MECH 467
Lab #2
By Bobsy Narayan
Date: November 17, 2024

Contents

Table of Figures

Table of Equations

Table of Tables

Abstract

This lab focuses on designing control algorithms that deliver desired performance characteristics for a given Mechatronics System. Aimed at evaluation system response speed, accuracy, and robustness of system actuators, this lab provides necessary information to design and control various proportional, lead compensator, and lead-lag compensator controllers. This lab will connect theoretical knowledge with practical application to assist users with learning the concepts behind digital controller design.

Introduction

Controller design plays a vital role in most industries, as it enables precise operation of manufacturing, robotics, transportation, and other applicable systems. Controllers can help ensure stability, improve performance, and optimize system responses to meet given desired system specifications. This report begins with a discussion our analyzed system, and the setup procedures used to create proper controller for system control. Afterwards, we analyze our experimental results measured from our system apparatus and provide a detailed discussion of the similarities and differences between our expected and measured results, followed by conclusions and supporting documentation in the end.

System Analyzed:

In this lab, we will be creating control algorithms for the ball-screw driven table identified in Project 1. Our analyzed system model can be seen in the following figure.

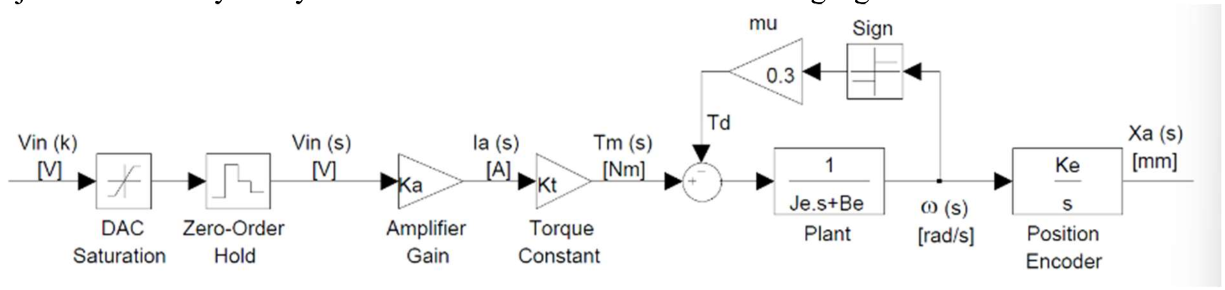


Figure 1: Open Loop Block Diagram of Ball Screw Feed Drive System

Due to the small impact and non-linear effects of torque friction, we can ignore it for our analysis, making this a linear singular path transfer function.

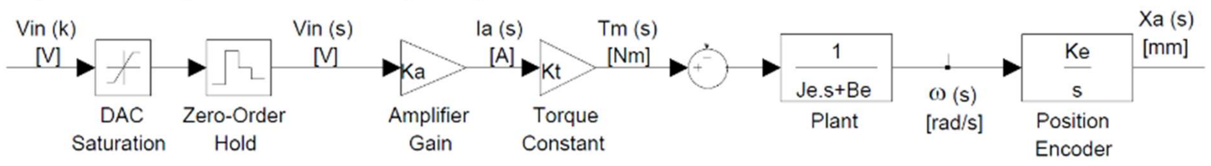


Figure 2: Simplified System Open Loop Block Diagram

In our *Lab 2 instructions document,* we are given values for system constants, as shown in the following table.

Table 1: Defined System Constants

System Constant	Value
Amplifier Gain (Ka)	0.887 A/V
Torque Constant (Kt)	0.72 Nm/A
Position Encoder Constant (Ke)	$20/2\pi$ mm/rad
Inertia Value (Je)	$7x10^{-4}\ kgm^2$
Vicsous Friction Constant (Be)	0.00612 Nm/rad/s
Sampling Time (Ts)	0.0002s

Prelab 1 – Discrete Transfer Function Derivation

1) Manually obtain Zero-Order hold equivalent of open loop transfer function for system analyzed in Lab 1. Compare to Matlab's C2D command/simulated results.

Open loop transfer function can be determined using Black's Formula, as seen in the following equation.

Equation 1: Open Loop Transfer Function for Experimental System

$$G(s) = \frac{X_a(s)}{V_{in}(s)} = K_a * K_t * \frac{1}{(J_e s + B_e)} * \frac{K_e}{s}$$

Using this transfer function, we can determine the Zero-order hold function using function decomposition. We can solve for our constants A & B by finding solutions where system poles are 0.

Equation 2: Partial Fraction Decomposition Setup & Constant Solutions

$$G(s) = \frac{K_a K_t K_e}{s(J_e s + B_e)} = \frac{A}{s} + \frac{B}{(J_e s + B_e)}$$

$$A = \lim_{s=0} s \frac{K_a K_t K_e}{s(J_e s + B_e)}$$

$$A = \frac{K_a K_t K_e}{B_e}$$

$$B = \lim_{s=-B_e/J_e} (J_e s + B_e) \frac{K_a K_t K_e}{s(J_e s + B_e)}$$

$$B = -\frac{J_e K_a K_t K_e}{B_e}$$

$$G(s) = \frac{K_a K_t K_e}{s(J_e s + B_e)} = \frac{K_a K_t K_e}{s B_e} - \frac{J_e K_a K_t K_e}{B_e (J_e s + B_e)}$$

$$G(s) = \frac{K_a K_t K_e}{s(J_e s + B_e)} = \frac{K_a K_t K_e}{s B_e} - \frac{K_a K_t K_e}{B_e (s + B_e/J_e)}$$

Now that we have successfully decomposed our transfer function, we can use this G(s) to find our Zoh function, as seen below.

Equation 3: Solution for G(z) Zero-Order hold function if T=0.0002

$$G(z) = (1 - z^{-1}) * Z\{\frac{G(s)}{s}\}$$

$$G(z) = (1 - z^{-1}) * Z\left\{\frac{K_a K_t K_e}{s^2 B_e} - \frac{K_a K_t K_e}{s B_e (s + B_e/J_e)}\right\}$$

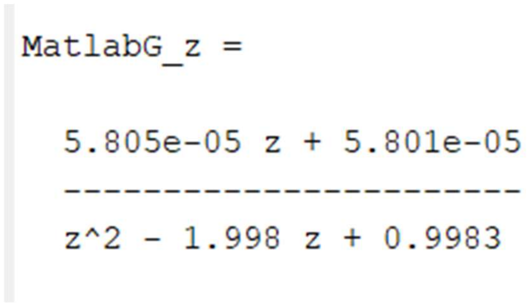
$$G(z) = (1 - z^{-1}) * Z\left\{\frac{K_a K_t K_e}{B_e}\frac{1}{s^2} - \frac{K_a K_t K_e}{B_e}\frac{1}{s(s + B_e/J_e)}\right\}$$

$$G(z) = (1 - z^{-1}) * Z\left\{\frac{K_a K_t K_e}{B_e}\frac{1}{s^2} - \frac{K_a K_t K_e}{B_e}\frac{J_e}{B_e}\frac{B_e/J_e}{s(s + B_e/J_e)}\right\}$$

$$G(z) = (1 - z^{-1}) * \left\{ \frac{K_a K_t K_e}{B_e} \frac{T z^{-1}}{(1 - z^{-1})^2} - \frac{J_e K_a K_t K_e}{B_e^2} \frac{z^{-1}(1 - e^{-\frac{Be}{Je}T})}{(1 - z^{-1})(1 - e^{-\frac{Be}{Je}T} z^{-1})} \right\}$$

When we plug this into MATLAB, our responses match our C2D values exactly, proving our manual solution. Bode Responses & Step responses are also provided, which both show the same result for both the C2D & Manual derivations.

Figure 3: Final G(z) for both Manual Solution



```
MatlabG_z =

  5.805e-05 z + 5.801e-05
  -----------------------
  z^2 - 1.998 z + 0.9983
```

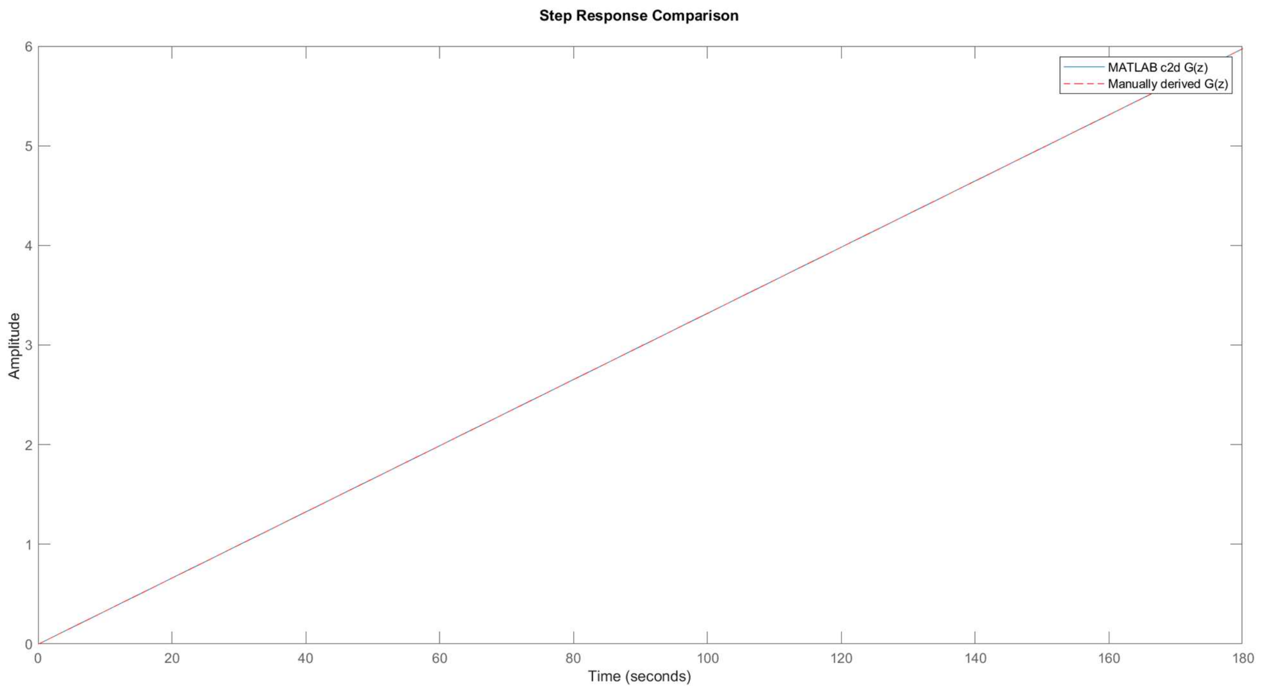


Figure 4: Bode Response for Manual & C2D Derivations

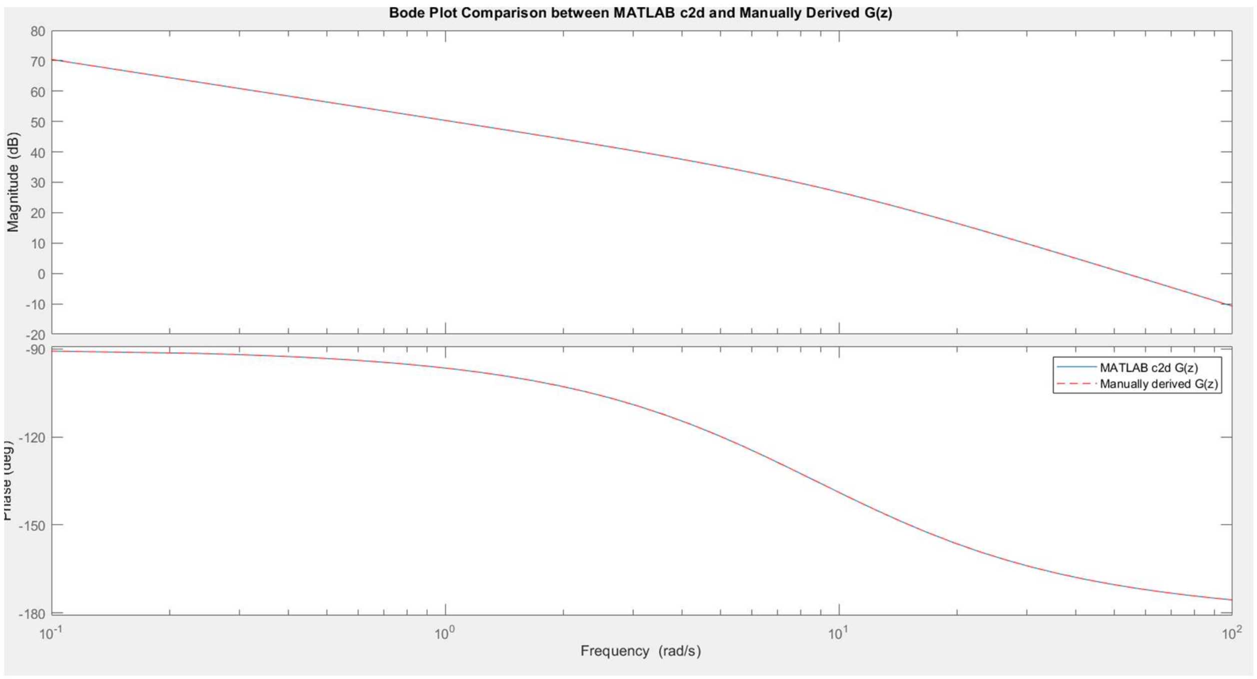


Figure 5: Step Response for Manual & C2D Derivations

Prelab 2 – State Space Model

Obtain the discrete time state space model of the machine shown in Figure 1. Simulate the step response of the system using the discrete state space model of the machine given in Figure 1. Compare the results obtained from discrete transfer function and state space models.

Firstly, let's find the continuous state space model from our system. Let's first defined relevant defined system matrices, state variables, and state equations.

- X – State Vector
- Y – Output Vector
- U – Input Vector
- A – State Matrix -
- B – Input Matrix
- C – Output Matrix
- D – Feedthrough Matrix

Equation 4: State Space General Equations

$$\{\dot{X}\} = [A]\{X\} + [B]\{U\}$$
$$\{Y\} = [C]\{X\} + [D]\{U\}$$

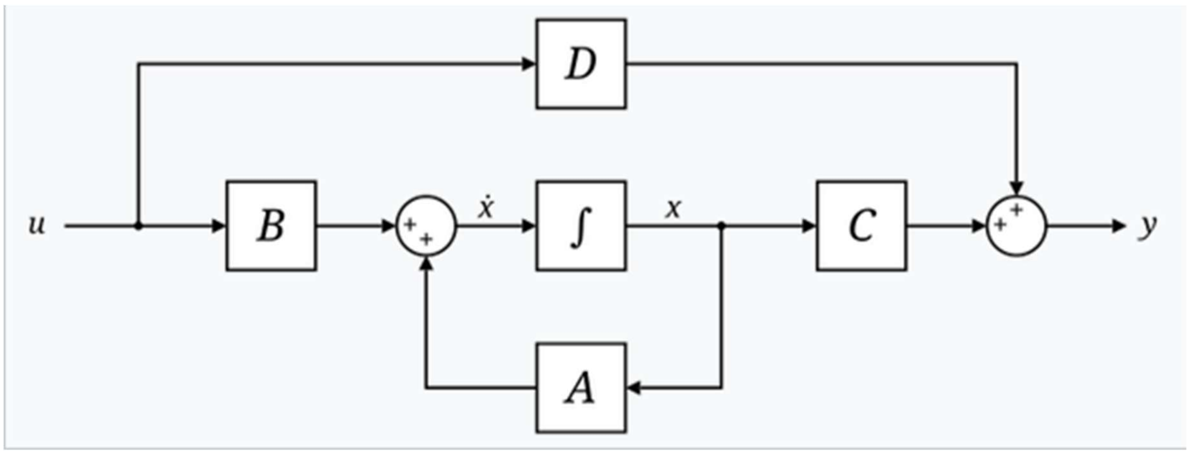


Figure 6: Block Diagram of Linear State-Space Equations (From Wikipedia)

We can define all linear equations and state variables from our main system function. To start, we can choose our input and output vectors.

Equation 5: State-Space Input, Output, and State Vectors

$$\{X\} = \begin{bmatrix} w(s) \\ X_a(s) \end{bmatrix}, \{U\} = \begin{bmatrix} V_{in}(s) \\ T_d(s) \end{bmatrix}; Y = \begin{bmatrix} X_a(s) \\ 0 \end{bmatrix};$$

Using these state variables, we can define our space-state equations.

Equation 6: State-Space Matrix Solutions

$$\begin{bmatrix} w\dot{(}s) \\ X_a\dot{(}s) \end{bmatrix} = [A] \begin{bmatrix} w(s) \\ X_a(s) \end{bmatrix} + [B] \begin{bmatrix} V_{in}(s) \\ T_d(s) \end{bmatrix}$$

$$[X_a(s)] = [C] \begin{bmatrix} w(s) \\ X_a(s) \end{bmatrix} + [D] \begin{bmatrix} V_{in}(s) \\ T_d(s) \end{bmatrix}$$

$$\frac{w(s)}{T(s)} = \frac{1}{J_e s + B_e}$$

$$J_e w\dot{(}t) + B_e w(t) = K_a K_t V_{in}(t) - T_d(t)$$

$$w\dot{(}t) = \frac{K_a K_t}{J_e} V_{in}(t) - \frac{1}{J_e} T_d(t) - \frac{B_e}{J_e} w(t)$$

$$w\dot{(}s) = \frac{K_a K_t}{J_e} V_{in}(s) - \frac{1}{J_e} T_d(s) - \frac{B_e}{J_e} w(s)$$

$$\begin{bmatrix} w\dot{(}s) \\ X_a\dot{(}s) \end{bmatrix} = \begin{bmatrix} -\frac{B_e}{J_e} & 0 \\ K_e & 0 \end{bmatrix} \begin{bmatrix} w(s) \\ X_a(s) \end{bmatrix} + \begin{bmatrix} \frac{K_a K_t}{J_e} & -\frac{1}{J_e} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{in}(s) \\ T_d(s) \end{bmatrix}$$

$$\begin{bmatrix} X_a(s) \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} w(s) \\ X_a(s) \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} V_{in}(s) \\ T_d(s) \end{bmatrix}$$

Now, we can simulate the results of a step-response input using our above state-space model, and our discrete transfer function found in Step 1. These results can be seen below.

One important note is the use of our friction torque. For our state-space model, we consider both Voltage & friction torque inputs for clarity. However, in our MATLAB comparison, we will ignore the friction input and focus solely on the Voltage input, as stated in our *System Analyzed* section.

Code 1: MATLAB State-Space Solution

```
% State-space matrices (continuous time)
A = [-Be/Je, 0; Ke, 0];
B = [Ka*Kt/Je, 1/Je; 0, 0];
C = [0, 1];
D = [0, 0];

% Continuous-time state-space system
sys_cont = ss(A, B, C, D);

% First Input – Discrete State-Space System
StateSpaceG_z_Xa = ss(StateSpaceG_z.A, StateSpaceG_z.B(:,1), StateSpaceG_z.C,
StateSpaceG_z.D(:,1), Ts);
```

Our Step-Response simulations, alongside our function bode plots for our state-space model and our MATLAB C2D model can be seen below.

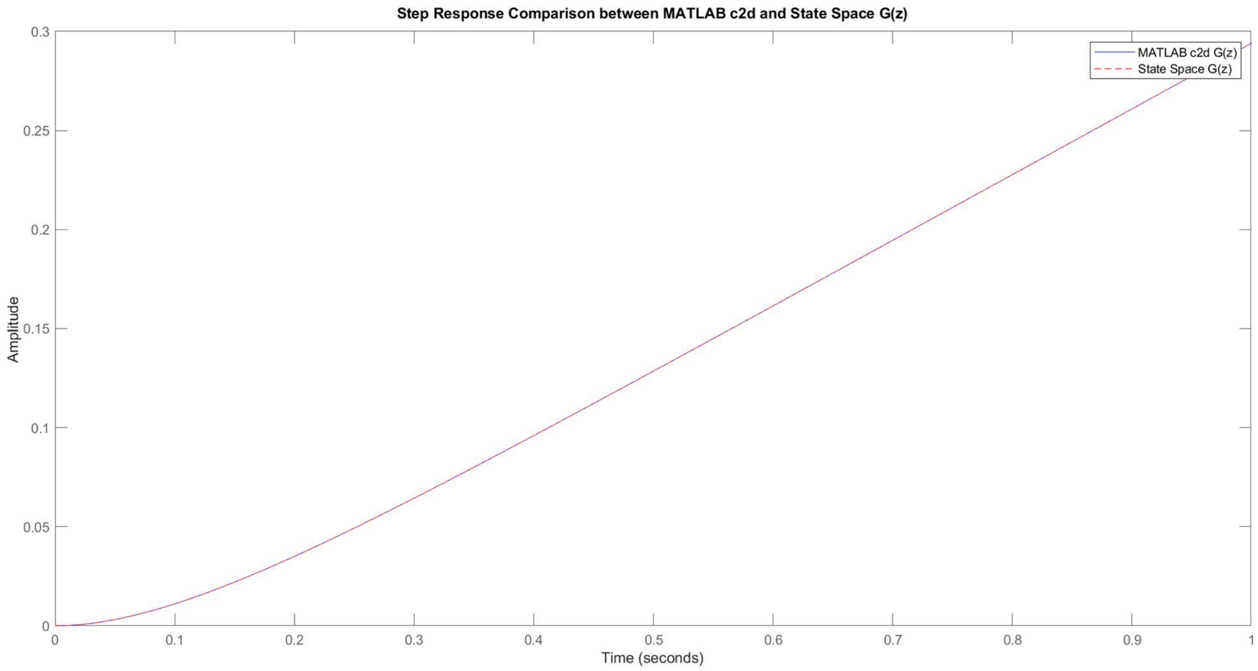


Figure 7: Step Response Comparison for State-Space Model & C2D Function

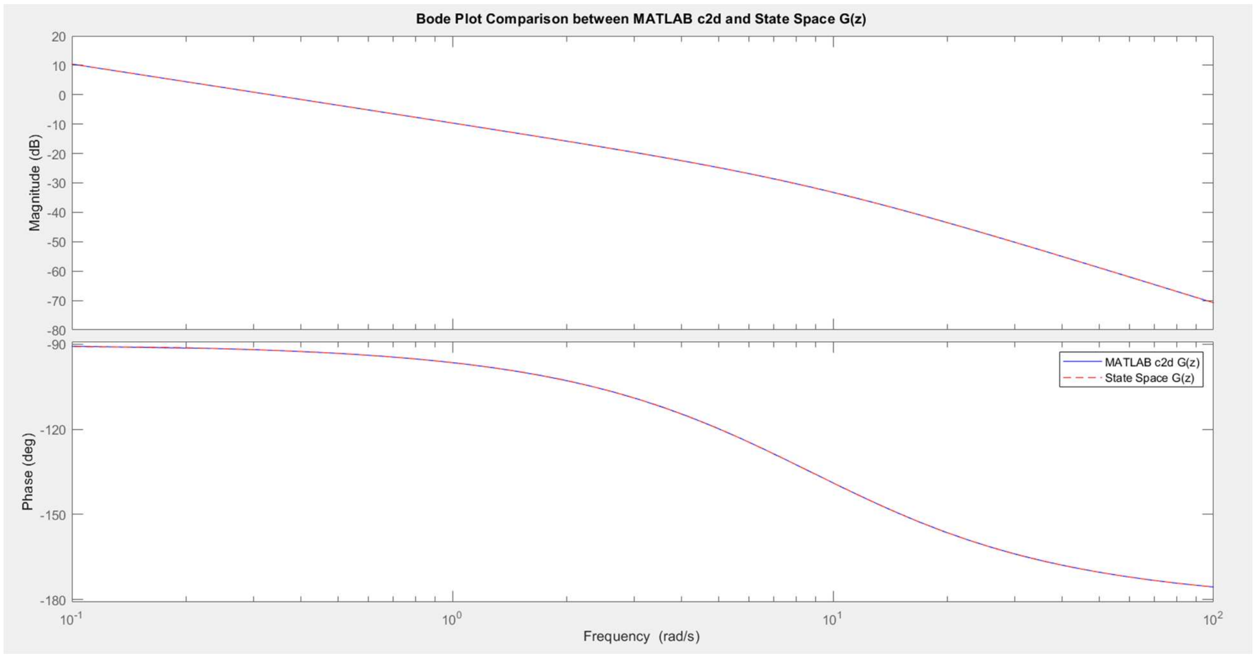


Figure 8: Bode Plot Comparison for State-Space Model & C2D Function

As we can see, our responses align perfectly, confirming this state-space model.

Prelab 3 - Stability Analysis

Assuming Closed-Loop P Controller, plot Root Locus of the drive Gol in s & z plane and observe how closed-loop poles changes as Kp increases from 0 to infinity. Derive the basic expressions manually.

Root locus is the graphical representation of the possible locations for closed loop poles for varying values of a certain system parameter. For our open loop system, our transfer function is G(s). For our closed-loop system with unity feedback, we can derive our basic continuous transfer function as follows.

Equation 7: Closed-Loop Transfer Function

$$G_{closedloop}(s) = \frac{K_p G_{ol}(s)}{1 + K_p G_{ol}(s)}$$

In simplistic terms, our closed-system poles, which determine the behaviour of our system, will change as Kp changes. Root locus will show us how our system & our closed-loop poles will change as Kp increases from 0 to inf. Our Root-locus solutions for our system in the s-plane & z-plane can be seen below.

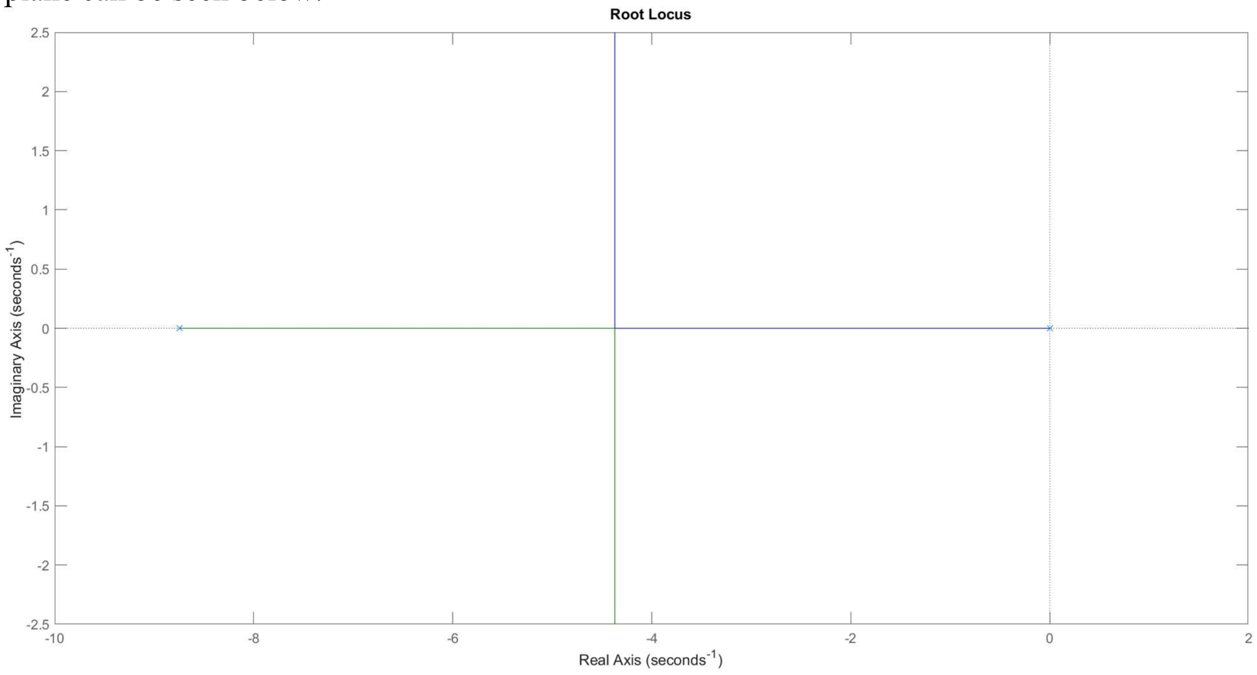


Figure 9: S-Plane Root Locus Diagram

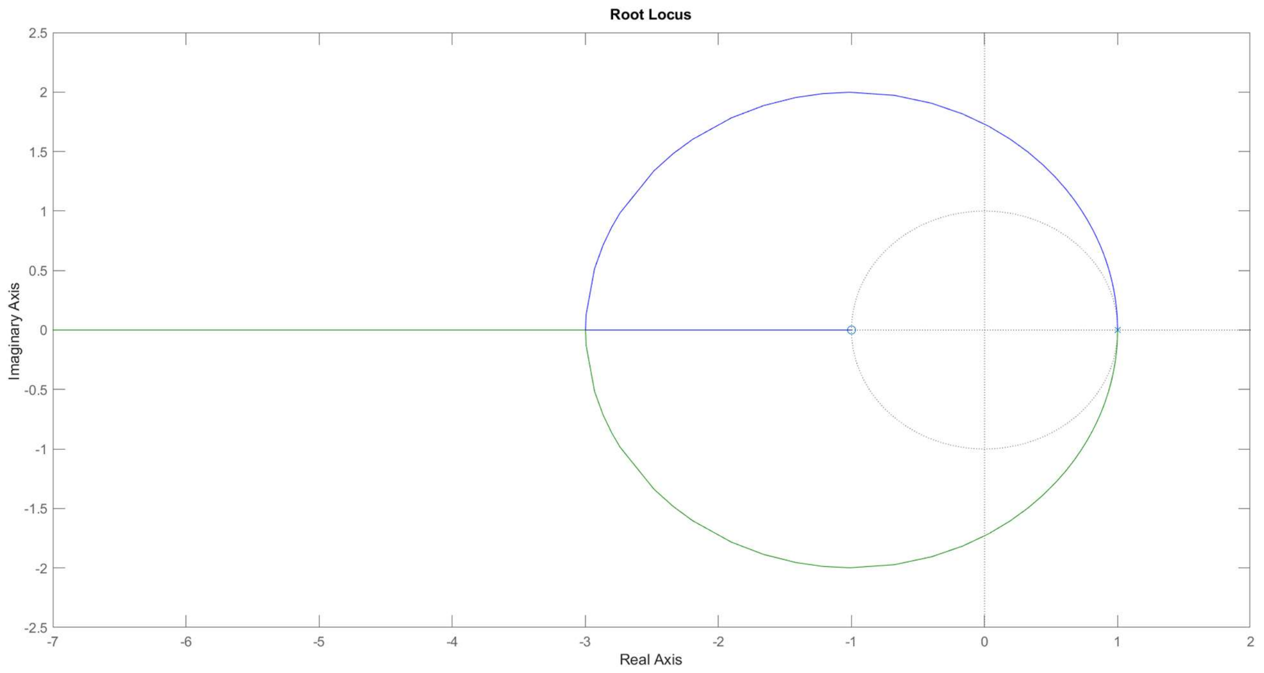


Figure 10: Z-Plane Root Locus Diagram

We can also derive the basic expressions for the Root Locus manually, as seen in the following equations.

Equation 8: Manual Derivation for Root Locus

$$G_{ol}(s) = \frac{K_p K_a K_t K_e}{s(J_e s + B_e)}$$

$$G(s) = \frac{G_{ol}(s)}{1 + G_{ol}(s)} = \frac{K_a K_t K_e K_p}{J_e s^2 + B_e s + K_a K_t K_e K_p}$$

$$\text{Poles} = \frac{-\mathrm{B_e} \pm \sqrt{\mathrm{B_e^2} - 4 J_e K_a K_t K_e K_p}}{2J_e}$$

$$\text{If } \mathrm{K_p} = 0 \rightarrow \frac{-\mathrm{B_e} \pm B_e}{2J_e}$$

$$\mathrm{P_1} = 0; P_2 = -\frac{B_e}{J_e}$$

$$\text{If } \mathrm{K_p} = \text{inf} \rightarrow \frac{-\mathrm{B_e} \pm \sqrt{-inf}}{2J_e}$$

$$\frac{-B_e \pm \mathrm{j(inf)}}{2J_e}$$

$$\mathrm{P_1} = -\frac{B_e}{J_e} - j * inf; P_2 = -\frac{B_e}{J_e} + j * inf$$

Find the phase and gain margins of Gol(s) and Gol(z) in the s-plane & z-plane. Comment on the stability in the closed loop systems.

Below is the Bode Plots for G_OpenLoop in the s-plane & z-plane.

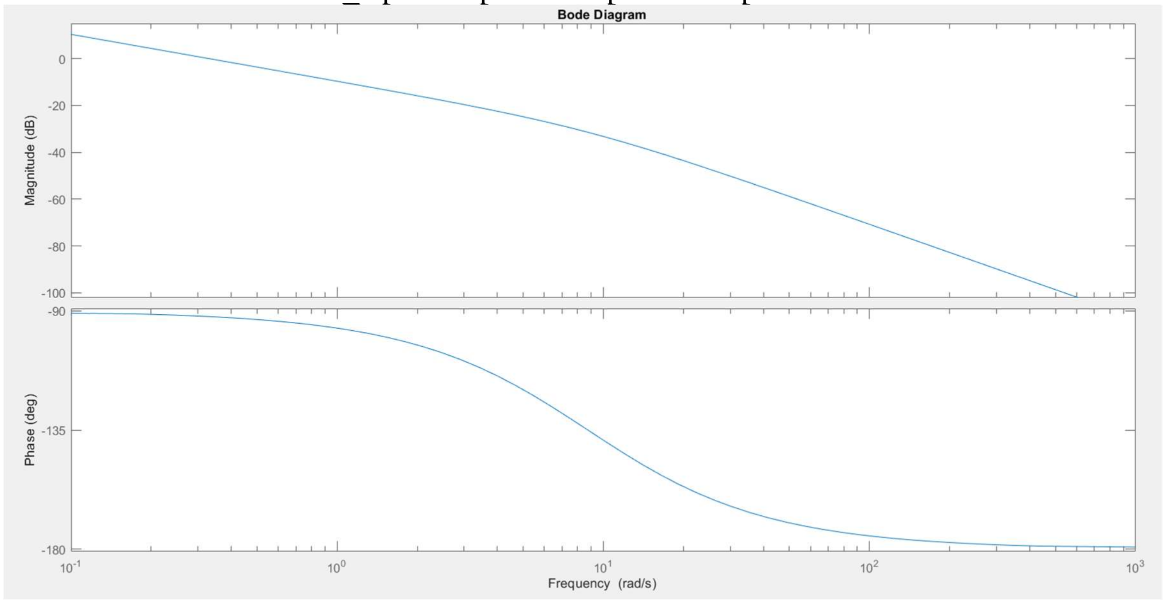


Figure 11: S-Plane Open-Loop Bode Plot

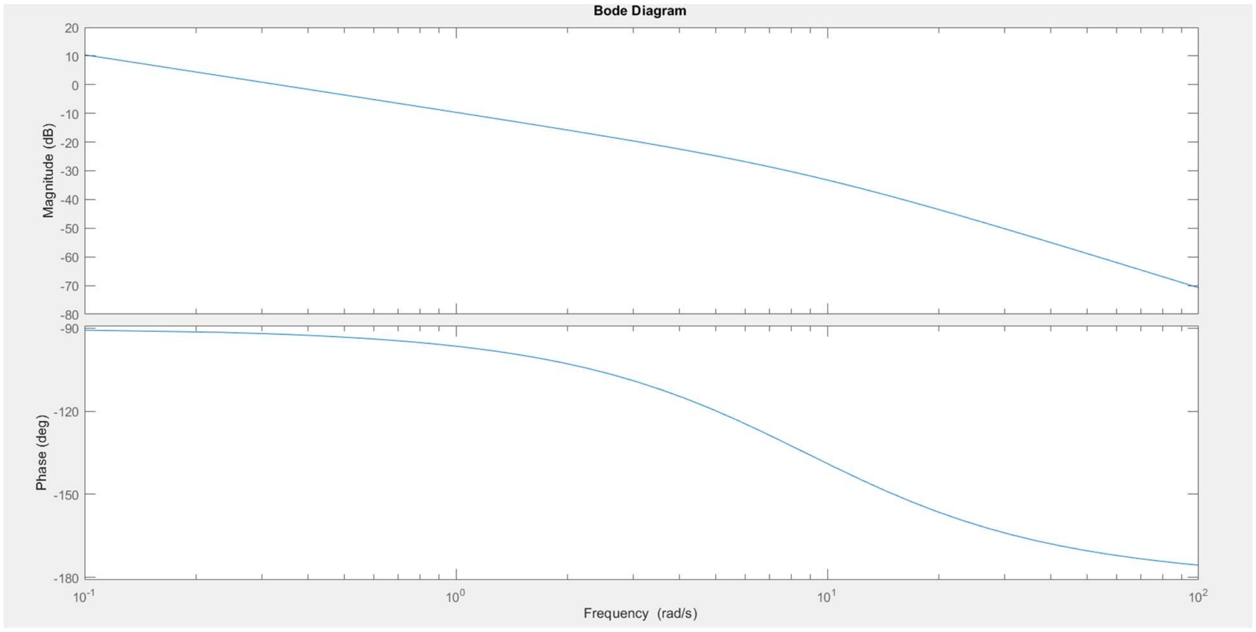


Figure 12: Z-Plane Open-Loop Bode Plot

As we can see, our bode plots for the continuous and discrete systems are the same. Since they are the same transfer function in different time domains. To determine closed-loop stability, we will observe both our bode plots and our root locus diagrams as done below.

In this S-plane, our poles begin in the negative real axis and 0 imaginary axis with Kp=0. As Kp increases, our poles begin to diverge and come closer together, until they meet at the origin of the real and imaginary axis. As Kp continues to increase past this point, our poles will diverge again, with decay rate=0 and our oscillation increasing with a steady state-response. In this situation, our system is always stable. Stable Systems can be defined as any poles where real value is negative.

In the z-plane, our poles begin with a positive real stable pole at 1. As our Kp increases, our poles will travel around the circle, representing higher oscillatory systems. At some point, our poles will begin to diverge, with some poles exiting the stable circle area, telling us that our system will become unstable at some high Kp value. Stable Systems can be defined as any poles where inside the radius of our defined system.

In short, our system will always be stable in the continuous frequency domain. As Kp increases, our decay and oscillatory responses of our system will change. In the discrete range, our decay and oscillatory responses will also change. However, at some high Kp value, our system will stop being stable and become unstable, due to the speed of our ZOH sampling rate compared to the frequency of the system itself.

Finally, the gain and phase margins of the system can be seen in the following table.

Table 2: Gain & Phase Margins of Gol in S-plane & z-plane

System	Phase Margin (deg)	Gain Margin (dB)
Gol(s)	10	Inf
Gol(z)	9	30

Discussion: Is stability in continuous and discrete domains always equivalent? Why? Using MATLAB, find the gain margin of Gol(z) for three different sampling time of 0.02, 0.002, and 0.0002. Which one is more stable? What do you conclude?

The Bode Plots for G_OpenLoop(z) for three different Sampling Times are shown below.

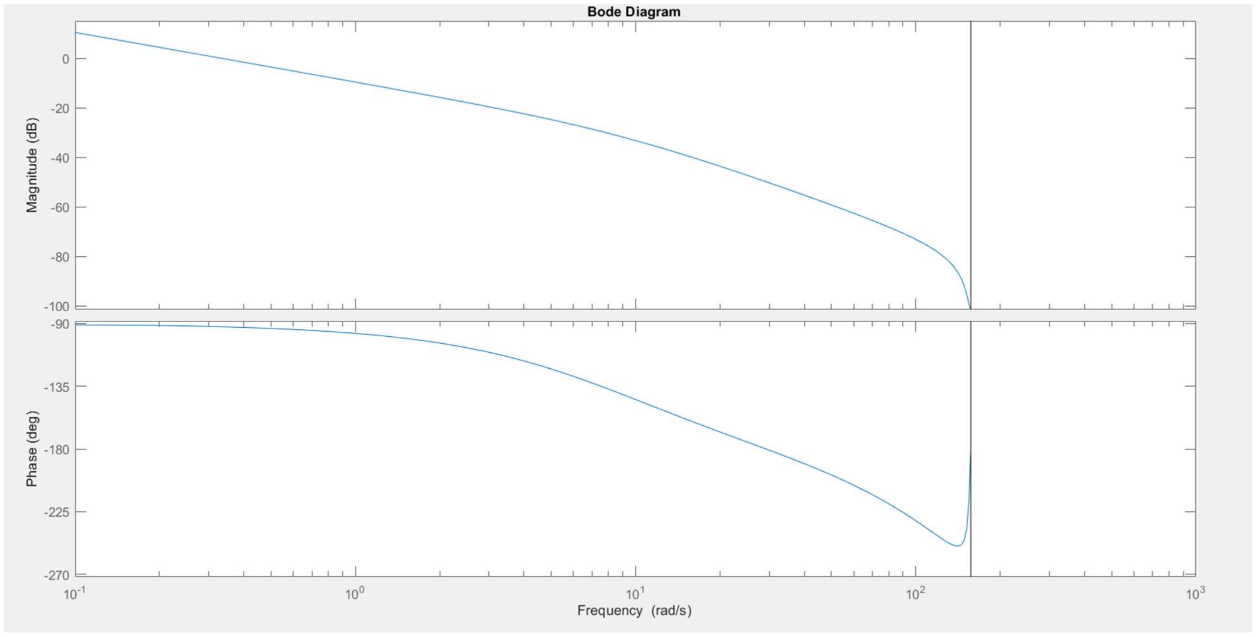


Figure 13: Z-Plane Open Loop Bode Diagram for Sampling Time = 0.02s

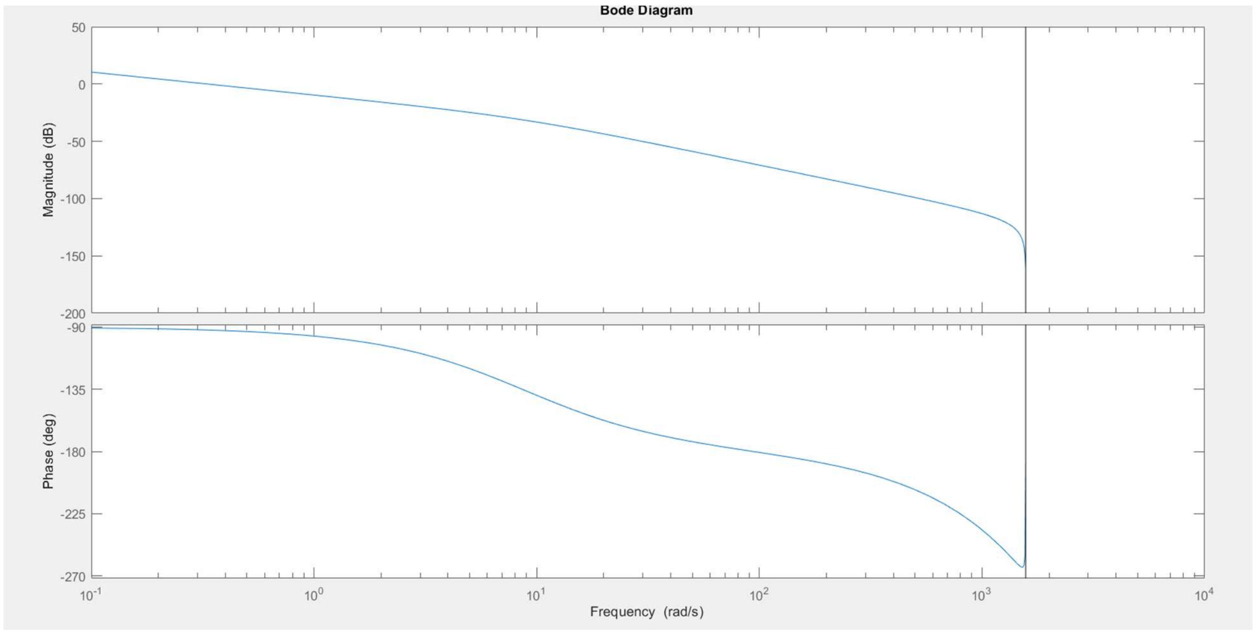


Figure 14: Z-Plane Open Loop Bode Diagram for Sampling Time = 0.002s

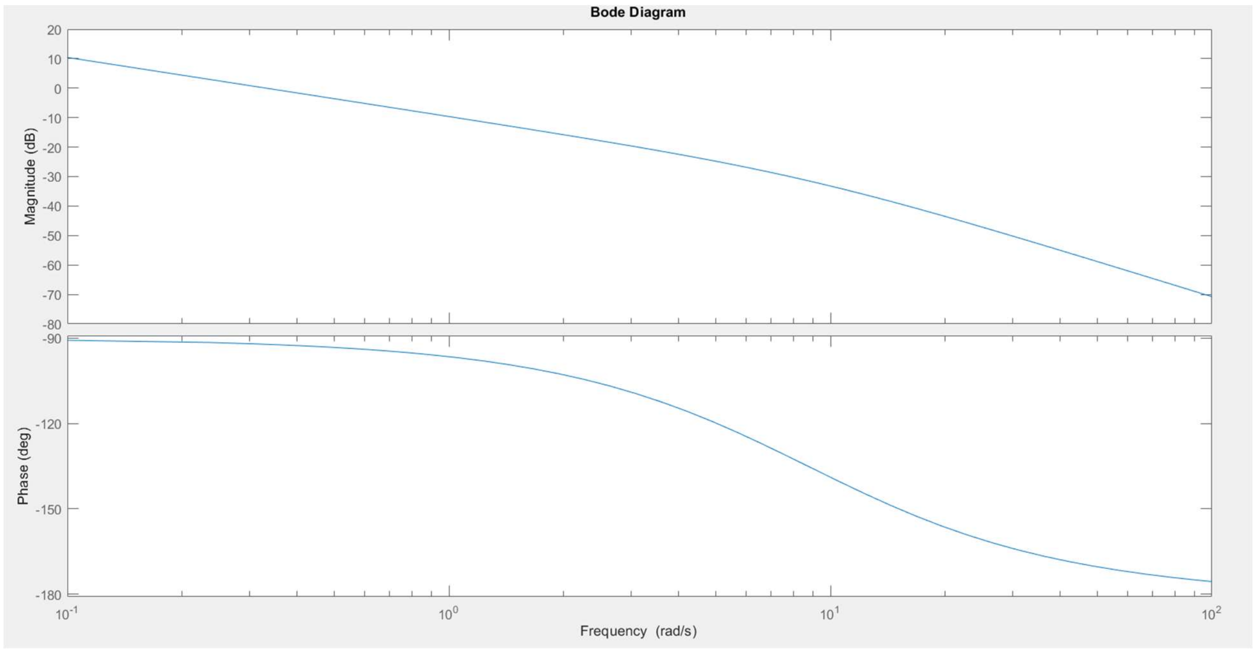


Figure 15: Z-Plane Open Loop Bode Diagram for Sampling Time = 0.0002s

The gain margins for the three values are listed in the table below. It was found by finding the difference between 0db and the gain at the point where the frequency crosses -180deg.

Table 3: Gain Margins at Various Timesteps

Time Step (s)	Gain Margin
0.02	-10.6
0.002	9.68
0.0002	27.8

As stated earlier, stability in the continuous and discrete domains are *NOT* the same. For our system, our continuous system is always stable, but our discrete system is not at some high Kp values. As Kp increases, our system will oscillate faster and our sampling time will not be able to catch up, leading to aliasing, causing our discrete system to "see" high-frequency values, leading to unstable behaviours. We need our sampling rate to be at least twice the highest frequency in the system, but as Kp increases, our sampling time will be low for this criterion, which leads to instability from our analog to digital conversion.
For the three plots above, we can determine that our third figure is the most stable. Our final system has a longer operating range and gain margin, showing a more stable system. As our sampling time increases, the gain at our crossover frequency increases, leading to a higher gain margin as we compare our crossover frequency gain to 0DB. A higher gain margin demonstrates a more stable system. This confirms our previous solution, where a higher sampling frequency leads to better ZOH analog to digital conversions and more stable systems.

Prelab 4 – P Controller Design

Using G_OpenLoop, find proportional gain Kp such that unity gain crossover frequency in z-domain is 60rad/s.

To understand how to find Kp properly, let's go into the relationship between Kp, Gcl, and Gol a bit further. As stated earlier, our relationship between our closed-loop system, Kp, and our open-loop system is as follows.

Equation 9: Closed-Loop Transfer Function

$$G_{closedloop}(s) = \frac{K_p G_{ol}(s)}{1 + K_p G_{ol}(s)}$$

In part 3, we discussed our Kp will affect the poles of our system and change system stability. However, Kp will also affect our system bandwidth. As lower frequencies, our gain function is approximately 1, representing our operating bandwidth range. As Kp increases, our bandwidth increases. However, as found in part 3, our poles also changes as Kp increases. If we increase our Kp too much, we will cause instability at higher frequencies, causing system issues.
In short, we want to increase Kp to increase our system bandwidth, but not too high so that we cause system instability.
As seen in equation 8, our Kp is directly apart of our Loop Return Ratio, L. As L increases, our bandwidth increases. The frequency at which our Loop Return Ratio crosses a gain of 1 is called our unity gain crossover frequency. At this frequency, our gain will begin to decrease. At this frequency, our system will have a specific phase. If our phase at unity gain crossover is too close to -180, instability will occur.
In short, we can use our relationship between Kp, unity gain crossover frequency, and output gain to determine Kp. Afterwards, we will check the phase margin at the given Kp to ensure it isn't close to -180deg. We would check phase margin by finding the phase where the gain is 0DB; our unity gain cross over frequency. At this point, we would determine the phase and compare it to -180deg to determine phase margin.
Using the relationships defined above, we can calculate Kp. Our transfer function for our system is defined below, as found earlier in the prelab. We can enter our s=0+jw value into our transfer function to determine our Kp_threshold value.

Equation 10: Kp Threshold Solution with s=jw=60j

$$G(s) = \frac{K_a K_t K_e}{s(J_e s + B_e)}$$
$$G(60j) = \frac{K_a K_t K_e}{60j(J_e * 60j + B_e)}$$
$$k_p = \frac{1}{G(60j)} = 1.252$$

To find our phase margin, let's plot our Loop Return Ratio, which is our plant & our Kp controller combined, and find where our gain is 0.

Equation 11: Loop Return Ratio Equation

$$LRR(s) = K_p \frac{K_a K_t K_e}{s(J_e s + B_e)}$$

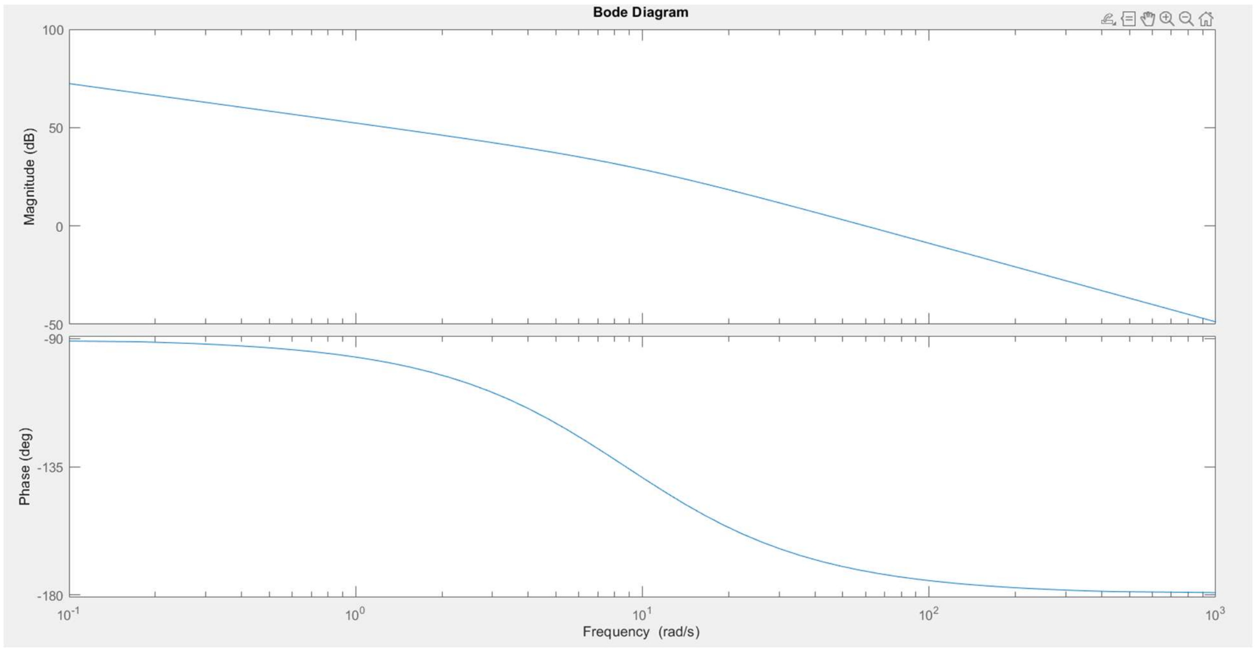


Figure 16: LRR Bode Plot & Phase Margin

Using MATLAB, I extracted our current plant phase, which is -180.8312, which tells us that our system can easily become unstable. In the future, we'd like to add more phase at this location to compensate our small phase margin.

Create a Simulink model for our system with P-Controller including zero order hold. Include Coulumb friction & Saturation Blocks.

A Simulink model was created for the closed-loop system with a p-controller, as shown in the following figure.

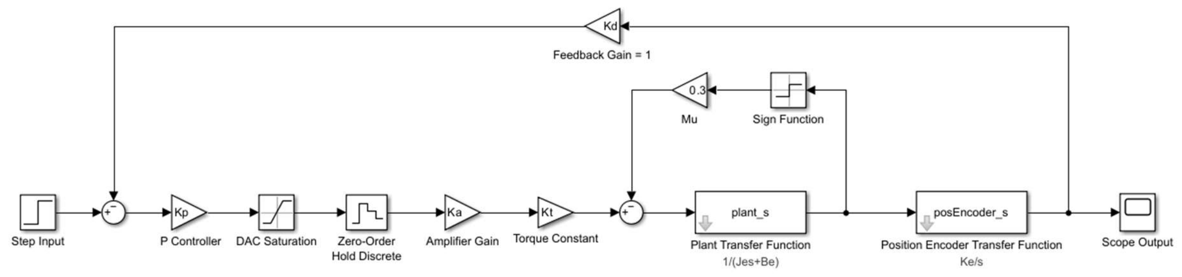


Figure 17: Simulink P-Controller System Model

Obtain step response of digital system with and without friction models. Comment on Coulumb friction effect on overshoot, rise time, and settling time.

Using this model, I will determine the step response of the digital system with and without the friction model. We will use our Kp value determined previously, and constant values defined in the *Lab Manual*. To better comment on Coloumb Friction's effects, we will use multiple other friction values besides 0 and 0.3

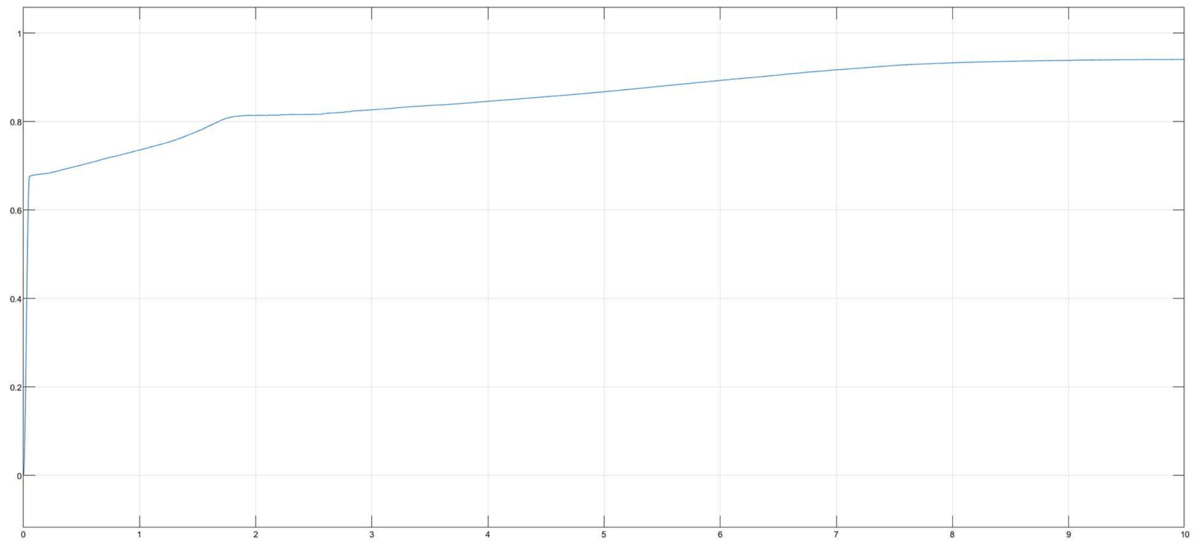


Figure 18: Step Response Output for System with Coulomb Friction = 0.5Nm

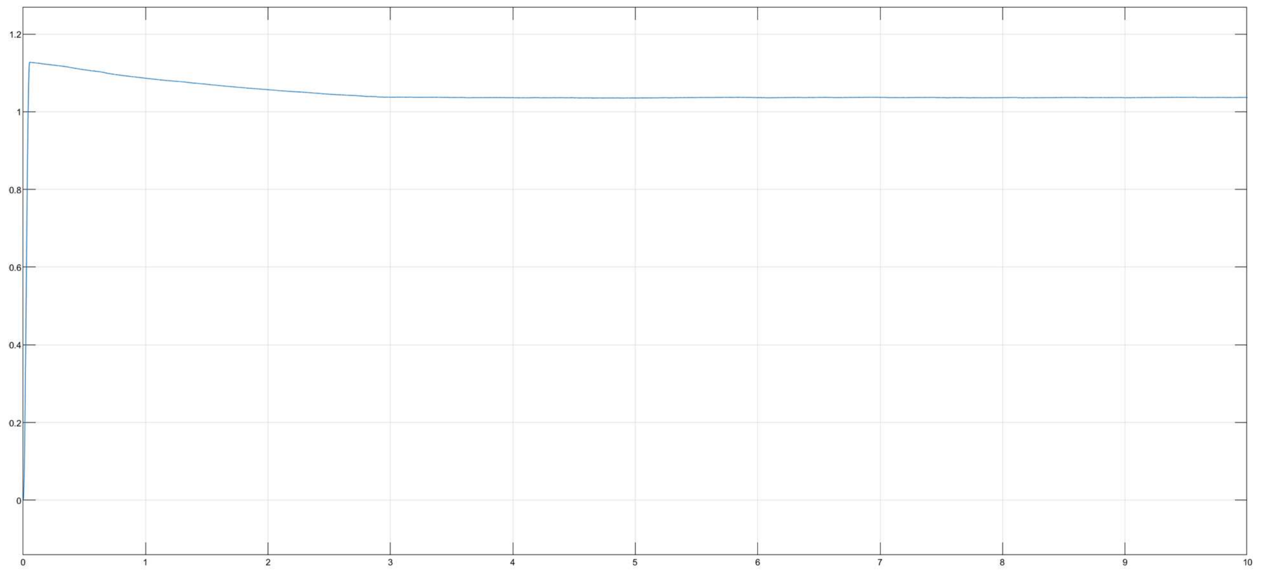


Figure 19: Step Response Output for System with Coulomb Friction = 0.3Nm

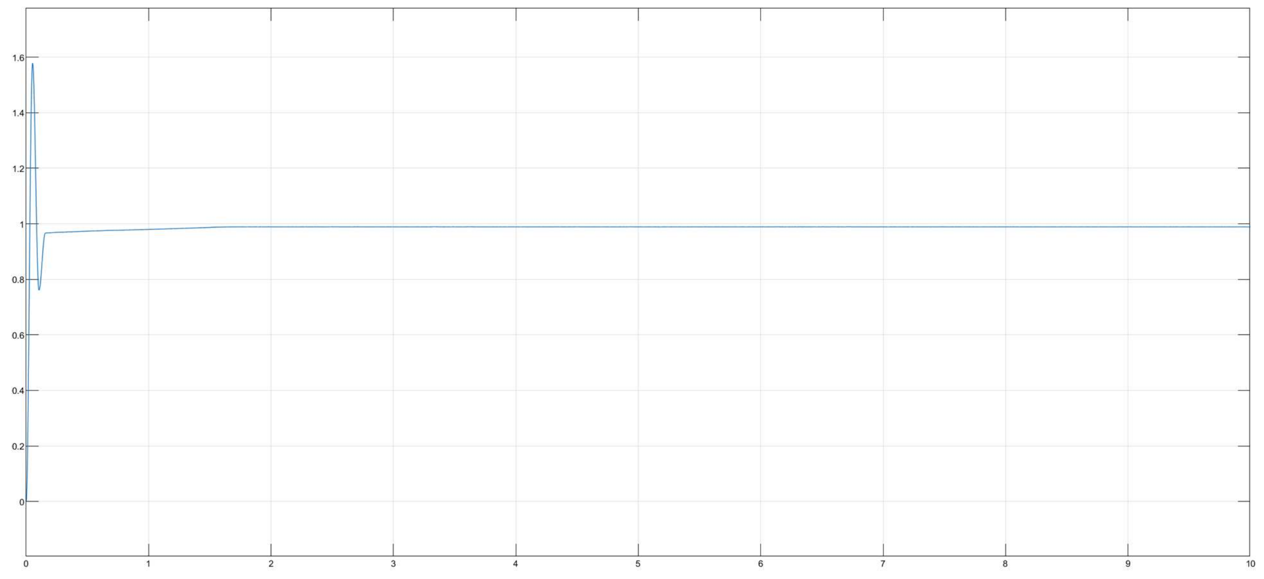


Figure 20: Step Response Output for System with Coulomb Friction = 0.1Nm

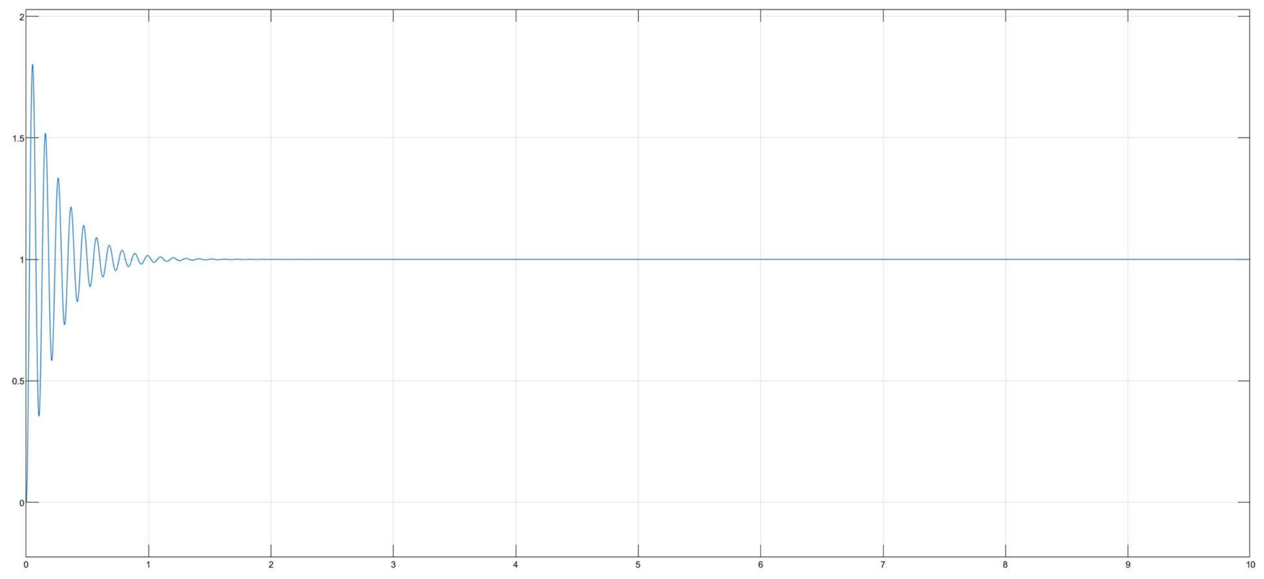


Figure 21: Step Response Output for System with Coulomb Friction = 0Nm

As seen in the following graphs. Coulomb Friction plays a major role in overshoot, rise time, & settling time.

- When Coulomb Friction is high, we have an overdamped system, where settling time is high, rise time is high, and overshoot doesn't occur, as the system slowly builds up to our final value.
- When Coulomb Friction is low, we have an underdamped system, where settling time is faster, rise time is faster, and overshoot does occur, as the system oscillates as its gets to our final position.

- For Coulomb Friction = 0.3, our system overshoots a bit, but decreases to its steady-state value, showing an underdamped system that's closer to a perfectly damped system as there are no oscillations.

How does saturation block affect overshoot, rise time, and settling time.
Similar to the last question, we will provide plots of multiple outputs with different saturation limits. Our coulomb friction constant will remain at 0.3.

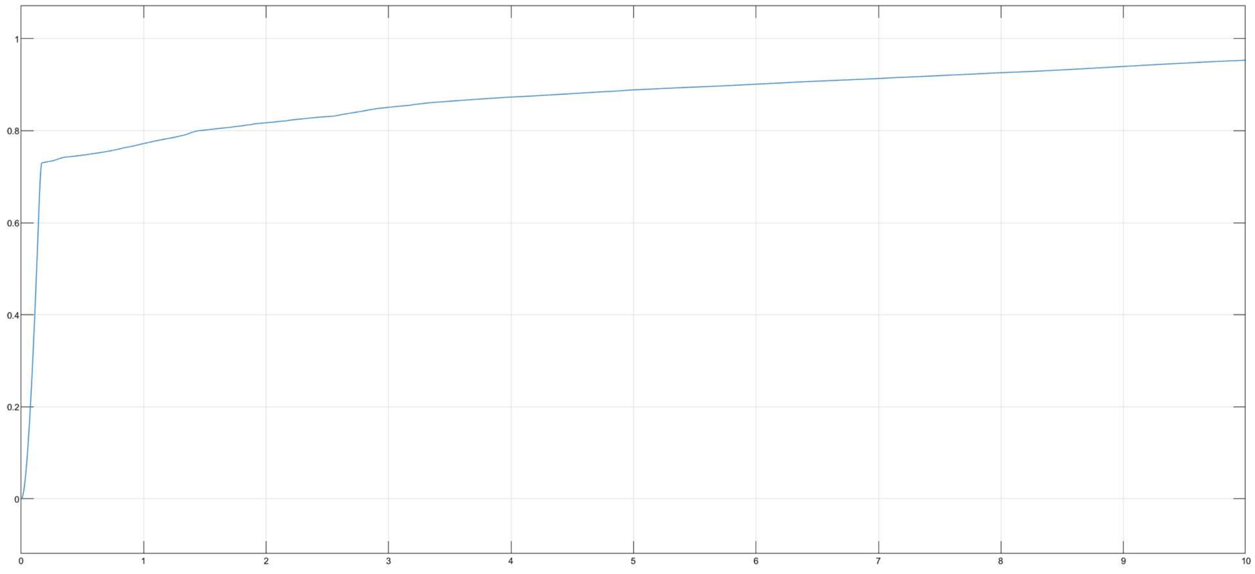


Figure 22: Step Response Output for System with Saturation = +-0.5

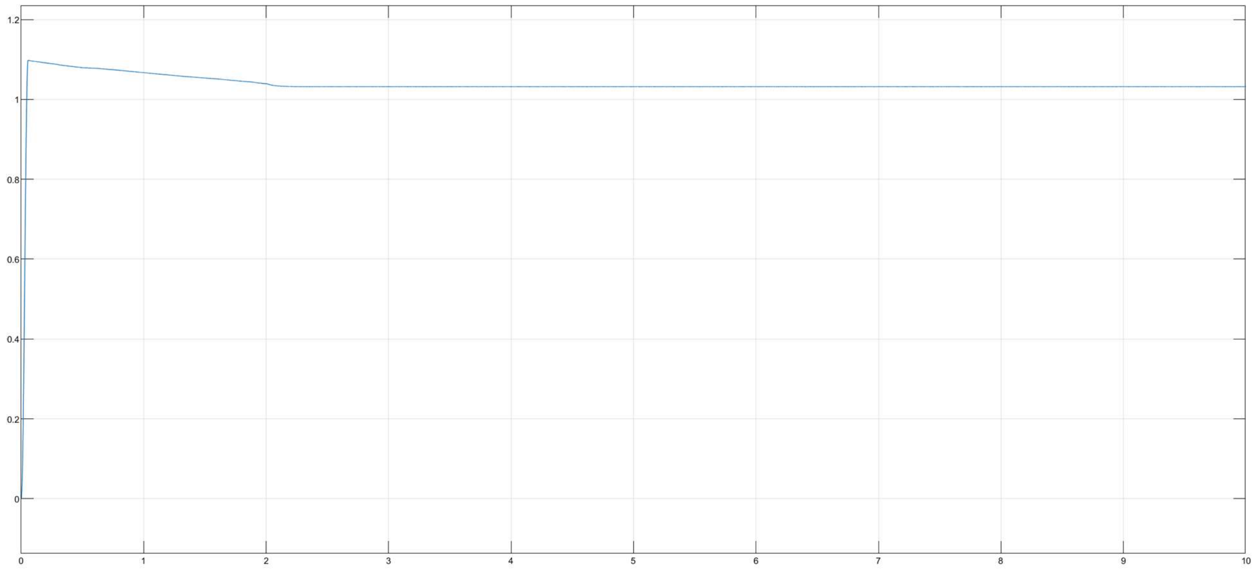


Figure 23: Step Response Output for System with Saturation = +-1

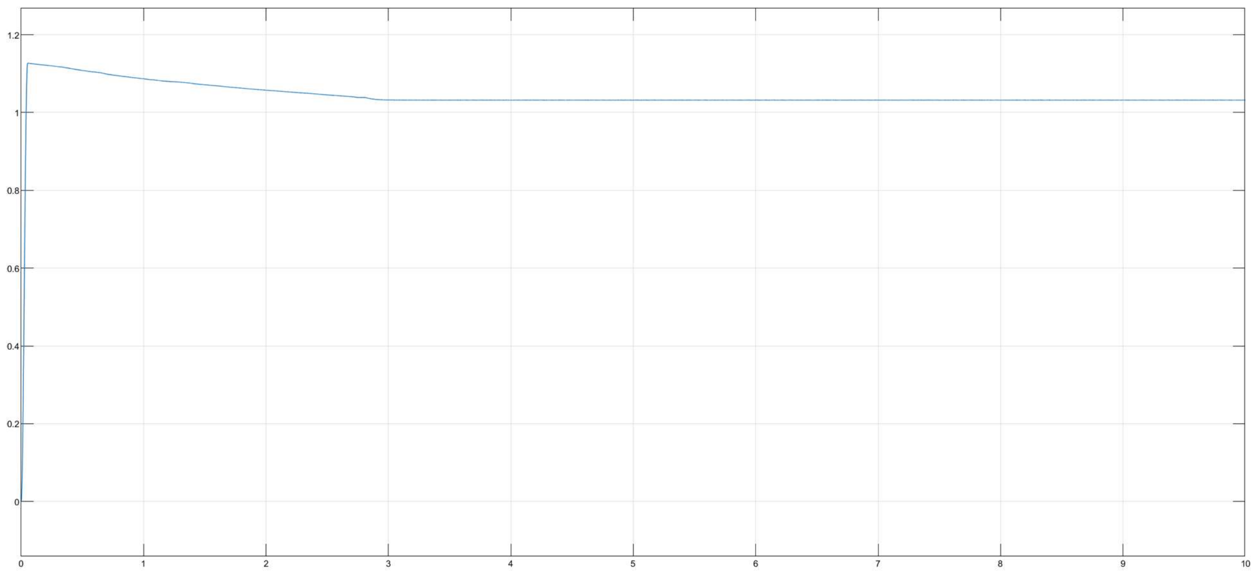


Figure 24: Step Response Output for System with Saturation = +-3

The saturation block tries to model the physical limits of a system. In this system, it limits the input voltages into the system so it could match what the actuator and hardware can deliver. At lower limits, the system might try to hit these higher voltages, but are unable to, leading to slower overall responses, like a damped system.
We can use this information to determine how saturation affects rise time, settling time, and overshoot. At lower limits, our rise time and settling times are slow, with no overshoot occurring, similar to a damped system. At higher limits, our system will react faster, with possible overshoot occurring. However, the system will be limited by the other friction constants into the system. For example, with higher saturation blocks, our output is still the same, as seen in Figure 21 and 22. This is because our system is limited by the Coulomb Friction still.

Prelab 5 – Lead Lag Compensator Controller Design

Determine Lead Compensator Parameters if gain crossover frequency is 377rad/s. Plot Loop Return Ratio System to confirm design criteria is met.

Lead compensators can be designed to add additional phase near the unity-gain cross over frequency. As we saw in Part 4, we had a small phase margin at cross-over, which can lead to instability at faster sampling times. We can use a lead compensator to add additional phase, preventing this.
Our target crossover frequency is 377 rad/s. Our target phase margin is 60deg, as per the *Lab 2 Manual*. We found our current phase in Part 2, which is -172deg. Using these values, we can determine the required peak phase of our lead compensator using the following equation.

Equation 12: Peak Phase Calculation

$$\emptyset_c = \emptyset_m - \emptyset_p - 180°$$
$$\emptyset_c = 60° - (-180.831) - 180°$$
$$\emptyset_c = 60.831°$$

Using our required peak phase, we can calculate our required lead controller parameters.

Equation 13: Lead Compensator Parameter Calculations

$$\alpha = \frac{1+\sin\emptyset_c}{1-\sin\emptyset_c} = \frac{1+\sin 60.831°}{1-\sin 60.831°} = 14.7711$$
$$\tau = \frac{1}{w_c\sqrt[2]{\alpha}} = \frac{1}{377\sqrt[2]{14.7711}} = 6.9016 * 10^{-4}$$

Using these values, we can calculate Proportional Gain Kp for this system. We can calculate Kp using the following equation.

Equation 14: Lead Compensator Proportional Gain Calculation

$$|L(jw_c) = 1 = |KC_0(jw_c)G(jw_c)|$$
$$C_0(377j) = \frac{\alpha\tau(377j)+1}{\tau(377j)+1}$$
$$G(377j) = \frac{K_aK_tK_e}{60j(J_e * 60j + B_e)}$$
$$k_p = \frac{1}{|C_0(60j)G(60j)|} = 12.7376$$

To confirm that our lead lag compensator is working as expected, we can plot our Loop Return Ratio Bode Plot, as seen in the following figure.

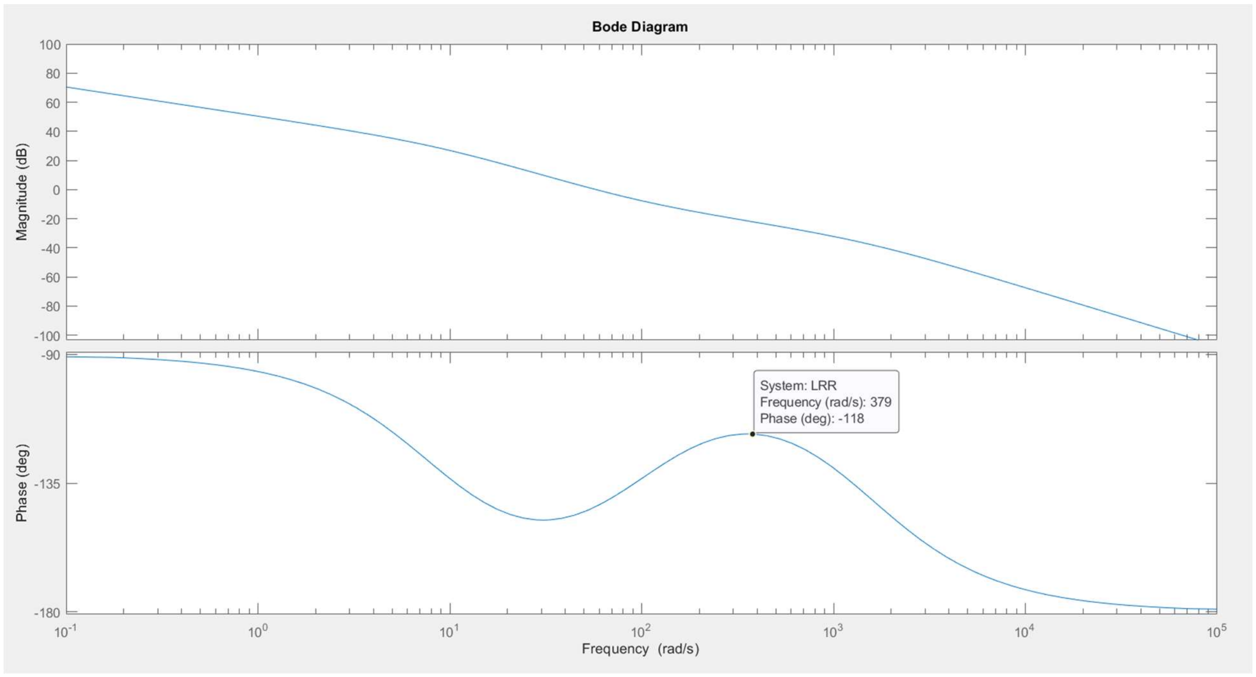


Figure 25: Lead Lag Compensator LRR Bode Plot

As you can see in the above plot, we have injected phase into our system at our crossover frequency, which removes the stability issues and positive feedback issues we had earlier.

Add Integral action (ki+s)/s to lead lag compensator. Ki=wc/10. Simulate Step & ramp input responses with friction and show the effect of integral action on steady-state input. Compare results with and without integral controller.

First, let's find the results of our system using the lead-lag compensator alone. We use the block diagram function shown in the following figure.

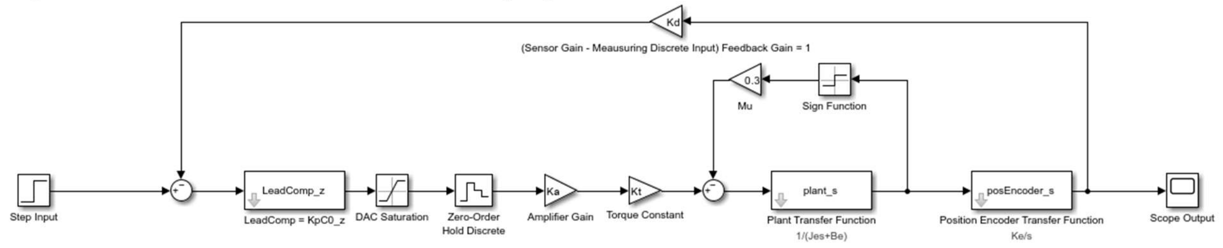


Figure 26: Lead Compensator Simulink Model

Our output results for a step-response and ramp response can be seen below. Our input response signal is yellow, and our output response signal is blue.

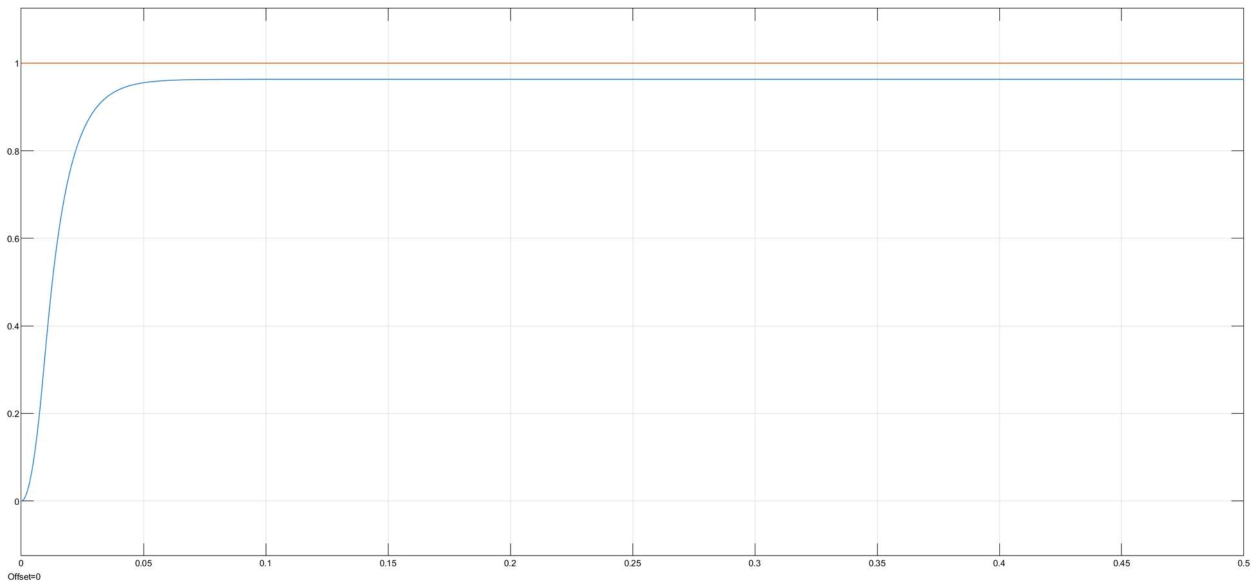


Figure 27: Step Response for Lead Compensator System

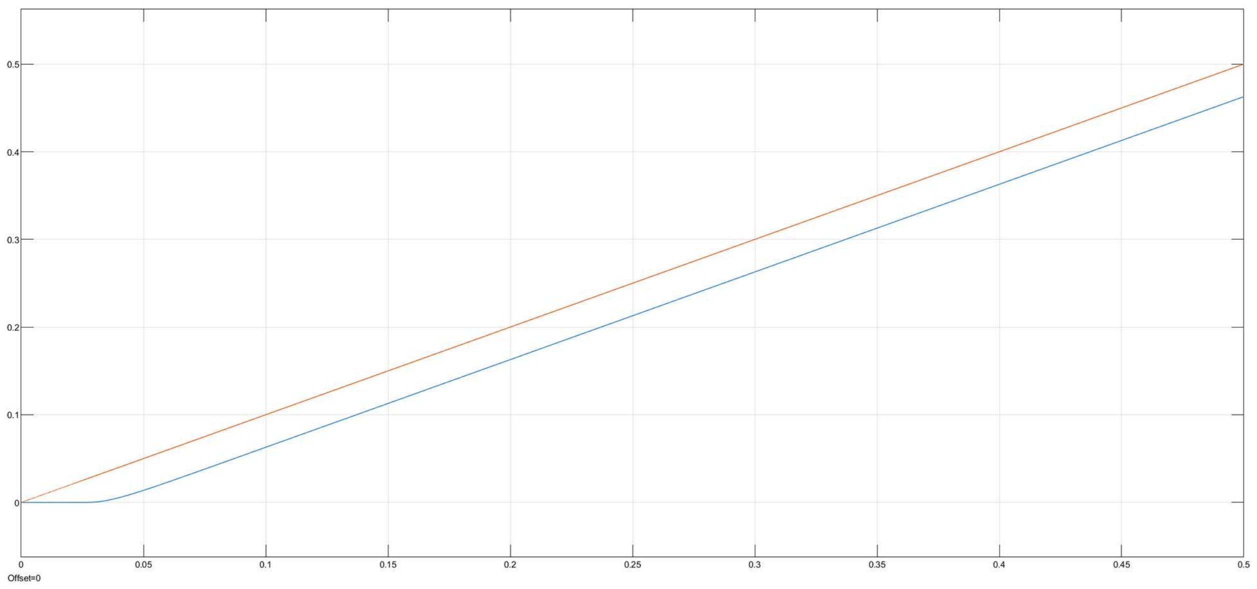


Figure 28: Ramp Response for Lead Compensator System

Our integration block diagram can be seen below. Notice how the lead compensator gain block is multiplied by the integrator function.

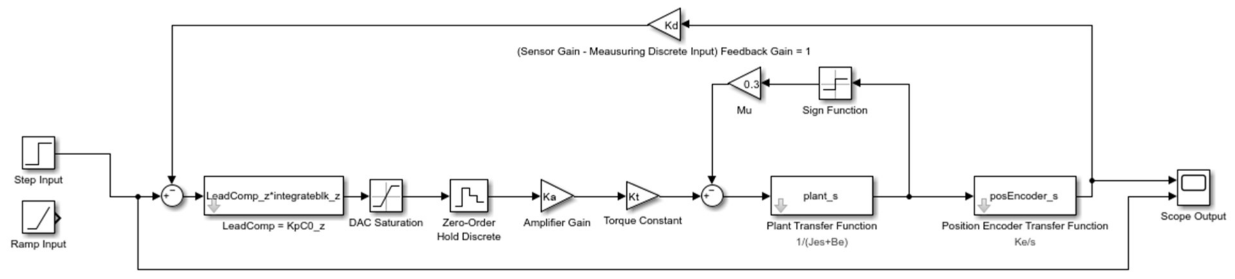


Figure 29: Lead Compensator + Integration Block Simulink Model

Our output results for this integrator system for a step-response and ramp response can be seen below. Our input response signal is yellow, and our output response signal is blue.

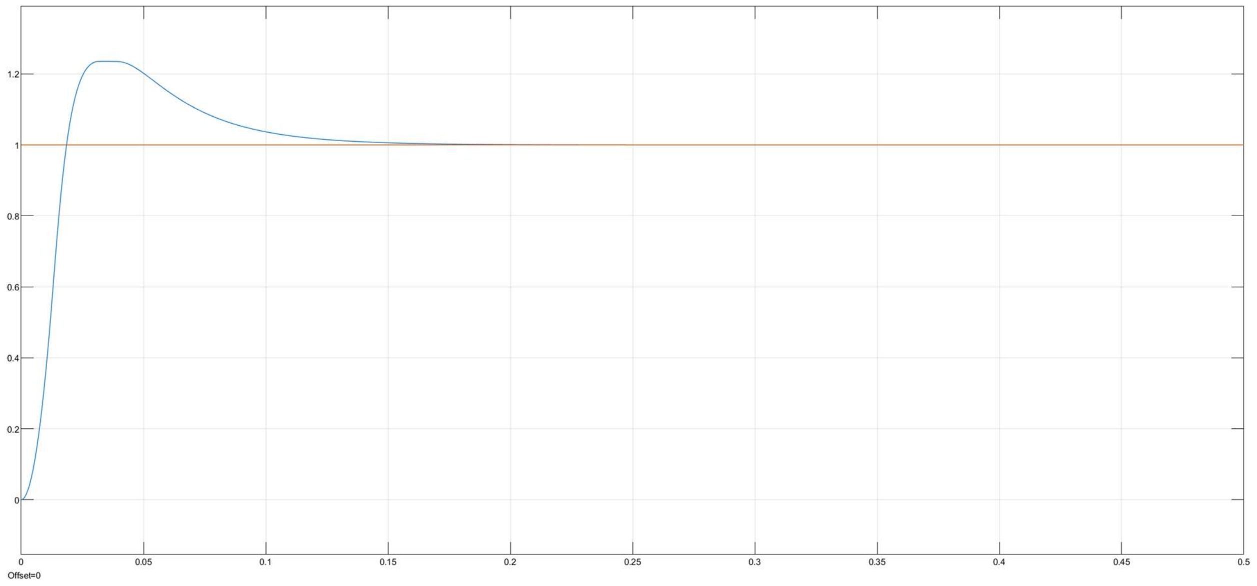


Figure 30: Step Response for Lead Compensator + Integration Block System

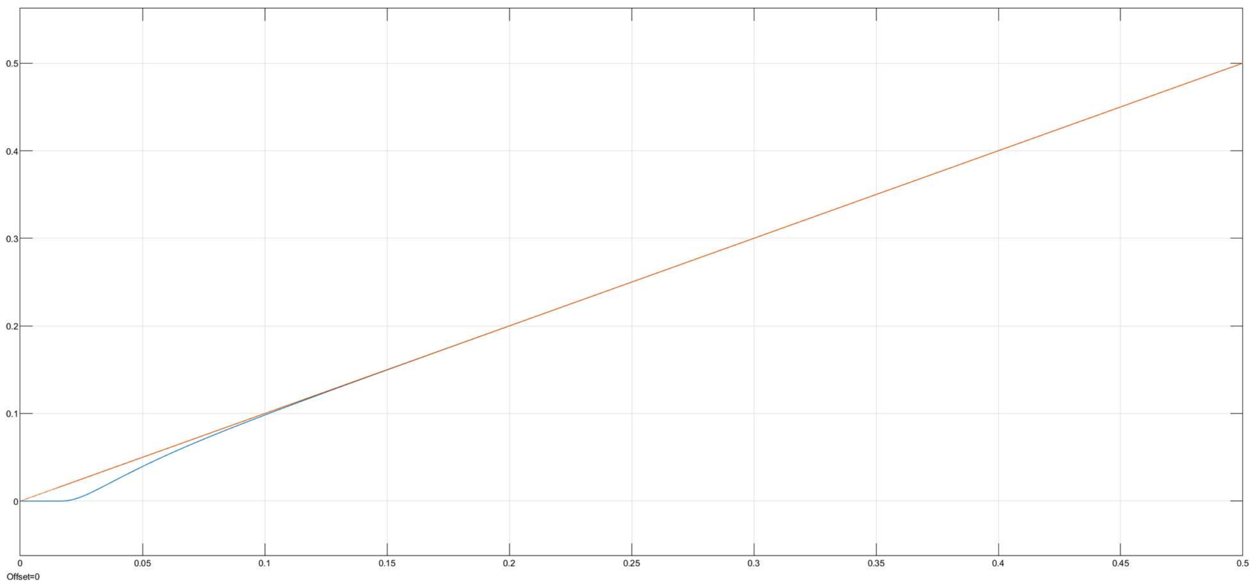


Figure 31: Ramp Response for Lead Compensator + Integration Block System

As we can see, the integration transfer function affects our system dramatically. For the original lead-compensator system without integration. Our output results match our input expected value, but our final steady-state values never fully hit our expected values. I assume that is due to friction. When we implement our integration transfer block, we see that our final output will match our intended input. For the step input, our final output will overshoot and come to steady-state at the intended final value. For the ramp function, our final output will have an increasing slope that decreases over time to match our final expected value.

To finalize this section, the following table shows our parameter values for all three Controller utilized in this prelab.

Table 4: Final Controller Parameters

Controller	Wc (rad/s)	Kp	Alpha	Tau	Ki
K Controller	60	1.252	N/A	N/A	N/A
Lead Compensator	377	12.7376	14.7711	$6.9016 * 10^{-4}$	N/A
Lead Compensator + Integration	377	12.7376	14.7711	$6.9016 * 10^{-4}$	37.7

Prelab 6 – Discussion

*Plot Magnitude & Phase of LRR, Lead Compensator, Lead Compensator+Integration, LRR*LL, LRR*LLI. Comment on how lead compensator and integrator affect magnitude and phase. Qualitively discuss how DC gain & gain crossover frequency affect steady-state error and rise time.*

The Bode Plots of all 5 systems can be seen below.

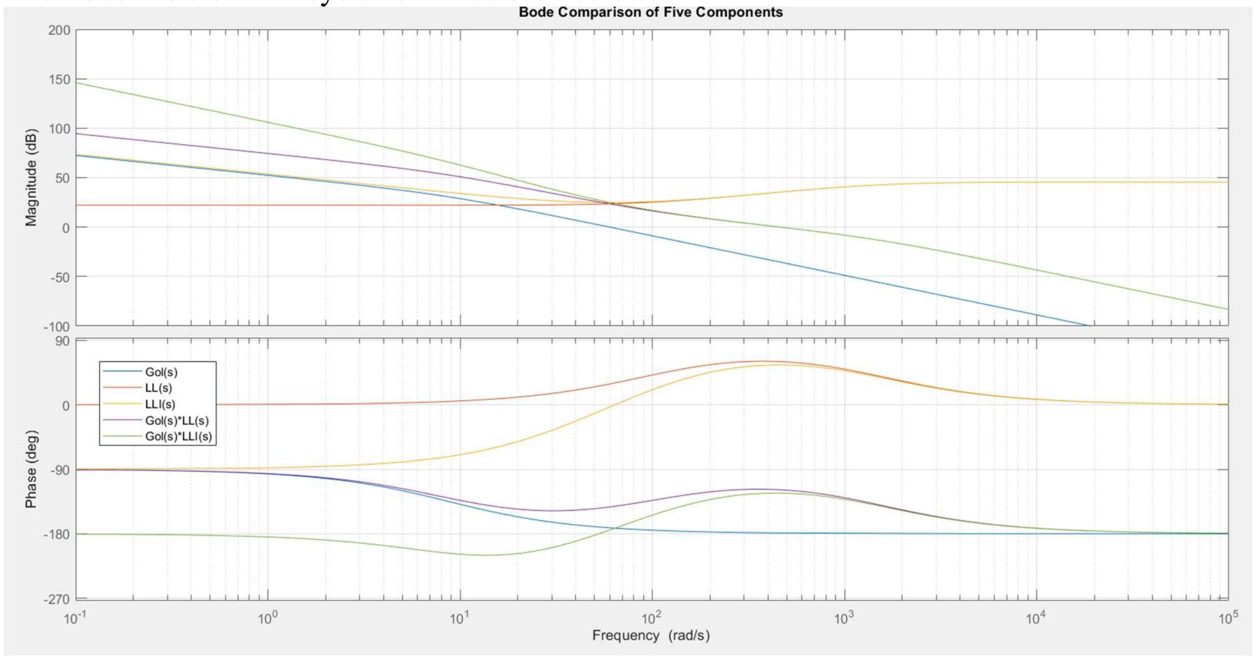


Figure 32: Bode Plots for Multiple Transfer Functions

Since the lead compensator and Integration Block is multiplied by our LRR, our Bode Responses are also multiplied by each other. However, since they are plotted on a log-log axis, it's the same as adding each bode plot together. For example, we can see that Gol*LL is the same as adding the Gol & LL transfer functions blocks together.

DC gain will reduce steady-state errors as the system can more accurately follow input functions without significant deviation. If the gain is too high, the system could respond faster, leading to faster rise times and oscillations. However, DC gain more directly impact steady-state error then rise time.

Gain Crossover frequency allows the system to handle higher frequencies while remaining stable, which will help steady-state error being minimized as the system is stabilized. A higher gain crossover frequency will lead to a system more responsive to changes, reducing rise time. In general, gain crossover frequency more directly impacts rise time than steady-state error.

Analysis – Plotting Experimental & Simulated Results

The following plots shows the experimental and Simulink results for each input and for each controller, as per the lab instructions. Our step response is a 1mm input and our ramp input is 5mm/s, used for each of the three controllers in this system.

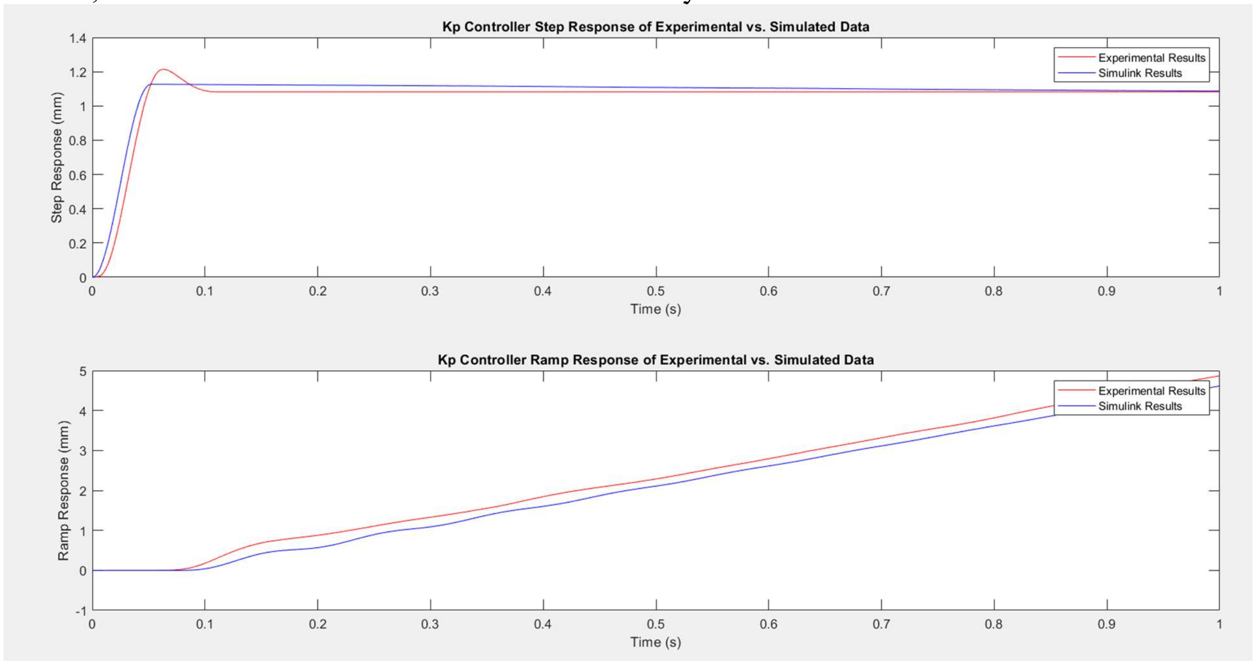


Figure 33: Kp Controller Experimental & Simulation Responses

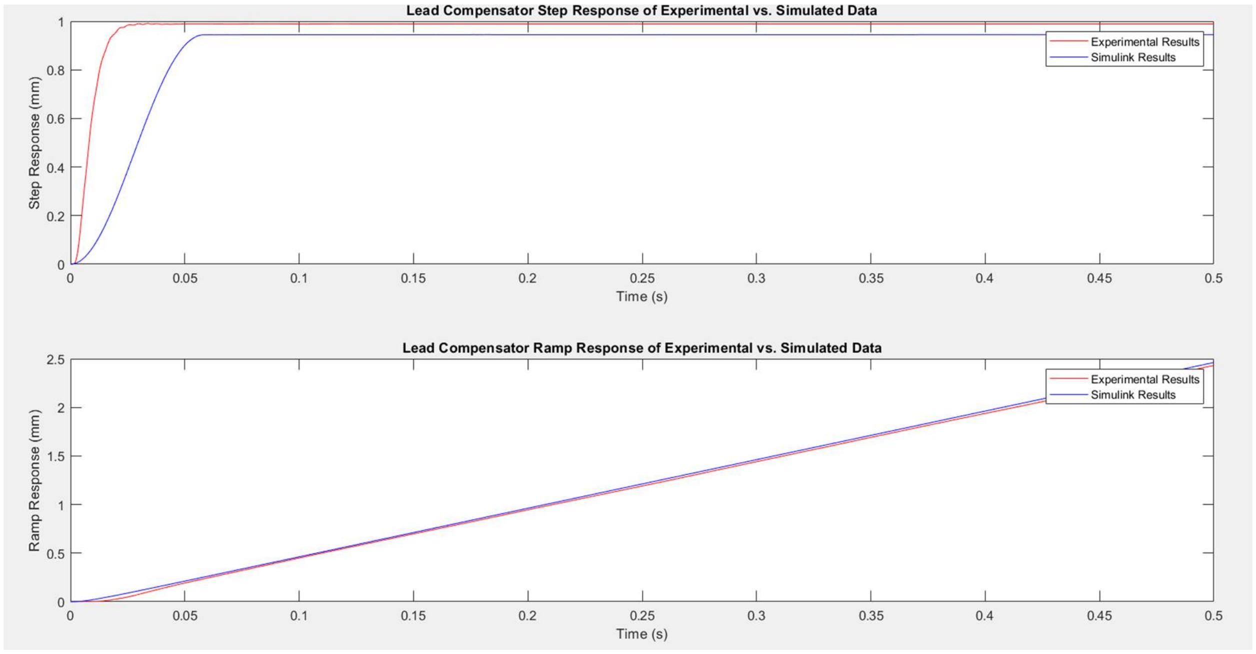


Figure 34: Lead Compensator Controller Experimental & Simulation Responses

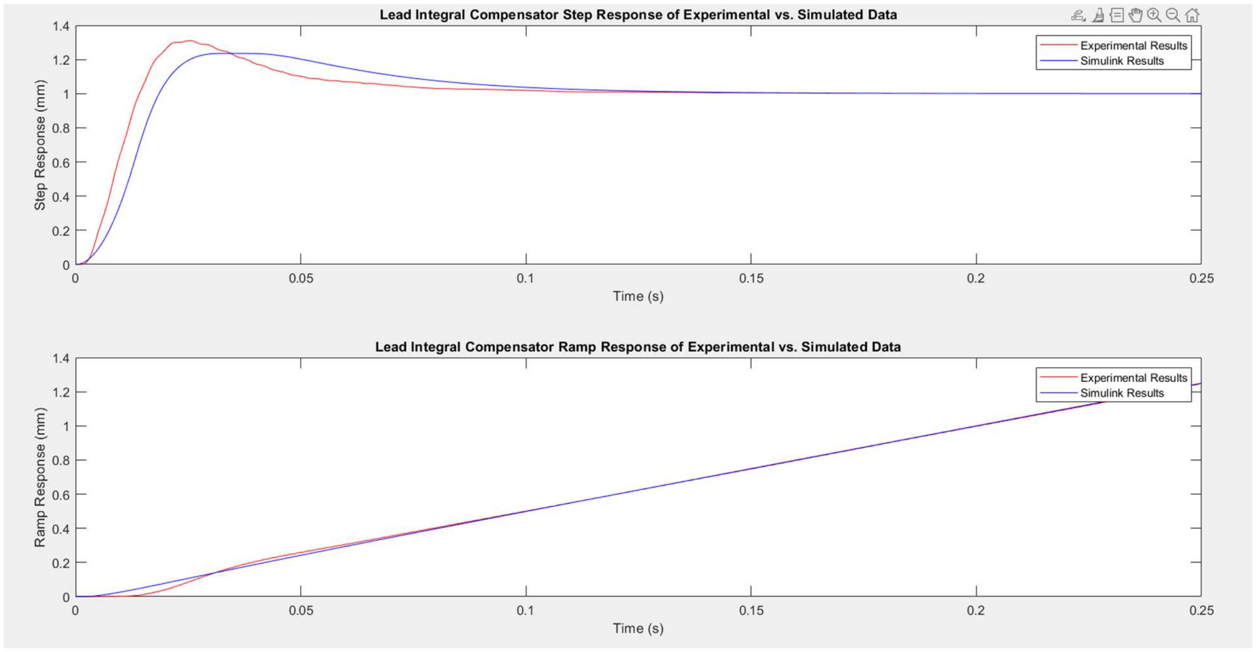


Figure 35: Lead Lag Compensator + Integral Controller Experimental & Simulation Responses

As we can see, we find that our measured and experimental results align very well. We find some differences, with our Kp controller step input having some overshoot and some other issues with offsets between experimental and measured data. I believe this is due to friction modelling, as real-world friction has a non-linear effect that wasn't entirely modeled in our system.

Comparison of Experiments & Simulation

Rise Times & Overshoot %

The table below shows our measured and simulated rise times & Steady-state values. This was taken using MATLAB's Stepinfo() function. The appendix shows this code in its entirety.
Table 5: Simulated & Experimental Rise Times & Overshoot

Controller	Experimental Rise Time (s)	Simulated Rise Time (s)	Experimental Overshoot (%)	Simulated Overshoot (%)
Kp	0.0291	0.0275	12.2170	9.1955
LL	0.0119	0.0345	0.2427	0
LLI	0.0088	0.0116	30.9583	23.5620

Most of our experimental and simulated values align within expectations. Most have similar values within the same magnitude, which tells us that our responses align.
There are a few differences that will be highlighted here. Firstly, our simulated rise time is 3x larger than our experimental rise time for the LL controller. Secondly, there are some larger than expected differences between our overshoot percentages for the Kp and LLI controllers.
Firstly, one factor that can cause this discrepancy is friction. We used a 0.3Nm friction in this system, which is slightly different than the 0.22Nm friction value we determined in Lab 1. Secondly, as described earlier, our simulated friction values does not fully account for all the non-linear effects of friction throughout the system, which will lead to some differences between our results. Unmodelled dynamics, real-world nonlinearities, & experimental errors could also cause differences between our results. Finally, actual damping in our system can be different than our simulated results, causing further differences between our results.

Steady-State Error

The table below shows a comparison for steady-state errors for our three controllers. I manually changed the steady-state time index for every controller, based on the earlier plots. The appendix shows this code in its entirety.
Table 6: Steady-State Errors for Measured & Simulated Ramp Responses

Controller	Measured Steady-State Error (%)	Simulated Steady-State Error (%)
Kp	0.125	0.387
LL	0.051	0.038
LLI	0.002	0.000

Most of our experimental and simulated values align within expectations. Most have similar values within the same magnitude, which tells us that our responses align. The main difference seen is that our simulated steady-state error is 3x larger than our experimental rise time for the Kp controller. Otherwise, our values align well and within expectations.
The discrepancies for the steady-state error is similar to the discrepancies found in the earlier section. Unmodelled friction, real-world non-linearities, differences in system parameters, &

damping differences could all have affected our simulated results compared to our real-world parameters.

Comparison of Controllers

In this section, we will answer the following questions provided to us in the lab manual.

1.1 How does an increased bandwidth affect the performance with regards to rise time of the step response and steady-state error of the ramp response for a Kp Controller?

An increased bandwidth leads to a decreased rise time as the system will respond faster to changes in input. Increasing bandwidth will also reduce steady-state error for a ramp function as Kp & bandwidth increases.

1.2 How does an increased bandwidth affect the performance with regards to rise time of the step response and steady-state error of the ramp response for a Lead Controller?

A lead controller improves phase margin, effectively increasing the system's bandwidth and reducing rise time overall. However, the system's steady state error will remain the same unless the Lead controller is connected with an integral system to overcome friction.

1.3 What is the reason for this difference?

This is due to each controller's inherent different characteristics. Increases a proportional controller's bandwidth is proportional to increasing it's Kp value, which changes the system's ability to respond to changes to input. This leads to lower rise times and steady-state error values. In comparison, increasing the bandwidth for a lead-compensator controller increases its gain at higher frequencies to increase phase margin and lowers its gain at lower frequencies, leading to reduced rise times. However, it won't directly affect steady-state error if an integral system isn't used, compared to the Kp controller.

2.1 Why is it not possible to design a lead-lag compensator with large bandwidth?

A lead-compensator boosts high frequency gain to improve phase margin, but this also increases high frequency noise in the system. If we increase bandwidth, our system will act in this high frequency range and have noise affect our system, degrading system performance and possible system instability.

2.2 What factors limit the system?

Some factors that limit the system include system dynamics, including actuator and sensor dynamics and their characteristics. As stated earlier, higher bandwidth can lead to noise amplification. Stability margins must also be considered, limiting our bandwidth and our system overall.

2.3 Assuming no ZOH delay, is it still possible to design a lead compensator with an infinite bandwidth?

Other factors limiting bandwidth above will stop us from designing a lead compensator with infinite bandwidth. Mechanical and real-world limitations will limit our system and saturate our outputs, limiting our bandwidth. Furthermore, an infinite bandwidth leads to a massive system gain, leading to instability as well.

3.1 What benefits does the integrator provide?

The main benefit the integrator provides is decreasing steady-state error. The integrator helps adjust system's control effort to overcome friction and to minimize error over time.

3.2 Provide this idea mathematically using the error transfer function and final value theorem.

Equation 15: Steady-State Error Formula & R(s) Solution

$$e_{ss} = \lim_{s\to 0} s[1 - G(s)]R(s)$$
$$R(s) = \frac{1}{s^2}$$

To solve G(s), we will use G_OpenLoop and our cascade integral transfer function. MATLAB was used to determine G(s). The code can be seen in the appendix.

Equation 16: G(s) Calculation

$$G_{ol}(s) = \frac{K_a K_t K_e}{s(J_e s + B_e)}$$
$$C(s) = \frac{K_i + s}{s}$$
$$G(s) = \frac{G_{ol}(s)C(s)}{1 + G_{ol}(s)C(s)} = \frac{K_a K_t K_e (K_i + s)}{(J_e s^3 + B_e s^2 + K_a K_t K_e s + K_a K_t K_e Ki)}$$

Using G(s) & R(s), we can solve for Ess.

Equation 17: Steady-State Error Calculation

$$e_{ss} = \lim_{s\to 0} s\left[1 - \frac{K_a K_t K_e (K_i + s)}{(J_e s^3 + B_e s^2 + K_a K_t K_e s + K_a K_t K_e Ki)}\right]\frac{1}{s^2}$$
$$e_{ss} = \lim_{s\to 0}\left[1 - \frac{K_a K_t K_e (K_i + s)}{(J_e s^3 + B_e s^2 + K_a K_t K_e s + K_a K_t K_e Ki)}\right]\frac{1}{s}$$
$$e_{ss} = \lim_{s\to 0}\left[\frac{1}{s} - \frac{K_a K_t K_e (K_i + s)}{s(J_e s^3 + B_e s^2 + K_a K_t K_e s + K_a K_t K_e Ki)}\right]$$
$$e_{ss} = 0$$

Since Ess is 0 in our solution, we proved that the integrator helps minimize steady-state error.

4.1 Why does the integrator cause an overshoot?

An integrator continuously accumulates error, which persists until error is minimized. This can lead to overshoot as this larger error drives our control effort higher than our setpoint. At that point, the error becomes negative and the error decreases, leading to settling or to decaying oscillations.

4.2 How would you reduce this affect?

We can reduce this effort by reducing our Integral Gain values, meaning that our system becomes less sensitive and applies less control effort. Similarly, we can also increase the system damping ratio as well. We can also add a Derivative term to damped oscillations to slow system as we move towards the setpoint.

4.3 What is the trade-off?

In the first two cases, creating a less sensitive system will lead to an overdamped system, resulting in low convergence to zero-steady state error. This increases the rise time and settling time, making a slower system overall that responses slower to inputs. For the final case, a derivative term amplifies high-frequency noise, increasing complexity & degrading performance, as the system will require intense filtering to remove any high-frequency noise.

5.1 Discuss possible scenarios where a Kp controller would be preferred over a lead integrator compensator.

While Lead-integrators compensator controllers have superior results compared to a Kp controller, they require significant resources and information to implement successful. Comparatively, Kp controllers are simple and quick to implement, which is useful for situations where speed is important.
Furthermore, lead compensators help ensure high accuracy and prevent overshoot and system oscillation. If the situation doesn't require these strict regulations, a Kp controller can be implemented quickly and simply.

Conclusion

In this lab, I learned how to design and analyze digital control algorithms for achieving desirable performance characteristics for Mechatronics systems. I learned how to create a digital controller by converting a continuous transfer function into a digital controller by using Zero-Order Hold Equivalent. I learned how this conversion changes system stability and how to analyze stability using Bode Plots & Root Locus. I then learned about the Kp Controller, Lead Compensator Controller, and Lead-Integral Compensator controller and explored each controller's advantages and disadvantages. The proportional controller is simple and fast to implement but limited in performance. The lead controller improved damping and transient performance, while the lead-integrator controller eliminated steady-state error but introduced overshoot that required proper tuning. Finally, I observed the effects of nonlinear elements and saturation on the system's performance as I compared real-world data to simulated results. This laboratory provided valuable insights into each controller's trade-offs and the practical implications of designing digital controllers for real-world implementation.

Appendix

Appendix 1: Simulink Block Diagrams

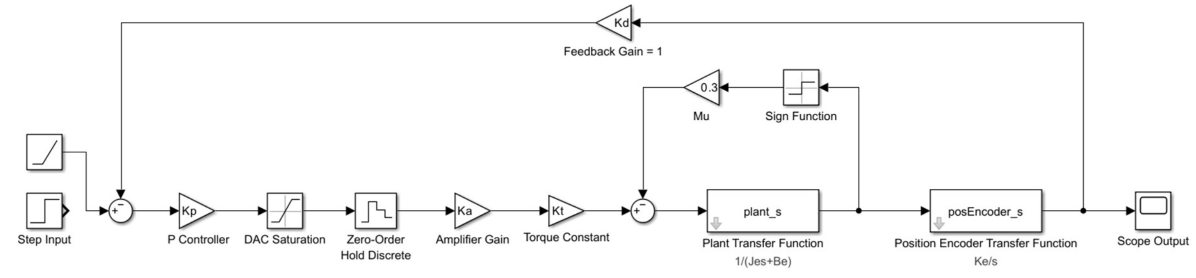


Figure 36: Kp Controller Block Diagram

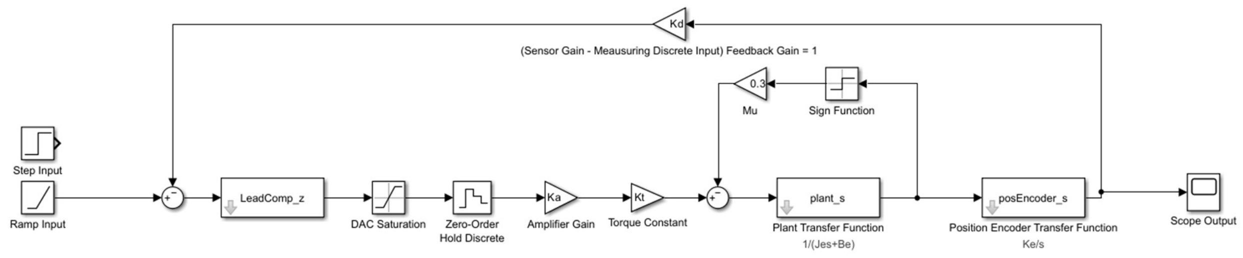


Figure 37: LL Controller Block Diagram

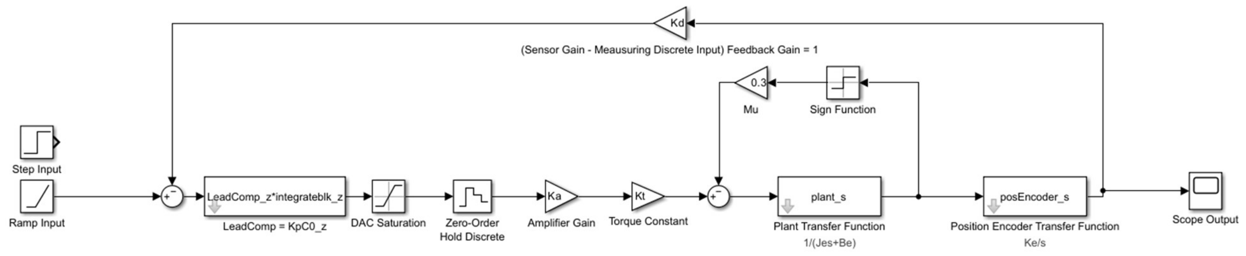


Figure 38: LLI Controller Block Diagram

Appendix 2: Prelab 1 Code

```
% Define system parameters
Ka = 0.887;  %[A/V]
Kt = 0.72;  %[Nm/A]
Ke = (20/(2*pi));  %[mm/rad]
Je = 7e-4;  %[kgm^2]
Be = 0.00612;  %[Nm/rad/s]
mu = 0.3;  %{?]
Ts = 0.0002; %[s]
hp = 20/1000; %[m]

% Define 's' & 'z' as a transfer function variable
s = tf('s');
z= tf('z',Ts);

% Define the numerator of the open transfer function
numerator = Ka * Kt * Ke / (s * (Je * s + Be));
% Create the transfer function G(s)
G_s = numerator

% Create Discrete Transfer Function using matlab model
MatlabG_z = c2d(G_s,Ts)

%Manually defined G(z)
% Define the manually derived G(z) using the equation
% First term:
first_term = (Ka * Kt * Ke / Be) * (Ts * z^(-1)) / ((1 - z^(-1))^2);
first_term=minreal(first_term)

% Second term:
st1=(Je * Ka * Kt * Ke / Be^2);
zpower=z^(-1);
est=exp(-Be*Ts/Je);
numst=zpower*(1-est);
denomst=(1-zpower)*(1-est*zpower);
second_term=st1*numst/denomst;
second_term=minreal(second_term);

% Combine both terms for G(z)
midcalc=(first_term - second_term)
midcalc=minreal(midcalc);
G_z = (1 - z^(-1)) * midcalc;
G_z=minreal(G_z);

% Plot the Bode plot for both transfer functions
figure;
bode(MatlabG_z, G_z, 'r--');
legend('MATLAB c2d G(z)', 'Manually derived G(z)');
title('Bode Plot Comparison between MATLAB c2d and Manually Derived G(z)');
% Step response comparison
figure;
step(MatlabG_z, G_z, 'r--');
legend('MATLAB c2d G(z)', 'Manually derived G(z)');
title('Step Response Comparison');
```

Appendix 3: Prelab 2 Code

```matlab
% Define 's' as a transfer function variable
s = tf('s');
% Define system parameters
Ka = 0.887;  %[A/V]
Kt = 0.72;  %[Nm/A]
Ke = (20/(2*pi));  %[mm/rad]
Je = 7e-4;  %[kgm^2]
Be = 0.00612;  %[Nm/rad/s]
mu = 0.3;  %{?]
Ts = 0.0002; %[s]
hp = 20/1000; %[m]

% State-space matrices (continuous time)
A = [-Be/Je, 0; Ke, 0];
B = [Ka*Kt/Je, 1/Je; 0, 0];
C = [0, 1];
D = [0, 0];

% Continuous-time state-space system
sys_cont = ss(A, B, C, D);

% Convert continuous-time state-space model to discrete-time model
StateSpaceG_z = c2d(sys_cont, Ts, 'zoh');  % Zero-order hold method for
discretization
% Select only the first input of the state-space system
StateSpaceG_z_Xa = ss(StateSpaceG_z.A, StateSpaceG_z.B(:,1), StateSpaceG_z.C,
StateSpaceG_z.D(:,1), Ts);


% Define the numerator of the open transfer function
numerator = Ka * Kt * Ke / (s * (Je * s + Be));
% Define the denominator of the transfer function
denominator = 1;
% Create the transfer function G(s)
G_s = numerator / denominator;

% Create Discrete Transfer Function
MatlabG_z=c2d(G_s,Ts);

%StateSpace Model

% Plot the Bode plot for both transfer functions
figure;
bode(MatlabG_z, 'b-', StateSpaceG_z_Xa, 'r--');
legend('MATLAB c2d G(z)', 'State Space G(z)');
title('Bode Plot Comparison between MATLAB c2d and State Space G(z)');

% Step response comparison
figure;
step(MatlabG_z, 'b-', StateSpaceG_z_Xa, 'r--');
legend('MATLAB c2d G(z)', 'State Space G(z)');
title('Step Response Comparison between MATLAB c2d and State Space G(z)');
xlim([0 1]);
```

Appendix 4: Prelab 3 Code

```
% Define 's' as a transfer function variable
s = tf('s');
% Define system parameters
Ka = 0.887;  %[A/V]
Kt = 0.72;  %[Nm/A]
Ke = (20/(2*pi));  %[mm/rad]
Je = 7e-4;  %[kgm^2]
Be = 0.00612;  %[Nm/rad/s]
mu = 0.3;  %{?]
Ts = 0.02; %[s]
hp = 20/1000; %[m]


% Define the numerator of the open transfer function
numerator = Ka * Kt * Ke / (s * (Je * s + Be));
% Define the denominator of the transfer function
denominator = 1;
% Create the transfer function G(s)
G_s = numerator / denominator;

% Create Discrete Transfer Function
MatlabG_z=c2d(G_s,Ts);

% Root Locus Stability Analysis
%R is complex root locations, K is feedback vector
%figure;
%rlocus(G_s); %Plots Gs Locus
%[rs,kps]=rlocus(G_s); %Gets values from locus

%figure;
%rlocus(MatlabG_z); %Plots Gz Locus
%[rz,kpz]=rlocus(MatlabG_z);

figure;
bode(G_s);

figure;
bode(MatlabG_z);
```

Appendix 5: Prelab 4 Code

```
%Expecting a KP ~1m or smtg like that
% Define system parameters
Ka = 0.887;  %[A/V]
Kt = 0.72;  %[Nm/A]
Ke = (20/(2*pi));  %[mm/rad]
Je = 7e-4;  %[kgm^2]
Be = 0.00612;  %[Nm/rad/s]
mu = 0.3;  %{?]
Ts = 0.0002; %[s]
hp = 20/1000; %[m]

%Shelby's Method
% Define 's' as j60.
s=60j;
% Define the numerator of the open transfer function
numerator = Ka * Kt * Ke / (s * (Je * s + Be));
% Create the transfer function G(s)
Kp=1/abs(numerator)

%Determine Gain Margin by finding LRR & finding phase when LRR gain = 1
s=tf('s');
LRR = Kp * Ka * Kt * Ke / (s * (Je * s + Be));
bode(LRR);
```

Appendix 6: Prelab 5 Code

```
% Define system parameters
Ka = 0.887;  %[A/V]
Kt = 0.72;  %[Nm/A]
Ke = (20/(2*pi));  %[mm/rad]
Je = 7e-4;  %[kgm^2]
Be = 0.00612;  %[Nm/rad/s]
mu = 0.3;  %{?]
Ts = 0.0002; %[s]
hp = 20/1000; %[m]
alpha = 14.7711;
tau = 6.9016e-4;

%Shelby's Method
% Define 's' as j60.
s=377j;
%Find Kp
Gs = Ka * Kt * Ke / (s * (Je * s + Be));
c0 = (alpha*tau*s+1)/(tau*s+1);
Kp=1/abs(Gs*c0)

%Determine Gain Margin by finding LRR & finding phase when LRR gain = 1
s=tf('s');
leadlag=(alpha*tau*s+1)/(tau*s+1);
LRR = leadlag* Ka * Kt * Ke / (s * (Je * s + Be));
bode(LRR);
```

Appendix 7: Prelab 6 Code

```
% Given system values for motor and control parameters
Je = 7e-4;          % Motor inertia constant
Be = 0.00612;       % Motor damping coefficient
Ka = 0.887;         % Amplifier gain
Kt = 0.72;          % Torque constant
Ke = 20 / (2 * pi); % Back EMF constant
Kp = 1.252;         % Proportional gain
T = 0.0002;         % Sampling time
alpha = 14.7711;    % Lead-lag compensation constant
tau = 6.9016e-4;      % Lead-lag time constant
K = 12.7376;        % System gain
w_c = 60 * 2 * pi; % Crossover frequency (rad/s)
Ki = w_c / 10;      % Integrator gain based on crossover frequency

% Calculate the constant term in the numerator of the open-loop transfer function
constant_numerator = Kt * Ka * Ke;

% Define the continuous-time open-loop transfer function without Kp
numerator = [constant_numerator];   % Numerator of G_open_s
denominator = [Je, Be, 0];          % Denominator of G_open_s
G_open_s = tf(numerator, denominator); % Open-loop transfer function
G_open_s = series(G_open_s,Kp); % Factor in Kp
% Define the lead-lag compensator transfer function LL(s)
LL_s = tf([alpha * tau * K, K], [tau, 1]);

% Define the integrator with gain Ki
integrator = tf([1, Ki], [1, 0]);

% Define the combined lead-lag and integrator transfer function LLI(s)
LLI_s = series(LL_s, integrator);

% Define the open-loop system with lead-lag compensation
G_open_LL = series(G_open_s, LL_s);

% Define the open-loop system with lead-lag and integrator compensation
G_open_LLI = series(G_open_s, LLI_s);

% Plot Bode diagrams to compare frequency responses
figure;
hold on; % Hold to overlay multiple plots on the same figure

% Plot Bode for the open-loop transfer function without compensators
bode(G_open_s);

% Plot Bode for lead-lag compensator only
bode(LL_s);

% Plot Bode for combined lead-lag and integrator compensator
bode(LLI_s);

% Plot Bode for open-loop system with lead-lag compensation
bode(G_open_LL);
```

```
% Plot Bode for open-loop system with lead-lag and integrator compensation
bode(G_open_LLI);

% Add grid and legend for better visualization and understanding
grid on;
legend('Gol(s)', 'LL(s)', 'LLI(s)', 'Gol(s)*LL(s)', 'Gol(s)*LLI(s)', 'Location',
'best');
title('Bode Comparison of Five Components'); % Title for Bode plot comparison
```

Appendix 8: Full Lab Code for Part 1, 2, & 3

```
%% Plot Graphs
% Define folder containing the data files
dataFolder = "Lab2Data";

% Get all CSV and MAT files in the folder
csvFiles = dir(fullfile(dataFolder, "*.csv"));
matFiles = dir(fullfile(dataFolder, "*.mat"));

% Initialize structures for data storage
measuredData = struct();
simulatedData = struct();

% Process CSV files (Measured Data)
for i = 1:length(csvFiles)
    % Get file name and path
    filePath = fullfile(csvFiles(i).folder, csvFiles(i).name);
    [~, fileName, ~] = fileparts(csvFiles(i).name);

    % Read and process data
    tableData = readtable(filePath);
    measuredData.(fileName).Time = tableData.Time / 1000; % Convert time to seconds
    measuredData.(fileName).Response = tableData.Enc2_ActPos_mm_; % Adjust column
name if necessary
end

% Process MAT files (Simulated Data)
for i = 1:length(matFiles)
    % Get file name and path
    filePath = fullfile(matFiles(i).folder, matFiles(i).name);
    [~, fileName, ~] = fileparts(matFiles(i).name);

    % Load and process data
    matData = load(filePath); % Assuming variables are named `Time` and `Response`
    simulatedData.(fileName).Time = matData.Time; % Adjust if variable names differ
    simulatedData.(fileName).Response = matData.Signal; % Adjust if variable names
differ
end

% Plot Kp Data
figure;
subplot(2,1,1);
plot(measuredData.Kp_Step.Time, measuredData.Kp_Step.Response, 'r');
hold on;
plot(simulatedData.Kp_Step.Time, simulatedData.Kp_Step.Response, 'b');
xlim([0 1]);
ylabel('Step Response (mm)');
xlabel('Time (s)');
legend('Experimental Results', 'Simulink Results');
title('Kp Controller Step Response of Experimental vs. Simulated Data');

subplot(2,1,2);
plot(measuredData.Kp_Ramp.Time, measuredData.Kp_Ramp.Response, 'r');
```

```
hold on;
plot(simulatedData.Kp_Ramp.Time, simulatedData.Kp_Ramp.Response, 'b');
xlim([0 1]);
ylabel('Ramp Response (mm)');
xlabel('Time (s)');
legend('Experimental Results', 'Simulink Results');
title('Kp Controller Ramp Response of Experimental vs. Simulated Data');

% Plot LL Data
figure;
subplot(2,1,1);
plot(measuredData.LL_Step.Time, measuredData.LL_Step.Response, 'r');
hold on;
plot(simulatedData.LL_Step.Time, simulatedData.LL_Step.Response, 'b');
xlim([0 0.5]);
ylabel('Step Response (mm)');
xlabel('Time (s)');
legend('Experimental Results', 'Simulink Results');
title('LL Controller Step Response of Experimental vs. Simulated Data');

subplot(2,1,2);
plot(measuredData.LL_Ramp.Time, measuredData.LL_Ramp.Response, 'r');
hold on;
plot(simulatedData.LL_Ramp.Time, simulatedData.LL_Ramp.Response, 'b');
xlim([0 0.5]);
ylabel('Ramp Response (mm)');
xlabel('Time (s)');
legend('Experimental Results', 'Simulink Results');
title('LL Controller Ramp Response of Experimental vs. Simulated Data');

% Plot LLI Data
figure;
subplot(2,1,1);
plot(measuredData.LLI_Step.Time, measuredData.LLI_Step.Response, 'r');
hold on;
plot(simulatedData.LLI_Step.Time, simulatedData.LLI_Step.Response, 'b');
xlim([0 0.25]);
ylabel('Step Response (mm)');
xlabel('Time (s)');
legend('Experimental Results', 'Simulink Results');
title('LLI Controller Step Response of Experimental vs. Simulated Data');

subplot(2,1,2);
plot(measuredData.LLI_Ramp.Time, measuredData.LLI_Ramp.Response, 'r');
hold on;
plot(simulatedData.LLI_Ramp.Time, simulatedData.LLI_Ramp.Response, 'b');
xlim([0 0.25]);
ylabel('Ramp Response (mm)');
xlabel('Time (s)');
legend('Experimental Results', 'Simulink Results');
title('LLI Controller Ramp Response of Experimental vs. Simulated Data');

%% Calculate Rise Time & Overshoot
% Initialize arrays to store rise times and overshoots
riseTimesMeasured = [];
```

```
riseTimesSimulated = [];
overshootsMeasured = [];
overshootsSimulated = [];

% Slicing for P Controller (Kp)
KpStartIdx = find(measuredData.Kp_Step.Time >= 0, 1);
KpEndIdx = find(measuredData.Kp_Step.Time <= 1, 1, 'last');
KpMeasuredTime = measuredData.Kp_Step.Time(KpStartIdx:KpEndIdx);
KpMeasuredResponse = measuredData.Kp_Step.Response(KpStartIdx:KpEndIdx);

% Calculate for P Controller (Kp)
expInfo = stepinfo(KpMeasuredResponse, KpMeasuredTime);
simInfo = stepinfo(simulatedData.Kp_Step.Response, simulatedData.Kp_Step.Time);
riseTimesMeasured = [riseTimesMeasured; expInfo.RiseTime];
riseTimesSimulated = [riseTimesSimulated; simInfo.RiseTime];
overshootsMeasured = [overshootsMeasured; expInfo.Overshoot];
overshootsSimulated = [overshootsSimulated; simInfo.Overshoot];

% Slicing for Lead Compensator (LL)
LLStepStartIdx = find(measuredData.LL_Step.Time >= 0, 1);
LLStepEndIdx = find(measuredData.LL_Step.Time <= 0.5, 1, 'last');
LLMeasuredTime = measuredData.LL_Step.Time(LLStepStartIdx:LLStepEndIdx);
LLMeasuredResponse = measuredData.LL_Step.Response(LLStepStartIdx:LLStepEndIdx);

% Calculate for Lead Compensator (LL)
expInfo = stepinfo(LLMeasuredResponse, LLMeasuredTime);
simInfo = stepinfo(simulatedData.LL_Step.Response, simulatedData.LL_Step.Time);
riseTimesMeasured = [riseTimesMeasured; expInfo.RiseTime];
riseTimesSimulated = [riseTimesSimulated; simInfo.RiseTime];
overshootsMeasured = [overshootsMeasured; expInfo.Overshoot];
overshootsSimulated = [overshootsSimulated; simInfo.Overshoot];

% Slicing for Lead Integral Compensator (LLI)
LLIStepStartIdx = find(measuredData.LLI_Step.Time >= 0, 1);
LLIStepEndIdx = find(measuredData.LLI_Step.Time <= 0.25, 1, 'last');
LLIMeasuredTime = measuredData.LLI_Step.Time(LLIStepStartIdx:LLIStepEndIdx);
LLIMeasuredResponse = measuredData.LLI_Step.Response(LLIStepStartIdx:LLIStepEndIdx);

% Calculate for Lead Integral Compensator (LLI)
expInfo = stepinfo(LLIMeasuredResponse, LLIMeasuredTime);
simInfo = stepinfo(simulatedData.LLI_Step.Response, simulatedData.LLI_Step.Time);
riseTimesMeasured = [riseTimesMeasured; expInfo.RiseTime];
riseTimesSimulated = [riseTimesSimulated; simInfo.RiseTime];
overshootsMeasured = [overshootsMeasured; expInfo.Overshoot];
overshootsSimulated = [overshootsSimulated; simInfo.Overshoot];

% Final results in arrays
riseTimesMeasured % Rise times from measured data
riseTimesSimulated % Rise times from simulated data
overshootsMeasured % Overshoots from measured data
overshootsSimulated % Overshoots from simulated data

%% Steady-state calculations
% Define ramp parameters
t = 0:0.01:2; % Time vector
```

```
slope = 5; % Ramp slope
expectedRamp = slope * t; % Expected ramp function

% Define the time range for steady-state error calculation
startTime = 1; % Start of steady state - Changes Per Controller
endTime = 2; % End of steady state - Changes Per Controller

% Initialize storage for steady-state errors
controllers = {'Kp', 'LL', 'LLI'}; % Controllers
steadyStateErrors = table(); % Table to store results

for i = 1:length(controllers)
    controller = controllers{i};

    % Experimental data
    measuredTime = measuredData.([controller '_Ramp']).Time;
    measuredResponse = measuredData.([controller '_Ramp']).Response;
    expIndices = measuredTime >= startTime & measuredTime <= endTime;
    expectedRampInterp = interp1(t, expectedRamp, measuredTime(expIndices));
    steadyStateErrorMeasured = mean(abs(measuredResponse(expIndices) -
expectedRampInterp));

    % Simulated data
    simulatedTime = simulatedData.([controller '_Ramp']).Time;
    simulatedResponse = simulatedData.([controller '_Ramp']).Response;
    simIndices = simulatedTime >= startTime & simulatedTime <= endTime;
    expectedRampInterp = interp1(t, expectedRamp, simulatedTime(simIndices));
    steadyStateErrorSimulated = mean(abs(simulatedResponse(simIndices) -
expectedRampInterp));

    % Append to table
    steadyStateErrors = [steadyStateErrors; table({controller},
steadyStateErrorMeasured, steadyStateErrorSimulated, ...
        'VariableNames', {'Controller', 'MeasuredError', 'SimulatedError'})];
end

% Display the results table
disp(steadyStateErrors);

% Plot comparison (optional)
figure;
bar(categorical(steadyStateErrors.Controller), [steadyStateErrors.MeasuredError,
steadyStateErrors.SimulatedError]);
ylabel('Steady-State Error (mm)');
xlabel('Controller');
legend('Experimental', 'Simulated');
title('Comparison of Steady-State Errors');
grid on;
```

Appendix 9: G(s) Calculation

```
s=tf('s');
% Define symbolic variables
syms s Ka Kt Ke Je Be Ki

% Define the open-loop transfer function G_ol(s)
G_ol = (Ka * Kt * Ke) / (s * (Je * s + Be));

% Define the controller transfer function C(s)
C = (Ki + s) / s;

% Calculate the closed-loop transfer function G(s)
G = simplify(G_ol * C / (1 + G_ol * C));

% Display the result
disp('Closed-loop transfer function G(s):');
disp(G);
```