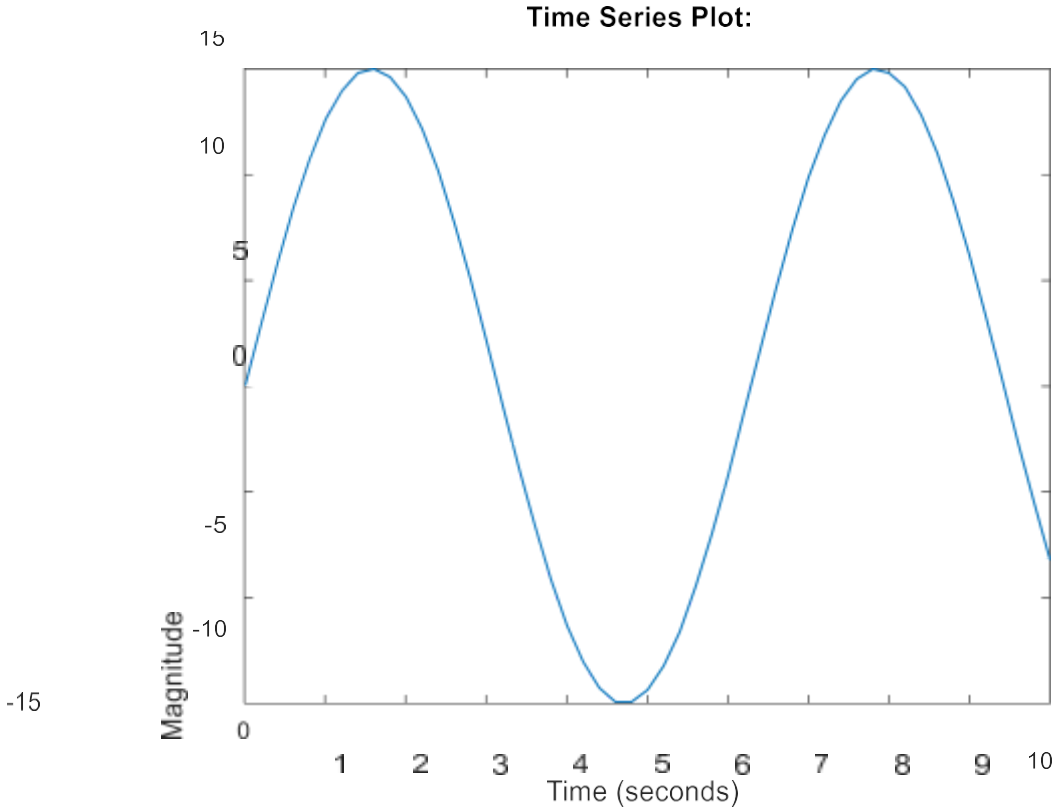


Fig.1.10: Output plot for block diagram of fig.1.8.

Here, it is noticed that the ouput signal is a sine wave having an amplitude of 15V which is the sum of two input signals. Hence, the output is displayed correctly.

Subtraction of two signals:

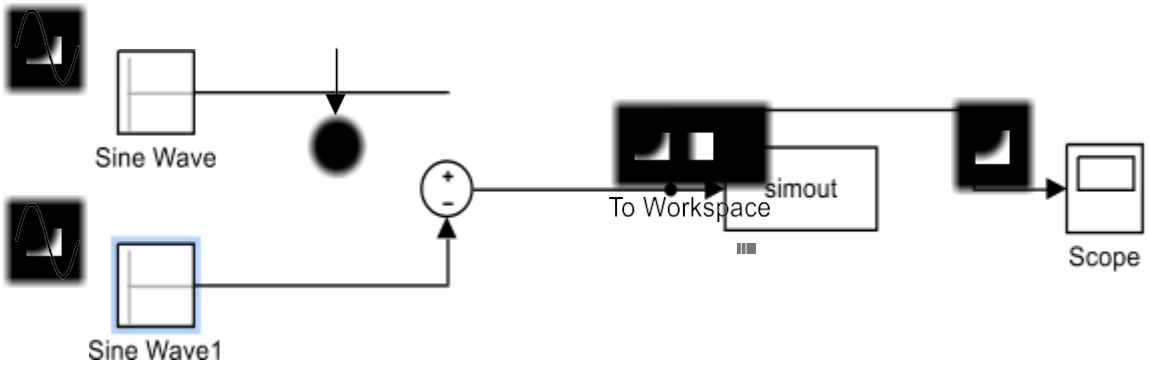


Fig. 1.11: Block diagram for subtracting two sine wave signals in Simulink.

Two sine waves having amplitudes of 10V and 5V respectively are connected to the sum block. The output is plotted in MATLAB using plot() function. The workspace block links the variables of Simulink with the MATLAB platform.

Plotted output is as below:

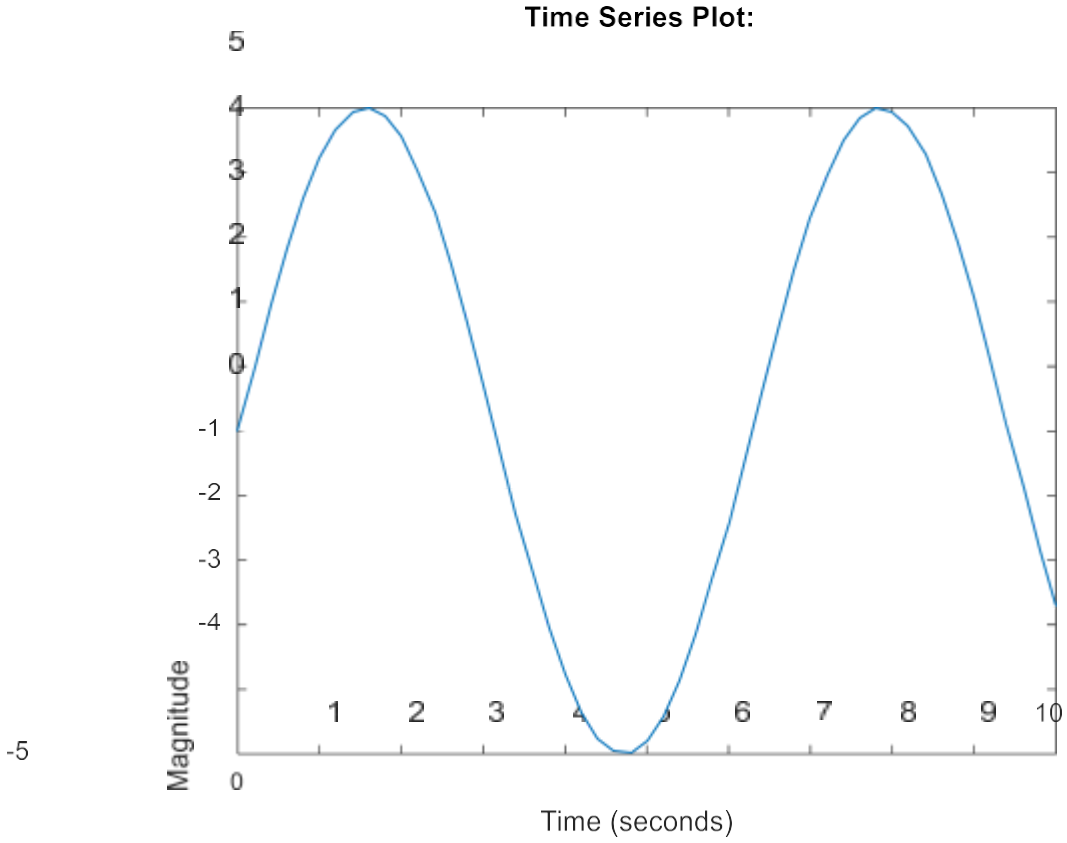


Fig.1.12: Output plot for block diagram of fig.1.10.

From fig.1.12 it is noticed that the output signal is a sine wave having an amplitude of 5V which is the difference between the 10V input signal connected to '+' terminal and 5V input signal connected to '-' terminal of the sum block. Hence, the output is displayed as expected.

Multiplication of two signals:

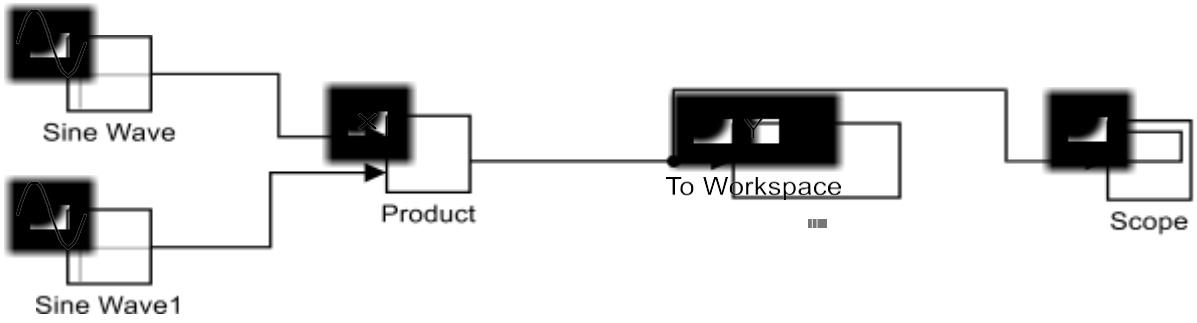


Fig. 1.13: Block diagram for multiplication of two sine wave signals in Simulink.

In fig.1.13 two sine waves having amplitudes of 10V and 5V are connected to the Product block. The output is plotted in MATLAB using plot() function. The workspace block links the variables of Simulink with the MATLAB platform. The output signal is expected to be a sine wave with an amplitude of 50V.

Plotted output is as below:

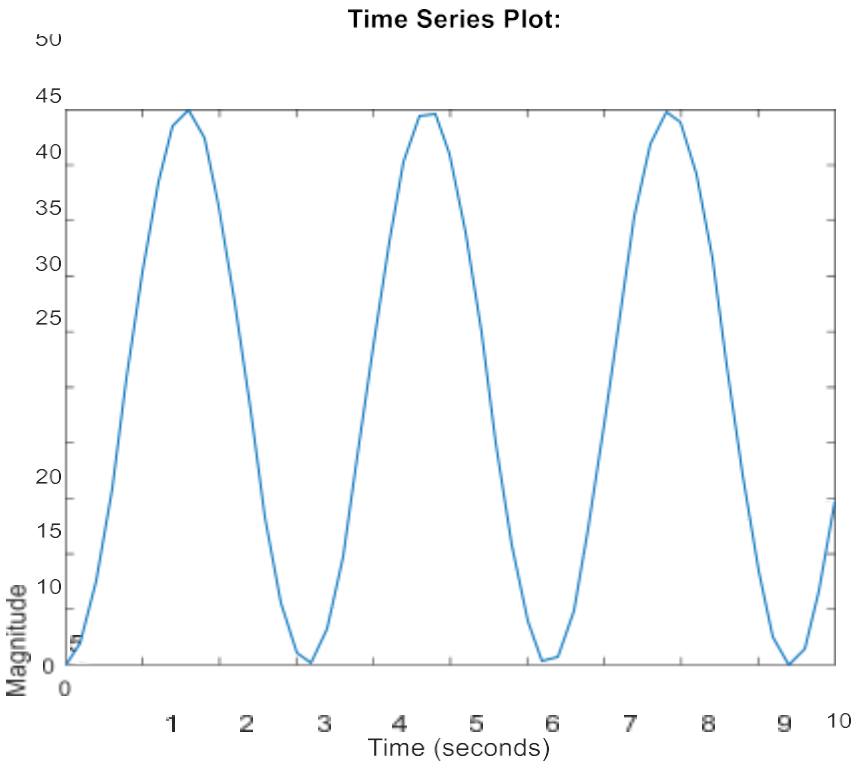


Fig.1.14: Output plot for block diagram of fig.1.12.

Multiplexing:

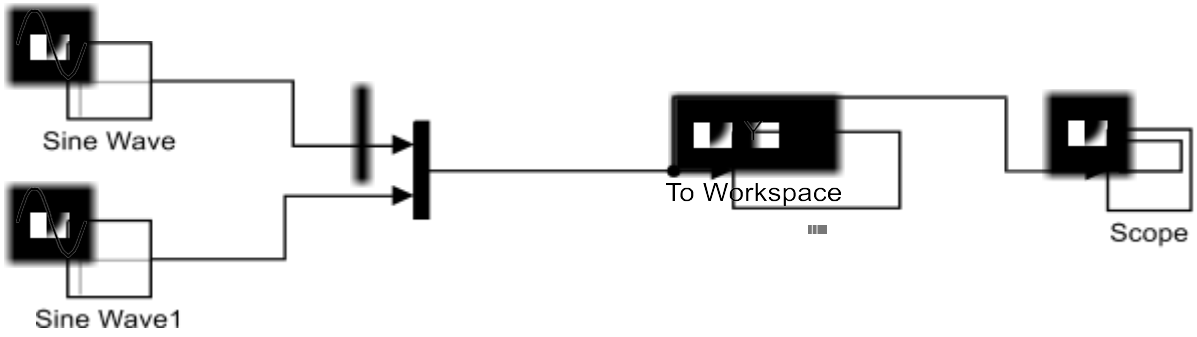


Fig. 1.15: Block diagram for multiplexing two sine wave signals in Simulink.

In fig.1.15 two sine waves having amplitudes of 10V and 5V are connected to the Mux block. Both signals are transmitted to the output port without any change of the signal through the Mux block. That means multiplexing of the signal has been done.The output is plotted in MATLAB using plot() function. The workspace block links the variables of Simulink with the MATLAB platform.

Plotted output is as below:

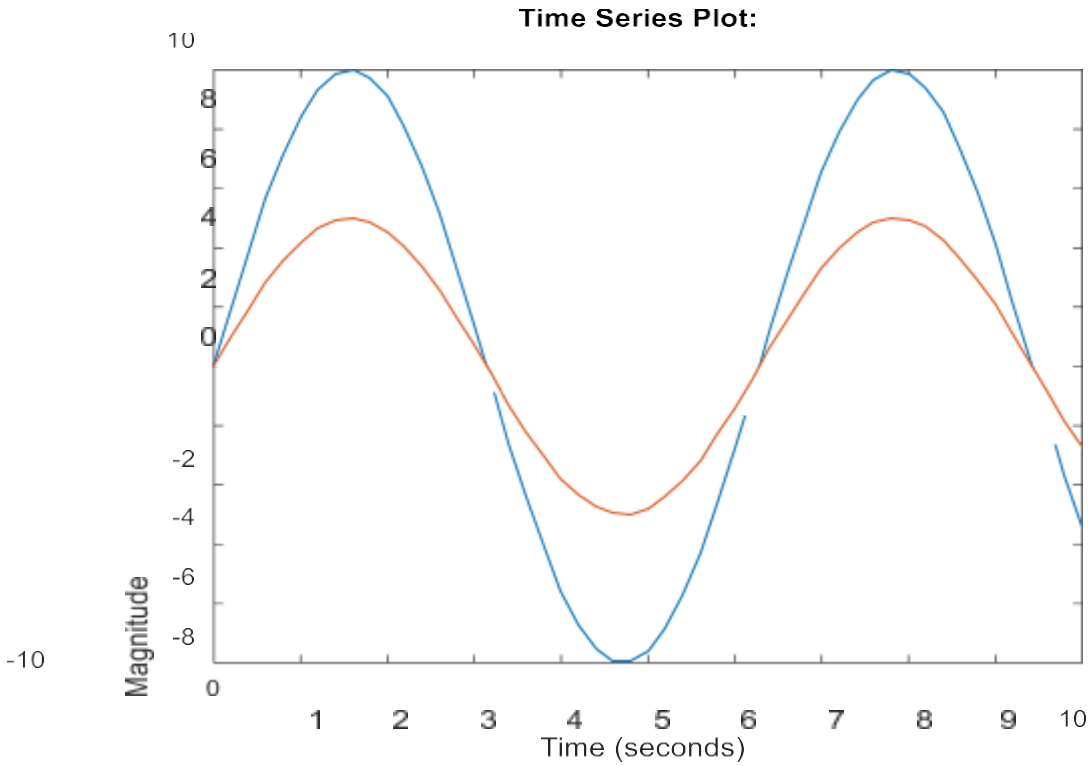


Fig.1.16: Output plot for block diagram of fig.1.15.

From fig.1.16 it is noticed that the output is both input sine waves having amplitudes of 5V and 10V. Hence, the input signal is multiplexed to the output as expected.

Discussion:

In this experiment, familiarization with graphical user interface based design and analysis platform Simulink has been performed. Getting acquainted with different libraries and blocks of this block diagram based design software have been performed. Addition, subtraction, multiplication and multiplexing of various input signals have been done through designing block diagrams and connecting respective blocks. The output signal was plotted using MATLAB and was also viewed through the Scope block.

All the functions or blocks worked correctly and the output shown was

perfect. Hence, it can be said that the experiment was performed properly.

Experiment No. 03

Name of the Experiment: Analysis of a Mass-Spring-Damper System

Objective:

The objective of this experiment is to analyze the dynamics of a mass-spring-damper system and simulate its behaviour using MATLAB Simulink. By the end of this experiment, students will be able to:

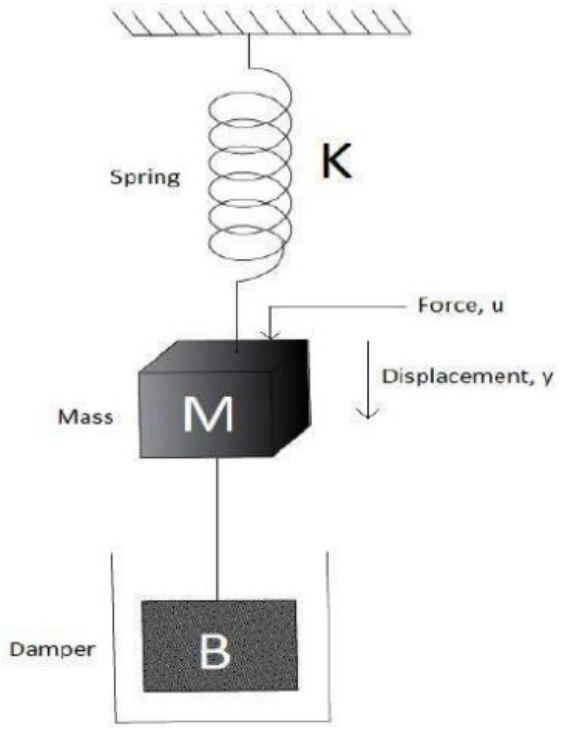
- Understand the fundamental principles of mass-spring-damper systems.
- Create a Simulink model of a mass-spring-damper system.
- Perform a step response analysis of the system.
- Interpret the simulation results to understand the system's performance parameters, such as oscillatory behavior and damping.

Introduction:

A mass-spring-damper system is a classic example of a second-order dynamic system and is commonly used in engineering to study vibration, control systems, and mechanical dynamics. The system consists of a mass attached to a spring and a damper. The spring provides a restoring force proportional to the displacement, while the damper provides a force proportional to the velocity of the mass.

Understanding the behavior of such systems is crucial for designing effective control systems and predicting the system's response to various inputs. In this experiment, MATLAB Simulink will be used to model and simulate the mass-spring-damper system, allowing students to visualize its response to a step input and analyze key performance metrics.

Diagram:



$u(t) = \text{Input}$

K :Spring

B : Damping constant

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

$$\Rightarrow Ms^2Y(s) + BsY(s) + KY(s) = U(s)$$

$$\Rightarrow \frac{Y(s)}{U(s)} = G(s) = \frac{1}{Ms^2 + Bs + K}$$

Fig.: Mass-spring-damper system.

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

$$\Rightarrow \frac{d^2y}{dt^2} = \frac{1}{M}u(t) - \frac{B}{M}\frac{dy}{dt} - \frac{k}{M}y$$

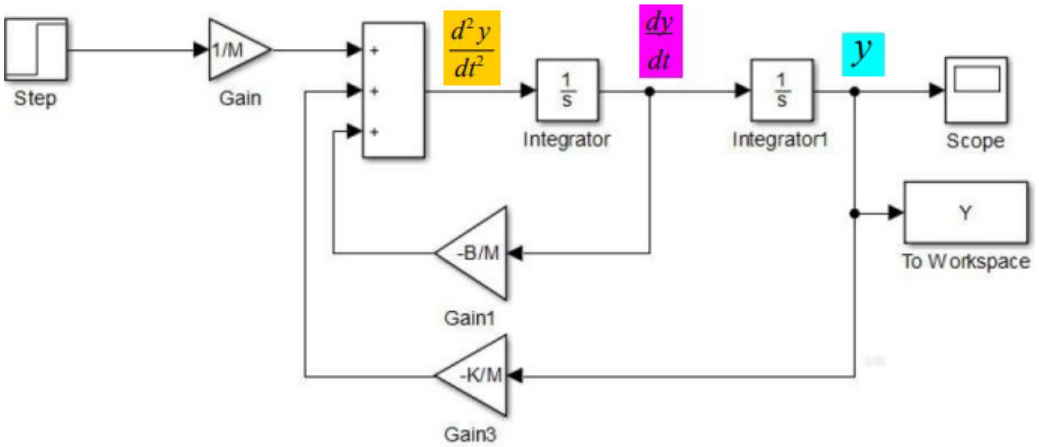


Fig.: Block diagram representation of a mass-spring-damper system.

Procedure:

1. **Open Simulink:**
    - Start MATLAB.
    - Open Simulink by typing simulink in the MATLAB command window.
2. **Create a New Model:**
    - Click on "Blank Model" in the Simulink start page.

Add Blocks:

- Drag and drop the following blocks from the Simulink Library Browser to your model:
    - **Integrator:** From Continuous.
    - **Sum:** From Math Operations.
    - **Gain:** From Math Operations.
    - **Step:** From Sources.
    - **Scope:** From Sinks.
    - **MATLAB Function:** From User-Defined Functions (for custom equations).

Configure Blocks:

- **Integrator Blocks:** Two integrator blocks will be used to represent the integration of acceleration to velocity and velocity to displacement.
- **Sum Block:** Configure the sum block to add and subtract forces.
- **Gain Blocks:** Set the gains to represent the spring constant (k) and damping coefficient (c).
- **MATLAB Function Block:** Define the differential equations for the mass-spring-damper system. Example function for mass_spring_damper:

Connect Blocks:

- Connect the blocks as shown in the diagram:
    - The Step block connects to the Sum block.
    - The Sum block connects to the MATLAB Function block.
    - The MATLAB Function block connects to the Integrator blocks.
    - The Integrator blocks connect to the Scope block for visualization.

Run the Simulation:

- Click on the "Run" button in the Simulink toolbar.

View Results:

- Double-click the Scope block to view the displacement and velocity responses of the mass-spring-damper system.

Results:

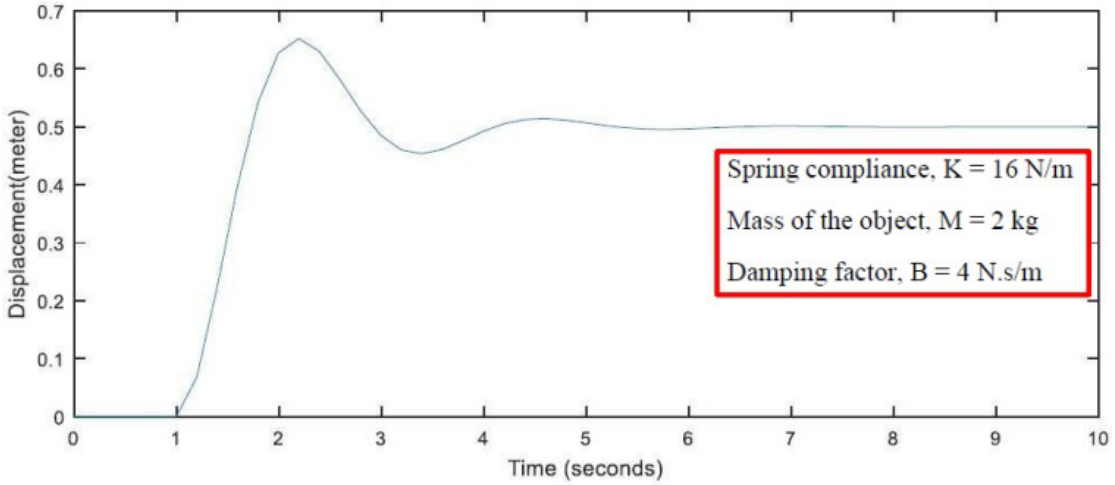


Fig.: Displacement response a mass-spring-damper system.

Results:

1. Amplitude of Oscillations: The peak displacement of the mass from its equilibrium position.
2. Transient Response: How the system responds initially to a disturbance before settling.
3. Steady-State Response: The final displacement of the system after transient effects have dissipated.
4. Overshoot: The extent to which the system exceeds its final steady-state value during transient response.
5. Settling Time: The time it takes for the system to settle within a certain percentage of its final value.
6. Phase Shift: The shift in phase between the input force and the displacement response.

Applications:

1. Electromechanical Systems: Design and control precision movements in robotic arms and industrial automation.
2. Vibration Control in Electrical Machinery: Model and design damping solutions for generators, transformers, and motors.
3. Power System Stability: Design damping controllers to mitigate oscillations and maintain system stability.
4. Control of Flexible Structures: Minimize vibrations in satellite antennas, solar panels, and transmission towers.
5. Seismic Protection for Electrical Infrastructure: Design seismic dampers for substations and control centers.
6. Signal Processing and Filtering: Design filters to remove noise and unwanted frequencies from signals.
7. Design of MEMS Devices: Optimize performance and reliability of MEMS sensors and actuators.
8. Control System Education and Research: Teach concepts of dynamic response, stability, and control design.

Discussion: Understanding the mass-spring-damper system is fundamental in engineering and control systems because it represents a simple yet powerful model for analyzing dynamic behavior and response. Learning about this system provides critical insights into how systems respond to forces and disturbances, which is essential for designing effective control strategies and predicting system behavior. By studying the mass-spring-damper system, engineers gain the ability to evaluate key performance metrics such as oscillatory behavior, damping effects, and settling times, which are crucial for ensuring stability and performance in real-world applications. Additionally, this knowledge helps in tuning system parameters to achieve desired performance, making it a foundational concept for optimizing and controlling mechanical and dynamic systems in various engineering fields.

Conclusion:

The experiment demonstrated the analysis and simulation of a mass-spring-damper system using MATLAB Simulink. Understanding the characteristics of such systems is crucial for control system design and dynamic analysis.

Experiment No. 04

Name of the Experiment: Analysis of the Characteristics of an RLC Series Circuit in MATLAB Simulink

Objective:

The objective of this experiment is to analyze the frequency response and transient behavior of an RLC series circuit using MATLAB Simulink. By the end of this experiment, students will be able to:

- Model an RLC series circuit in Simulink.
- Perform frequency response analysis using Bode plots.
- Observe and interpret the transient response of the circuit to a step input.

Introduction:

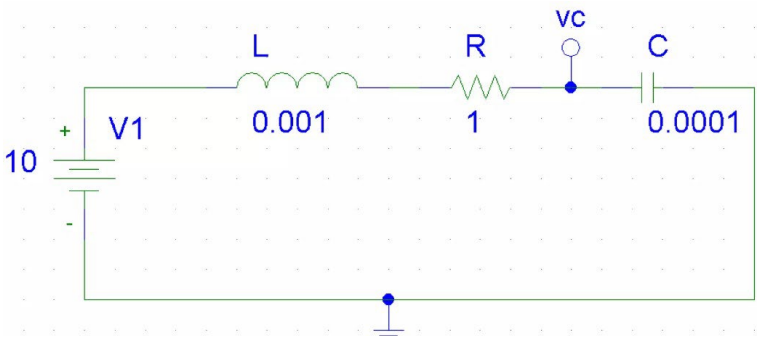
RLC series circuits, composed of a resistor (R), inductor (L), and capacitor (C) connected in series, are fundamental in electrical engineering for applications such as filtering, tuning, and impedance matching. Understanding the behavior of these circuits, particularly their transient and steady-state responses, is essential for designing and analyzing electrical systems.

In this experiment, MATLAB Simulink will be utilized to model and analyze an RLC series circuit. Simulink offers a graphical environment for modeling and simulating dynamic systems using block diagrams. By creating a Simulink model of the RLC circuit, students can visually examine its response to different inputs.

The experiment will focus on the circuit's response to a step input, observing how the voltage across the resistor, inductor, and capacitor changes over time. This analysis will provide insights into the circuit's transient response, resonance, and damping characteristics. Through this hands-on experience, students will enhance their understanding of RLC circuits and learn

to apply theoretical concepts to practical scenarios, which is crucial for control system design and optimization.

Diagram:



Fiquare: RLC series circuit

Solution:

Using Kvl in loop,

$$L\frac{di}{dt} + Ri + \int idt = V_i(t) \quad \ldots\ldots\ldots\ldots (1)$$

$$\frac{1}{C}\int idt = V_0(t) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Now, taking Laplace Transform of Equation 1 and 2,

$$LI(s) + RI(s) + \frac{1}{Cs}I(s) = V_i(s)$$

$$\Rightarrow \quad I(s)\left[L + R + \frac{1}{C}\right] = V_i(s) \quad \ldots\ldots\ldots\ldots\ldots (3)$$

$$V_o(s) = \frac{1}{Cs}I(s) \quad \ldots\ldots\ldots\ldots\ldots\ldots (4)$$

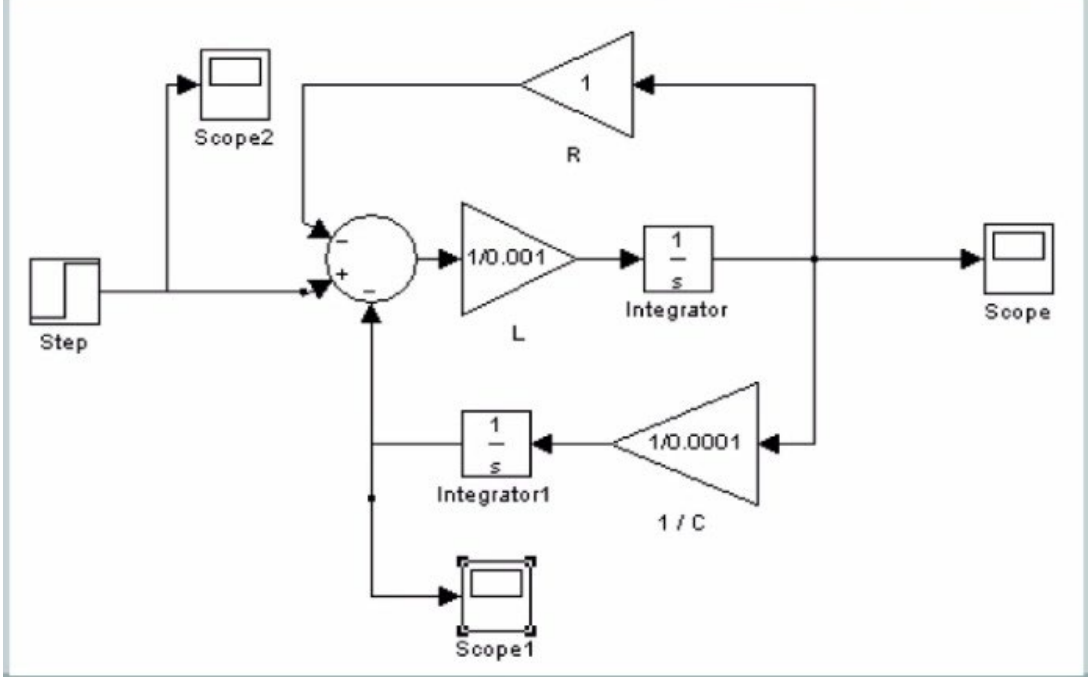
Now from equation 3 and 4, we have,

$$\frac{V_o(s)}{V_i(s)} = \frac{1}{Cs\left(L + R + \frac{1}{C}\right)}$$

$$= \frac{1}{LCs^2 + RCs + 1}$$

Analysis:

$$Ri(t) + L\frac{di}{dt} + \frac{1}{C}\int_{-\infty}^{\tau=t} i(\tau)\,d\tau = v(t)$$



Procedure:

1. **Open Simulink:**
    - Start MATLAB.
    - Open Simulink by typing simulink in the MATLAB command window.
2. **Create a New Model:**
    - Click on "Blank Model" in the Simulink start page.
    - Save your model with a relevant name, such as RLC_Series_Circuit.
3. **Add Blocks to the Model:**
    - Drag and drop the following blocks from the Simulink Library Browser:
        - **AC Voltage Source:** From Simscape > Electrical > Specialized Power Systems > Sources.
        - **Series RLC Branch:** From Simscape > Electrical > Specialized Power Systems > Passive Components.
        - **Scope:** From Sinks (for viewing output signals).

        - **Voltage Measurement:** From Simscape > Electrical > Specialized Power Systems > Sensors (to measure voltage across components).
        - **Current Measurement:** From Simscape > Electrical > Specialized Power Systems > Sensors (to measure current).
4. **Configure the Blocks:**
    - **AC Voltage Source:**
        - Double-click the block and set the amplitude and frequency of the AC source as needed for your analysis.
    - **Series RLC Branch:**
        - Double-click the block and configure the resistance (R), inductance (L), and capacitance (C) according to your experimental setup. These values should match those specified in your lab instructions.
    - **Voltage and Current Measurement:**
        - Place the voltage measurement block across the resistor or capacitor to observe the voltage drop.
        - Place the current measurement block to measure the current flowing through the circuit.
5. **Connect the Blocks:**
    - Connect the blocks as follows:
        - The AC Voltage Source connects to the Series RLC Branch.
        - The Series RLC Branch connects to the Voltage and Current Measurement blocks.
        - Connect the outputs of the Voltage and Current Measurement blocks to the Scope block to visualize the results.
6. **Configure Simulation Settings:**
    - Click on the Simulation tab and set the simulation parameters:
        - Choose a suitable solver (e.g., ode45).
        - Set the simulation time based on your analysis needs.
7. **Run the Simulation:**
    - Click on the "Run" button in the Simulink toolbar to start the simulation.

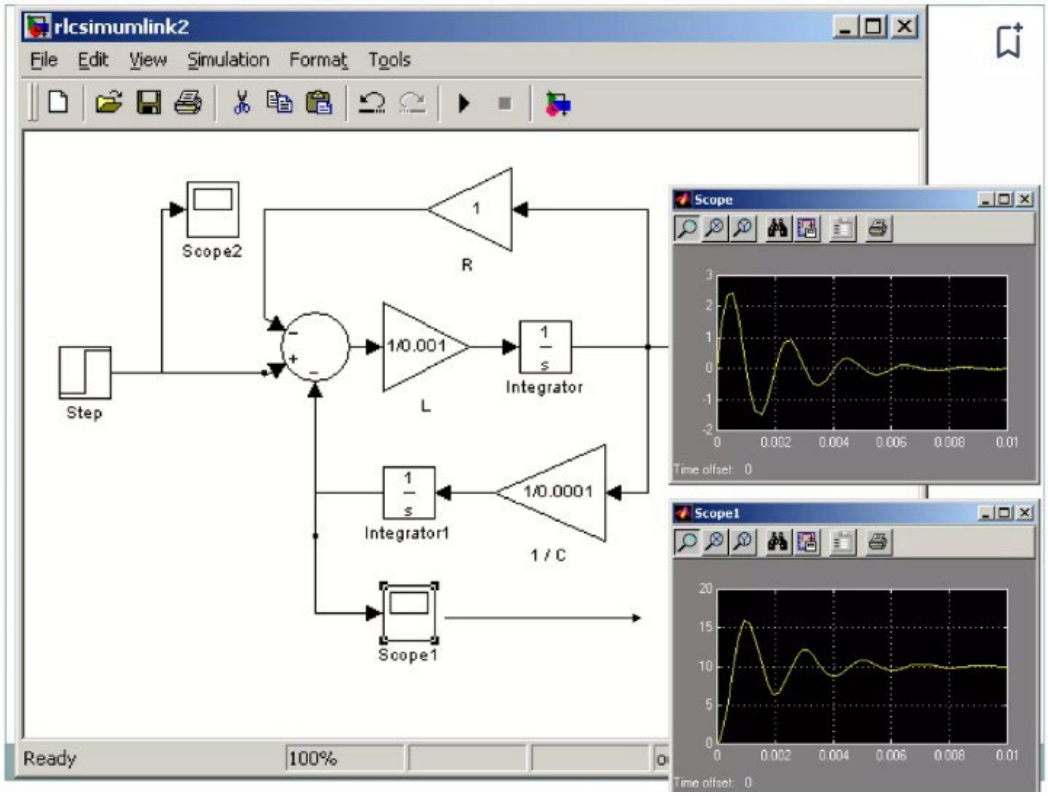
5. Analyze Results:

- **Frequency Response:**

- Use a Bode plot to analyze the frequency response of the RLC circuit. Add a **Bode Plot** block from the Simulink Library (or use the MATLAB command bodewith the transfer function obtained).
- Examine the resonance frequency, gain, and phase shift.

- **Transient Response:**
  - View the output signals (voltage across the components and current) on the Scope block.
  - Observe how the circuit responds to the AC input over time, including the effects of resonance and damping.

Results:



Fiquare: Characteristics of RLC series circuit.

Result:

The MATLAB Simulink experiment on the RLC series circuit revealed key insights into its transient and steady-state responses. Initially, the circuit exhibited oscillations due to inductive and capacitive elements, which settled into a steady state over time. The resonance frequency, where impedance is minimized and current maximized, was observed, along with

the damping behavior influenced by resistance. Higher resistance led to increased damping, reducing oscillations. The experiment underscored the importance of RLC circuit analysis in practical applications like filter design and power quality management, providing a deeper understanding of resonance, damping, and dynamic behavior essential for control system design.

Application:

1. Power Quality Analysis: Evaluate and improve the quality of power in electrical networks by analyzing resonance and filtering characteristics.
2. Filter Design: Design and optimize filters for signal processing applications by understanding frequency response.
3. Transient Response Analysis: Study and mitigate the effects of transient disturbances in power systems.
4. Impedance Matching: Ensure efficient power transfer in communication and transmission lines by analyzing impedance characteristics.
5. Circuit Stability: Assess and enhance the stability of electrical circuits by examining damping effects.
6. Energy Storage Systems: Analyze the charging and discharging behavior of capacitors and inductors in energy storage applications.
7. Resonant Circuits: Design resonant circuits for applications like radio frequency (RF) circuits and oscillators.
8. Educational Tool: Provide practical understanding and visualization of RLC circuit behavior for educational purposes.

Discussion:

The analysis of the RLC series circuit using MATLAB Simulink provided valuable insights into the dynamic behavior of these fundamental electrical components. The transient response of the circuit demonstrated oscillatory behavior due to the interplay between the inductor and capacitor. These oscillations were heavily influenced by the damping effect of the resistor, which controlled how quickly the system returned to steady state.

The resonance frequency, where the inductive and capacitive reactances canceled each other out, resulted in maximum current and minimum impedance. This characteristic is crucial for applications requiring tuned circuits, such as radio frequency (RF) filters and oscillators. The

damping ratio, determined by the resistance value, showed its importance in managing the circuit's oscillatory behavior, highlighting the need for precise component selection in practical designs.

Through simulation, it was evident that varying the component values (R, L, and C) significantly affected the circuit's performance. This variability underscores the importance of understanding each component's role and its impact on the overall system. The energy exchange between the inductor and capacitor, and its eventual dissipation through the resistor, illustrated the principles of energy conservation and loss in real-world circuits.

Conclusion:

The experiment successfully demonstrated the characteristics of an RLC series circuit, emphasizing the importance of transient and steady-state analysis. Key findings include the identification of the resonance frequency, the impact of damping on oscillations, and the critical role of component values in circuit behavior. These insights are essential for designing efficient and reliable electrical systems, such as filters, tuning circuits, and power management systems. The hands-on experience with MATLAB Simulink provided a practical understanding of these theoretical concepts, enhancing the ability to apply them in control system design and optimization.

Experiment No. 05

Name of the experiment: Analysis of the Characteristics of a DC Motor and DC Motor Simulation in MATLAB.

Objective:

The objective of this experiment is to analyze the characteristics of a DC motor and simulate its behaviour using MATLAB Simulink. By the end of this experiment, students will be able to:

1. Understand the fundamental principles of DC motor operation.
2. Create a Simulink model of a DC motor.
3. Perform a step response analysis of the DC motor.
4. Interpret the simulation results to understand the motor's performance parameters, such as speed and torque.

Introduction:

DC motors are widely used in various applications ranging from household appliances to industrial automation due to their simplicity, reliability, and controllability. They convert electrical energy into mechanical energy through the interaction of magnetic fields and conductors. Understanding the behavior and characteristics of DC motors is crucial for designing effective control systems.

In this experiment, we will utilize MATLAB Simulink, a graphical programming environment, to model and simulate a DC motor. Simulink provides a user-friendly interface that allows for the construction and simulation of dynamic systems using block diagrams. By creating a DC motor model in Simulink, students can visually analyze its response to different inputs and understand key performance metrics. We will simulate a step response to observe how the motor's speed changes over time when subjected to a sudden change in input voltage. This analysis helps in understanding the motor's transient and steady-state behavior,

which are essential for control system design and optimization.

Diagram:

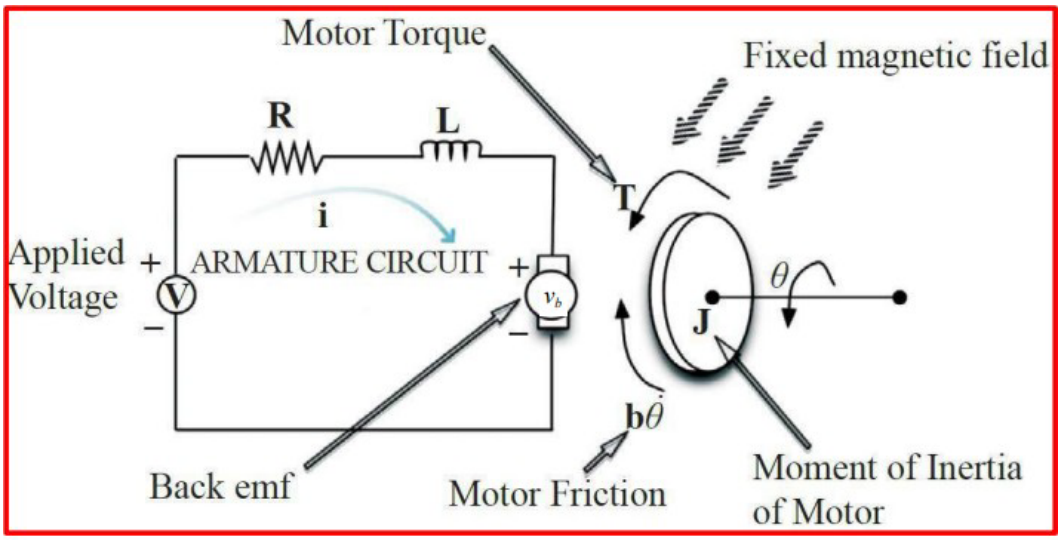


Figure : Simplified model of a separately excited DC motor

R = Armature resistance	θ= Angular position
L = Armatureinductance	ω= Angular velocity
i_a = Armature current	J = Rotor inertia
v_i = Input voltage	B = Viscous friction
v_b = Back emf	$\tau(t)$ = Motor torque

Now armature circuit:

$$L\frac{di_a}{dt} + Ri_a + v_b = v_i$$

$$\Rightarrow L\frac{di_a}{dt} + Ri_a + K_b\omega = v_i \text{-----(i)}$$

since $v_b = K_b\omega$

An equation describing the rotational motion of the inertial load:

$$J\frac{d\omega}{dt} + B\omega = K_t i_a \quad -----(ii)$$

and $$\frac{d\theta}{dt} = \omega \quad -----(iii)$$

Now from Eq. (i) and (iii)

$$L\frac{di_a}{dt} + Ri_a + K_b\frac{d\theta}{dt} = v_i$$

$$\Rightarrow LsI_a(s) + RI_a(s) + K_b s\theta(s) = V_i(s)$$

$$\Rightarrow I_a(s) = \frac{V_i(s) - K_b s\theta(s)}{Ls + R} \quad -----(iv)$$

Now from equation (ii) and (iii)

$$J\frac{d^2\theta}{dt^2} + B\frac{d\theta}{dt} = K_t i_a$$

$$\Rightarrow \quad Js^2\theta(s) + Bs\theta(s) = K_t i_a(s)$$

$$\Rightarrow \quad Js^2\theta(s) + Bs\theta(s) = K_t \frac{V_i(s) - K_b s\theta(s)}{Ls+R}$$

$$\Rightarrow \quad \frac{\theta(s)}{V_i(s)} = \frac{K_t}{s[(Js+B)(Ls+R) + K_t K_b]}$$

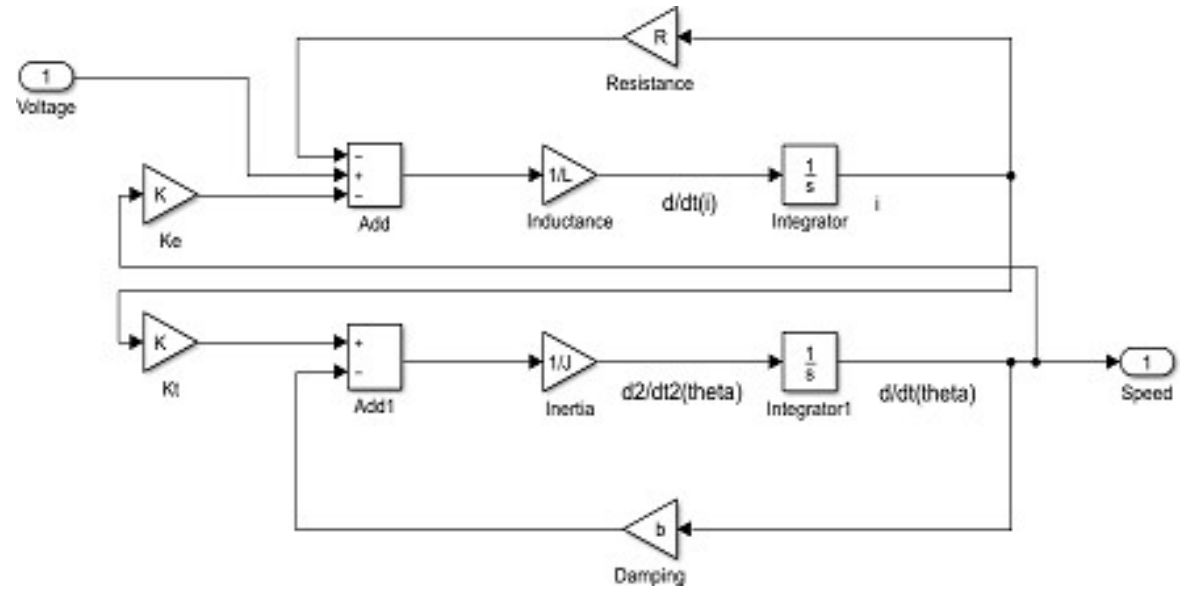
Now, $\frac{d\theta}{dt} = \omega$

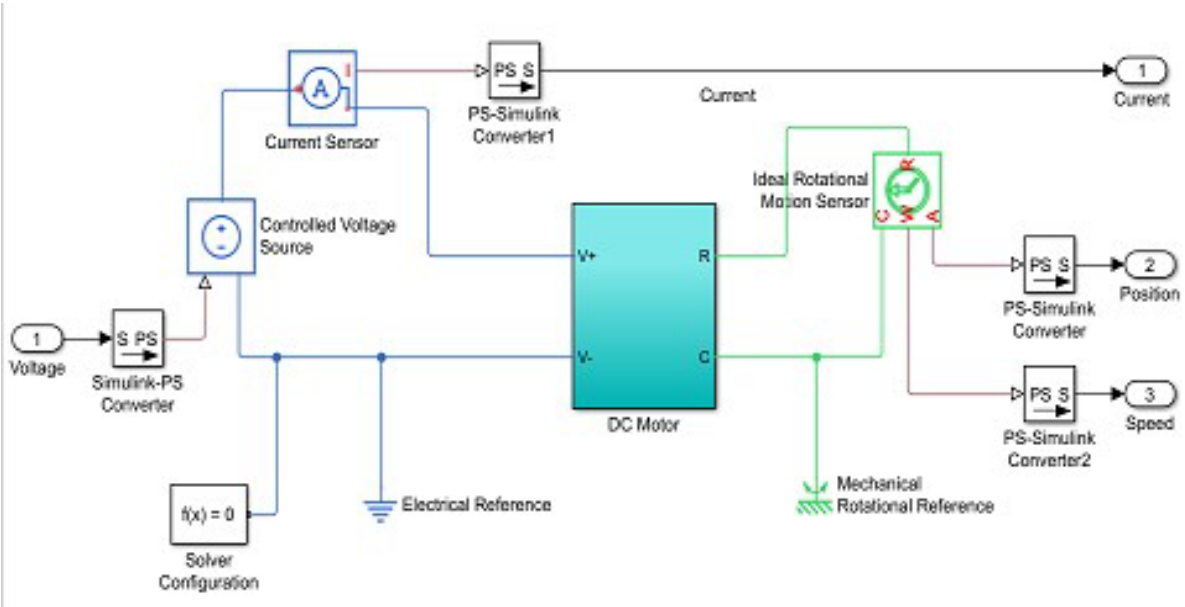
$\Rightarrow s\theta(s) = \omega$

$\Rightarrow \theta(s) = \omega/s$

$$\frac{\omega(s)}{V_i(s)} = \frac{K_t}{(Js+B)(Ls+R)+K_tK_b}$$

Model:





Procedure:

1. **Open Simulink:**
    - Start MATLAB.
    - Open Simulink by typing simulink in the MATLAB command window.

2. **Create a New Model:**
    - Click on "Blank Model" in the Simulink start page.
3. **Add Blocks:**
    - Drag and drop the following blocks from the Simulink Library Browser to your model:
        - **DC Motor:** From Simscape > Electrical > Specialized Power Systems > Machines.
        - **Step:** From Sources.
        - **Scope:** From Sinks.
        - **PS-Simulink Converter:** From Simscape > Utilities.
        - **Simulink-PS Converter:** From Simscape > Utilities.
4. **Connect Blocks:**
    - Connect the blocks as shown in the diagram:
        - The **Step** block connects to the **Simulink-PS Converter** block.
        - The **Simulink-PS Converter** block connects to the **DC Motor** block.
        - The **DC Motor** block connects to the **PS-Simulink Converter** block.
        - The **PS-Simulink Converter** block connects to the **Scope** block.
5. **Configure Blocks:**
    - Double-click the **Step** block and set its step time to 1, initial value to 0, and final value to 1.
    - Double-click the **DC Motor** block to check and understand its parameters (e.g., armature resistance, inductance, back EMF constant).

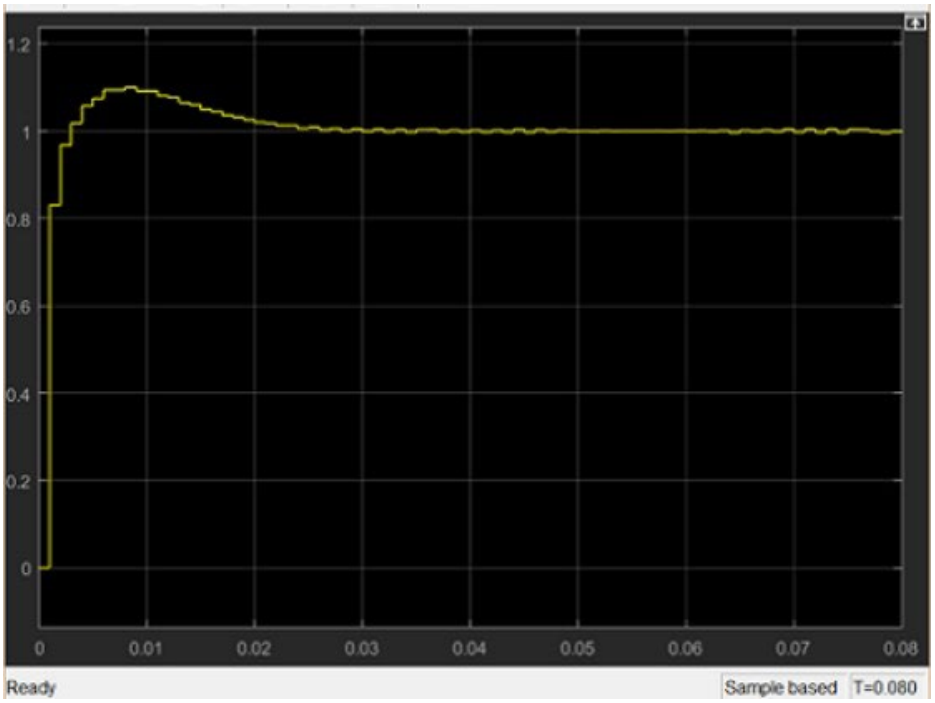
6. Run the Simulation:

- Click on the "Run" button in the Simulink toolbar.

7. View Results:

- Double-click the **Scope** block to view the motor's speed response and other characteristics.

Output:



Result:

The MATLAB Simulink experiment on DC motor characteristics revealed that the motor exhibits significant dynamic changes during the transient period, such as initial spikes in speed, torque, and current when a step input is applied. These parameters stabilise to steady-state values, indicating good damping characteristics. The analysis highlighted the motor's behaviour, essential for control system design, energy efficiency optimization, and fault detection. The experiment demonstrated the effectiveness of MATLAB Simulink for modelling and simulating DC motor performance, providing valuable insights for engineering applications and further research.

Application:

1. **Motor Control Design:** Develop and optimize speed and torque control algorithms.
2. **Performance Analysis:** Evaluate motor behavior under various conditions.
3. **Fault Diagnosis:** Implement fault detection and monitoring systems.
4. **Energy Efficiency:** Improve energy consumption in different load scenarios.
5. **Drive System Integration:** Design and simulate drive systems for various applications.
6. **Educational Use:** Teach DC motor principles and control techniques.

7. **Load Simulation:** Analyse motor response to load variations.
8. **Renewable Energy:** Optimise performance in renewable energy applications.
9. **Power Electronics:** Study interactions with inverters and converters.
10. **R&D:** Innovate and develop new motor control technologies.

Discussion:

The analysis of the characteristics of a DC motor using MATLAB Simulink provided valuable insights into its dynamic and steady-state behavior. The simulations allowed for a detailed examination of how the motor responds to various inputs, including step changes in voltage and load variations. Key performance metrics such as speed, torque, and current were observed, highlighting the motor's response characteristics and the effects of different control strategies.

The ability to simulate faults and monitor the motor's performance under different conditions demonstrated the importance of fault detection and diagnostic algorithms. Energy efficiency analysis showed how load conditions impact energy consumption, emphasizing the need for optimization in real-world applications.

Integration with drive systems and power electronics was another critical aspect of the experiment. By modeling the motor's interaction with inverters and converters, the simulation provided insights into improving system efficiency and reliability. This comprehensive analysis is essential for designing effective control systems and optimizing motor performance in various industrial applications.

Conclusion:

The experiment successfully demonstrated the use of MATLAB Simulink for analyzing the characteristics of a DC motor. The simulations provided a thorough understanding of the motor's dynamic and steady-state responses, fault diagnosis, energy efficiency, and integration with power electronics and drive systems. These insights are crucial for EEE engineers in designing, optimizing, and implementing control systems in industrial automation, renewable energy, and other applications. The hands-on experience with MATLAB Simulink enhances practical skills, bridging the gap between theoretical knowledge and real-world applications.

Experiment No: 06

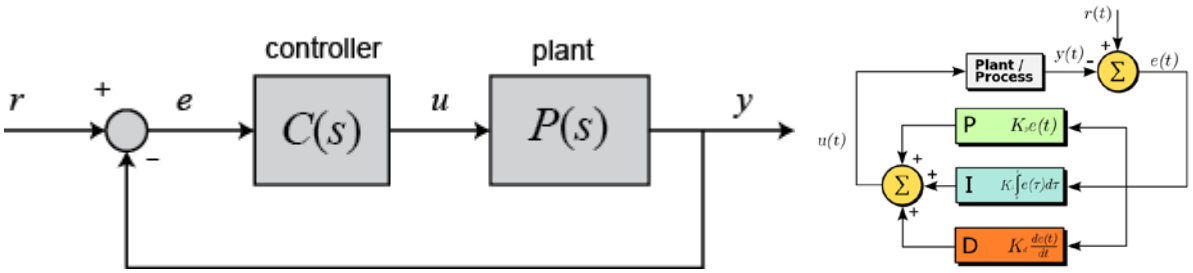
Name of the experiment: Modeling of Physical systems and study of their closed loop response

Objective:

(i) The objective of this experiment is the modeling of physical systems and study of their closed loop response.
(ii) Study closed loop system with the variation of different parameters of the system
(iii) Simulation of a dc motor
(iv) Study of PID controller
(v) Study of feedback
(vi) Check the state-space representation of the system.

Introduction:

We will consider the following unity feedback controlled system, where the plant is assumed as a dc motor and the controller is used as a P, I, D, PI, PD or PID controller.



System Modeling: Motor position, θ

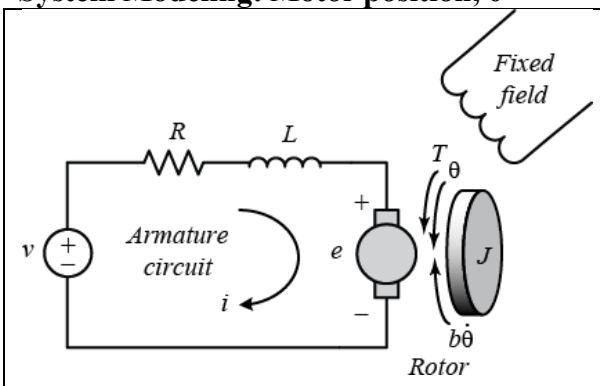


Figure: A dc motor

The physical parameters for the dc motor are:

(J)	moment of inertia of the rotor	0.01 kg.m^2
(b)	motor viscous friction constant 0.1 N.m.s	
(Ke)	electromotive force constant	0.01 V/rad/sec (Kt)
motor torque constant		0.01 N.m/Amp
(R)	electric resistance	1 Ohm
(L)	electric inductance	0.5 H

System Equations:

$$TT = kk_1 ii_{ff}\, ii = KK_{tt}\, ii$$

$$ee = kk_2 ii_{ff}\, \omega\omega = KK_{ee}\, \omega\omega = KK_{ee}\, \theta\dot{\theta}$$

In SI units, the motor torque and back emf constants are equal, that is, $K_t = K_e$; therefore, we will use K to represent both the motor torque constant and the back emf constant.

$$JJ\ddot{\theta\theta} + bb\dot{\theta\theta} = KKii$$

$$LL\frac{ddii}{ddtt} + RRii = VV - KK\dot{\theta\theta}$$

1. Transfer Function

Applying the Laplace transform, the above modeling equations can be expressed in terms of the Laplace variable s.

$$ss(JJss + bb)\theta\theta(ss) = KKKK(ss)$$

$$(LLss + RR)\mathbb{K}(ss) = VV(ss) - KKss\theta\theta(ss)$$

We arrive at the following open-loop transfer function by eliminating $I(s)$ between the two above equations, where the rotational speed is considered the output and the armature voltage is considered the input.

$$PP(ss) = \frac{\theta\theta(ss)}{VV(ss)} = \frac{KK}{(JJss + bb)(LLss + RR) + KK^2} \quad \left[\frac{rrrrdd/sseess}{VV}\right]$$

2. State-Space

In state-space form, the governing equations above can be expressed by choosing the rotational speed and electric current as the state variables. Again the armature voltage is treated as the input and the rotational speed is chosen as the output.

$$\frac{dd}{dddtt}\begin{bmatrix}\theta\theta\\ ii\end{bmatrix} = \begin{bmatrix}-\frac{bb}{JJ} & \frac{KK}{JJ}\\ -\frac{KK}{LL} & -\frac{RR}{LL}\end{bmatrix}\begin{bmatrix}\theta\theta\\ i\end{bmatrix} + \begin{bmatrix}0\\ \frac{1}{LL}\end{bmatrix}VV$$

$$yy = [1 \quad 0]\begin{bmatrix}\theta\theta\\ ii\end{bmatrix}$$

Design requirements

First consider that our uncompensated motor rotates at 0.1 rad/sec in steady state for an input voltage of 1 Volt. Since the most basic requirement of a motor is that it should rotate at the desired speed, we will require that the steady- state error of the motor speed be less than 1%. Another performance requirement for our motor is that it must accelerate to its steady-state speed as soon as it turns on. In this case, we want it to have a settling time less than 2 seconds. Also, since a speed faster than the reference may damage the equipment, we want to have a step response with overshoot of less than 5%. In summary, for a unit step command in motor speed, the control system's output should meet the following requirements.

- Settling time less than 2 seconds
- Overshoot less than 5%
- Steady-state error less than 1%

MATLAB representation

1. Transfer Function

We can represent the above open-loop transfer function of the motor in MATLAB by defining the parameters and transfer function as follows. Running this code in the command window produces the output shown below.

```
J = 0.01;
b = 0.1;
K = 0.01;
R = 1;
L = 0.5;
s = tf('s');
P_motor = K/((J*s+b)*(L*s+R)+K^2)
```

```
P_motor =

                0.01
  ---------------------------
  0.005 s^2 + 0.06 s + 0.1001

Continuous-time transfer function.
```

2. State Space

We can also represent the system using the state-space equations. The following additional MATLAB commands create a state-space model of the motor and produce the output shown below when run in the MATLAB command window.

```
A = [-b/J    K/J
     -K/L    -R/L];
B = [0
     1/L];
C = [1    0];
D = 0;
motor_ss = ss(A,B,C,D)
```

The above state-space model can also be generated by converting your existing transfer function model into state-space form. This is again accomplished with the ss command as shown below.

```
motor_ss = ss(P_motor);
```

PID controller:

The output of a PID controller, equal to the control input to the plant, in the time-domain is as follows:

$$uu(tt) = KK_{pp}\, ee(tt) + KK_{ii} \int ee(tt)\, ddtt + KK_{dd} \frac{ddee}{ddtt} \quad (1)$$

The transfer function of a PID controller is found by taking the Laplace transform of Eq.(1).

$$KK_{pp} + \frac{KK_{ii}}{ss} + KK_{d}\, ss = \frac{KK_{dd}\, ss^2 + KK_{pp}\, ss + KK_{ii}}{ss}$$

, where K_p= Proportional gain K_i= Integral gain K_d= Derivative gain

We can define a PID controller in MATLAB using the transfer function directly, for example:

```
Kp = 1;
Ki = 1;
Kd = 1;
s = tf('s');
C = Kp + Ki/s + Kd*s
```

Alternatively, we may use MATLAB's **pid controller object** to generate an equivalent continuous-time controller as follows:

```
C = pid(Kp,Ki,Kd)
Controller = tf(C)
```

The Characteristics of P, I, and D Controllers

A proportional controller (K_P) will have the effect of reducing the rise time and will reduce but never eliminate the **steady-state error**. An integral control (K_i) will have the effect of eliminating the steady- state error for a constant or step input, but it may make the transient response slower. A derivative control (K_d) will have the effect of increasing the stability of the system, reducing the overshoot, and improving the transient response. The effects of each of controller parameters, K_p, K_i, and K_d on a closed- loop system are summarized in the table below.

CL RESPONSE	RISE TIME	OVERSHOOT	SETTLING TIME	S-S ERROR
K_p	Decrease	Increase	Small Change	Decrease
K_i	Decrease	Increase	Increase	Eliminate
K_d	Small Change	Decrease	Decrease	No Change

Note that these correlations may not be exactly accurate, because K_p, K_i, and K_d are dependent on each other. In fact, changing one of these variables can change the effect of the other two. For this reason, the table should only be used as a reference when you are determining the values for K_p, K_i, and K_d .

General Tips for Designing a PID Controller

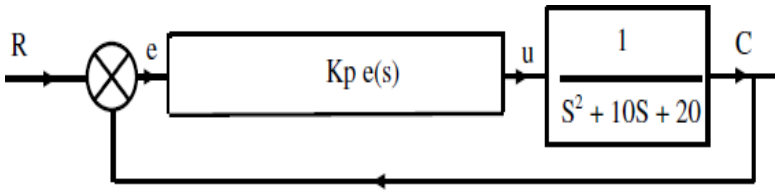
When you are designing a PID controller for a given system, follow the steps shown below to obtain a desired response.

1. Obtain an open-loop response and determine what needs to be improved
2. Add a proportional control to improve the rise time
3. Add a derivative control to improve the overshoot
4. Add an integral control to eliminate the steady-state error
5. Adjust each of 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 what characteristics.

Lastly, please keep in mind that you do not need to implement all three controllers (proportional, derivative, and integral) into a single system, if not necessary. For example, if a PI controller gives a good enough response (like the above example), then you don't need to implement a derivative controller on the system. Keep the controller as simple as possible.

Proportional Control:

Let's first try employing a proportional controller with a gain of 100, that is, $C(s) = 100$. To determine the closed-loop transfer function, we use the feedbackcommand. Add the following code to the end of your m- file.



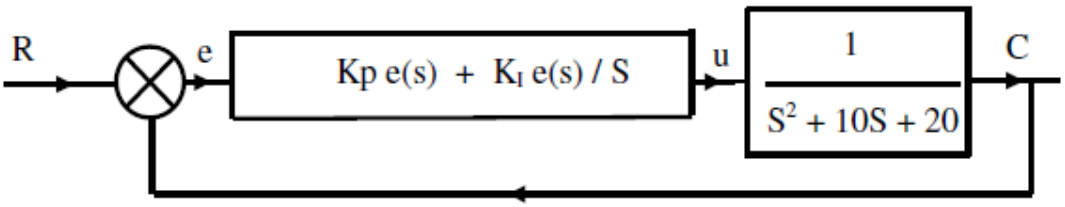

```
Kp = 100;   %% Use Kp = 300;
C = pid(Kp);
sys_cl = feedback(C*P_motor,1);
```

$$\text{Plant} = \frac{1}{S^2 + 10S + 20}$$

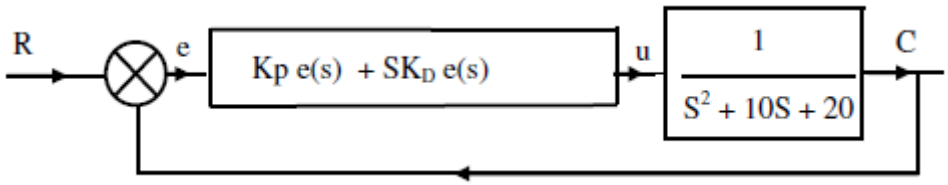
Now let's examine the closed-loop step response. Add the following commands to the end of your m-file and run it in the command window. You should generate the plot shown below. You can view some of the system's characteristics by right-clicking on the figure and choosing **Characteristics** from the resulting menu. In the figure below, annotations have specifically been added for **Settling Time**, **Peak Response**, and **Steady State**.

```
t = 0:0.01:5;
step(sys_cl,t) grid
title('Step Response with Proportional Control')
```

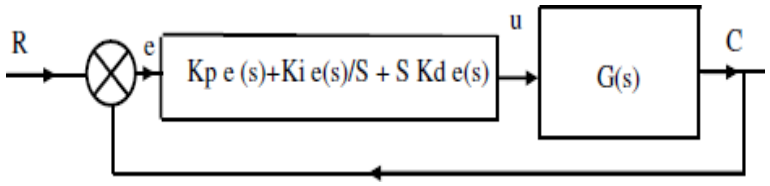
Proportional-Integral Control: $K_I = 30$ & $K_I = 70$



Proportional-Integral Control: $K_d = 10$ & $K_d = 20$



PID control:



$$G(s) = \frac{6575.25}{S^3 + 10.81S^2 + 848.43S + 6443.25}$$

Use MatLab tools, RLTOOL, design a PID controller with the following specifications: (i) Settling time of 0.5 seconds, (ii) Overshoot of <10%.

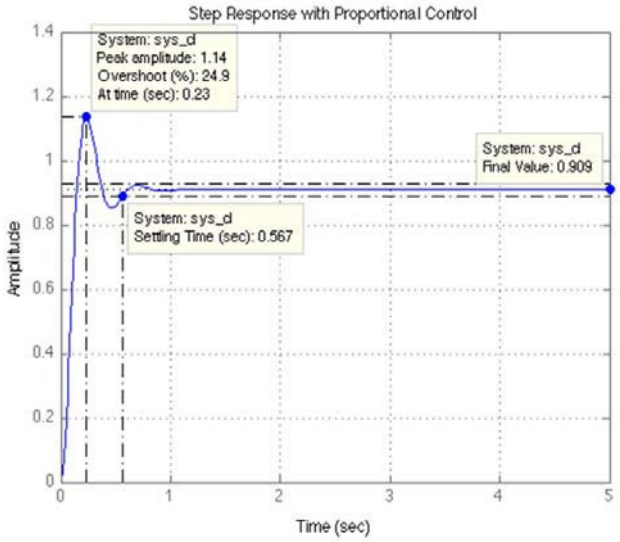
PID control

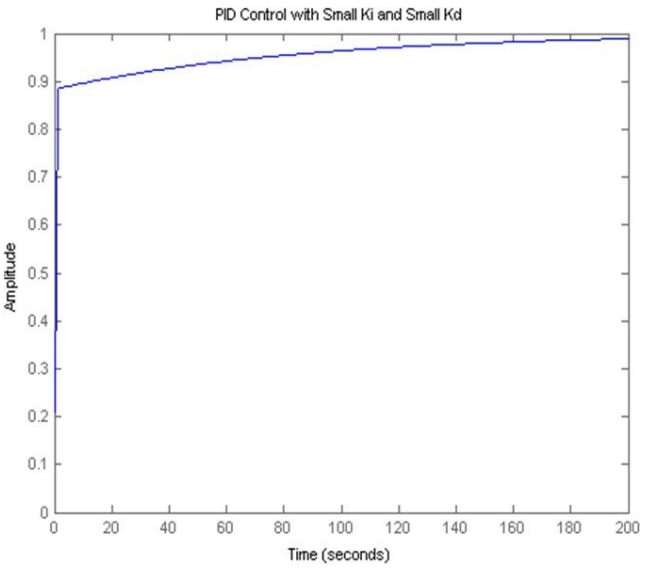
Let's try a PID controller with small *Ki* and *Kd*. Modify your m-file so that the lines defining your control are as follows. Running this new m-file gives you the plot shown below.

```
Kp = 75;
Ki = 1;
Kd = 1;
```

```
C = pid(Kp,Ki,Kd);
sys_cl = feedback(C*P_motor,1); step(sys_cl,[0:1:200])
title('PID Control with Small Ki and Small Kd')
```

Inspection of the above indicates that the steady-state error does indeed go to zero for a step input. However, the time it takes to reach steady-state is far larger than the required settling time of 2 seconds.





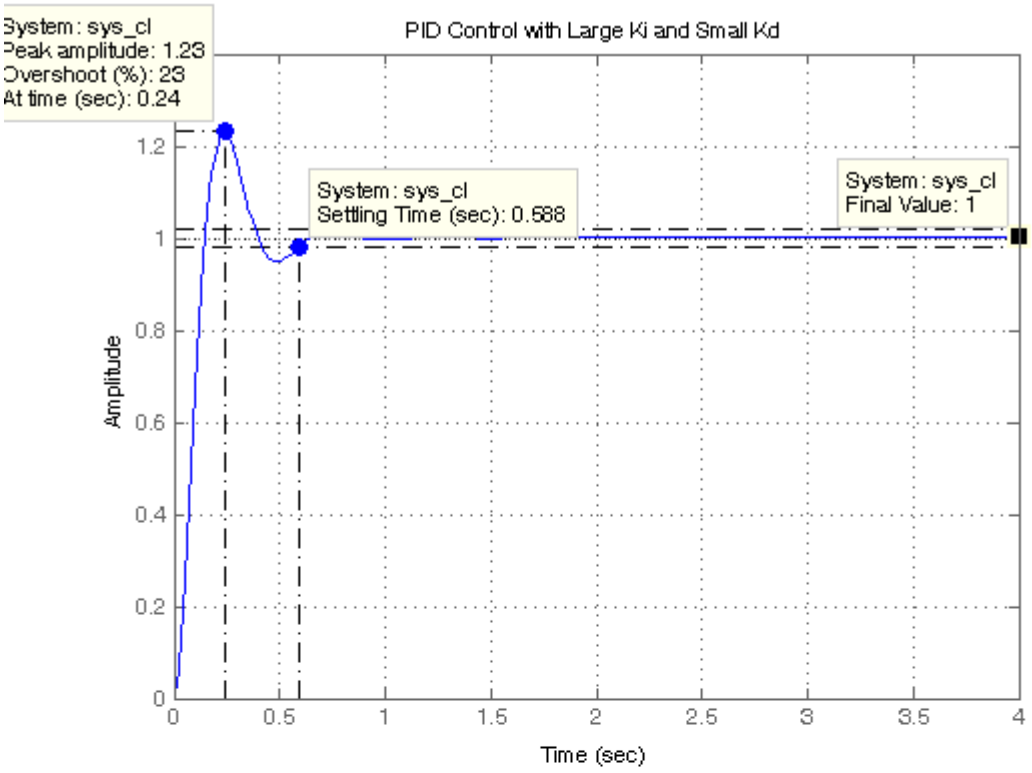
Tuning the gains

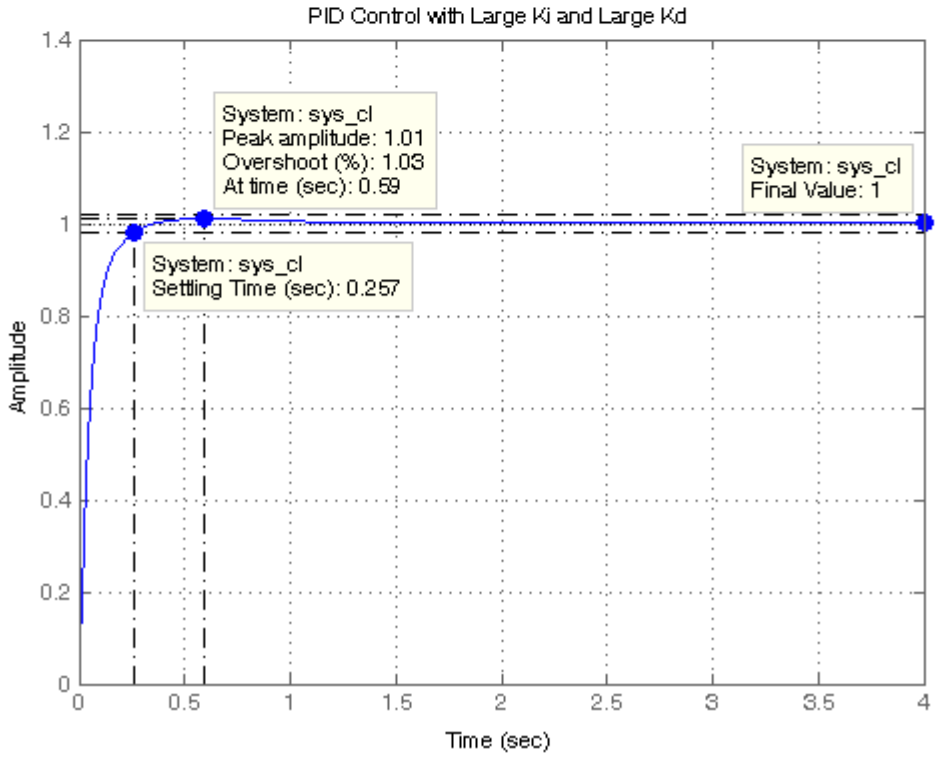
In this case, the long tail on the step response graph is due to the fact that the integral gain is small and, therefore, it takes a long time for the integral action to build up and eliminate the steady-state error. This process can be sped up by increasing the value of *Ki*. Go back to your m-file and change *Ki* to 200 as in the following. Rerun the file and you should get the plot shown below. Again the annotations are added by right- clicking on the figure and choosing **Characteristics** from the resulting menu.

```
Kp = 100;
Ki = 200;
Kd = 1;
C = pid(Kp,Ki,Kd);
sys_cl = feedback(C*P_motor,1); step(sys_cl, 0:0.01:4)
grid
title('PID Control with Large Ki and Small Kd')
```

As expected, the steady-state error is now eliminated much more quickly than before. However, the large *Ki* has greatly increased the overshoot. Let's increase *Kd* in an attempt to reduce the overshoot. Go back to the m-file and change *Kd* to 10 as shown in the following. Rerun your m-file and the plot shown below should be generated.

```
Kp = 100;
Ki = 200;
Kd = 10;
C = pid(Kp,Ki,Kd);
sys_cl = feedback(C*P_motor,1); step(sys_cl, 0:0.01:4)
grid
title('PID Control with Large Ki and Large Kd')
```





As we had hoped, the increased *Kd* reduced the resulting overshoot. Now we know that if we use a PID controller with *Kp* = 100, *Ki* = 200, and *Kd* = 10, all of our design requirements will be satisfied.

Experiment No. 07

Name of the experiment: Root Locus Controller Design of Physical systems

Key MATLAB commands: feedback, rlocus, step, sisotool

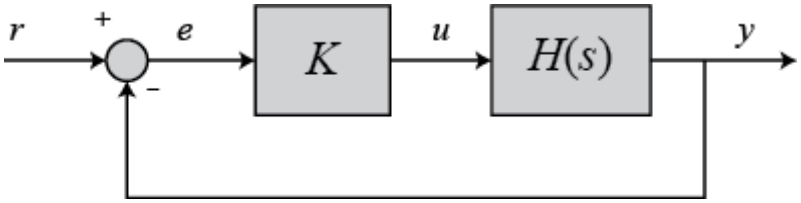
Objective:

(i) The objective of this experiment is to study of their root locus controller design.
(ii) Closed-Loop Poles
(iii) Plotting the Root Locus of a Transfer Function
(iv) Choosing a Value of K from the Root Locus
(v) Closed-Loop Response
(vi) Using SISOTOOL for Root Locus Design

Introduction to Root Locus Controller Design**:**

Closed-Loop Poles:

The root locus of an (open-loop) transfer function $HH(ss)$ is a plot of the locations (locus) of all possible closed-loop poles with proportional gain Kand unity feedback.



The closed-loop transfer function is:

$$\frac{YY(ss)}{RR(ss)} = \frac{KKHH(ss)}{1+KKHH(ss)} \qquad (1)$$

and thus the poles of the closed-loop poles of the closed-loop system are values of ss such that

$$1 + KKHH(ss) = 0 \qquad (2)$$

If we write $HH(ss) = bb(ss)/aa(ss)$, then this equation has the form:

$$aa(ss) + KKbb(ss) = 0 \qquad (3)$$

$$\frac{aa(ss)}{KK} + bb(ss) = 0 \qquad (4)$$

Let nn = order of $aa(ss)$ and mm = order of $bb(ss)$ (the order of a polynomial is the highest power of ss that appears in it).

We will consider all positive values of K. In the limit as $KK \to 0$, the poles of the closed-loop system are $aa(ss) = 0$ or the poles of $HH(ss)$. In the limit as $KK \to \infty$, the poles of the closed-loop system are $bb(ss) = 0$ or the zeros of $HH(ss)$.

No matter what we pick K to be, **the closed-loop system must always have** nn **poles**, where nn is the number of poles of $HH(ss)$. **The root locus must have** nn **branches**, each branch starts at a pole of $HH(ss)$ and goes to a zero of $HH(ss)$. If $HH(ss)$ has more poles than zeros (as is often the case), $mm < nn$ and we say that $HH(ss)$ has **zeros at infinity**. In this case, the limit of $HH(ss)$ as $ss \to \infty$ is zero. The number of zeros at infinity is $nn - mm$, the number of poles minus the number of zeros, and is the number of branches of the root locus that go to infinity (asymptotes).

Since the root locus is actually the locations of all possible closed-loop poles, from the root locus we can select a gain such that our closed-loop system will perform the way we want. If any of the selected poles are on the right half plane, the closed-loop system will be unstable. The poles that are closest to the imaginary axis have the greatest influence on the closed-loop response, so even though the system has three or four poles, it may still act like a second or even first order system depending on the location(s) of the dominant pole(s).

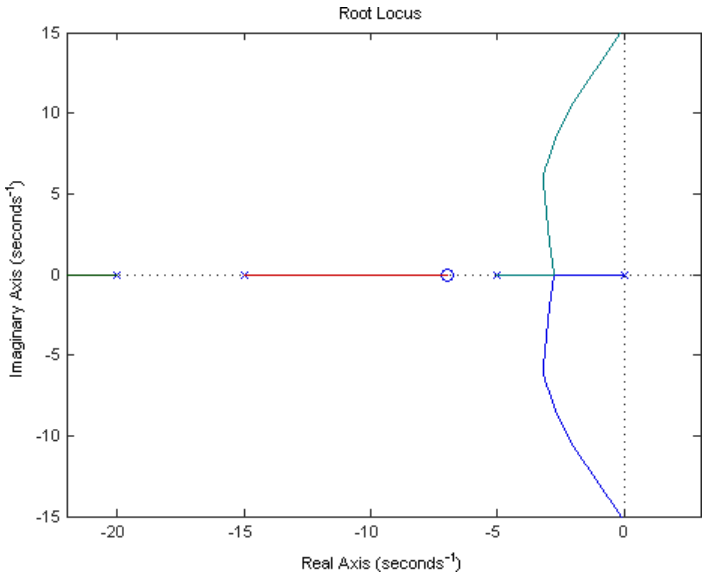
Plotting the Root Locus of a Transfer Function

Consider an open-loop system which has a transfer function of

$$HH(ss) = \frac{YY(ss)}{UU(ss)} = \frac{ss+7}{ss(ss+5)(ss+15)(ss+20)} \qquad (4)$$

How do we design a feedback controller for the system by using the root locus method? Say our design criteria are 5% overshoot and 1 second rise time. Make a MATLAB file called rl.m. Enter the transfer function, and the command to plot the root locus:

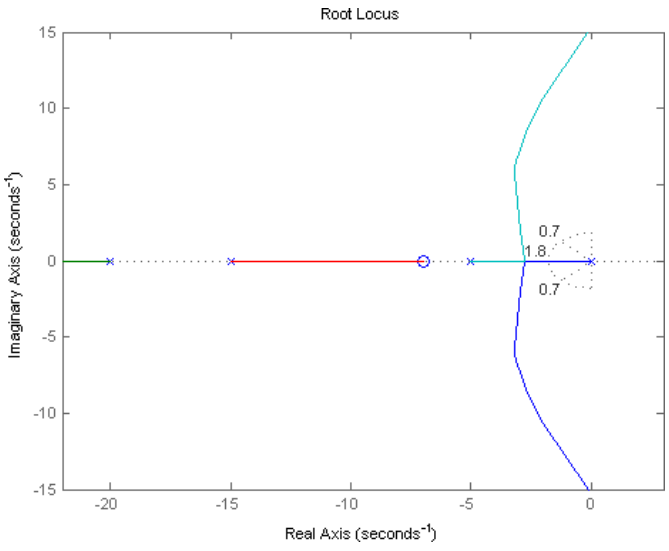
```
s = tf('s');
sys = (s + 7)/(s*(s + 5)*(s + 15)*(s + 20)); rlocus(sys)
axis([-22 3 -15 15])
```



Choosing a Value of K from the Root Locus

The plot above shows all possible closed-loop pole locations for a pure proportional controller. Obviously not all of those closed-loop poles will satisfy our design criteria, To determine what part of the locus is acceptable, we can use the command sgrid(Zeta,Wn) to plot lines of constant damping ratio and natural frequency. Its two arguments are the damping ratio ( ) and natural frequency ( ) [these may be vectors if you want to look at a range of acceptable values]. In our problem, we need an overshoot less than 5% (which means a damping ratio of greater than 0.7) and a rise time of 1 second (which means a natural frequency greater than 1.8). Enter the following in the MATLAB command window:

```
Zeta = 0.7;
Wn = 1.8;
sgrid(Zeta,Wn)
```



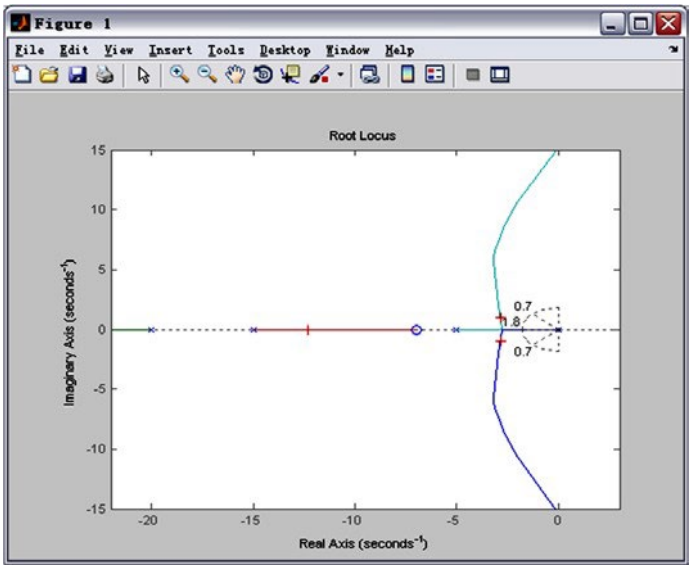
On the plot above, the two dotted lines at about a 45^0 angle indicate pole locations with $\zeta = 0.7$; in between these lines, the poles will have $\zeta > 0.7$ and outside of the lines $\zeta < 0.7$. The semicircle indicates pole locations with a natural frequency $\omega\omega_{nn} = 1.8$; inside the circle, $\omega\omega_{nn} < 1.8$ and outside the circle $\omega\omega_{nn} > 1.8$.

Going back to our problem, to make the overshoot less than 5%, the poles have to be in between the two white dotted lines, and to make the rise time shorter than 1 second, the poles have to be outside of the white dotted semicircle. So now we know only the part of the locus outside of the semicircle and in betwen the two lines are acceptable. All the poles in this location are in the left-half plane, so the closed-loop system will be stable.

From the plot above we see that there is part of the root locus inside the desired region. So in this case, we need only a proportional controller to move the poles to the desired region. You can use the rlocfind command in MATLAB to choose the desired poles on the locus:

```
[k,poles] = rlocfind(sys)
```

Click on the plot the point where you want the closed-loop pole to be. You may want to select the points indicated in the plot below to satisfy the design criteria.



Note that since the root locus may have more than one branch, when you select a pole, you may want to find out where the other pole (poles) are. Remember they will affect the response too. From the plot above, we see that all the poles selected (all the "+" signs) are at reasonable positions. We can go ahead and use the chosen Kas our proportional controller.

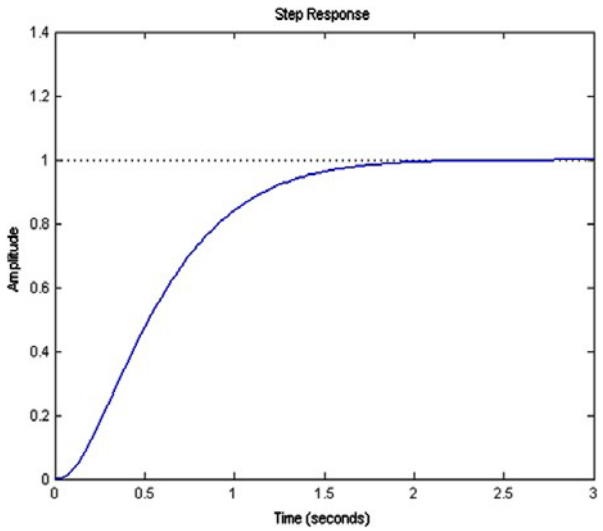
Closed-Loop Response

In order to find the step response, you need to know the closed-loop transfer function. You could compute this using the rules of block diagrams, or let MATLAB do it for you (there is no need to enter a value for Kif the rlocfindcommand was used):

```
K = 350;
sys_cl = feedback(K*sys,1) sys_cl =

                    350 s + 2450
----------------------------------------------------
s^4 + 40 s^3 + 475 s^2 + 1850 s + 2450

Continuous-time transfer function.
```

The two arguments to the function feedback are the numerator and denominator of the open-loop system. You need to include the proportional gain that you have chosen. Unity feedback is assumed.
If you have a non-unity feedback situation, look at the help file for the MATLAB function feedback, which can find the closed-loop transfer function with a gain in the feedback loop.
Check out the step response of your closed-loop system:

```
step(sys_cl)
```



As we expected, this response has an overshoot less than 5% and a rise time less than 1 second.

Using SISOTOOL for Root Locus Design

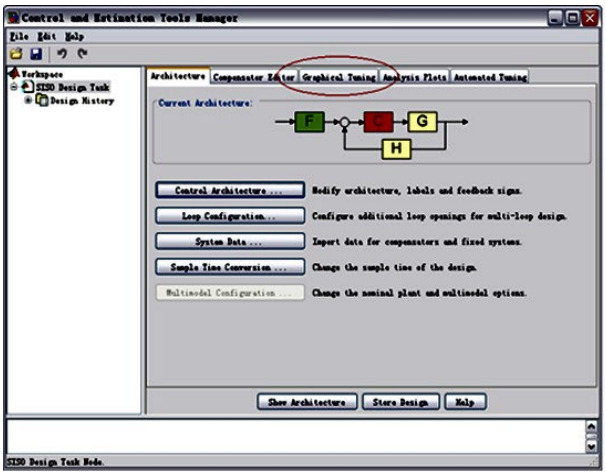
Another way to complete what was done above is to use the interactive MATLAB GUI called sisotool. Using the same model as above, first define the plant, $HH(ss)$.

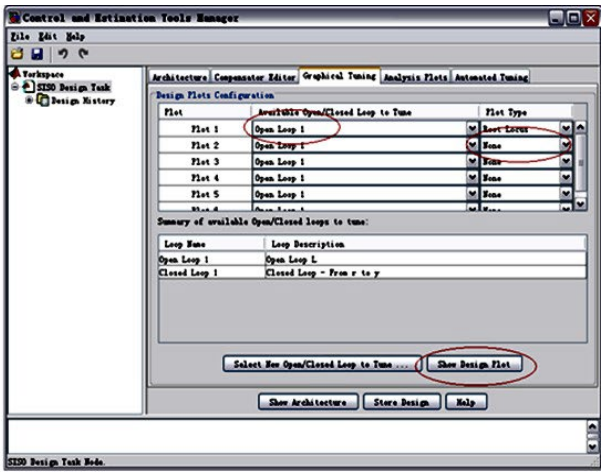
```
s = tf('s');
plant = (s + 7)/(s*(s + 5)*(s + 15)*(s + 20));
```

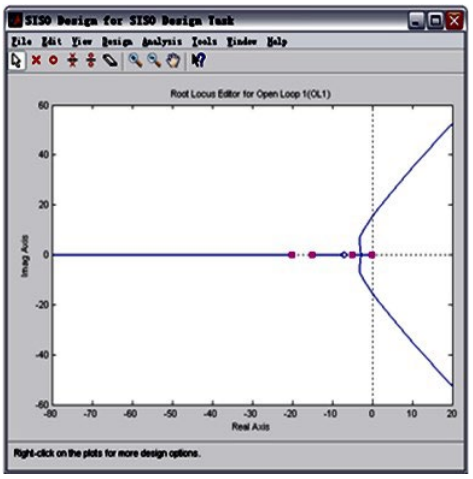
The sisotool function can be used for analysis and design. In this case, we will focus on using the Root Locus as the design method to improve the step response of the plant. To begin, type the following into the MATLAB command window:

```
sisotool(plant)
```

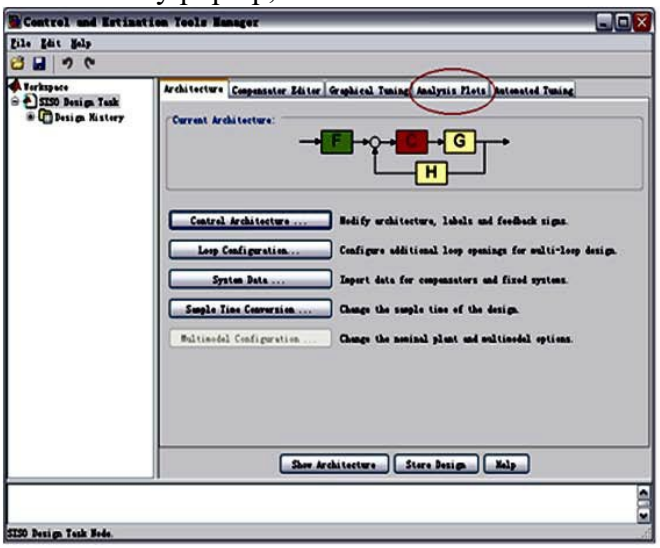
The following window should appear. To start, select the tab labeled **Graphical Tuning**. Within this window, turn off **Plot 2** and make sure **Plot 1** is the Root Locus and verify that **Open Loop 1** is selected. Finally, click the button labeled **Show Design Plot** to bring up the tunable Root Locus plot.

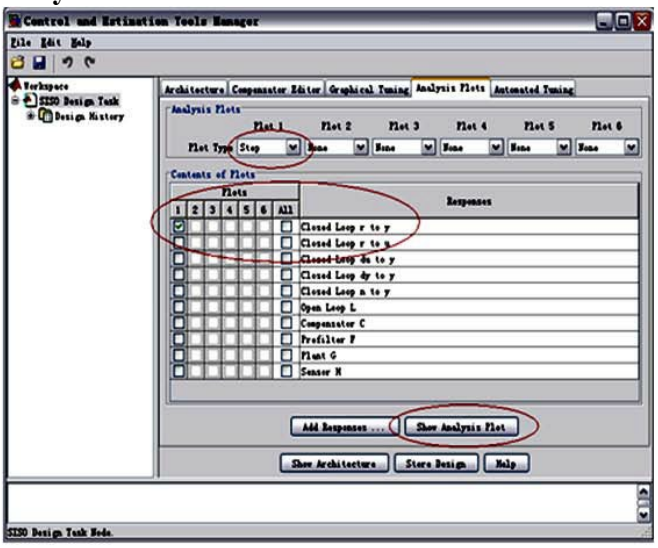


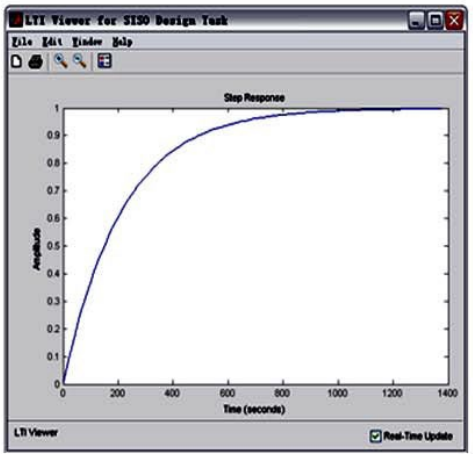




In the same fashion, select the tab labeled **Analysis Plots**. Within this window, for **Plot 1**, select **Step**. In the **Contents of Plots** subwindow, select **Closed Loop r to y** for **Plot 1**. If the window does not automatically pop up, click the button labeled **Show Analysis Plot**.

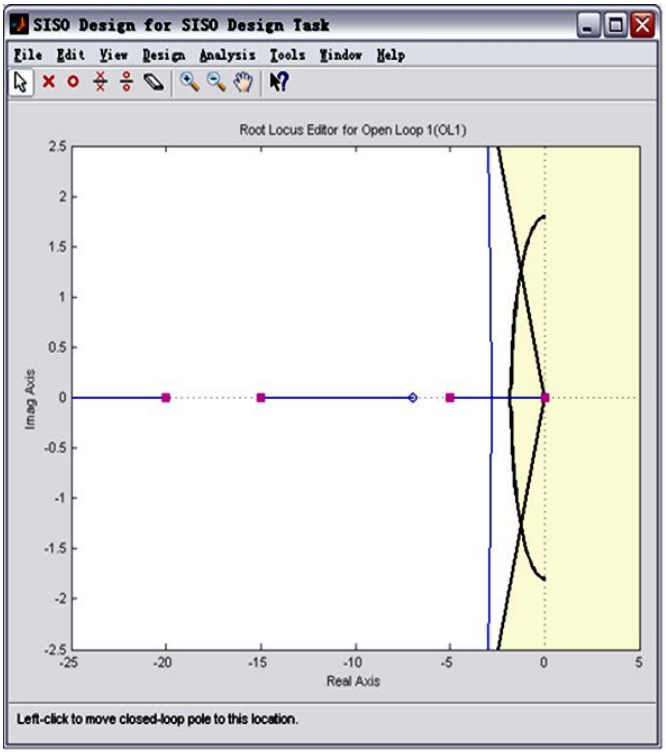


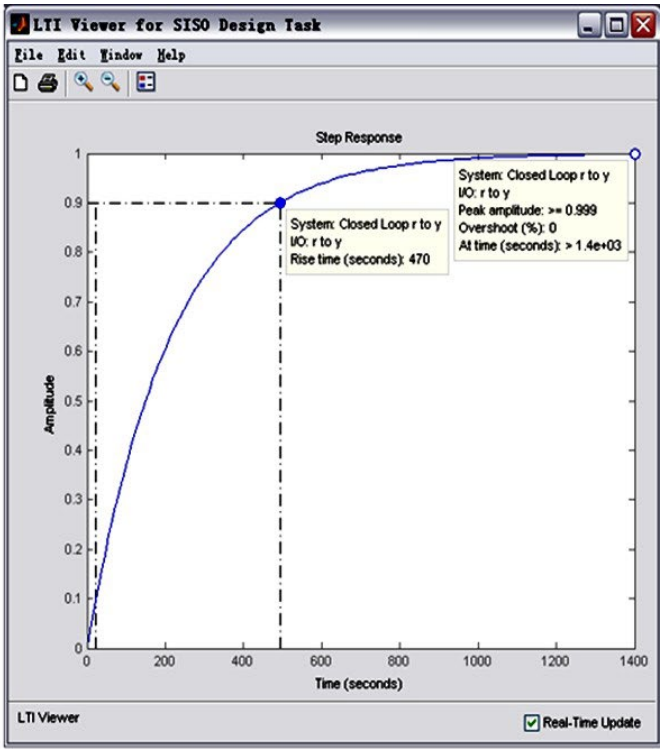




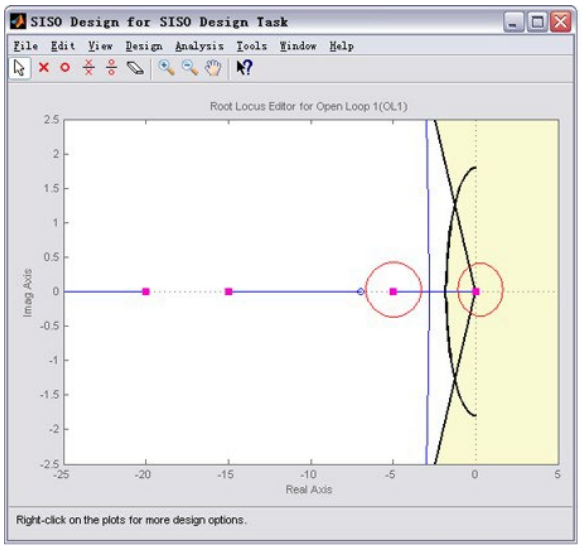
The next thing to do is to add the design requirements to the Root Locus plot. This is done directly on the plot by right-clicking and selecting **Design Requirements**, **New**. Design requirements can be set for the Settling Time, the Percent Overshoot, the Damping Ratio, the Natural Frequency, or a Region Constraint. There is no direct requirement for Rise Time, but the natural frequency can be used for this. Here, we will set the design requirements for the damping ratio and the natural frequency just like was done with sgrid. Recall that the requirements call for $\zeta = 0.7$ and $\omega_n = 1.8$. Set these within the design requirements. On the plot, any area which is still white, is an acceptable region for the poles.

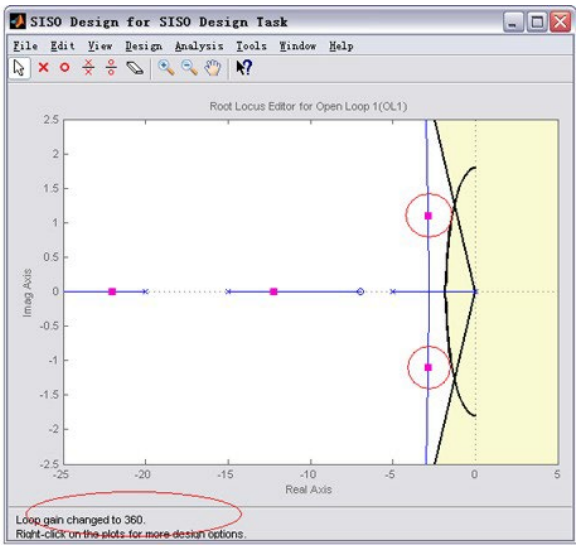
Zoom into the Root Locus by right-clicking on the axis and select **Properties**, then click the label **Limits**. Change the real axis to -25 to 5and the imaginary to -2.5 to 2.5.
Also, we can see the current values of some key parameters in the response. In the Step response, right-click on the plot and go to **Characteristics** and select **Peak Response**. Do the same for the **Rise Time**. There should now be two large dots on the screen indicating the location of these parameters. Click each of these dots to bring up a screen with information. Both plots should appear as shown here:



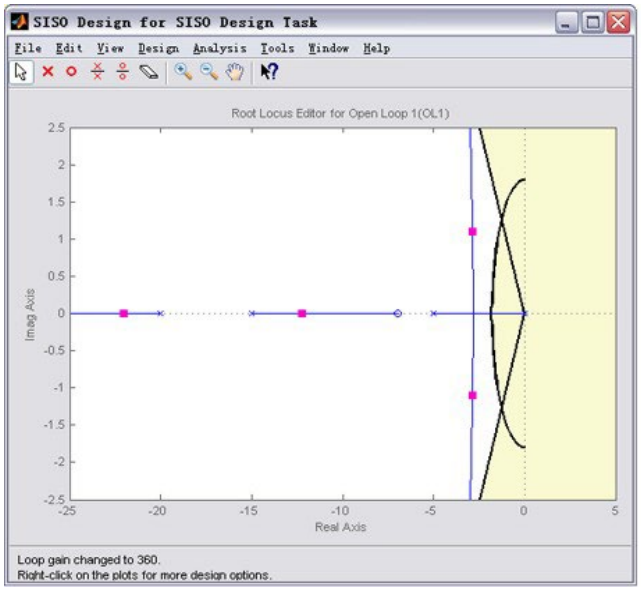


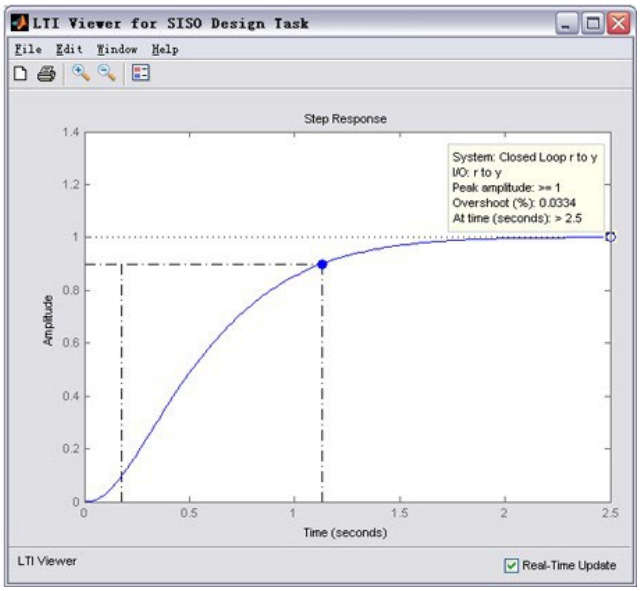
As the characteristics show on the Step response, the overshoot is acceptable, but the rise time is incredibly off.
To fix this, we need to choose a new value for the gain K. Similarly to the rlocfind command, the gain of the controller can be changed directly on the root locus plot. Click and drag the pink box on the origin to the acceptable area where the poles have an imaginary component as shown below.





At the bottom of the plot, it can be seen that the loop gain has been changed to 361. Looking at the Step response, both of the values are acceptable for our requirements.





DC Motor Speed: Root Locus Controller Design

Key MATLAB commands used in this tutorial are: tf, sisotool

Contents:

- *Drawing the open-loop root locus*
- *Finding the loop gain*
- *Adding a lag controller*
- *Finding the loop gain with a lag controller*

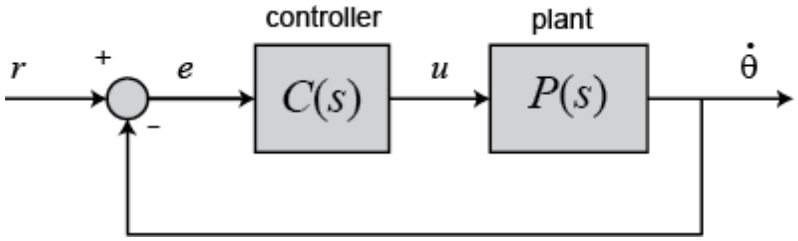
From the main problem, the dynamic equations in the Laplace domain and the open-loop transfer function of the DC Motor are the following.

$$ss(JJss + bb)\theta\theta(ss) = KKKK(ss) \quad (1)$$

$$(LLss + RR)KK(ss) = VV(ss) - KKss\theta\theta(ss) \quad (2)$$

$$PP(ss) = \frac{\theta\theta(ss)}{VV(ss)} = \frac{K}{(JJss + bb)(LLss + RR) + KK^2} \quad \left[\frac{rraarr/ssssss}{VV}\right] \quad (3)$$

The structure of the control system has the form shown in the figure below.



For a 1-rad/sec step reference, the design criteria are the following.

- Settling time less than 2 seconds
- Overshoot less than 5%
- Steady-state error less than 1%

Now let's design a controller using the methods introduced in the Introduction: Root Locus Controller Design part. Create a new m-file and type in the following commands.

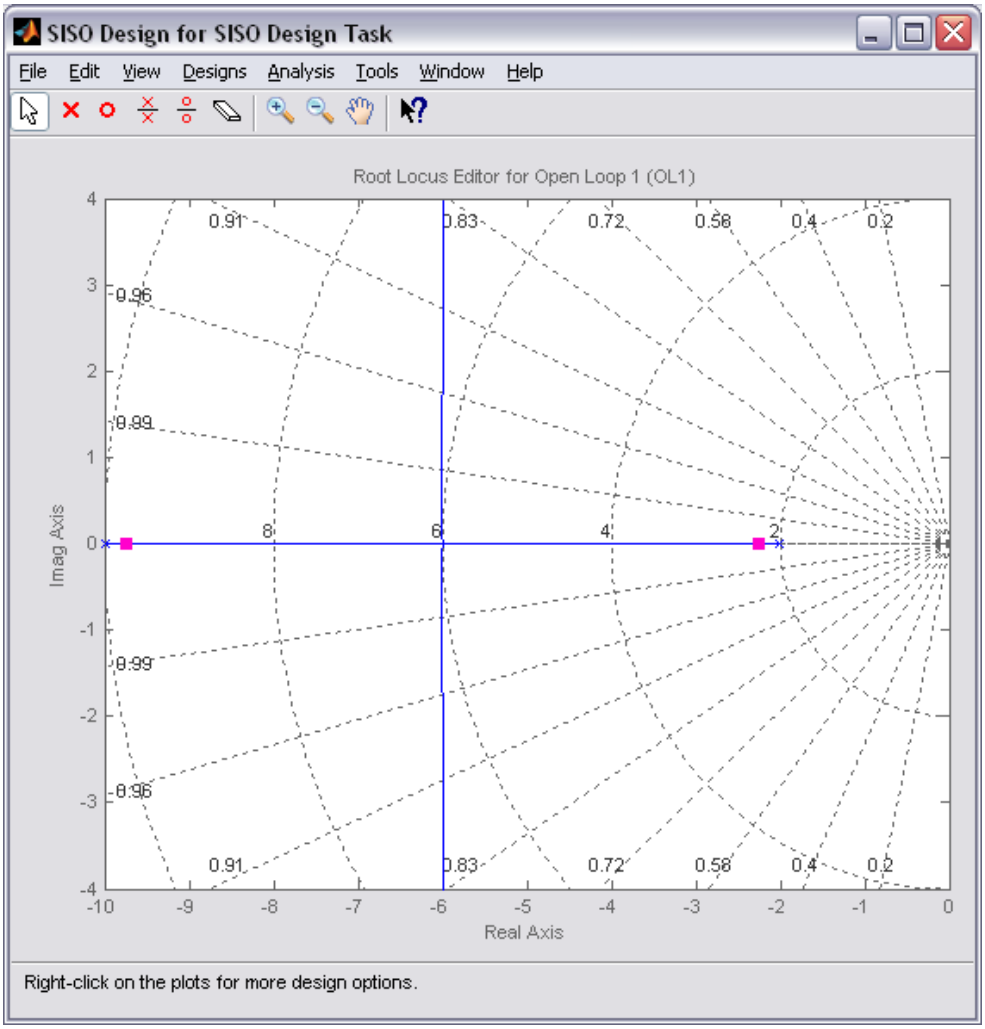
```
J = 0.01;
b = 0.1;
K = 0.01;
R = 1;
L = 0.5;

s = tf('s');
P_motor = K/((J*s+b)*(L*s+R)+K^2);
```

Drawing the open-loop root locus

The main idea of root locus design is to predict the closed-loop response from the root locus plot which depicts possible closed-loop pole locations and is drawn from the open-loop transfer function. Then by adding zeros and/or poles via the controller, the root locus can be modified in order to achieve a desired closed-loop response.

We will use for our design the **SISO Design Tool** graphical user interface. This tool allows the you to graphically tune the controller via the root locus plot. Let's first view the root locus for the uncompenstated plant. This is accomplished by adding the command `sisotool('rlocus', P_motor)` to the end of your m-file and running the file at the command line.

Two windows will initially open, one is the **SISO Design Task** which will open with the root locus of the uncompensated plant, and the other is **Control and Estimation Tool Manager** which allows you to design compensators, analyze plots, etc. Right-click on the root locus plot and click on **Grid**. Your plot will then appear as follows.

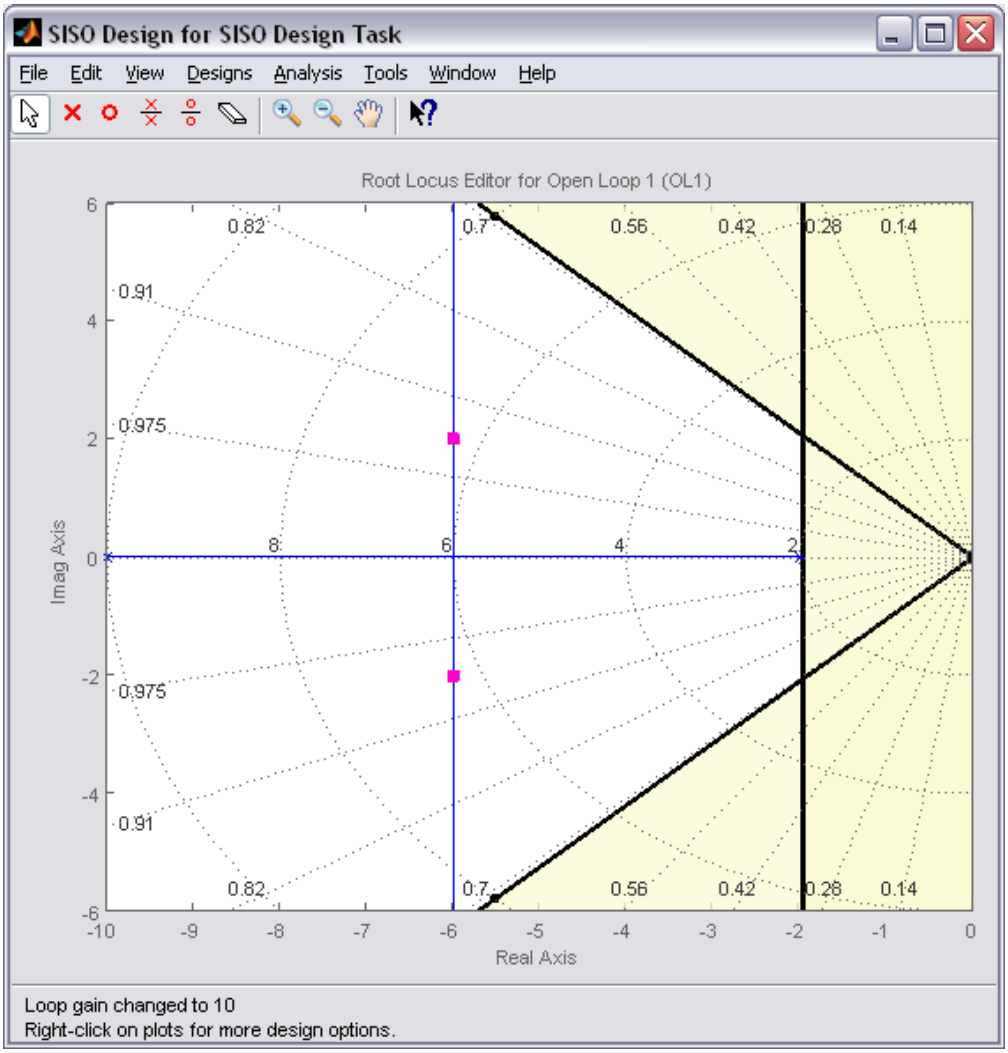

SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Root Locus Editor for Open Loop 1 (OL1)
0.91
0.83
0.72
0.58
0.4
0.2
0.96
0.99
8
6
4
2
Imag Axis
Real Axis
Right-click on the plots for more design options.

Finding the loop gain

Recall that our design requirements specify that the settling time be less than 2 seconds and that the overshoot be less than 5%. The location of the system's closed-loop poles provide information regarding the system's transient response. The **SISO Designt Tool** allows you to specify the region in the complex *s*-plane corresponding to specific design requirements. The provided regions correspond to a canonical second-order system, but in general are a good place to start from even for higher-order systems or systems with zeros.

These desired regions can be added to the root locus plot by right-clicking on the plot and choosing **Design Requirements > New** from the resulting menu. You can add many design requirements including Settling time, Percent overshoot, Damping ratio, Natural frequency, and generic Region constraint.

Adding our settling time and percent overshoot requirements to the root locus plot produces the following figure.

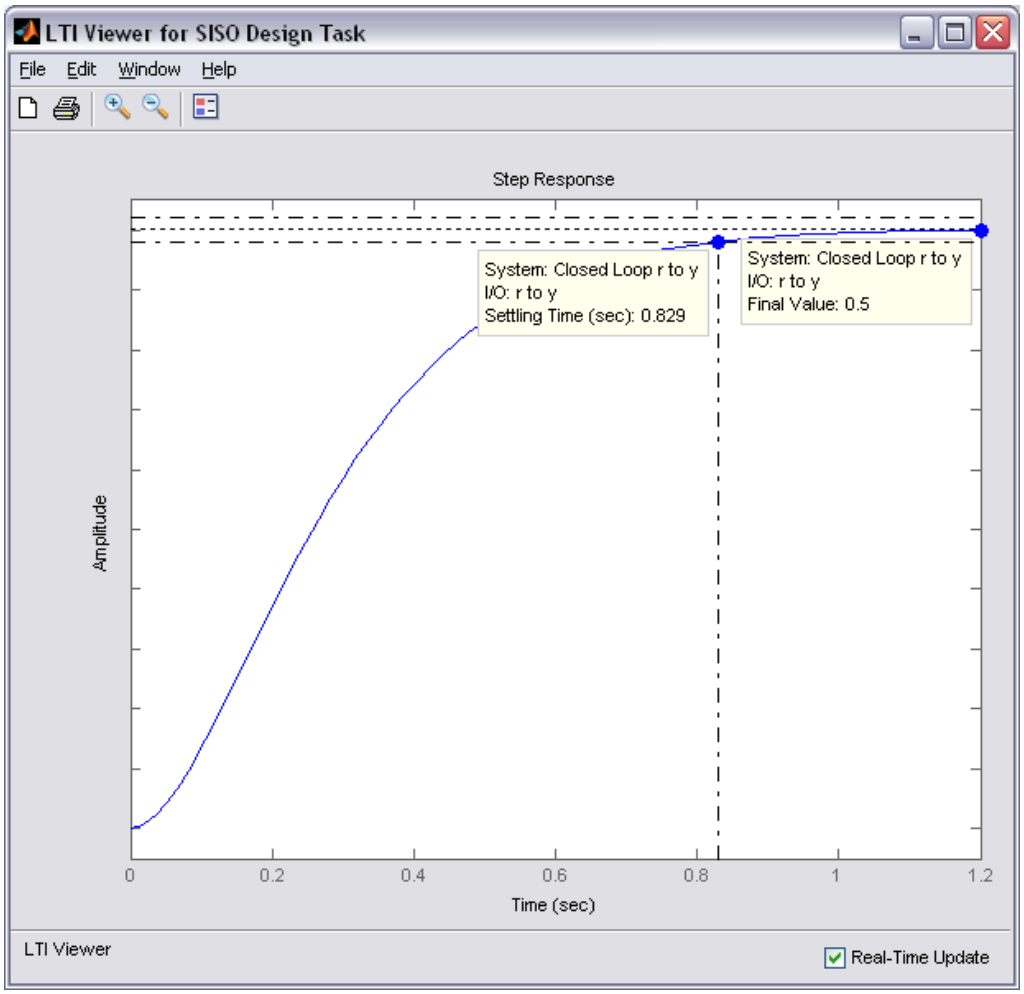


The resulting desired region for the closed-loop poles is shown by the unshaded region of the above figure. More specifically, the two rays centered at the origin represent the overshoot requirement; the smaller the angle these rays make with the negative real-axis, the less overshoot is allowed. The vertical line at s = -2 represents the settling time requirement, where the farther to left the closed-loop poles are located the smaller the settling time is. From examination of the above figure, there are values of the loop gain that will place both closed-loop poles in the desired region. This can be seen from the fact that the two branches of the root locus are symmetric and pass through the unshaded region. Furthermore, since the closed-loop system has two poles with no zeros, placing the closed-loop poles in the shown region will guarantee satisfaction of our transient response requirements.

You can select a specific pair of closed-loop poles from the resulting figure in order to determine the corresponding loop gain that places the poles at that location. For our system, let's choose to place the closed-loop poles so that they are located on the vertical branches of the root-locus between the real axis and the damping requirement. The pink boxes on the root locus indicate the location of the closed-loop

poles for the current loop gain. Clicking on the pink boxes and dragging them along the root locus to the desired location automatically modifies the controller to place the closed-loop poles at the indicated position. Let us drag a closed-loop pole to a location near -6 + 2i. The pole location will be indicated at the bottom of the window along with the corresponding damping ratio and natural frequency. Releasing the mouse button will further show at the bottom of the window the corresponding loop gain, which in this case is approximately 10.

We can also generate the closed-loop step response for the system with this new gain. From the **Control and Estimation Tool Manager**, click on the **Analysis Plots** tab and under **Plot1**, choose Step, a blank window titled **LTI Viewer for SISO Design Task** will appear. Right-click on this window and then from **Systems** menu choose the first item which is **Closed Loop r to y (blue)**. The closed-loop step response will then appear in the figure. You can also identify some characteristics of the step response. Specifically, right-click on the figure and under **Characteristics** choose **Settling Time**. Then repeat for **Steady State**. Your figure will appear as shown below.



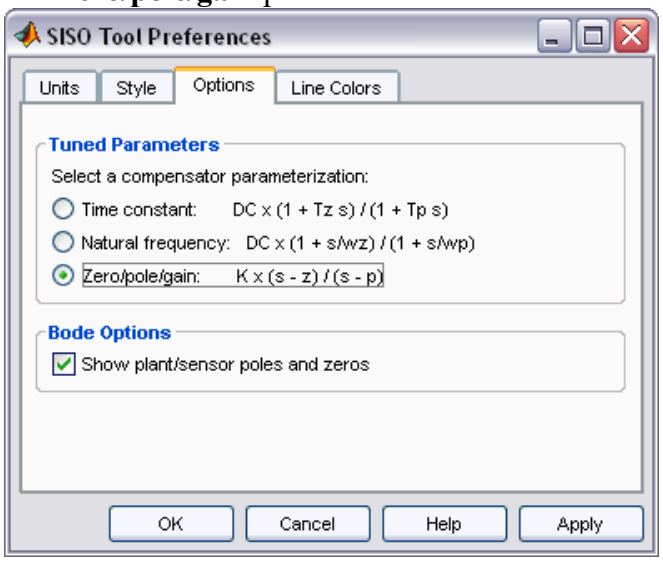
From inspection of the above, one can see that there is no overshoot and the settling time is less than one second, therefore, the overshoot and settling time requirements are satisfied. However, we can also observe that the steady-state error is approximately 50%. If we increase the loop gain to reduce the steady-state error, the overshoot will become too large. You can see this for yourself by graphically moving the closed-loop poles vertically upward along the root locus, this corresponds to increasing the loop gain. The step response plot will change automatically to reflect the modified loop gain. We will attempt to add a lag controller to reduce the steady-state error requirement while still satisfying the transient requirements.

Adding a lag controller

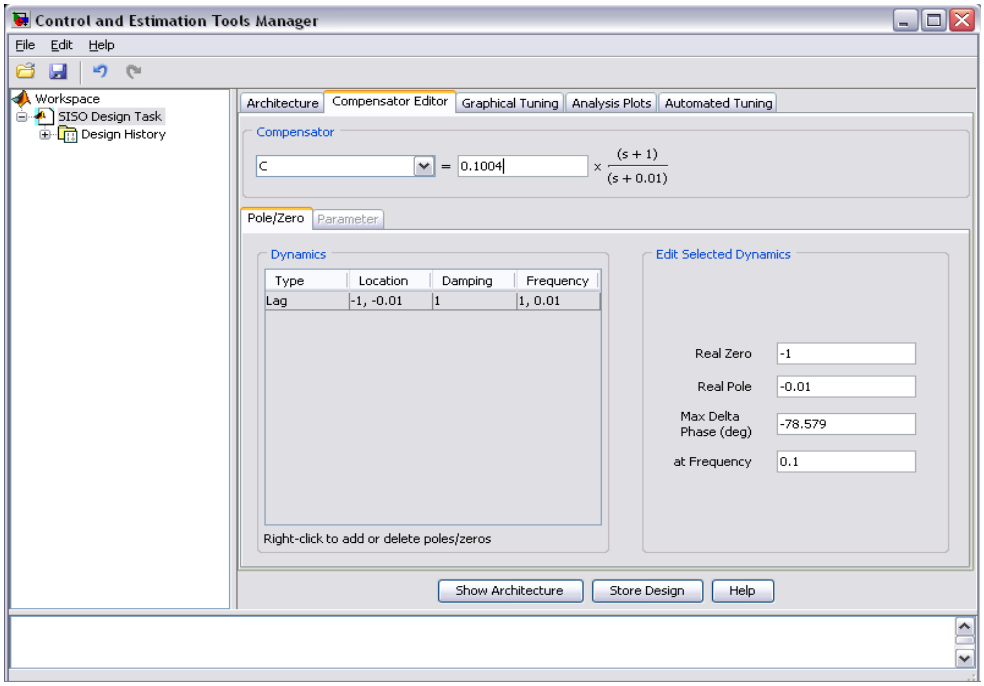
In the above we saw that the overshoot and settling time criteria were met with the proportional controller, but the steady-state error requirement was not. A **lag compensator** is one type of controller known to be able to reduce steady-state error. However, we must be careful in our design to not increase the settling time too much. Let's first try adding a lag compensator of the form given below.

$$CC(ss) = \frac{(ss+1)}{(ss+0.01)} \quad (4)$$

We can use the **SISO Design Tool** to design our lag compensator. To make the **SISO Design Tool** have a compensator parameterization corresponding to the one shown above, click on the **Edit** menu at the top of the **Control and Estimation Tools Manager** window and choose **SISO Tool Preferences**. Then From the **Options** tab, select a **Zero/pole/gain** parameterization as shown below.



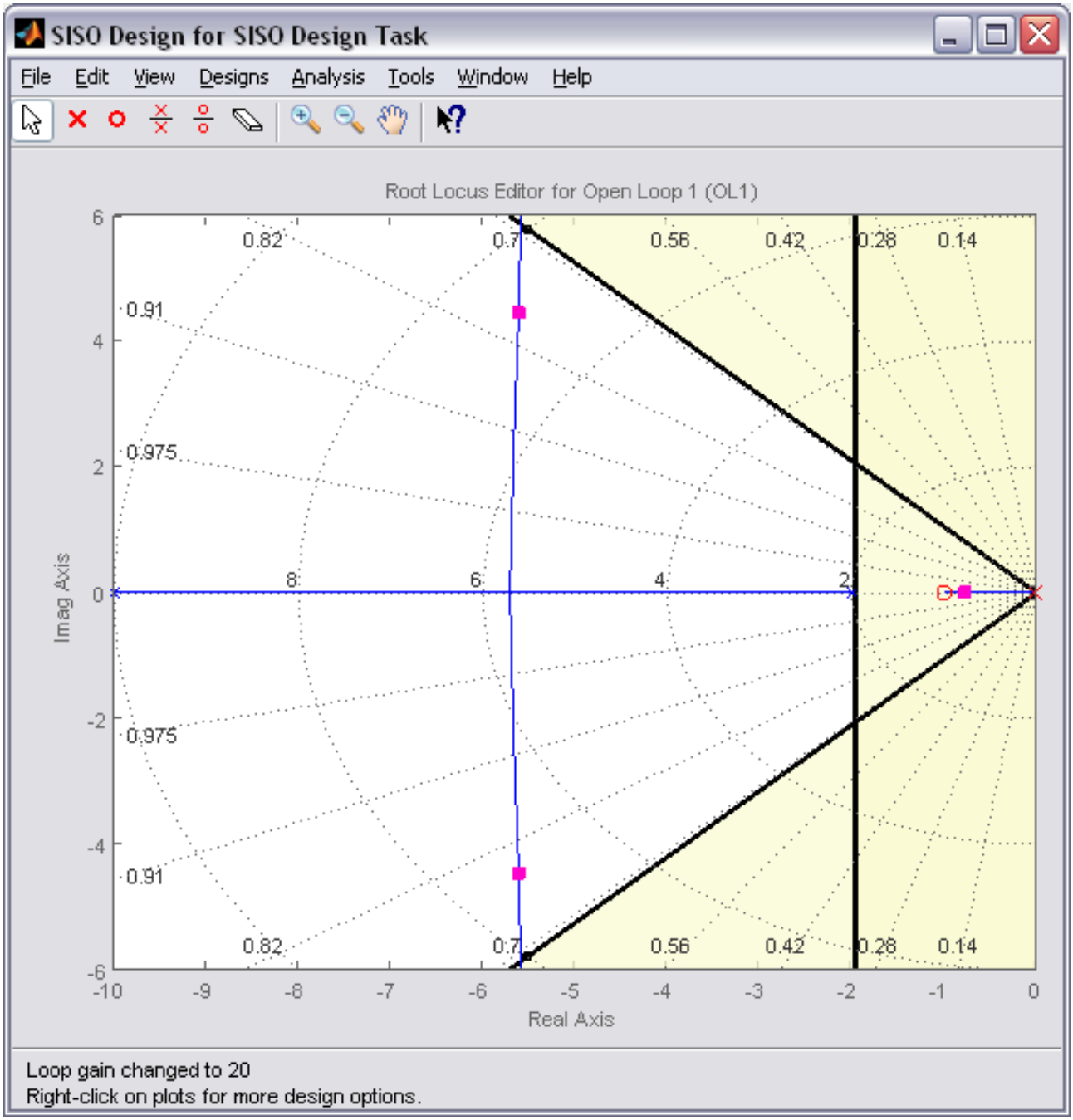
You can then add the lag compensator from under the **Compensator Editor** tab of the **Control and Estimation Tools Manager** window. Specifically, right-click in the **Dynamics** section of the window and select **Add Pole/Zero > Lag**. Then enter the **Real Zero** and **Real Pole** locations as shown in the following figure.



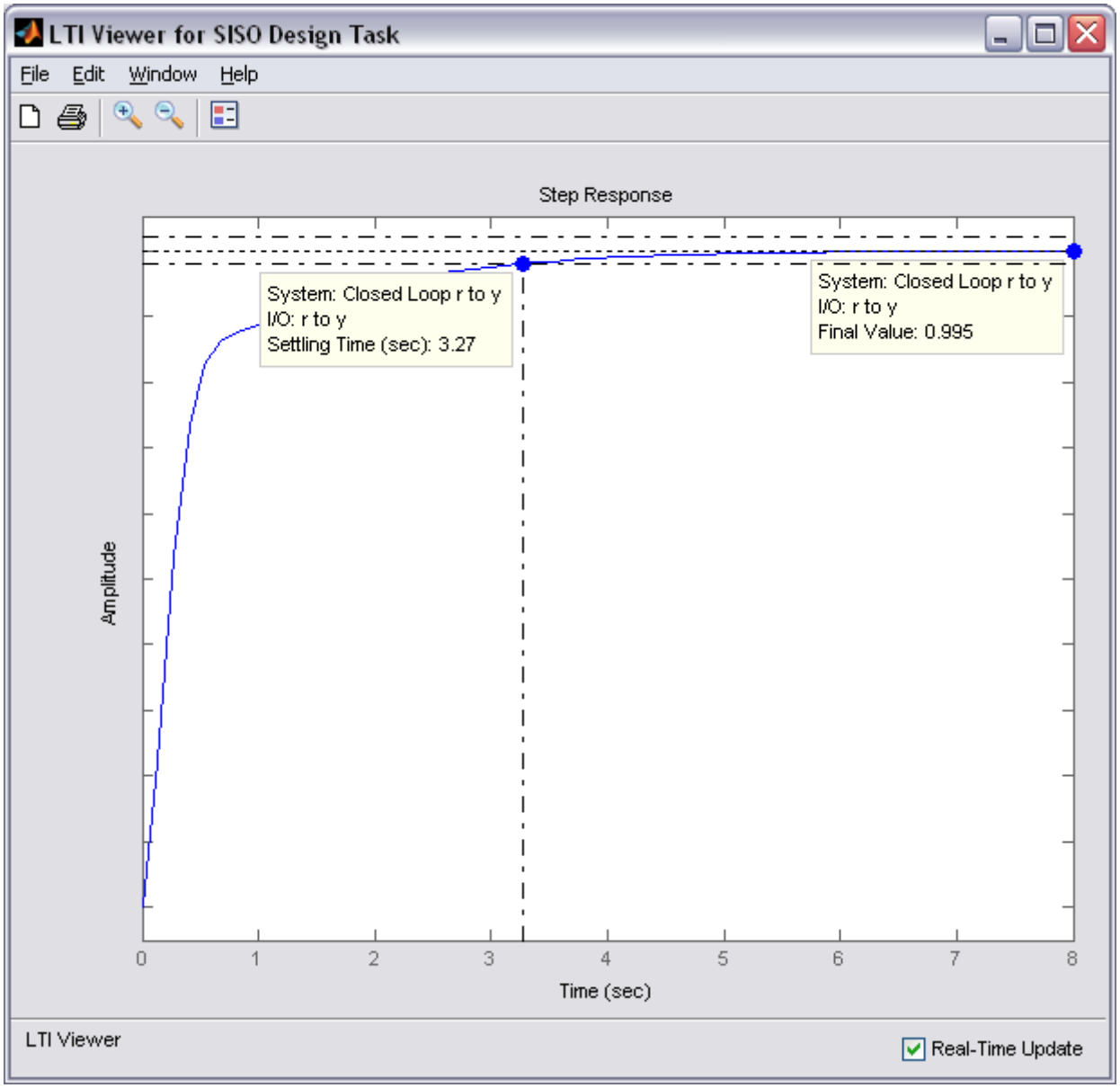
Note that the phase lag contributed by the compensator and the frequency where it is located are updated to match the pole and zero locations chosen.

Finding the loop gain with a lag controller

Notice how the root locus has changed to reflect the addition of the pole and zero from the lag compensator as shown in the figure below. We can again choose closed-loop pole locations to attempt to achieve our desired transient requirements. Let's attempt to place two of the closed-loop poles in our desired region near the boundary of the overshoot requirement. For example, a loop gain of approximately 20 will place the poles at the positions shown in the figure below.

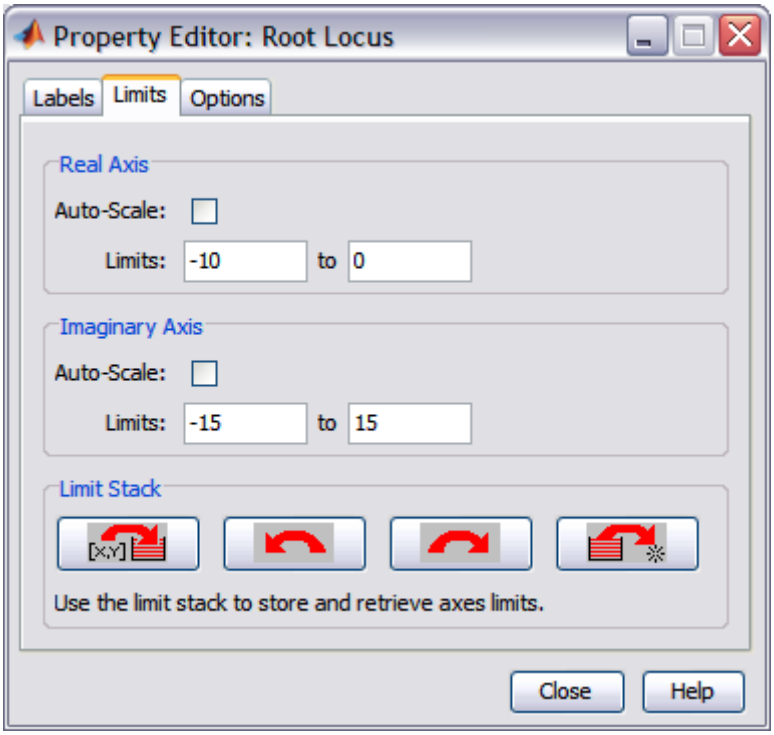


The corresponding closed-loop step response will then update automatically to match the figure shown below.

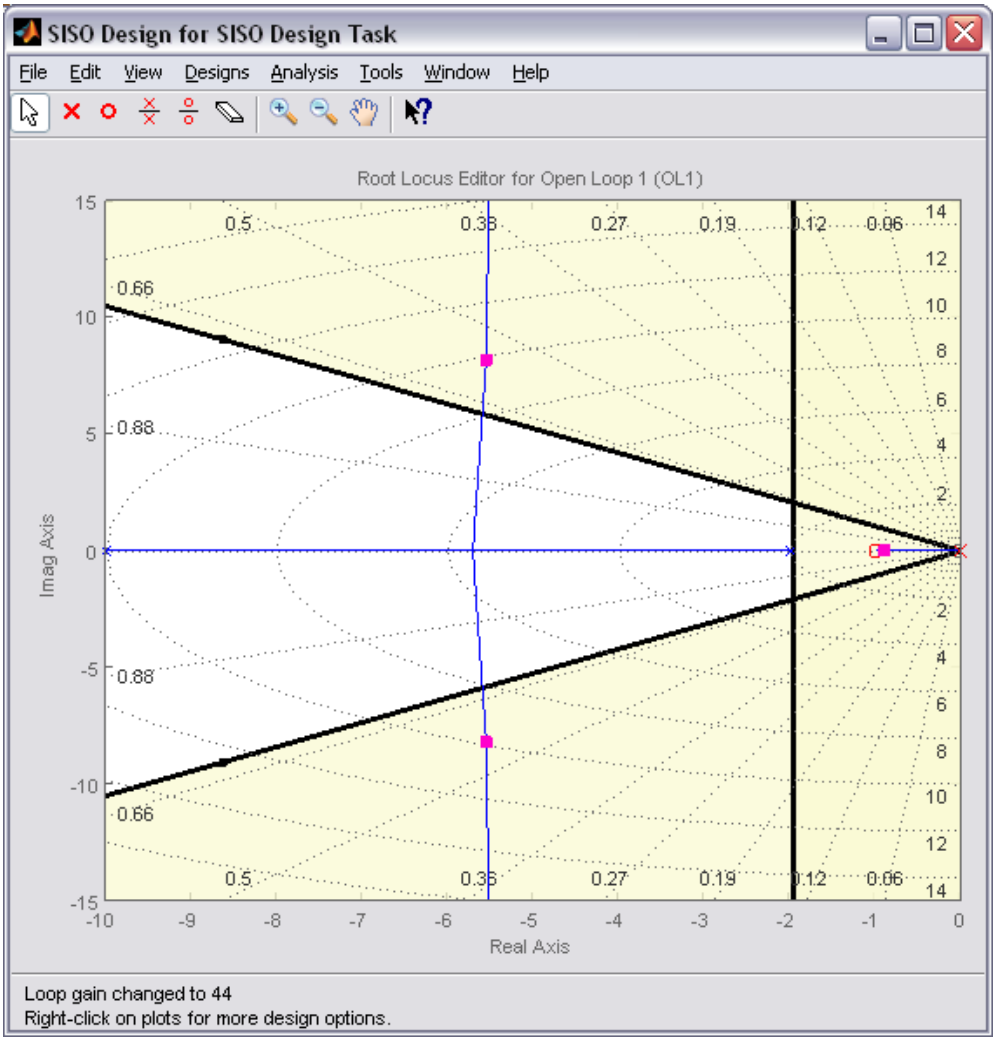


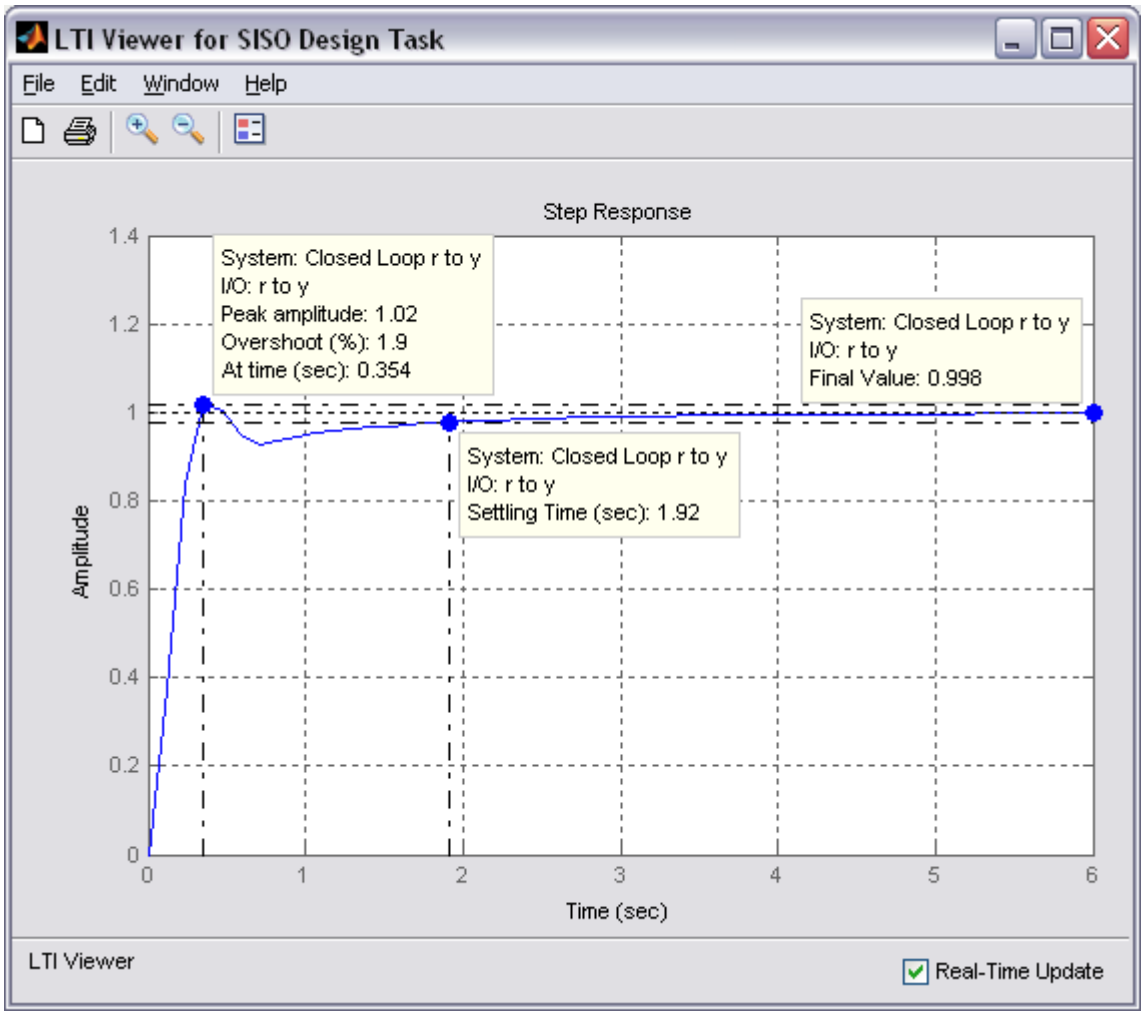
As you can see, the response is not quite satisfactory even though two of the closed-loop poles were placed in the desired region. The reason for this is because the closed-loop system no longer has the form of a canonical second-order system. Specifically, there is a third pole on the real axis indicated in the root locus plot above that is outside of the desired region. The fact that this third pole is to the right of the two conjugate poles placed above means that it will slow the system response down, that is why the settling time requirement is no longer met. Additionally, the overshoot requirement is met easily even though the two conjugate poles are near the edge of the allowed region. This is due again to the third pole which is well damped and tends to dominate the response because it is "slower" than the other poles. What this means is that we can further increase the loop gain such that the conjugate poles move beyond the diagonal lines while still meeting the overshoot requirement.

You can now return to the root locus plot and graphically move the conjugate poles farther away from the real axis; this corresponds to increasing the loop gain. Before you do this, however, you likely need to change the limits on the imaginary axis so that you can move the poles a sufficient distance. In order to change these limits, double-click on the root locus plot to open the **Property Editor**, then click on the **Limits** tab and change the imaginary axis limits to [-15,15] as shown below.



Experiment with different gains (closed-loop pole locations) until you achieve the desired response. Below is the root locus with a loop gain of 44 and the corresponding closed-loop step response.





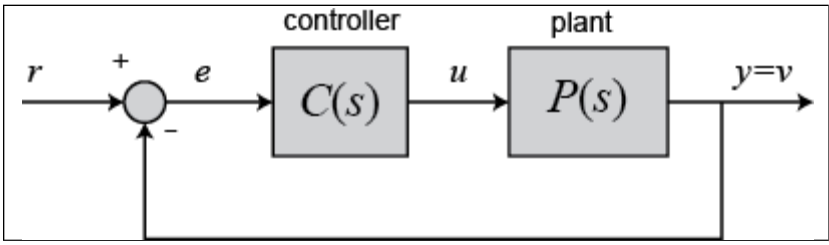
Now the settling time is less than 2 seconds and the steady-state error and overshoot requirements are still met. As you can see, the root locus design process requires some trial and error. The **SISO Design Tool** is very helpful in this process. Using the **SISO Design Tool**, it is very easy to tune your controller and immediately see the effect on the root locus and various analysis plots, like the closed-loop step response. If we had not been able to get a satisfactory response by tuning the loop gain, we could have tried moving the pole and zero of the lag compensator or we could have tried a different type of dynamic compensator (additional poles and/or zeros).

Work: Design root locus controller for cruise control with the following specifications

For this example, let's assume that the parameters of the system are

(m)	vehicle mass	1000 kg
(b)	damping coefficient	50 N.s/m
(r)	reference speed	10 m/s

and the block diagram of a typical unity feedback system is shown below.



Performance specifications

- Rise time < 5 sec
- Overshoot < 10%
- Steady-state error < 2%

Experiment No. 08

Name of the experiment: Frequency Domain Methods for Controller Design of Physical systems Key

MATLAB commands: tf, bode, margin, step, feedback

Objective:

(i) To study of the frequency domain methods for controller design of physical system
(ii) Drawing the original Bode plot
(iii) Adding proportional gain
(iv) Plotting the closed-loop response
(v) Adding a lag compensator

Introduction: Frequency Domain Methods for Controller Design

The frequency response method of controller design has certain advantages, especially in real-life situations such as modeling transfer functions from physical data. In this tutorial, we will see how we can use the open-loop frequency response of a system to predict its behavior in closed-loop.

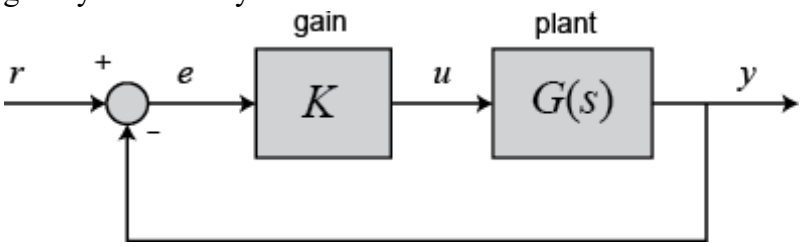
Key MATLAB commands used in this tutorial are: bode, nyquist, margin, lsim, step, feedback, sisotool

Contents

- Gain and Phase Margin
- Nyquist Diagram
- The Cauchy Criterion
- Closed-Loop Performance from Bode Plots
- Closed-Loop Stability from the Nyquist Diagram

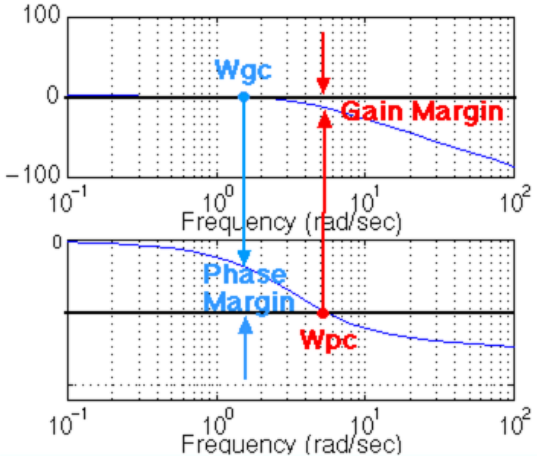
Gain and Phase Margin

Consider the following unity feedback system:



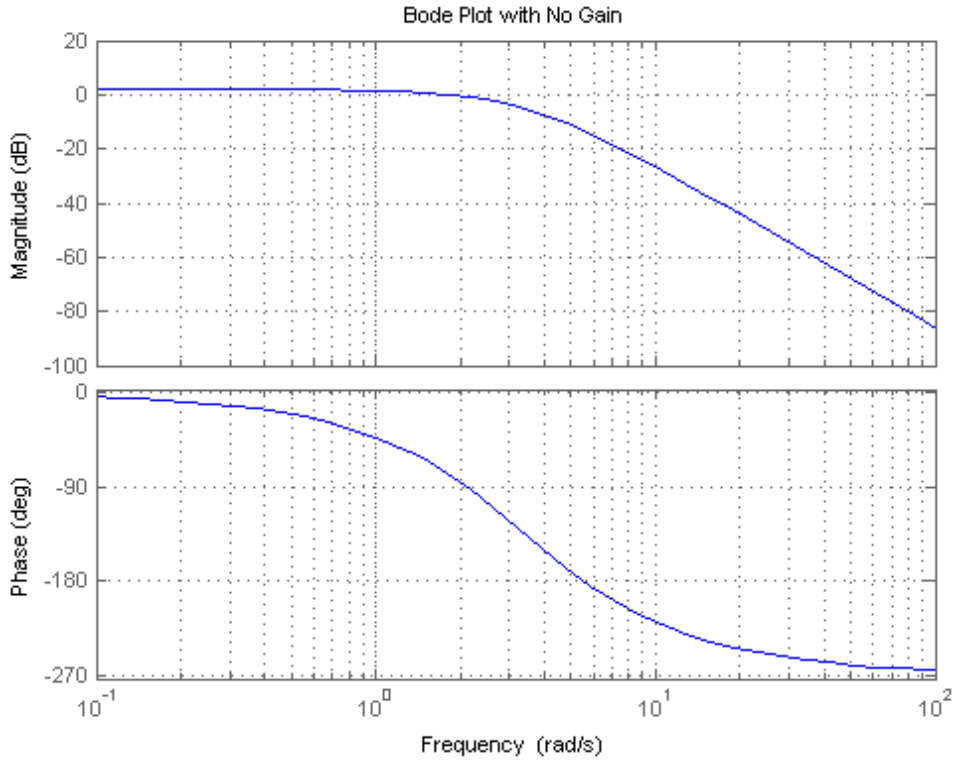
where K is a variable (constant) gain and $G(s)$ is the plant under consideration. The **gain margin** is defined as the change in open-loop gain required to make the system unstable. Systems with greater gain margins can withstand greater changes in system parameters before becoming unstable in closed-loop.
The **phase margin** is defined as the change in open-loop phase shift required to make a closed-loop system unstable.
The phase margin also measures the system's tolerance to time delay. If there is a time delay greater than $180/W_{pc}$ in the loop (where W_{pc} is the frequency where the phase shift is 180 deg), the system will become unstable in closed-loop. The time delay, τ_d can be thought of as an extra block in the forward path of the block diagram that adds phase to the system but has no effect on the gain. That is, a time delay can be represented as a block with magnitude of 1 and phase $\omega\tau_d$ (in radians/second).

The phase margin is the difference in phase between the phase curve and -180 degrees at the point corresponding to the frequency that gives us a gain of 0 dB (the gain crossover frequency, W_{gc}). Likewise, the gain margin is the difference between the magnitude curve and 0 dB at the point corresponding to the frequency that gives us a phase of -180 degrees (the phase crossover frequency, W_{pc}).



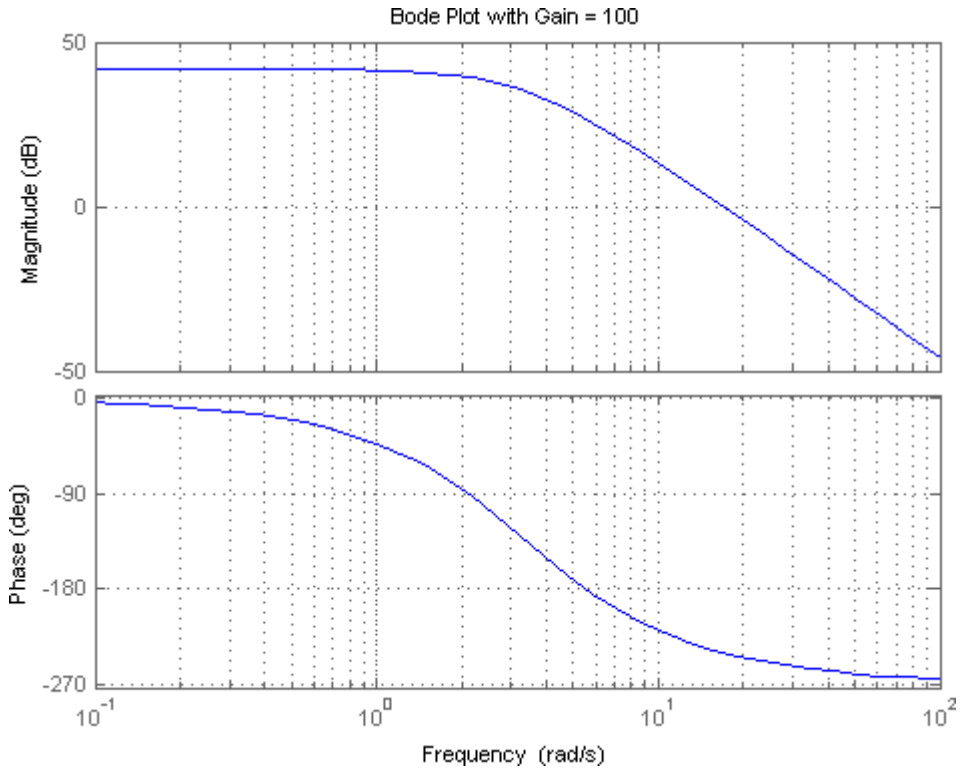
One nice thing about the phase margin is that you don't need to replot the Bode in order to find the new phase margin when changing the gains. If you recall, adding gain only shifts the magnitude plot up. This is equivalent to changing the y-axis on the magnitude plot. Finding the phase margin is simply a matter of finding the new cross-over frequency and reading off the phase margin. For example, suppose you entered the command bode(sys). You will get the following bode plot:

```
s = tf('s');
sys = 50/(s^3 + 9*s^2 + 30*s +40); bode(sys)
grid on
title('Bode Plot with No Gain')
```



You should see that the phase margin is about 100 degrees. Now suppose you added a gain of 100, by entering the command bode(100*sys). You should get the following plot:

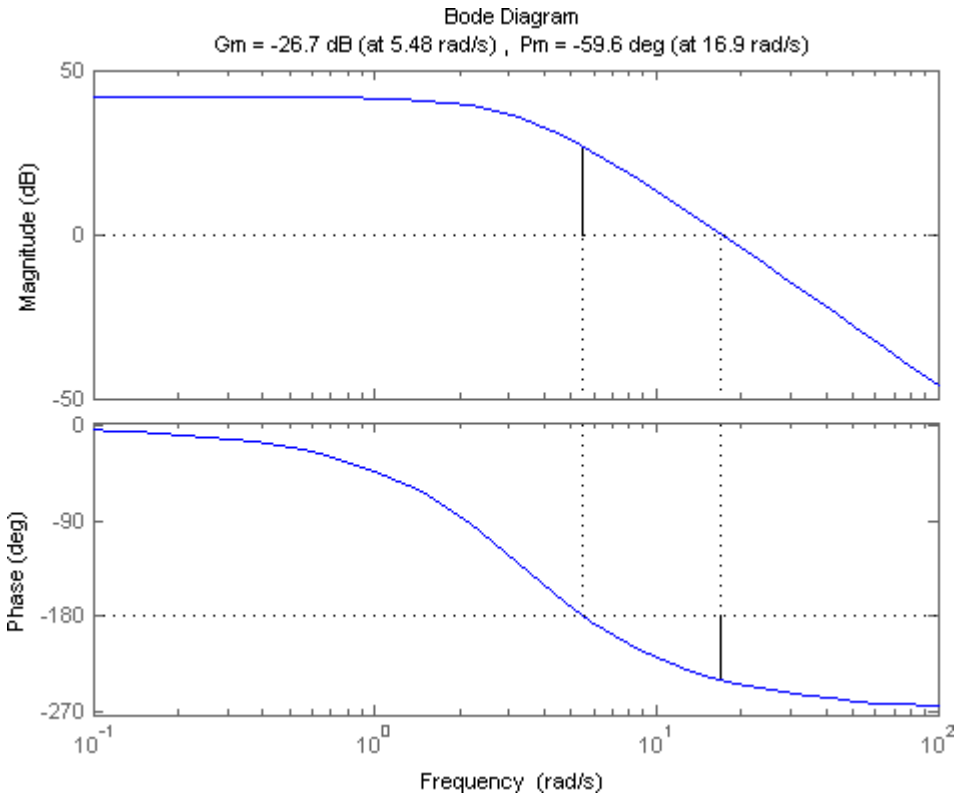
```
bode(100*sys) grid on
title('Bode Plot with Gain = 100')
```



As you can see the phase plot is exactly the same as before, and the magnitude plot is shifted up by 40 dB (gain of 100). The phase margin is now about -60 degrees. This same result could be achieved if the y-axis of the magnitude plot was shifted down 40 dB. Try this, look at the first Bode plot, find where the curve crosses the -40 dB line, and read off the phase margin. It should be about 90 degrees, the same as the second Bode plot.

We can have MATLAB calculate and display the gain and phase margins using the margin(sys) command. This command returns the gain and phase margins, the gain and phase cross over frequencies, and a graphical representation of these on the Bode plot. Let's check it out:

margin(100*sys)



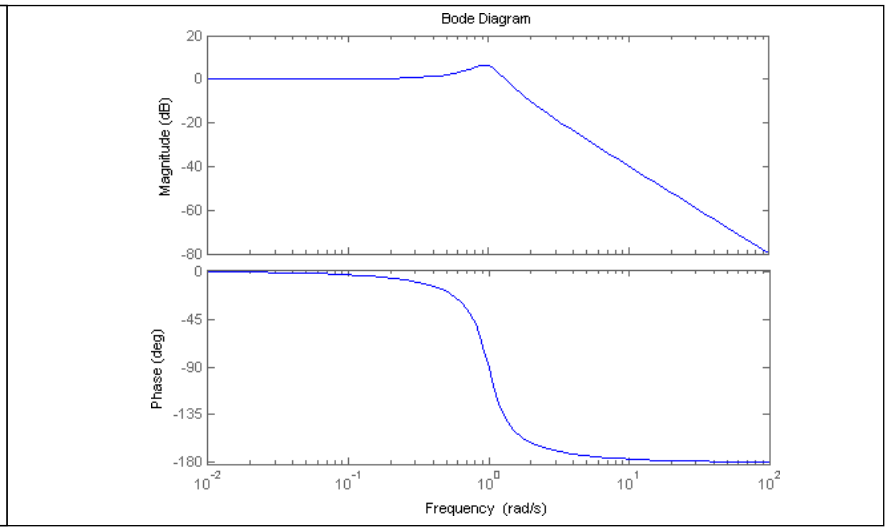
Bandwidth Frequency

The bandwidth frequency is defined as the frequency at which the closed-loop magnitude response is equal to -3 dB. However, when we design via frequency response, we are interested in predicting the closed-loop behavior from the open-loop response. Therefore, we will use a second-order system approximation and say that the bandwidth frequency equals the frequency at which the open-loop magnitude response is between -6 and -7.5 dB, assuming the open-loop phase response is between -135 deg and -225 deg. *For a complete derivation of this approximation, see any textbook.*

In order to illustrate the importance of the bandwidth frequency, we will show how the output changes with different input frequencies. We will find that sinusoidal inputs with frequency less than Wbw (the bandwidth frequency) are tracked "reasonably well" by the system. Sinusoidal inputs with frequency greater than Wbw are attenuated (in magnitude) by a factor of 0.707 or greater (and are also shifted in phase).

Let's say we have the following closed-loop transfer function representing a system: $GG(ss) = \frac{1}{ss^2+0.5ss+1}$ (1)

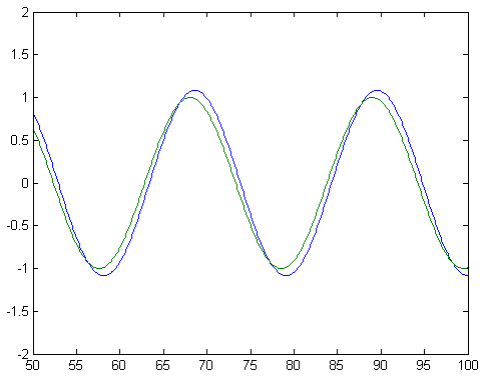
```
sys = 1/(s^2 + 0.5*s +
1);
bode(sys)
```



Since this is the closed-loop transfer function, our bandwidth frequency will be the frequency corresponding to a gain of -3 dB. Looking at the plot, we find that it is approximately 1.4 rad/s. We can also read off the plot that for an input frequency of 0.3 radians, the output sinusoid should have a magnitude about one and the phase should be shifted by perhaps a few degrees (behind the input). For an input frequency of 3 rad/sec, the output magnitude should be about -20 dB (or 1/10 as large as the input) and the phase should be nearly -180 (almost exactly out-of-phase). We can use the lsim command to simulate the response of the system to sinusoidal inputs.

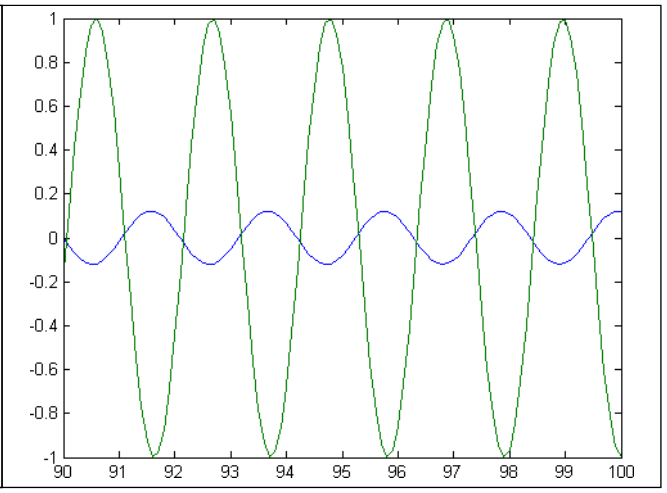
First, consider a sinusoidal input with a **frequency lower than Wbw**. We must also keep in mind that we want to view the steady state response. Therefore, we will modify the axes in order to see the steady state response clearly (ignoring the transient response).

```
sys = 1/(s^2 + 0.5*s + 1);
w = 0.3;
t = 0:0.1:100;
u = sin(w*t);
[y,t] = lsim(sys,u,t);
plot(t,y,t,u)
axis([50 100 -2 2])
```



Note that the output (blue) tracks the input (green) fairly well; it is perhaps a few degrees behind the input as expected. However, if we set the frequency of the input **higher than the bandwidth frequency** for the system, we get a very distorted response (with respect to the input):

```
sys = 1/(s^2 + 0.5*s + 1);
w = 3;
t = 0:0.1:100;
u = sin(w*t);
[y,t] = lsim(sys,u,t);
plot(t,y,t,u)
axis([90 100 -1 1])
```

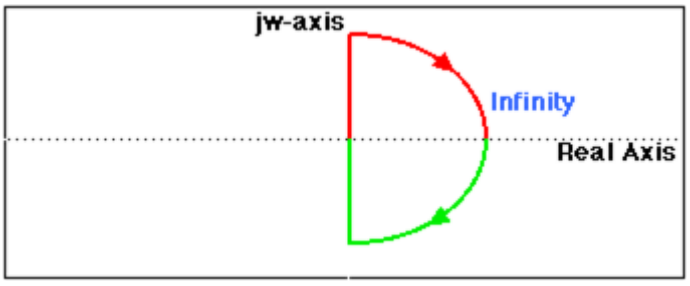


Again, note that the magnitude is about 1/10 that of the input, as predicted, and that it is almost exactly out of phase (180 degrees behind) the input. Feel free to experiment and view the response for several different frequencies ω, and see if they match the Bode plot.

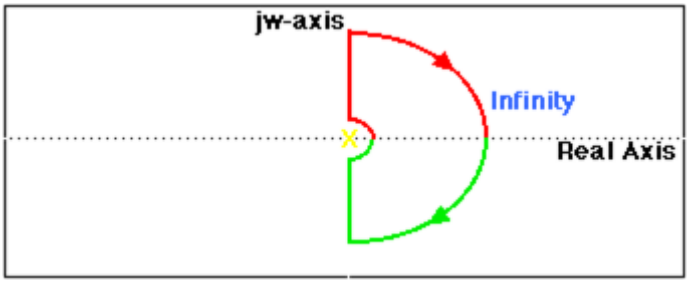
Nyquist Diagram

The Nyquist plot allows us to predict the stability and performance of a closed-loop system by observing its open-loop behavior. The Nyquist criterion can be used for design purposes regardless of open-loop stability (remember that the Bode design methods assume that the system is stable in open-loop). Therefore, we use this criterion to determine closed-loop stability when the Bode plots display confusing information.

Note: The MATLAB nyquist command does not provide an adequate representation for systems that have open-loop poles in the jw-axis. Therefore, we suggest that you copy the nyquist1.m file as a new m-file. This m-file creates more accurate Nyquist plots, since it correctly deals with poles and zeros on the jw-axis.
The Nyquist diagram is basically a plot of $G(j\omega)$ where $G(s)$ is the open-loop transfer function and ω is a vector of frequencies which encloses the entire right-half plane. In drawing the Nyquist diagram, both positive and negative frequencies (from zero to infinity) are taken into account. We will represent positive frequencies in red and negative frequencies in green. The frequency vector used in plotting the Nyquist diagram usually looks like this (if you can imagine the plot stretching out to infinity):



However, if we have open-loop poles or zeros on the jw axis, $G(s)$ will not be defined at those points, and we must loop around them when we are plotting the contour. Such a contour would look as follows:



Please note that the contour loops around the pole on the jw axis. As we mentioned before, the MATLAB nyquist command does not take poles or zeros on the jw axis into account and therefore produces an incorrect plot. To correct this, please download and use nyquist1.m. If we have a pole on the jw axis, we have to use nyquist1. If there are no poles or zeros on the jw-axis, or if we have pole-zero cancellation, we can use either the nyquistcommand or nyquist1.m.

The Cauchy Criterion

The Cauchy criterion (from complex analysis) states that when taking a closed contour in the complex plane, and mapping it through a complex function $G(s)$, the number of times that the plot of $G(s)$ encircles the origin is equal to the number of zeros of $G(s)$ enclosed by the frequency contour minus the number of poles of $G(s)$ enclosed by the frequency contour. Encirclements of the origin are counted as positive if they are in the same direction as the original closed contour or negative if they are in the opposite direction.

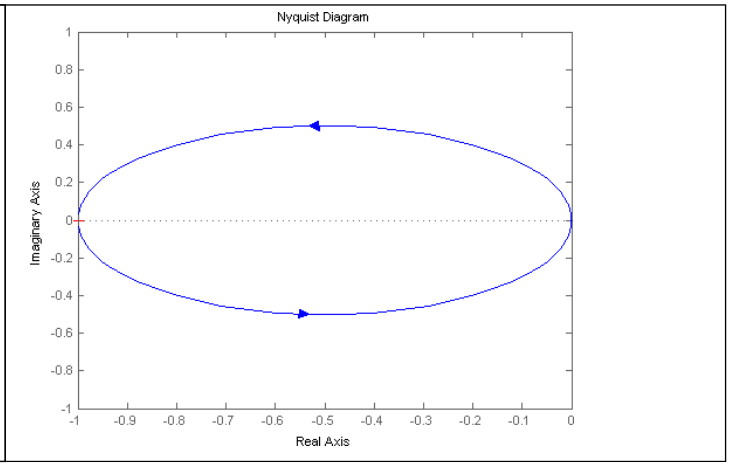
When studying feedback controls, we are not as interested in $G(s)$ as in the closed-loop transfer function:

$$\frac{GG(ss)}{1+GG(ss)} \quad (2)$$

If $1+G(s)$ encircles the origin, then $G(s)$ will enclose the point -1. Since we are interested in the closed-loop stability, we want to know if there are any closed-loop poles (zeros of $1+G(s)$) in the right-half plane.
Therefore, the behavior of the Nyquist diagram around the -1 point in the real axis is very important; however, the axis on the standard nyquistdiagram might make it hard to see what's happening around this point. To correct this, you can add the **lnyquist.m** function to your files. The **lnyquist.m** command plots the Nyquist diagram using a logarithmic scale and preserves the characteristics of the -1 point.
To view a simple Nyquist plot using MATLAB, we will define the following transfer function and view the Nyquist plot:

$$\frac{0.5}{ss-0.5} \quad (3)$$

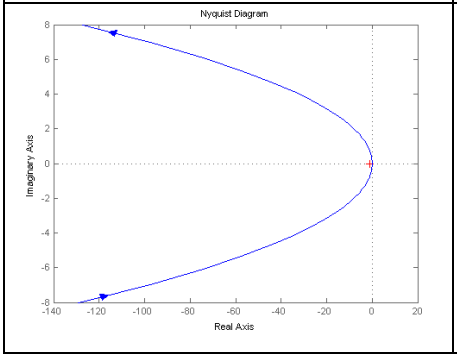
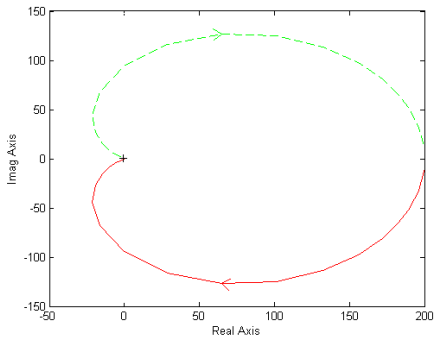
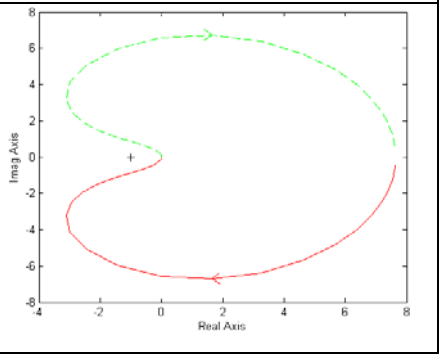
```
s = tf('s');
sys = 0.5/(s - 0.5);
nyquist(sys)
axis([-1 0 -1 1])
```



Now we will look at the Nyquist diagram for the following transfer function: $\frac{s+2}{s^2}$

(4)

Note that this function has a pole at the origin. We will see the difference between using the nyquist, nyquist1, and lnyquistcommands with this particular function.

sys = (s + 2)/(s^2); nyquist(sys)	nyquist1(sys)	lnyquist(sys)
		

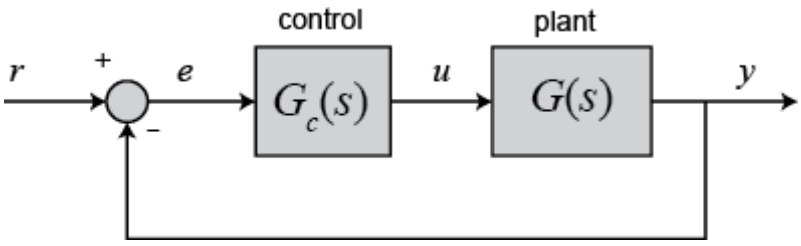
Note that the nyquistplot is not the correct one, the nyquist1plot is correct, but it's hard to see what happens close to the -1 point, and the lnyquistplot is correct and has an appropriate scale.

Closed-Loop Performance from Bode Plots

In order to predict closed-loop performance from open-loop frequency response, we need to have several concepts clear:

- The system must be stable in open-loop if we are going to design via Bode plots.
- If the **gain crossover frequency** is less than the **phase crossover frequency** (i.e. $W_{gc} < W_{pc}$), then the closed-loop system will be stable.
- For second-order systems, the closed-loop damping ratio is approximately equal to the phase margin divided by 100 if the phase margin is between 0 and 60 degrees. We can use this concept with caution if the phase margin is greater than 60 degrees.
- For second-order systems, a relationship between damping ratio, bandwidth frequency, and settling time is given by an equation described on the wbw.m file page.
- A very rough estimate that you can use is that the bandwidth is approximately equal to the natural frequency.

Let's use these concepts to design a controller for the following system:


control
plant
r
+
e
$G_c(s)$
u
$G(s)$
y
−

Where $G_c(s)$ is the controller and $G(s)$ is: $GG(ss) = \frac{10}{1.25ss+1}$

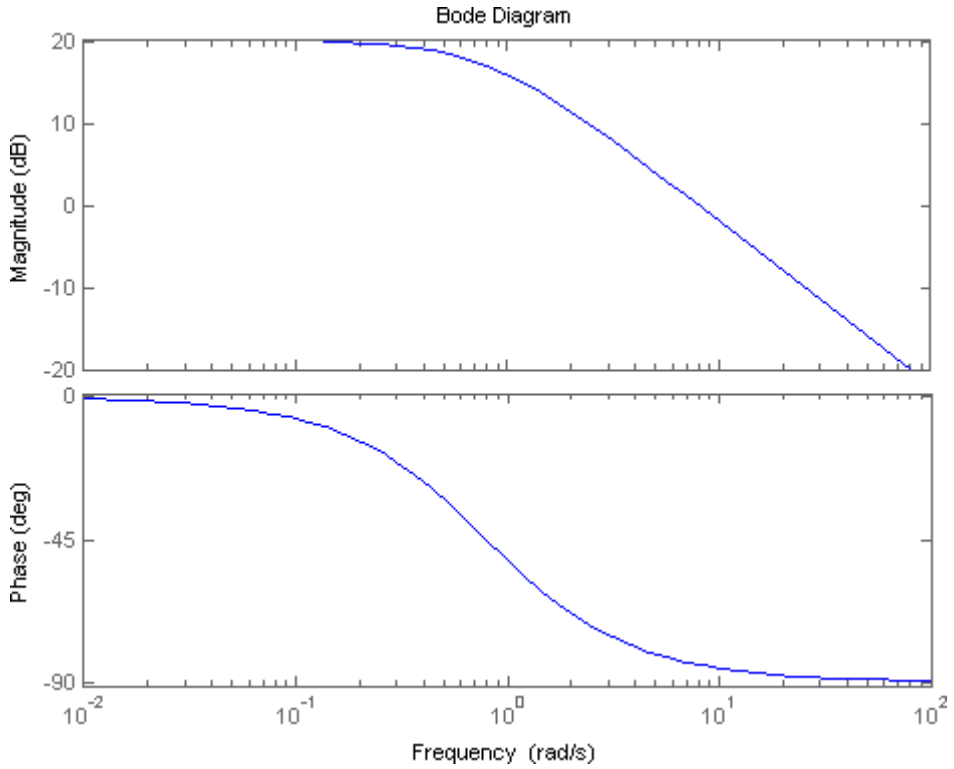
(5)

The design must meet the following specifications:

- Zero steady state error.
- Maximum overshoot must be less than 40%.
- Settling time must be less than 2 seconds.

There are two ways of solving this problem: one is graphical and the other is numerical. Within MATLAB, the graphical approach is best, so that is the approach we will use. First, let's look at the Bode plot. Create a m-filewith the following code:

```
sys = 10/(1.25*s + 1); bode(sys)
```

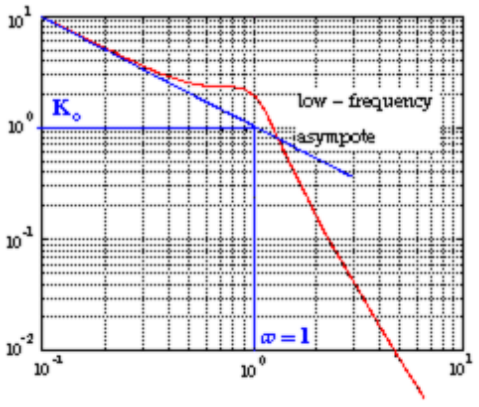


There are several characteristics of the system that can be read directly from this Bode plot. First of all, we can see that the bandwidth frequency is around 10 rad/sec. Since the bandwidth frequency is roughly the same as the natural frequency (for a first order system of this type), the rise time is 1.8/BW = 1.8/10 = 1.8 seconds. This is a rough estimate, so we will say the rise time is about 2 seconds.

The phase margin for this system is approximately 95 degrees. The relation damping ratio = PM/100 only holds for PM < 60. Since the system is first-order, there should be no overshoot.

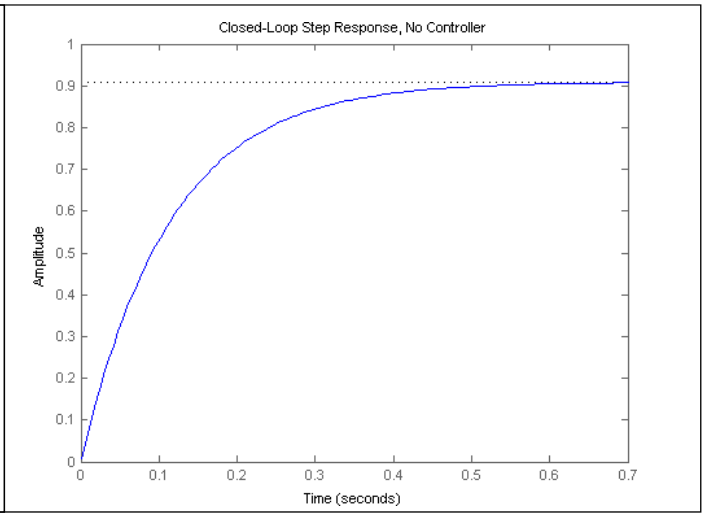
The last major point of interest is **steady-state error**. The steady-state error can be read directly off the Bode plot as well. The constant (K_p, K_v, or K_a) is found from the intersection of the low frequency asymptote with the w = 1 line. Just extend the low frequency line to the w = 1 line. The magnitude at this point is the constant. Since the Bode plot of this system is a horizontal line at low frequencies (slope = 0), we know this system is of type zero. Therefore, the intersection is easy to find. The gain is 20 dB (magnitude 10). What this means is that the constant for the error function is 10. The steady-state error is 1/(1+Kp) = 1/(1+10) = 0.091.

If our system was type one instead of type zero, the constant for the steady-state error would be found in a manner similar to the following.



Let's check our predictions by looking at a step response plot. This can be done by adding the following two lines of code into the MATLAB command window.

```
sys_cl = feedback(sys,1);
step(sys_cl)
title('Closed-Loop Step Response, No
Controller')
```



As you can see, our predictions were very good. The system has a rise time of about 2 seconds, has no overshoot, and has a steady-state error of about 9%. Now we need to choose a controller that will allow us to meet the design criteria. We choose a PI controller because it will yield zero steady-state error for a step input. Also, the PI controller has a zero, which we can place. This gives us additional design flexibility to help us meet our criteria. Recall that a PI controller is given by:

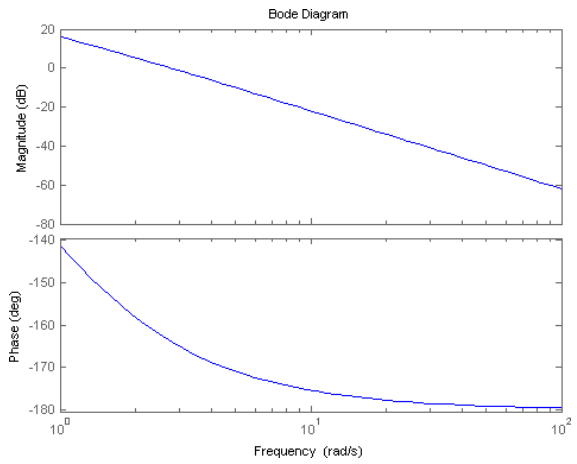
$$GG(ss) = \frac{KK(ss+aa)}{ss} \qquad (6)$$

The first thing we need to find is the damping ratio corresponding to a percent overshoot of 40%. Plugging in this value into the equation relating overshoot and damping ratio (or consulting a plot of this relation), we find that the damping ratio corresponding to this overshoot is approximately 0.28. Therefore, our phase margin should be at least 30 degrees. We must have a bandwidth frequency greater than or equal to 12 if we want our settling time to be less than 1.75 seconds which meets the design specs.

Now that we know our desired phase margin and bandwidth frequency, we can start our design. Remember that we are looking at the open-loop Bode plots. Therefore, our bandwidth frequency will be the frequency corresponding to a gain of approximately -7 dB.

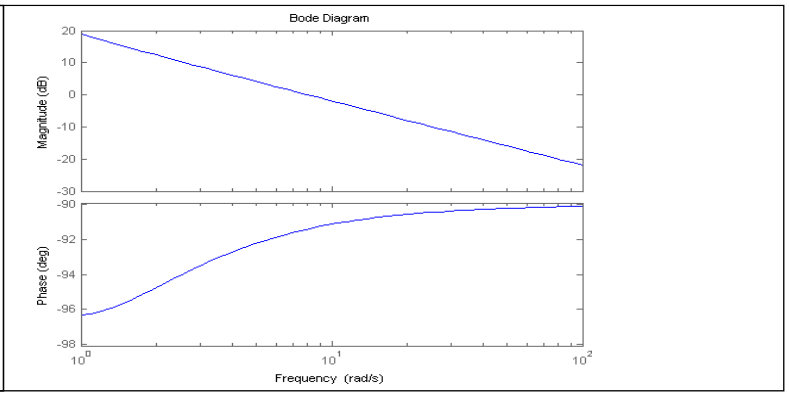
Let's see how the integrator portion of the PI or affects our response. Change your m-file to look like the following (this adds an integral term but no proportional term):

```
plant = 10/(1.25*s + 1);
contr = 1/s;
bode(contr*plant, logspace(0,2))
```



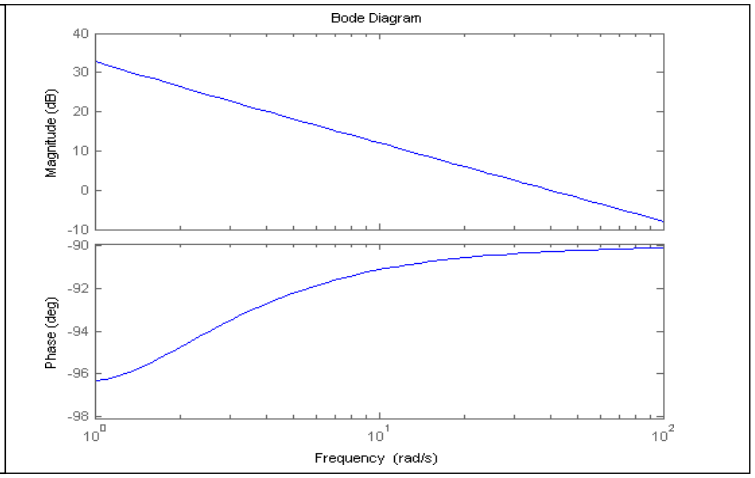
Our phase margin and bandwidth frequency are too small. We will add gain and phase with a zero. Let's place the zero at 1 for now and see what happens. Change your m-fileto look like the following:

```
plant = 10/(1.25*s + 1);
contr = (s + 1)/s;
bode(contr*plant,
logspace(0,2))
```



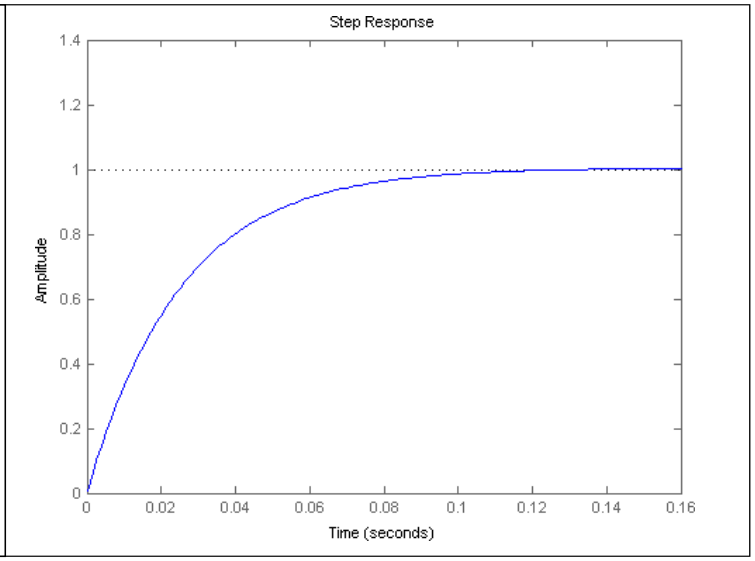
It turns out that the zero at 1 with a unit gain gives us a satisfactory answer. Our phase margin is greater than 60 degrees (even less overshoot than expected) and our bandwidth frequency is approximately 11 rad/s, which will give us a satisfactory response. Although satisfactory, the response is not quite as good as we would like. Therefore, let's try to get a higher bandwidth frequency without changing the phase margin too much. Let's try to increase the gain to 5 and see what happens. This will make the gain shift and the phase will remain the same.

```
plant = 10/(1.25*s + 1);
contr = 5 * ((s + 1)/s);
bode(contr*plant, logspace(0,2))
```



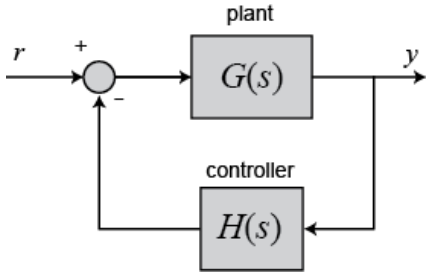
That looks really good. Let's look at our step response and verify our results. Add the following two lines to your m-file:

```
sys_cl = feedback(contr*plant,1);
step(sys_cl)
```



Closed-Loop Stability from the Nyquist Diagram

Consider the negative feedback system:



Remember from the Cauchy criterion that the number N of times that the plot of G(s)H(s) encircles -1 is equal to the number Z of zeros of 1 + G(s)H(s) enclosed by the frequency contour minus the number P of poles of 1 + G(s)H(s) enclosed by the frequency contour (N = Z - P). Keeping careful track of open- and closed-loop transfer functions, as well as numerators and denominators, you should convince yourself that:

- The zeros of 1 + G(s)H(s)are the poles of the closed-loop transfer function.
- The poles of 1 + G(s)H(s)are the poles of the open-loop transfer function. The

Nyquist criterion then states that:

- P= the number of open-loop (unstable) poles of G(s)H(s).
- N= the number of times the Nyquist diagram encircles -1.
- clockwise encirclements of -1 count as positive encirclements.
- counter-clockwise encirclements of -1 count as negative encirclements.
- Z= the number of right-half-plane (positive, real) poles of the closed-loop system.

The important equation which relates these three quantities is:

$$Z=P+N \qquad (7)$$

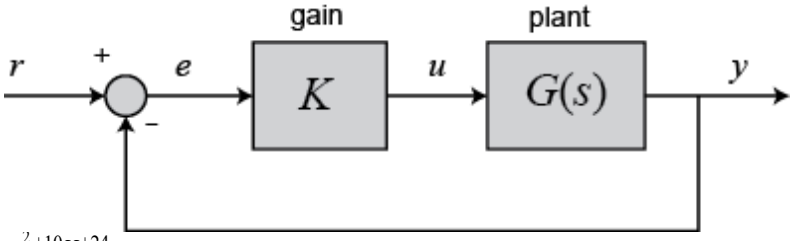
Note: This is only one convention for the Nyquist criterion. Another convention states that a positive N counts the counter-clockwise or anti-clockwise encirclements of -1. The P and Z variables remain the same. In this case the equation becomes Z = P - N. Throughout these tutorials, we will use a positive sign for clockwise encirclements.

It is very important (and somewhat tricky) to learn how to count the number of times that the diagram encircles -1. Therefore, we will go into some detail to help you visualize this.

Another way of looking at it is to imagine you are standing on top of the -1 point and are following the diagram from beginning to end. Now ask yourself: How many times did I turn my head a full 360 degrees? Again, if the motion was clockwise, N is positive, and if the motion is anti-clockwise, N is negative.

Knowing the number of right-half plane (unstable) poles in open loop (P), and the number of encirclements of -1 made by the Nyquist diagram (N), we can determine the closed-loop stability of the system. If Z = P + Nis a positive, nonzero number, the closed-loop system is unstable.

We can also use the Nyquist diagram to find the range of gains for a closed-loop unity feedback system to be stable. The system we will test looks like this:



where G(s) is: $GG(ss) = \frac{ss^{2}+10ss+24}{2ss\ -8ss+15}$ (8)

This system has a gain Kwhich can be varied in order to modify the response of the closed-loop system. However, we will see that we can only vary this gain within certain limits, since we have to make sure that our closed-loop system will be stable. This is what we will be looking for: the range of gains that will make this system stable in the closed-loop.

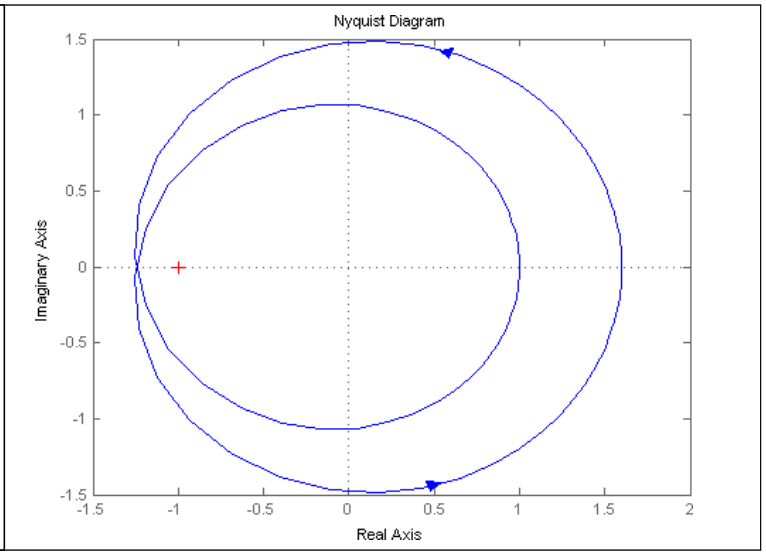
The first thing we need to do is find the number of positive real poles in our open-loop transfer function:

```
roots([1 -8 15]) ans =
     5
     3
```

The poles of the open-loop transfer function are both positive. Therefore, we need two anti-clockwise (N = -2) encirclements of the Nyquist diagram in order to have a stable closed-loop system (Z = P + N). If the number of encirclements is less than two or the encirclements are not anti-clockwise, our system will be unstable.

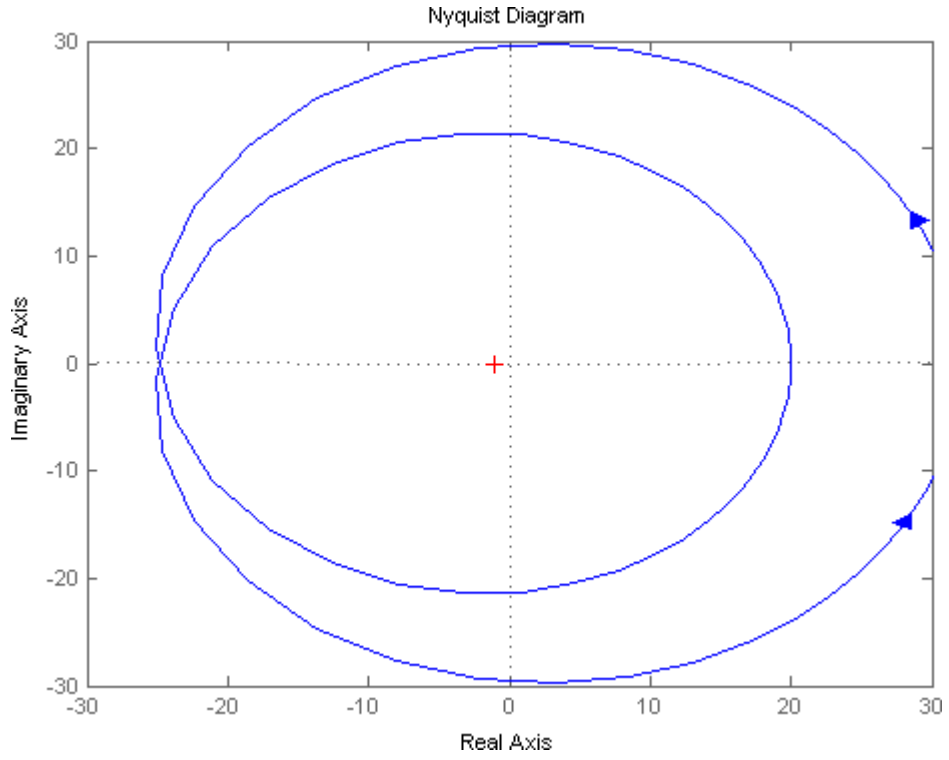
Let's look at our Nyquist diagram for a gain of 1:

```
sys = (s^2 + 10*s + 24)/(s^2 - 8*s
+ 15);
nyquist(sys)
```



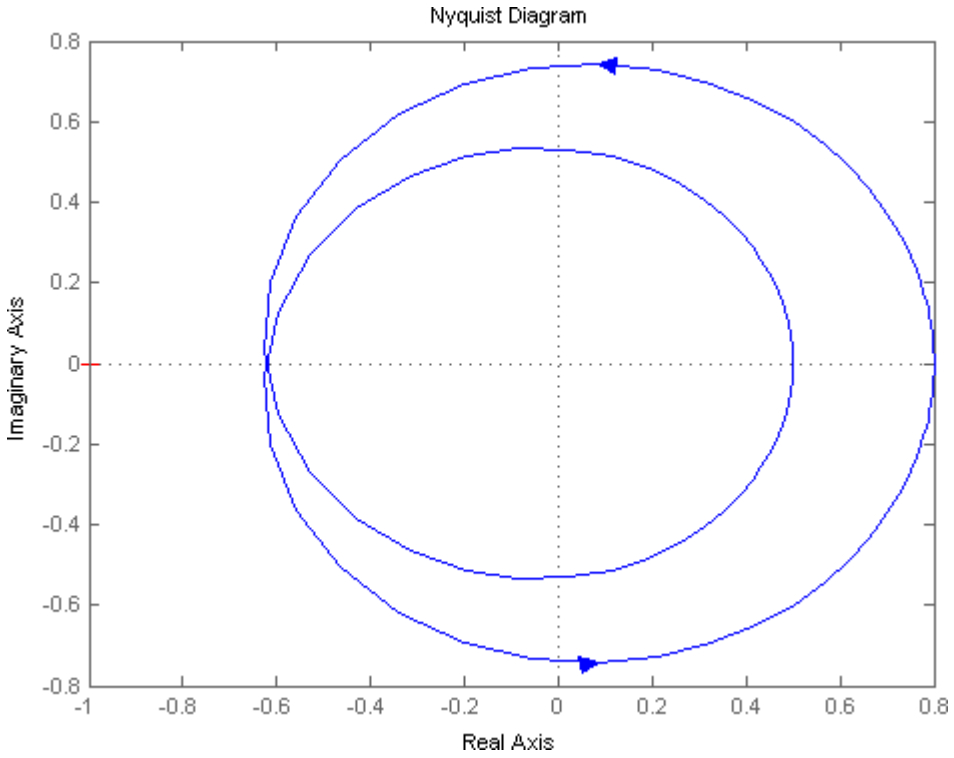
There are two anti-clockwise encirclements of -1. Therefore, the system is stable for a gain of 1. Now we will see how the system behaves if we increase the gain to 20:

```
nyquist(20*sys)
```



The diagram expanded. Therefore, we know that the system will be stable no matter how much we increase the gain. However, if we decrease the gain, the diagram will contract and the system might become unstable. Let's see what happens for a gain of 0.5:

```
nyquist(0.5*sys)
```



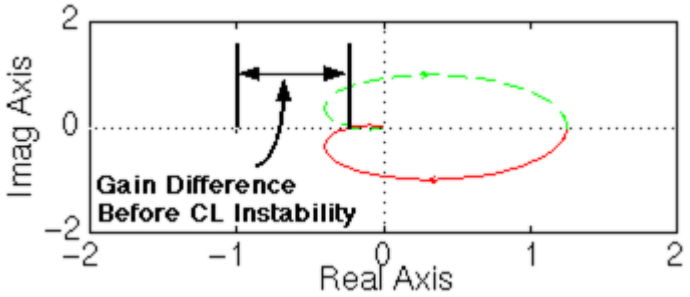
The system is now unstable. By trial and error we find that this system will become unstable for gains less than 0.80. We can verify our answers by zooming in on the Nyquist plots as well as by looking at the closed-loop steps responses for gains of 0.79, 0.80, and 0.81.

Gain Margin

We already defined the gain margin as the change in open-loop gain expressed in decibels (dB), required at 180 degrees of phase shift to make the system unstable. Now we are going to find out where this comes from. First of all, let's say that we have a system that is stable if there are no Nyquist encirclements of -1, such as:

$$\frac{50}{ss^3+9ss^2+30ss+40} \qquad (9)$$

Looking at the roots, we find that we have no open loop poles in the right half plane and therefore no closed-loop poles in the right-half-plane if there are no Nyquist encirclements of -1. Now, how much can we vary the gain before this system becomes unstable in closed-loop? Let's look at the following figure:



The open-loop system represented by this plot will become unstable in closed loop if the gain is increased past a certain boundary. The negative real axis area between -1/a (defined as the point where the 180 degree phase shift occurs...that is, where the diagram crosses the real axis) and -1 represents the amount of increase in gain that can be tolerated before closed-loop instability.

If we think about it, we realize that if the gain is equal to a, the diagram will touch the -1 point:

$$GG(jjjj) = \frac{-1}{aa} \qquad (10)$$

Or

$$aaGG(jjjj) = -1 \qquad (11)$$

Therefore, we say that the gain margin is aunits. However, we mentioned before that the gain margin is usually measured in decibels. Hence, the gain margin is:

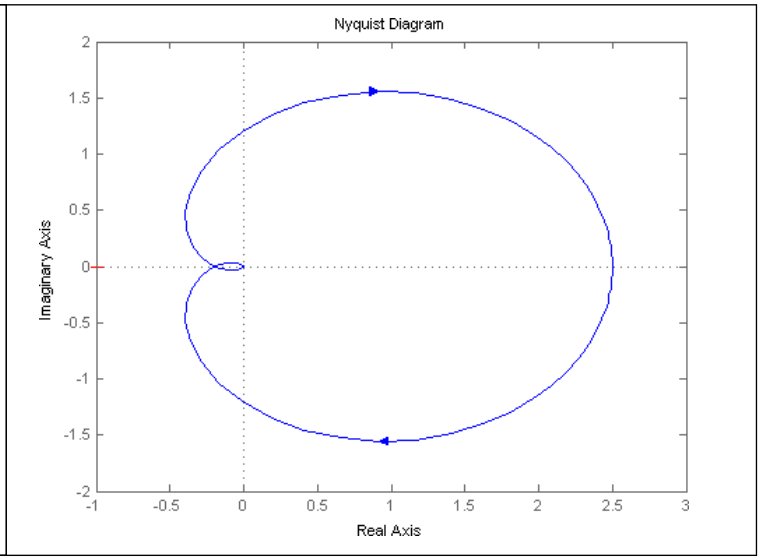
$$GGGG = 20llllll_{10}(aa) \qquad (12)$$

We will now find the gain margin of the stable, open-loop transfer function we viewed before. Recall that the function is:

$$\frac{50}{ss^3 + 9ss^2 + 30ss + 40} \qquad (13)$$

and that the Nyquist diagram can be viewed by typing:

```
sys = 50/(s^3 + 9*s^2 + 30*s + 20);
nyquist(sys)
```



As we discussed before, all that we need to do to find the gain margin is find a, as defined in the preceding figure. To do this, we need to find the point where there is exactly 180 degrees of phase shift. This means that the transfer function at this point is real (has no imaginary part). The numerator is already real, so we just need to look at the denominator. When s = jw, the only terms in the denominator that will have imaginary parts are those which are odd powers of s. Therefore, for G(jw) to be real, we must have:

$$-jjjj^3 + 30jjjj = 0 \qquad (14)$$

which means w = 0 (this is the rightmost point in the Nyquist diagram) or w = sqrt(30). We can then find the value of G(jw) at this point using polyval:

```
w = sqrt(30);
polyval(50,j*w)/polyval([1 9 30 40],j*w) ans =
      -0.2174
```

The answer is: -0.2174 + 0i. The imaginary part is zero, so we know that our answer is correct. We can also verify by looking at the Nyquist plot again. The real part also makes sense. Now we can proceed to find the gain margin.

We found that the 180 degrees phase shift occurs at -0.2174 + 0i. This point was previously defined as -1/a. Therefore, we now have a, which is the gain margin. However, we need to express the gain margin in decibels:

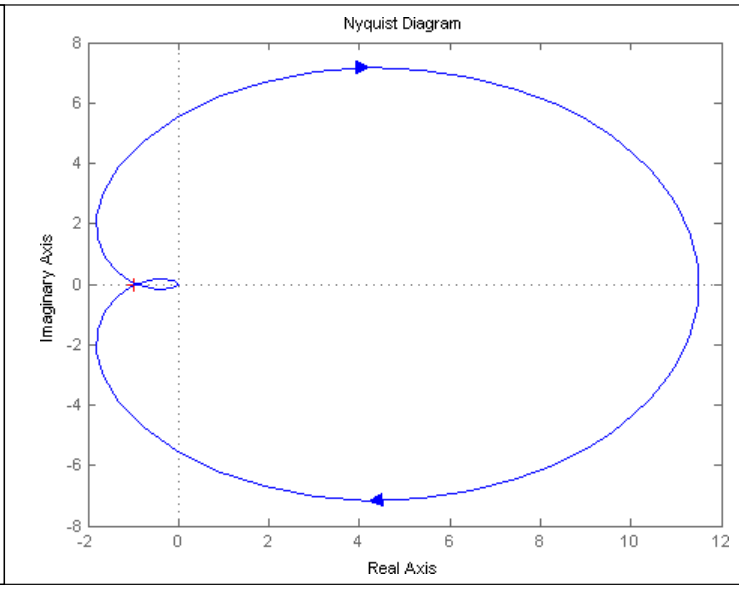
$$\frac{-1}{aa} = -0.2174 \qquad (15)$$

$$\Rightarrow aa = 4.6 \qquad (16)$$

$$\Rightarrow GGGG = 20llllll_{10}(4.6) = 13.26dddd \qquad (17)$$

We now have our gain margin. Let's see how accurate it is by using a gain of a = 4.6and zooming in on the Nyquist plot:

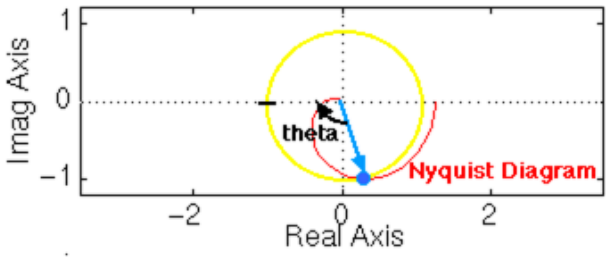
```
a = 4.6;
nyquist(a*sys)
```



The plot appears to go right through the -1 point. We will now verify the accuracy of our results by viewing the zoomed Nyquist diagrams and step responses for gains of 4.5, 4.6, and 4.7.

Phase Margin

We have already discussed the importance of the phase margin. Therefore, we will only talk about where this concept comes from. We have defined the phase margin as the change in open-loop phase shift required at unity gain to make a closed-loop system unstable. Let's look at the following graphical definition of this concept to get a better idea of what we are talking about.



Let's analyze the previous plot and think about what is happening. From our previous example we know that this particular system will be unstable in closed-loop if the Nyquist diagram encircles the -1 point. However, we must also realize that if the diagram is shifted by theta degrees, it will then touch the -1 point at the negative real axis, making the system marginally stable in closed-loop. Therefore, the angle required to make this system marginally stable in closed-loop is called the phase margin (measured in degrees). In order to find the point we measure this angle from, we draw a circle with radius of 1, find the point in the Nyquist diagram with a magnitude of 1 (gain of zero dB), and measure the phase shift needed for this point to be at an angle of 180 degrees.

DC Motor Speed: Frequency Domain Methods for Controller Design

From the main problem, the dynamic equations in the Laplace domain and the open-loop transfer function of the DC Motor are the following.

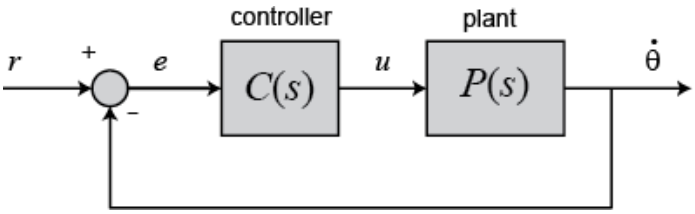
$$ss(JJss + bb)\theta\theta(ss) = KKKK(ss) \quad (1)$$

$$(LLss + RR)\text{KK}(ss) = VV(ss) - KKss\theta\theta(ss) \quad (2)$$

$$PP(ss) = \frac{\theta\theta\,(ss)}{K} = \frac{K}{\overline{VV(ss)}} \quad \left[\frac{rraadd\ /sssscc}{VV}\right] \quad (3)$$

$$(JJss + bb)(LLss+RR)+KK^2$$

The structure of the control system has the form shown in the figure below.



For a 1-rad/sec step reference, the design criteria are the following.

- Settling time less than 2 seconds
- Overshoot less than 5%
- Steady-state error less than 1%

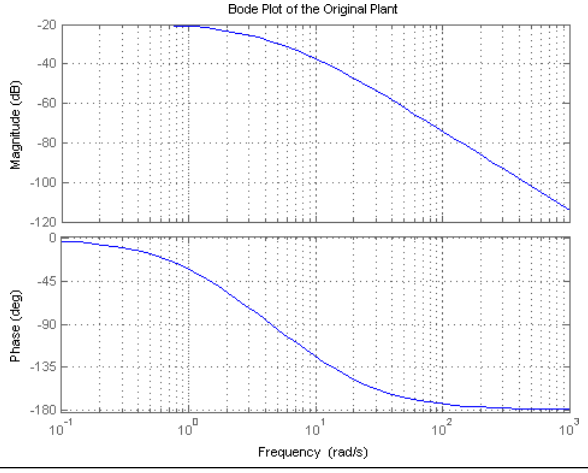
Now let's design a controller using the methods introduced in the Introduction: Frequency Domain Methods for Controller Design part. Create a new m-file and type in the following commands.

```
J = 0.01;
b = 0.1;
K = 0.01;
R = 1;
L = 0.5;
s = tf('s');
P_motor = K/((J*s+b)*(L*s+R)+K^2);
```

Drawing the original Bode plot

The main idea of frequency-based design is to use the Bode plot of the open-loop transfer function to estimate the closed-loop response. Adding a controller to the system changes the open-loop Bode plot, thereby changing the closed-loop response. It is our goal to design the controller to shape the open-loop Bode plot in such a way that the closed-loop system behaves in a desired manner. Let's first draw the Bode plot for the original open-loop plant transfer function. Add the following code to the end of your m-file and run it in the MATLAB command window. You should generate the Bode plot shown below.

```
bode(P_motor)
grid
title('Bode Plot of the Original
Plant')
```



Adding proportional gain

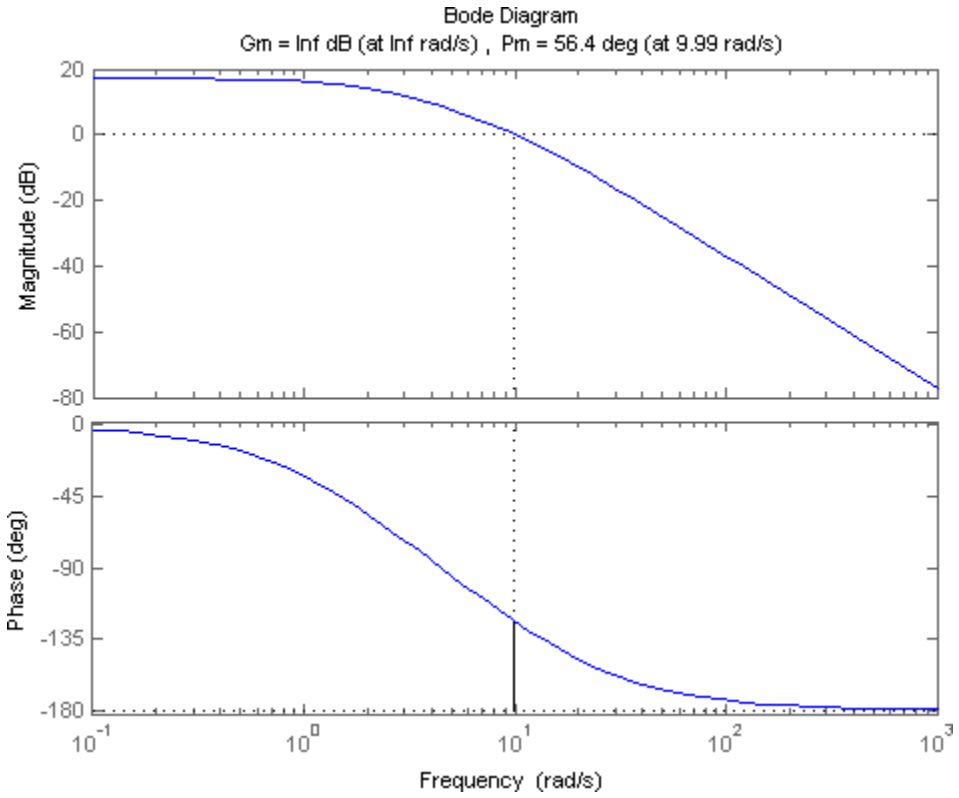
From the Bode plot above, it appears that the gain margin and phase margin of this system are currently infinite which indicates the system is robust and has minimal overshoot. The problem with this is that the phase margin is infinite because the magnitude plot is below 0 dB at all frequencies. This indicates that the system will have trouble tracking various reference signals without excessive error. Therefore, we would like to increase the gain of the system while still achieving enough phase margin.

A phase margin of 60 degrees is generally sufficient for stability margin. From the above Bode plot, this phase margin is achieved for a crossover frequency of approximately 10 rad/sec. The gain needed to raise the magnitude plot so that the gain crossover frequency occurs at 10 rad/sec appears to be approximately 40 dB. The exact phase and gain of the Bode plot at a given frequency can be determined by clicking on the graph at the corresponding frequency. The bode command, invoked with left-hand arguments, can also be used to provide the exact phase and magnitude at 10 rad/sec as shown below.

```
[mag,phase,w] = bode(P_motor,10) mag =
        0.0139
phase =
  -123.6835
w =
        10
```

Therefore, the exact phase margin for a gain crossover frequency of 10 rad/sec is 180 - 123.7 = 56.3 degrees. Since the exact magnitude at this frequency is 20 log 0.0139 = -37.1 dB, 37.1 dB of gain must be added to the system. Otherwise stated, a proportional gain of 1/0.0139 = 72 will achieve an open-loop gain of 1 at 10 rad/sec. Add the following commands to your m-file to observe the effect of this proportional controller on the system. In this case, we use the margin command instead of the bode command in order to explicitly see the new gain and phase margins and crossover frequencies.

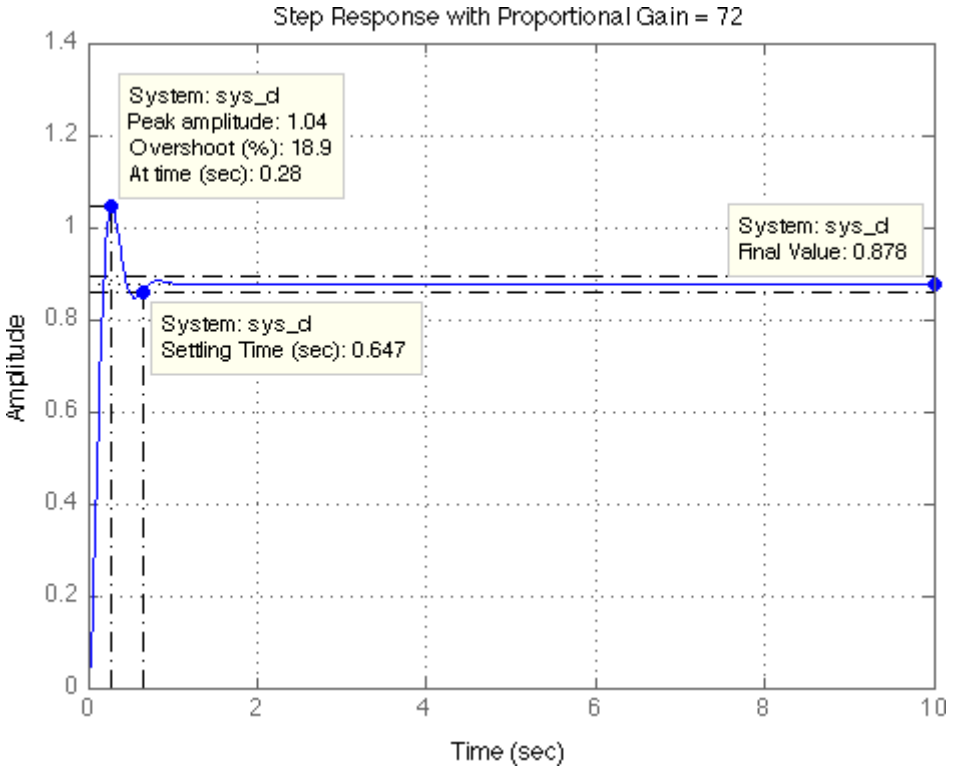
```
C = 72;
margin(C*P_motor);
```



Plotting the closed-loop response

From the plot above we see that the resulting phase margin and gain crossover frequency are as we expected. Let's see what the closed-loop response look like. Add a % in front of the bodeand margin commands to comment them out, then add the following code to the end of your m-file. Rerunning the m-file will produce the step response shown below where the annotations were added by right-clicking on the plot and choosing **Characteristics** from the resulting menu.

```
sys_cl = feedback(C*P_motor,1); t =
0:0.01:10;
step(sys_cl,t), grid
title('Step Response with Proportional Gain = 72')
```



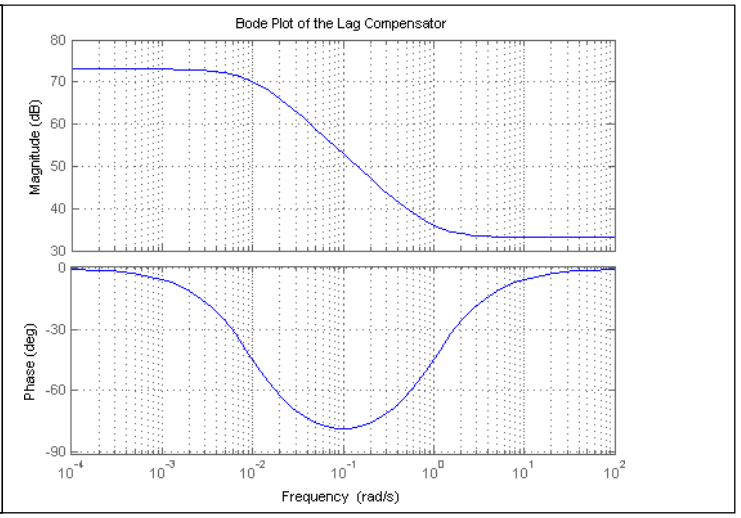
Note that the settling time is fast enough, but the overshoot and the steady-state error are too high. The overshoot can be reduced by decreasing the gain in order to achieve a larger phase margin, but this would cause the steady-state error to become even larger. A lag compensator could be helpful here in that it can decrease the gain crossover frequency in order to increase the phase margin without decreasing the system's DC gain.

Adding a lag compensator

Consider the following lag compensator: $CC(ss) = \frac{ss+1}{ss+0.01}$ (4)

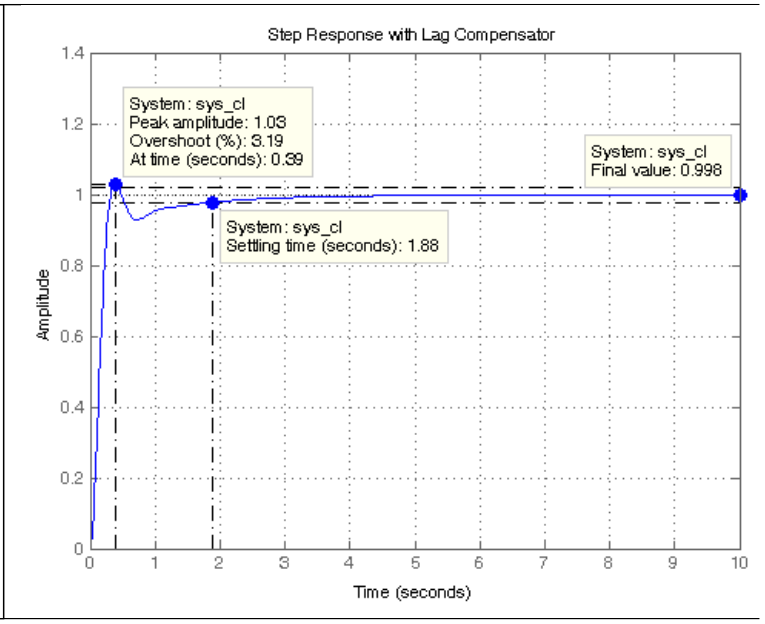
This lag compensator has a DC gain of 1/0.01 = 100 which means it will increase the system's static position error constant by a factor of 100 and will reduce the steady-state error associated with the system's closed-loop step response. In fact, it allows us to reduce the proportional gain of 72 used earlier, while still meeting the requirement on steady-state error. We will employ a gain of 45. Furthermore, since the corner frequencies of the pole and zero are a decade or more below the current gain crossover frequency of 10 rad/sec, the phase lag contributed by the compensator shouldn't adversely affect performance much. A Bode plot of the lag compensator can be generated employing the following commands.

```
C = 45*(s + 1)/(s + 0.01);
bode(C)
grid
title('Bode Plot of the Lag
Compensator')
```



The resulting step response can then be observed by modifying the code in your m-file as follows.

```
 sys_cl = feedback(C*P_motor,1);
 t = 0:0.01:10;
 step(sys_cl,t), grid
 title('Step Response with Lag
Compensator')
```



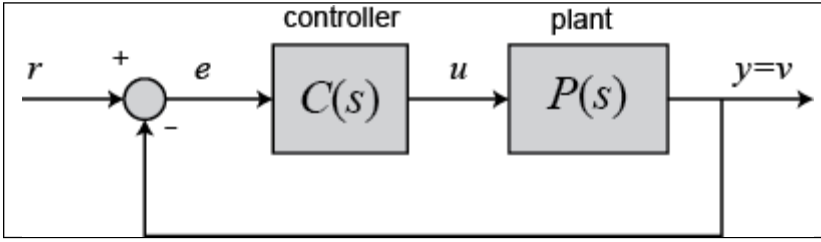
Inspection of the above demonstrates that all of the given requirements are now met when the lag compensator described above is employed.

Work: Design frequency domain method controller for cruise control with the following specifications

For this example, let's assume that the parameters of the system are

(m)	vehicle mass	1000 kg
(b)	damping coefficient	50 N.s/m
(r)	reference speed	10 m/s

and the block diagram of a typical unity feedback system is shown below.



Performance specifications

- Rise time < 5 sec
- Overshoot < 10%
- Steady-state error < 2%