MECH 467

Project 2: Digital Control of Motion Actuators

Lab Performed on November 20, 2025

Sean Lee, 23334642

Seanhlee411@gmail.com

Table of Contents

1. Abstract

The objective of this project is to design control algorithms that deliver the desired performance characteristics of the ball-screw driven table identified in Project I. We will apply control theory principles and our knowledge of the system to develop specific controllers, implement them under computer control, and evaluate their performance based on response speed, accuracy, and robustness. Furthermore, we will conduct analysis on the designed controllers, testing their ability to satisfy specified requirements, to determine the efficacy of the system in achieving the desired behavioral properties.

2. Introduction

Ball screw drives are widely utilized in industrial applications involving precise moving parts, such as CNC machines, due to their high accuracy and precision in positioning. They are significantly more common in small-scale applications compared to their alternative, linear motor drives, due to their compact size and lower cost. Before operating machines of this nature, modeling said machine is an important step that can help better understand the system for control, as it allows the user to predict the behavior of the machine and develop relevant automatic control systems without the risk of damaging the equipment during the testing of said systems.

The objective of this lab is to perform component identification as a crucial step in this modeling process. We will apply the laws of mechanics to use the information provided by identification to build an accurate model of the system. Furthermore, we will utilize this model to simulate the machine's behavior, establishing a safe environment for the development and verification of automatic control systems.

3. Pre-Lab

3.1. Discrete Transfer Function Derivation

Below is the manual derivation of the discrete domain transfer function of the open loop transfer function from lab 1. Note that we will treat T_d as a disturbance input and will not include in calculation of the transfer function.

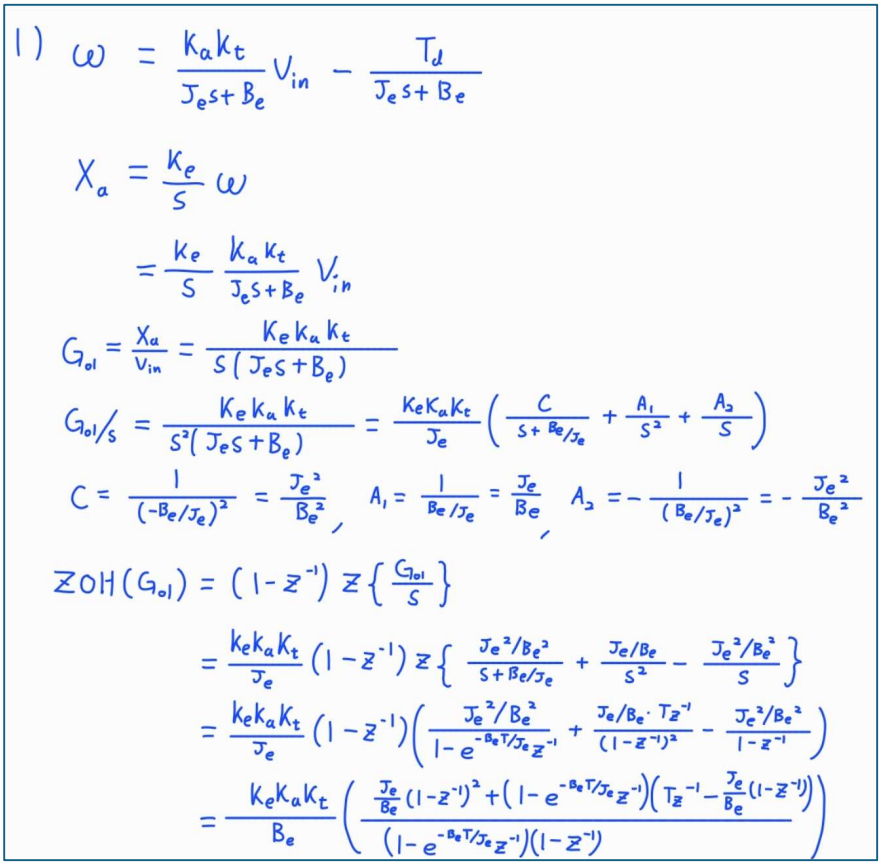


Figure 1: Manual derivation of Z-domain transfer function

To confirm, using MATLAB and the provided parameters, we can convert the transfer function from lab 1 from continuous domain to discrete domain. Then, we can manually input our found transfer function and compare the two. As expected, both transfer function are identical, yielding

$$G_{ol}(z) = \frac{5.805 \times 10^{-5} z + 5.801 \times 10^{-5}}{z^2 - 1.998z + 0.9982}$$

Naturally, their bode plots are also identical, as shown below:

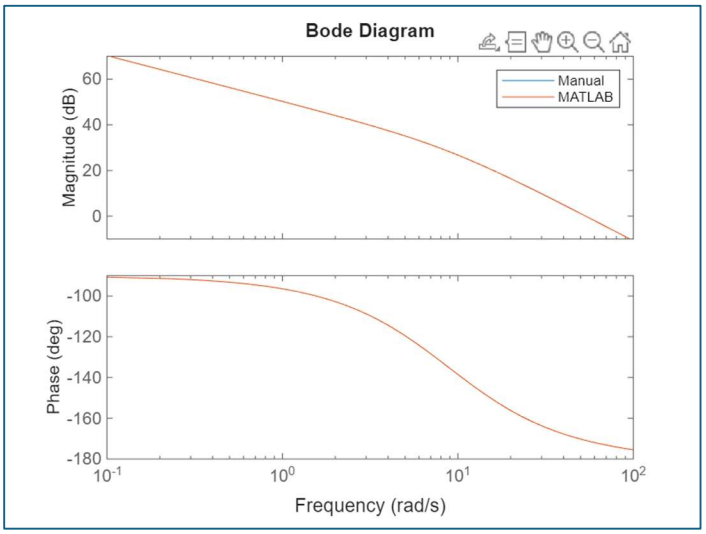


Figure 2: Bode plot comparison of manually obtained transfer function and MATLAB-driven transfer function

3.2. State Space Model

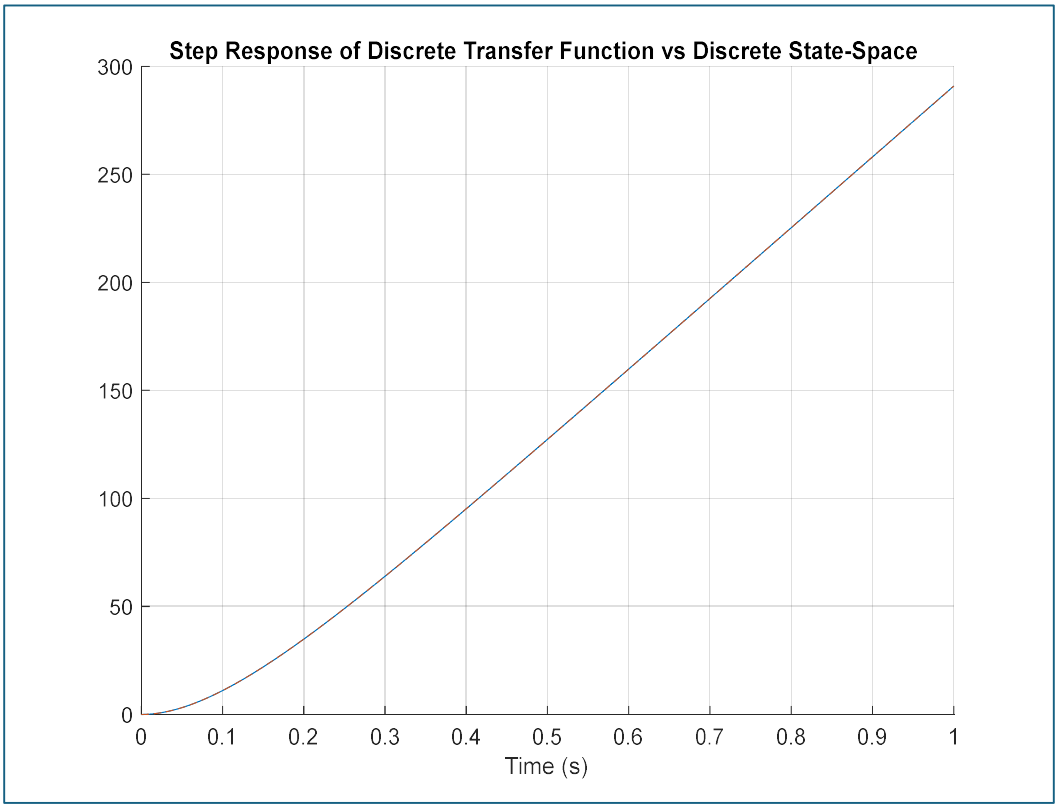


Figure 3: Step response of discrete transfer function vs discrete state-space model

As expected, the two responses are identical.

3.3. Stability Analysis

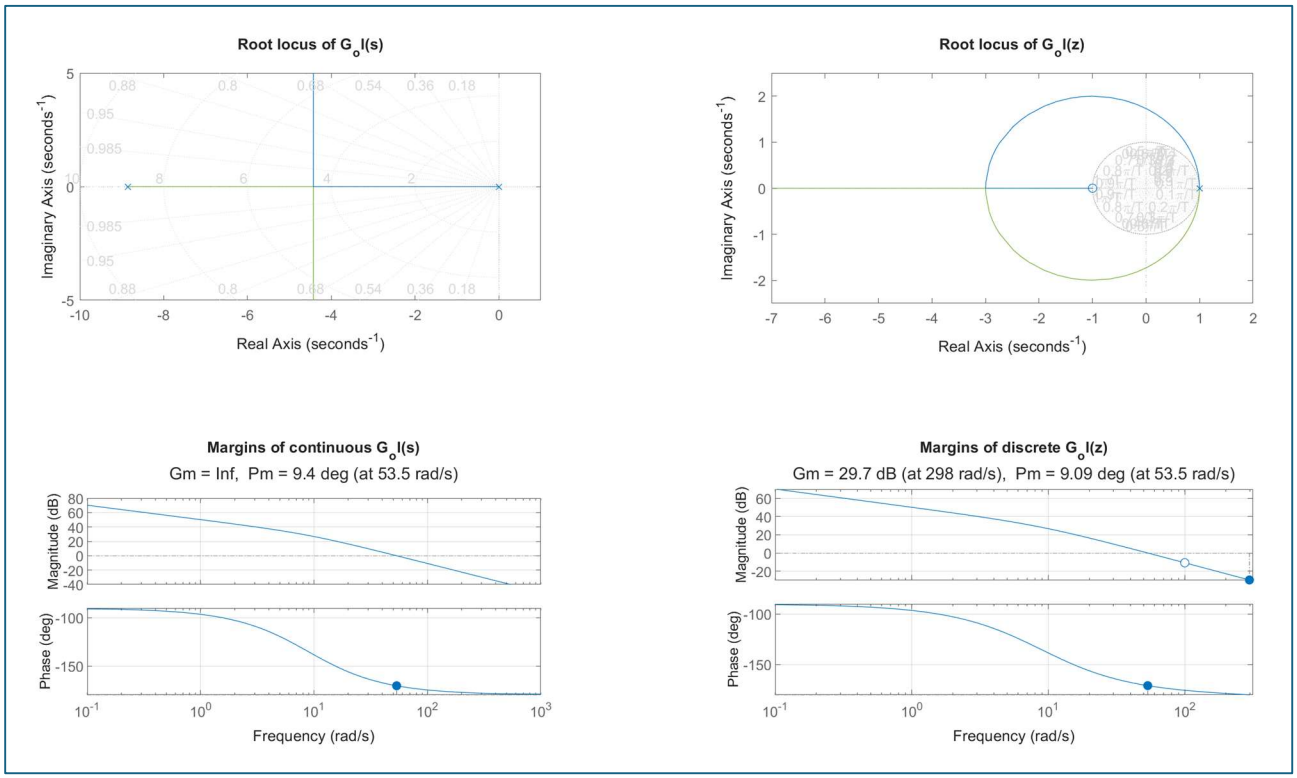


Figure 4: Root locus and margins of G_{ol} in continuous and discrete domain

a) See figure below for manual derivation of expressions for closed-loop poles.

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

Poles: $1 + K_p G_{ol}(s) = 0$

$$\frac{K_p K_e k_a k_t}{s(J_e s + B_e)} = -1$$

$$J_e s^2 + B_e s + K_p K_e k_a k_t = 0$$

$$P_{cl} = \frac{-B_e \pm \sqrt{B_e^2 - 4 J_e K}}{2 J_e} \quad (K = K_p K_e k_a k_t)$$

Figure 5: Manual derivation of closed-loop pole expression

b) The phase margins for G_{ol} are nearly identical for continuous and discrete domains. This suggests our sampling time was sufficiently small to accurately represent the system. The infinite gain margin of the continuous domain means our system is theoretically always stable, which in real application is limited by the sampling time, as observed in the limited value of discrete domain gain margin.

c) The stability in continuous and discrete domains is not always equivalent. If the sampling period of the discrete system is too large – meaning the sampling frequency is too small – the discrete model cannot accurately represent the continuous real system, resulting in discrepancies. In general, the higher the sampling frequency, the closer we are to the true continuous system, and thus the higher gain margin and bandwidth stability. We can see this is the case by testing the three sampling times; the gain margins, for the three sampling times, are as follows:

Table 1: Gain margins at various sampling times

0.02s	0.002s	0.0002s
0.314 dB	9.71 dB	29.69 dB

As we expect, the smallest sampling time produces the highest gain margin, which has the highest bandwidth for stability.

3.4. P-Controller Design

60 rad/s corresponds to 0.012 rad/sample. Using this per-sample frequency, we can calculate the magnitude of $G_{ol}(z)$, where $z = e^{j\omega_{persample}}$. K_p should be set such that at this frequency, $|K_p G_{ol}(z)| = 1$. Using MATLAB, we can plot this, gaining the bode plot below, and finding $K_p = 1.2531$.

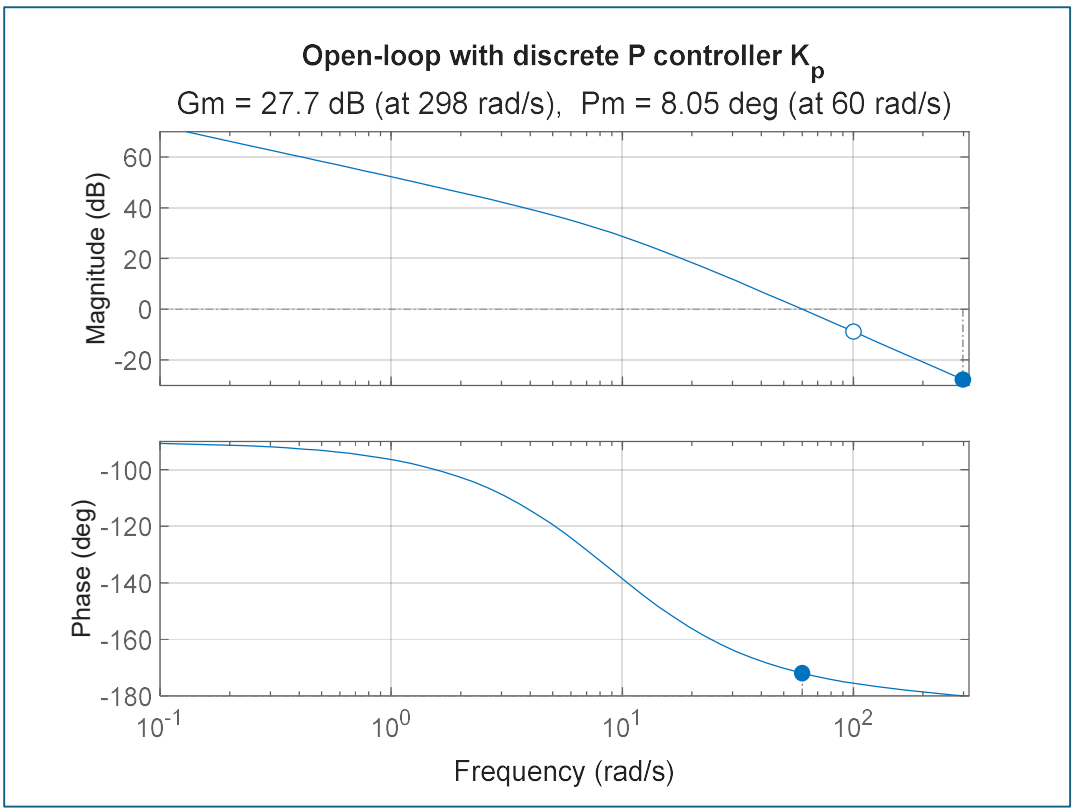


Figure 6: Bode Plot with K_p

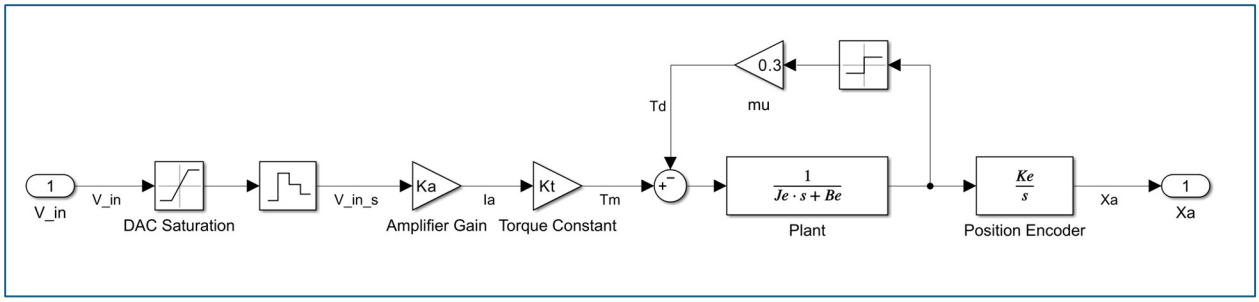


Figure 7: System block diagram on Simulink

Using the implemented Simulink block diagram from Figure 7, we can observe the step response like below:

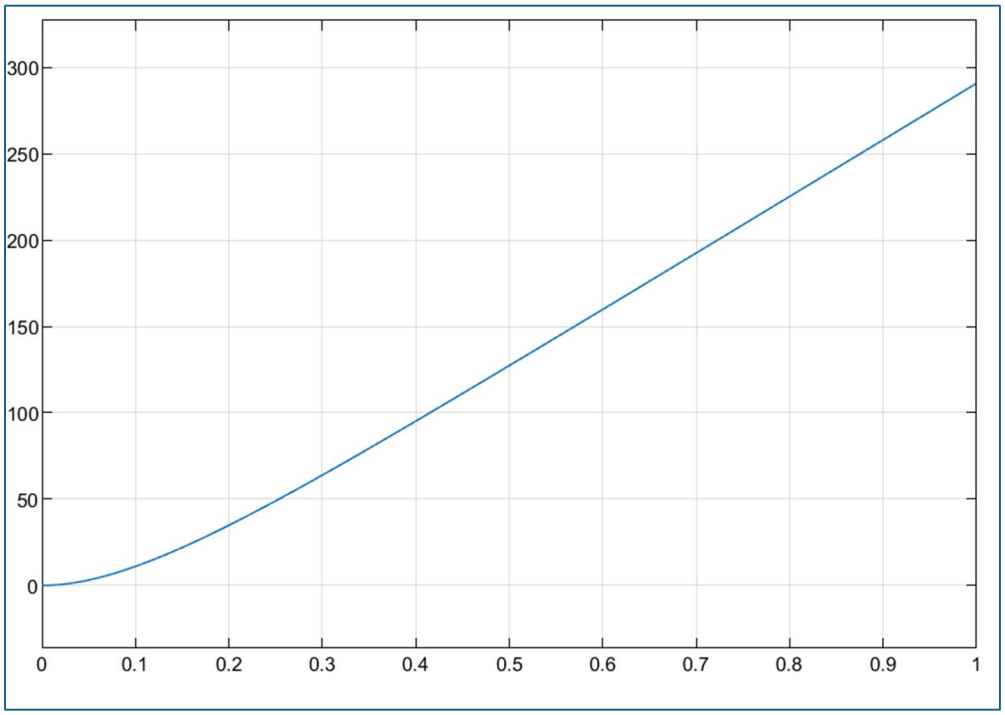


Figure 8: Step response of Simulink block diagram (without Coulomb friction)

This verifies our model from 3.2 as it behaves nearly identically. Note that the saturation used here was $\pm\ 3\ A$.

With coulomb friction, the overall shape of the response is still similar, but the magnitude of X_a becomes significantly smaller at all times. As the system rises indefinitely, overshoot, rise time, and settling time do not exist.

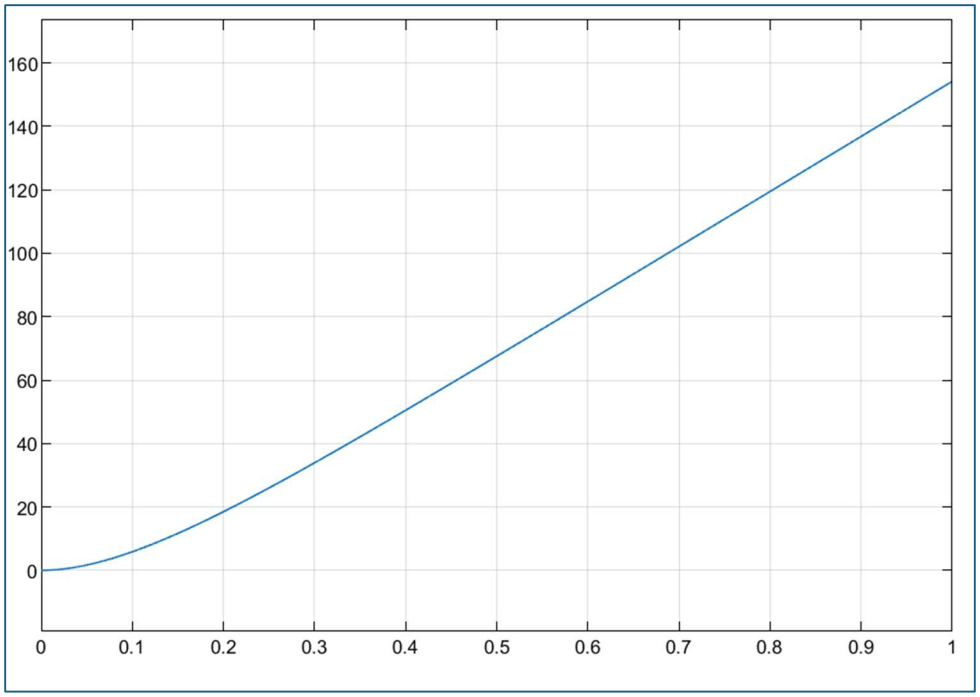


Figure 9: Step response of Simulink block diagram (with Coulomb friction)

Setting a lower saturation limit (at $\pm 0.5\ A$) lowers the magnitude by about one decrement, but it does not introduce overshoot, rise time, or settling time. See the plot below:

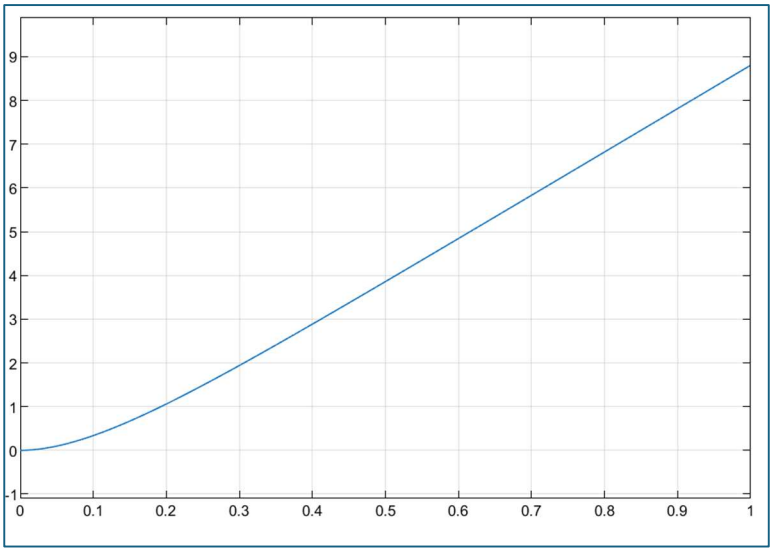


Figure 10: Step response of Simulink block diagram (with Coulomb friction and lower saturation limit)

3.5. Lead – Lag Compensator Design

a) The expressions for each component of the controller C(s) is provided below. The constants' values are found via MATLAB.

To find α, we set the max phase lead to 60 degrees and work backwards:

$$\alpha = \frac{1 + \sin(60°)}{1 - \sin(60°)} = 13.928$$

Then we can derive T from α:

$$T = \frac{1}{\omega_c \sqrt{\alpha}} = 7.1074 \times 10^{-4}$$

To find K, we look at the steady state gain, which should be 1:

$$K = \frac{1}{|C(j\omega_c)G_{ol}(j\omega_c)|} = 13.1174$$

Plotting, we obtain the frequency response below. As we can observe, the design criteria as mentioned before are met.

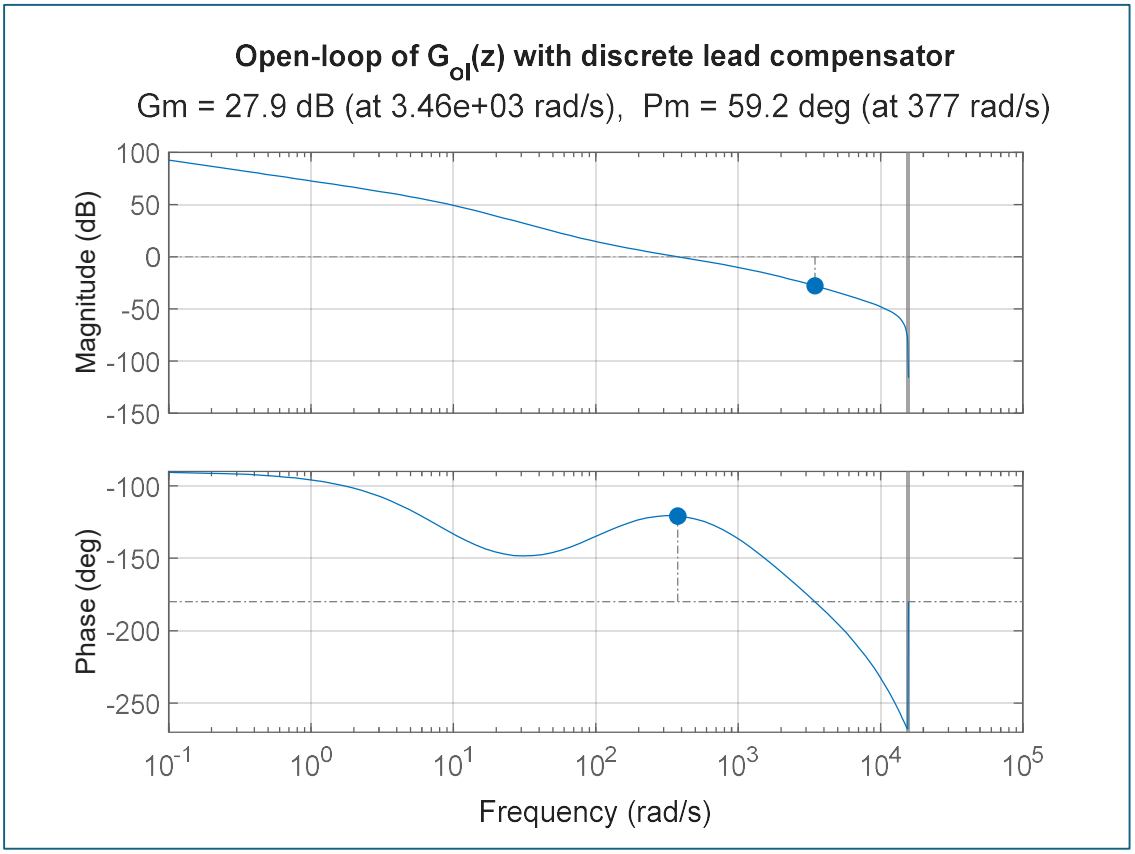


Figure 11: Frequency response of system with designed lead-lag compensator

b) Adding the integral action as well as the lead lag compensator has drastically changed the behaviour of the system. See below for the step and ramp responses, with mu of 0.3 and saturation limit of $\pm$ 3 A.

The step response is no longer close to linear. It instead resembles a parabola.

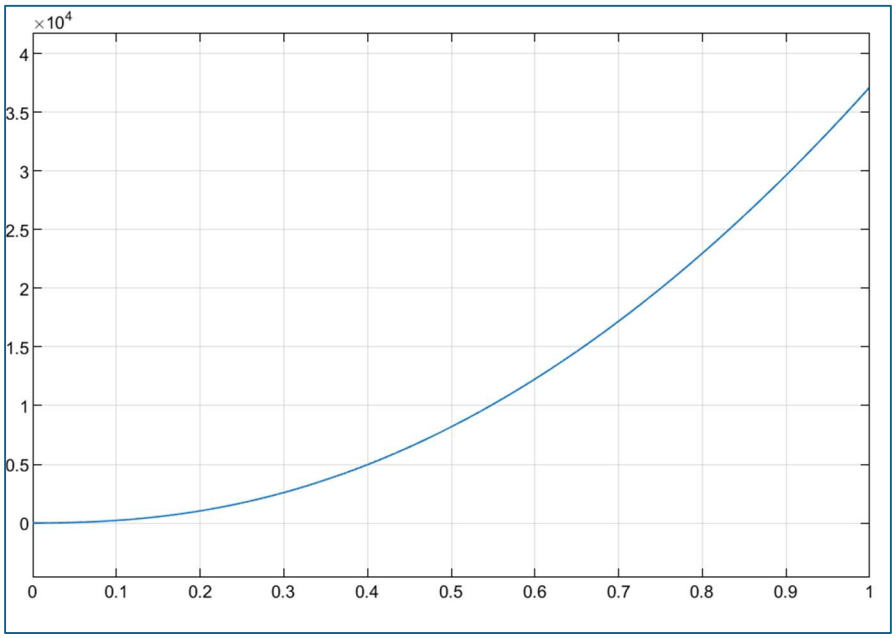


Figure 12: Step response of system with integral action and lead-lag compensator

And the ramp response has an even higher rate of growth, resembling a higher-degree polynomial graph.

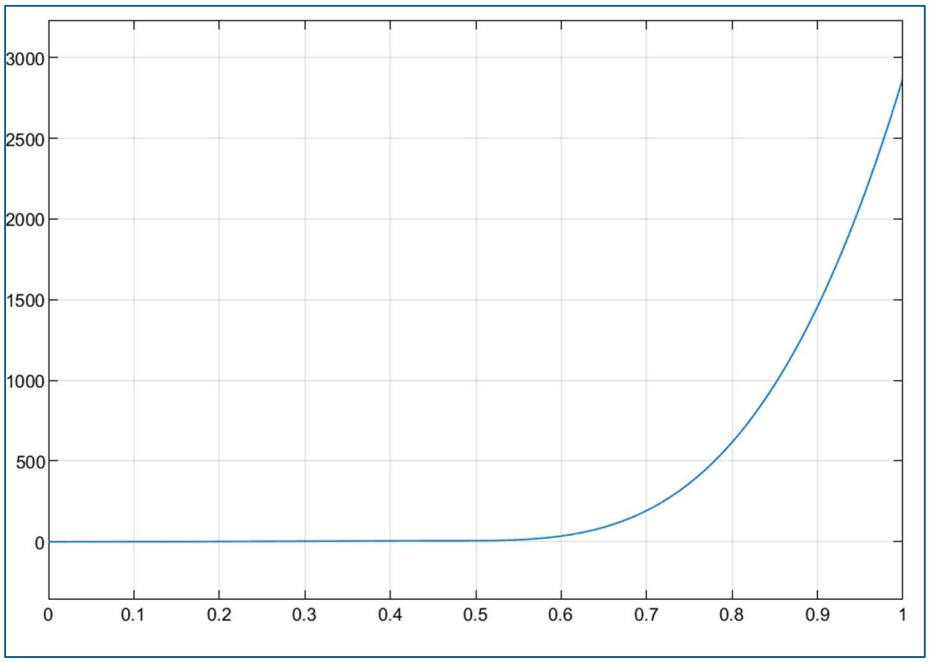


Figure 13: Ramp response of system with integral action and lead-lag compensator

Without the integral controller (and still with the lead-lag compensator), the responses look like:

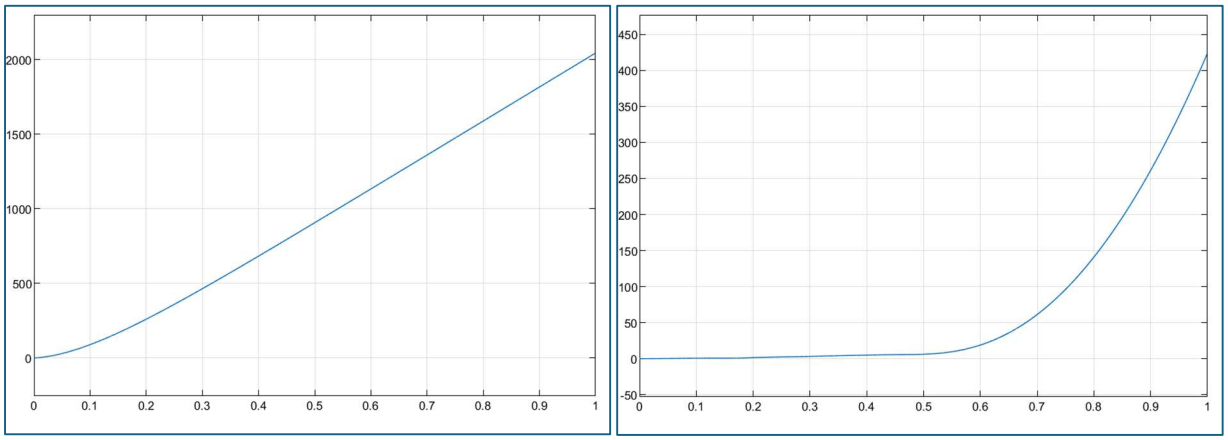


Figure 14: Step (left) and ramp (right) responses of system with only lead-lag compensator

These responses, especially the step response, confirm that the integral controller effectively makes the output behave almost like a polynomial one degree higher, allowing the system to change much more rapidly.

3.6. Discussion

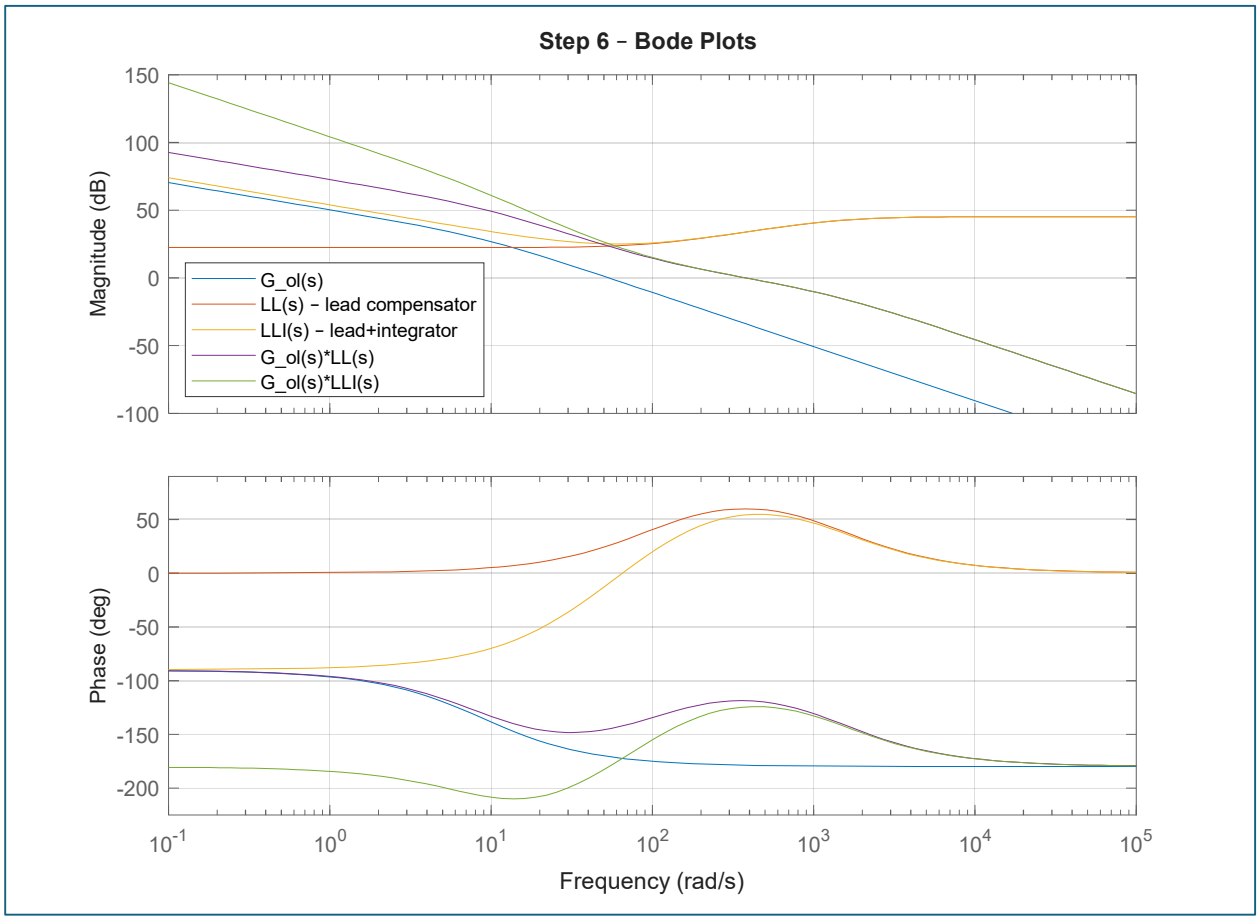


Figure 15: Bode plots of transfer functions

We see the expected behaviour of the transfer functions in the bode plots. For both magnitude and phase plots, the LL and LLI controllers can be seen converging to a same steady-state final value, with its gain cross over frequency around our target of 377 rad/s, with a phase margin of 60 degrees. We also see that the LL and LLI controllers, when multiplied to G_{ol}, modify its responses to resemble their respective plot shapes, increasing the response rates at low to medium frequencies, while keeping higher frequency responses very low.

Both controllers, which modifies DC gain of the system (both plots have significantly different value near 0 frequency), lowers the steady-state error. Especially with LLI controller, which has theoretically infinite DC gain, the steady-state error is nearly nonexistent.

Both controller also forcefully sets the gain crossover frequency at 377 rad/s, which shortens the rise time, as the system is allowed to grow faster at frequencies under bandwidth.

4. Body

4.1. Comparison of Experiments and Simulation

Again, for the modeled system, because the output is a position profile, which continuously rises based on our setup, the rise time and overshoot technically cannot be quantified without differentiating. It would exist if the output we are looking at is a velocity profile.

However, the real system seems to be different – it behaves as if the position encoder block was not an integrator (as if it was just a constant like K_e without s^{-1}), where step input results in a step position output. This allows us to measure the rise time and overshoot for the step responses, and the steady state error for the ramp responses.

1) For the real system for each controller types, the step response rise time and overshoots are as follows (using encoder 2 position data):

Table 2: Real step response overshoots and rise times for each controller

Controller Type	Overshoot (%)	Rise time (ms)
KP	14.89	27.69
LL	0.818	10.42
LLIC	34.83	8.3945

We can observe that the introduction of lead-lag controller significantly reduces rise time and overshoot. The addition of integral controller reduces rise time even further, but runs the risk of introducing significant overshoot.

2) Again using encoder 2 velocity data, but this time only using data in the first 2 seconds to analyze the response in one direction, we get the following steady state errors:

Table 3: Real ramp response steady state errors for each controller

Controller Type	Steady state error (%)
KP	2.45
LL	0.52
LLIC	0.08

This time, each introduction of lead-lag or integrator controller reduces the steady state error, which can be attributed to the controllers speeding up the response time of the system, allowing it to approach its steady state value much faster and therefore giving it more time to settle within the 2-second period we excite it for. Particularly, the introduction of integral controller practically eliminates the steady state error, as expected.

4.2. Part B: Comparison of Controllers

3) While increasing bandwidth generally improves the performance for both controllers, lead-lag controller does this more efficiently than a simple proportional controller by essentially decoupling the response speed from stability of the system. To increase the bandwidth on a proportional controller, the constant K_p must be increased, which bring the system closer and closer to instability. On the other hand, a lead-lag controller achieves this by adding or removing a lead/lag from the system, which boosts low-frequency gain significantly without affecting the bandwidth much

4) It is impossible to design a lead-lag controller with infinite bandwidth, because that would mean the system must be capable of instantly moving mechanical mass, which requires infinite acceleration and force. Additionally, even if the ZOH component had zero delay, we would need to sample the system at infinite frequency, which is impossible.

5) The integrator speeds up the system response by adding an additional term to the system response that varies with the previous feedback, which decreases rise time and nearly eliminates the steady state error.

6) The integrator causes overshoot because of the added term mentioned in Q5. This can be reduced either by using the integrator in moderation (reduce the coefficient K_I) or by introducing a derivative term to counteract the overshoot of integrator controller.

7) Some scenarios include:
    a. The inputted state being unreachable due to saturation limits – integrator would constantly try to drive the system beyond saturation
    b. High noise being amplified by the integrator term
    c. Applications where short settling time is preferred

5. Conclusion

In conclusion, this laboratory experiment successfully demonstrated the design and evaluation of control algorithms for the ball-screw driven table, bridging the gap between theoretical system modeling and practical motion control. The primary objective—to achieve specific performance targets regarding response speed, accuracy, and robustness—was met through an iterative process of mathematical modeling and experimental validation.

A key takeaway from this project is the critical role of accurate system identification in control design. While the mathematical model served as a highly effective, low-risk tool for initial tuning—allowing for rapid iteration without the physical risks of damaging the hardware—it inherently relied on approximations. As observed, theoretical values for parameters like effective inertia (J_e) and friction often fail to capture the full complexity of the physical system (e.g., unmodeled entropy or non-linearities).

However, by integrating experimental parameter identification, we were able to calibrate these uncertain variables, significantly improving the model's fidelity. This process highlighted that while simple controllers (like Proportional control) offer robustness and ease of tuning, more complex strategies (such as Lead-Lag compensation) are necessary to decouple bandwidth from stability, thereby minimizing steady-state error without sacrificing rise time. Ultimately, this lab confirmed that a hybrid approach—using calibrated models for design and real-hardware tests for verification—is essential for developing robust control systems that operate effectively within the physical constraints of actuator saturation and sensor noise.

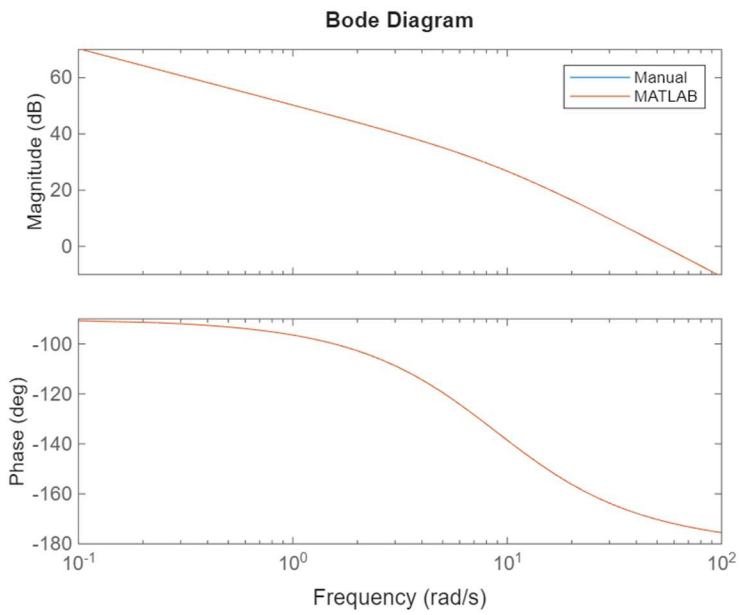
6. Appendices

6.1. Appendix A – MATLAB Code

```
%% MECH467/541
% Prelab 2

%% Initialization
clc
clear all
close all
format compact

%% Parameters
Ka = 0.887;
Kt = 0.72;
Je = 7e-4;
Be = 6.2e-3;
Ke = 20/2/pi;
T = 2e-4; % Sampling time
```

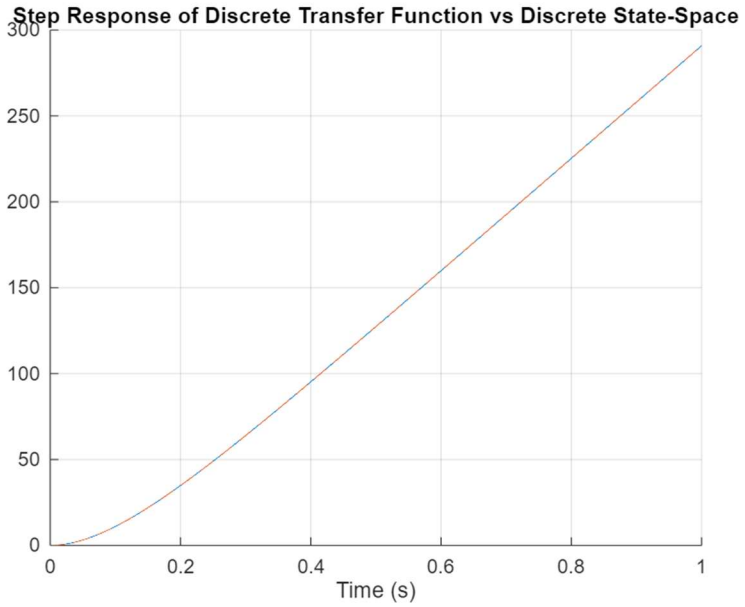
Part 1

```
% Tfs
s = tf('s');
Gol = Ka * Kt * Ke / (s * (Je * s + Be));

z = tf('z');
Gol_manual = Ke * Ka * Kt / Be * ...
    (Je/Be*(1-z^-1)^2 + (1-exp(-Be * T / Je) * z^-1) * (T * z^-1 - Je/Be * (1-
z^-1))) / ...
    ((1-exp(-Be * T / Je) * z^-1) * (1-z^-1));
Gol_manual = minreal(Gol_manual);
Gol_manual.Ts = T; % Assign the specific sample time

Gol_z = c2d(Gol, T, 'zoh');
figure; grid on;
bode(Gol_manual, Gol_z);
legend('Manual','MATLAB');
```



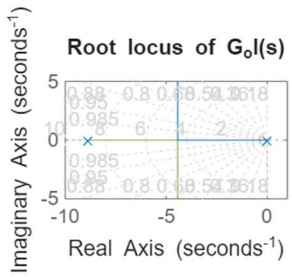
Part 2

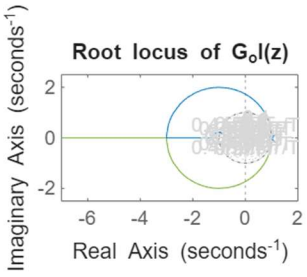
```
% Matricies
A = [-Be/Je, 0;
    Ke, 0];
B = [Ka*Kt/Je 1/Je;
    0 0];
C = [0 1];
D = [0 0];

contsys = ss(A,B,C,D);
discsys = c2d(contsys, T, 'zoh');

% Step Response
t = 0:T:1;
[y_tf, t_tf] = step(Gol_z, t);

[y_ss_3d, t_ss] = step(discsys(:,1), t);
y_ss = squeeze(y_ss_3d);

figure; grid on; hold on;
plot(t_tf, y_tf);
plot(t_ss, y_ss, '--');
xlabel('Time (s)');
title('Step Response of Discrete Transfer Function vs Discrete State-Space');
```

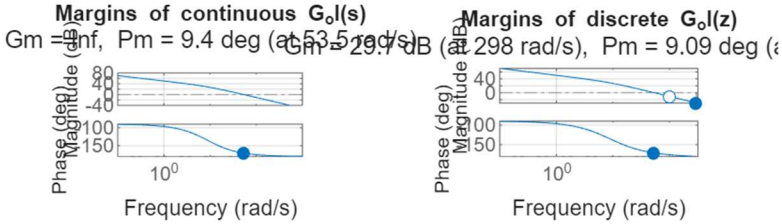


Part 3

```
figure;
% (a) Root loci
subplot(2,2,1);
rlocus(Gol);
grid on;
title('Root locus of G_ol(s)');

subplot(2,2,2);
rlocus(Gol_z);
grid on;
title('Root locus of G_ol(z)');

% (b) Phase and gain margins (individual margin plots)
subplot(2,2,3);
margin(Gol);
grid on;
title('Margins of continuous G_ol(s)');

subplot(2,2,4);
margin(Gol_z);
grid on;
title('Margins of discrete G_ol(z)');
```








```
% (c) Gain margin for three sampling times in z-domain
Ts_list = [0.02, 0.002, 0.0002];
gm_vec  = zeros(size(Ts_list));
gm_dB   = zeros(size(Ts_list));

fprintf('\nGain margins of G_ol(z) for different Ts:\n');
```

```
Gain margins of G_ol(z) for different Ts:
```

```
for k = 1:numel(Ts_list)
    Gz_k   = c2d(Gol, Ts_list(k), 'zoh');
    [gm, ~, ~, ~] = margin(Gz_k);
    gm_vec(k) = gm;
    gm_dB(k)  = 20*log10(gm);
    fprintf('  Ts = %.5f s:  GM = %.3f (%.2f dB)\n', ...
            Ts_list(k), gm_vec(k), gm_dB(k));
end
```

```
Warning: The closed-loop system is unstable.

  Ts = 0.02000 s:  GM = 0.314 (-10.05 dB)

  Ts = 0.00200 s:  GM = 3.059 (9.71 dB)

  Ts = 0.00020 s:  GM = 30.508 (29.69 dB)
```

Part 4

```
omega_c_P = 60;                 % desired crossover (rad/s)
```

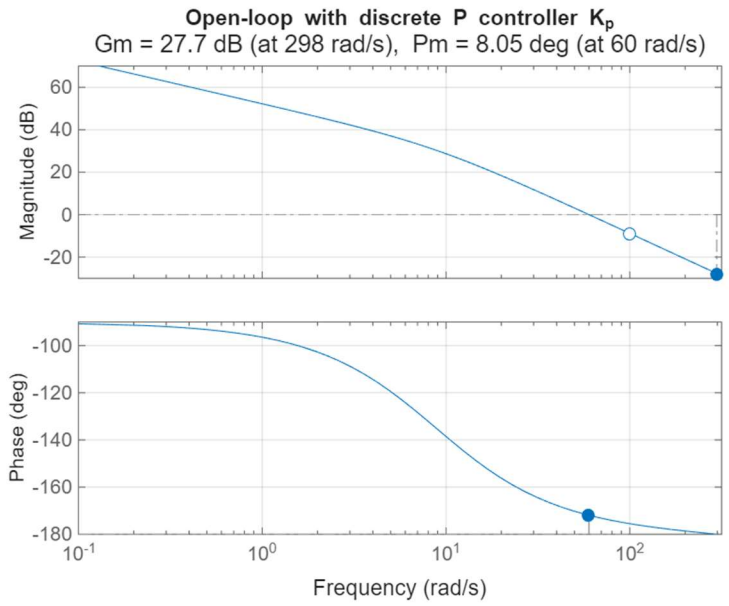
```
Omega_d   = omega_c_P * T;  % discrete freq (rad/sample)
z_P       = exp(1j * Omega_d);

[num_z, den_z] = tfdata(Gol_z, 'v');
Gz_ejO = polyval(num_z, z_P) / polyval(den_z, z_P);

Kp      = 1 / abs(Gz_ejO);     % gain for |Kp*G_ol(z)| = 1 at ω_c
P_ctrl_z = tf(Kp, 1, T); % discrete P controller
L_P_z    = series(P_ctrl_z, Gol_z);

fprintf('\nP-controller design:\n');
```

```
P-controller design:
```

```
fprintf('  Kp chosen for ω_c = 60 rad/s in z-domain: Kp = %.4f\n', Kp);
```

```
  Kp chosen for ω_c = 60 rad/s in z-domain: Kp = 1.2531
```

```
figure(4);
margin(L_P_z);
grid on;
title('Open-loop with discrete P controller K_p');
```



Part 5

```
omega_c_LL = 377;          % desired crossover (rad/s) = 60 Hz
phi_max    = 60 * pi/180;  % desired max phase lead (rad)
```

```
% a from  sin(phi_max) = (a-1)/(a+1)
a_lead = (1 + sin(phi_max)) / (1 - sin(phi_max));      % ≈ 13.9

% T from ω_max = 1 / (T * sqrt(a))
T_lead = 1 / (omega_c_LL * sqrt(a_lead));              % ≈ 7.1e-4 s

% Magnitude of plant at ω_c (continuous)
G_at_wc = squeeze(freqresp(Gol, omega_c_LL));
G_at_wc = G_at_wc(1);

% Magnitude of lead network (without K) at ω_c
C0_at_wc = (1 + 1j * a_lead * T_lead * omega_c_LL) / ...
           (1 + 1j * T_lead * omega_c_LL);

% K so that |K*C0(jωc)*G(jωc)| = 1
K_lead = 1 / (abs(C0_at_wc) * abs(G_at_wc));

% Continuous lead controller
C_lead_s = K_lead * (1 + a_lead * T_lead * s) / (1 + T_lead * s);

% Discrete lead controller (Tustin)
C_lead_z = c2d(C_lead_s, T, 'tustin');

% Open-loop with lead in discrete domain
L_lead_z = series(C_lead_z, Gol_z);

fprintf('\nStep 5 – Lead compensator design:\n');
```

```
Step 5 – Lead compensator design:
```

```
fprintf('  a      = %.3f\n', a_lead);
```

```
a      = 13.928
```

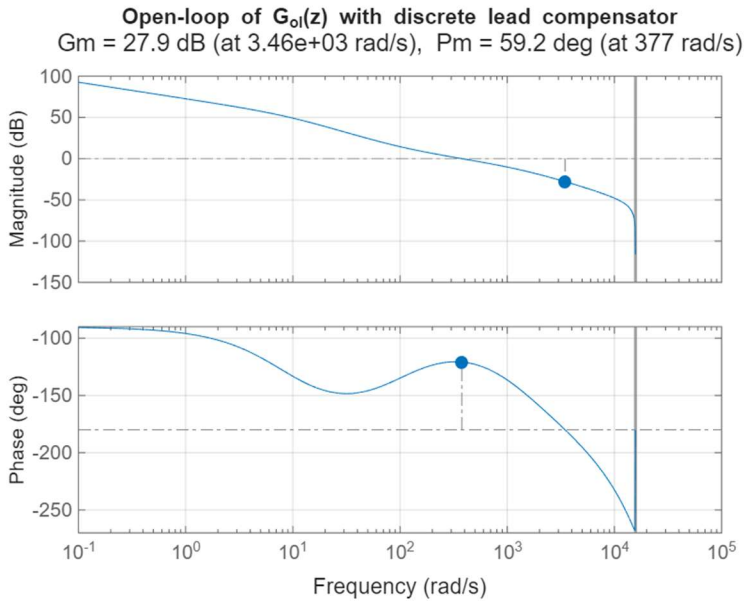
```
fprintf('  T      = %.4e s\n', T_lead);
```

```
T      = 7.1074e-04 s
```

```
fprintf('  K_lead = %.4f\n', K_lead);
```

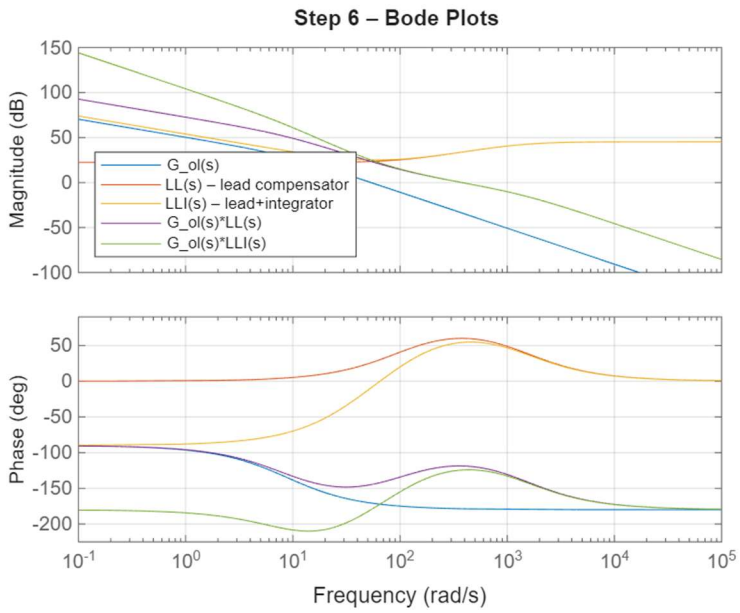
```
K_lead = 13.1174
```

```

figure;
margin(L_lead_z);
grid on;
title('Open-loop of G_o_l(z) with discrete lead compensator');
```



Part 6

```
Ki_int      = omega_c_LL / 10;          % ≈ 37.7
C_leadInt_s = (Ki_int + s)/ s * C_lead_s;  % lead + integrator

% Open-loop products in continuous domain
L_lead_s    = series(C_lead_s,    Gol);   % G_ol(s)*LL(s)
L_leadInt_s = series(C_leadInt_s, Gol);   % G_ol(s)*LLI(s)

figure(6);
bode(Gol, C_lead_s, C_leadInt_s, L_lead_s, L_leadInt_s);
grid on;
legend('G_ol(s)', ...
       'LL(s) – lead compensator', ...
       'LLI(s) – lead+integrator', ...
       'G_ol(s)*LL(s)', ...
       'G_ol(s)*LLI(s)', ...
       'Location','SouthWest');
title(['Step 6 – Bode Plots ', ...
       '']);
```




```
fprintf('\nIntegral action:\n');
```

```
Integral action:
```

```
fprintf('  K_i (integral gain) = %.4f\n', Ki_int);
```

```
  K_i (integral gain) = 37.7000
```

Lab 1

```
close all;
function T = import_xlsx_table(xlsxPath, nHeaderRows, varNames, sheet)
%IMPORT_XLSX_TABLE Read an Excel sheet as a table, skipping header rows and
empty tails.
%   T = IMPORT_XLSX_TABLE(xlsxPath, nHeaderRows, varNames)
%   T = IMPORT_XLSX_TABLE(xlsxPath, nHeaderRows, varNames, sheet)

    if nargin < 4 || isempty(sheet), sheet = 1; end
    if ~isfile(xlsxPath), error('File not found: %s', xlsxPath); end
    if ~isscalar(nHeaderRows) || nHeaderRows < 0 || nHeaderRows ~=
floor(nHeaderRows)
        error('nHeaderRows must be a non-negative integer.');
    end

    % Normalize varNames -> cellstr
    if isstring(varNames), varNames = cellstr(varNames(:)); end
    if ischar(varNames),   varNames = cellstr(varNames);    end
```

```
    if ~iscellstr(varNames)
        error('varNames must be a cell array or string array of names.');
    end

    % Build rows-only range: (nHeaderRows+1) : end-of-sheet
    startRow = nHeaderRows + 1;
    rowSpan  = sprintf('%d:%d', startRow, 1048576);  % XLSX max rows

    % Read data only (no headers taken from the sheet)
    T = readtable(xlsxPath, ...
        'Sheet', sheet, ...
        'Range', rowSpan, ...
        'ReadVariableNames', false, ...
        'VariableNamingRule', 'preserve');

    % Drop rows that are entirely empty/missing (trailing or anywhere)
    if ~isempty(T)
        T = rmmissing(T, 'MinNumMissing', width(T));   % remove rows where all
entries are missing
    end

    % Validate width vs provided names, then assign clean names
    if width(T) ~= numel(varNames)
        error('Column count mismatch: table has %d columns; you provided %d
names.', ...
              width(T), numel(varNames));
    end
    T.Properties.VariableNames = matlab.lang.makeValidName(varNames,
'ReplacementStyle','delete');
 end

 kpramp = import_xlsx_table(fullfile(pwd,'Data','kp_ramp.xlsx'), 11,
{'time','ref','pos1','pos2','vel1','vel2','vout'}, 'kp_ramp');
 kpstep = import_xlsx_table(fullfile(pwd,'Data','kp_step.xlsx'), 11,
{'time','ref','pos1','pos2','vel1','vel2','vout'}, 'kp_step');
 llramp = import_xlsx_table(fullfile(pwd,'Data','ll_ramp.xlsx'), 11,
{'time','ref','pos1','pos2','vel1','vel2','vout'}, 'll_ramp');
 llstep = import_xlsx_table(fullfile(pwd,'Data','ll_step.xlsx'), 11,
{'time','ref','pos1','pos2','vel1','vel2','vout'}, 'll_step');
 llicramp = import_xlsx_table(fullfile(pwd,'Data','llic_ramp.xlsx'), 11,
{'time','ref','pos1','pos2','vel1','vel2','vout'}, 'llic_ramp');
 llicstep = import_xlsx_table(fullfile(pwd,'Data','llic_step.xlsx'), 11,
{'time','ref','pos1','pos2','vel1','vel2','vout'}, 'llic_step');
```

Comparison of Experiments and Simulation

```
function [overshoot_pct, rise_time_ms] = analyze_step_response(T)
    % ANALYZE_STEP_RESPONSE Calculates overshoot and rise time from a table
    % Inputs:
    %   T: A table with columns 'time' (ms) and 'ref' (mm)

    % Extract arrays for easier handling
    t = T.time;
    y = T.pos2;

    %% 1. Determine Steady State and Step Params
    % Assume the last 5% of the data represents steady state
    num_points = length(y);
    ss_window = floor(0.05 * num_points);
    if ss_window < 1, ss_window = 1; end

    y_final = mean(y(end-ss_window:end)); % Steady state value
    y_initial = y(1);                     % Initial value
    step_amp = y_final - y_initial;       % Step height

    %% 2. Calculate Overshoot
    % Percent by which the peak exceeds the steady state
    [y_max, ~] = max(y);
    overshoot_val = y_max - y_final;

    if step_amp ~= 0
        overshoot_pct = (overshoot_val / step_amp) * 100;
    else
        overshoot_pct = 0;
    end

    % Cap at 0 if overdamped (no overshoot)
    if overshoot_pct < 0
        overshoot_pct = 0;
    end

    %% 3. Calculate Rise Time (10% to 90%)
    % Threshold values
    v10 = y_initial + 0.10 * step_amp;
    v90 = y_initial + 0.90 * step_amp;

    % Find exact times using Linear Interpolation helper
    t10 = find_crossing_time(t, y, v10);
    t90 = find_crossing_time(t, y, v90);
```

```
    if ~isnan(t10) && ~isnan(t90)
        rise_time_ms = t90 - t10;
    else
        rise_time_ms = NaN;
    end

    %% 4. Print Results
    fprintf('--- Step Response Analysis ---\n');
    fprintf('Initial Value:   %.4f mm\n', y_initial);
    fprintf('Final Value:     %.4f mm\n', y_final);
    fprintf('Peak Value:      %.4f mm\n', y_max);
    fprintf('--------------------------\n');
    fprintf('Overshoot:       %.2f %%\n', overshoot_pct);
    if isnan(rise_time_ms)
        fprintf('Rise Time:       Undefined (Did not reach 90%%)\n');
    else
        fprintf('Rise Time:       %.4f ms\n', rise_time_ms);
    end
end

function t_cross = find_crossing_time(t, y, threshold)
    % Finds the first time 't' where 'y' crosses 'threshold'
    % Uses linear interpolation between samples for accuracy.

    % Find the first index where y exceeds threshold
    idx = find(y >= threshold, 1);

    if isempty(idx) || idx == 1
        % Threshold never reached or reached at very first index
        t_cross = NaN;
        if idx == 1, t_cross = t(1); end
        return;
    end

    % Get points before (idx-1) and after (idx) the crossing
    y1 = y(idx-1);
    y2 = y(idx);
    t1 = t(idx-1);
    t2 = t(idx);

    % Linear Interpolation formula: t = t1 + (val - y1) * slope
    fraction = (threshold - y1) / (y2 - y1);
    t_cross = t1 + fraction * (t2 - t1);
end
```

```
[kpos, kprt] = analyze_step_response(kpstep);
```

```
--- Step Response Analysis ---
Initial Value:   0.0000 mm
Final Value:     1.0631 mm
Peak Value:      1.2214 mm
--------------------------
Overshoot:       14.89 %
Rise Time:       27.6894 ms
```

```
[llos, llrt] = analyze_step_response(llstep);
```

```
--- Step Response Analysis ---
Initial Value:   0.0000 mm
Final Value:     0.9780 mm
Peak Value:      0.9860 mm
--------------------------
Overshoot:       0.82 %
Rise Time:       10.4163 ms
```

```
[llicos, llicrt] = analyze_step_response(llicstep);
```

```
--- Step Response Analysis ---
Initial Value:   0.0000 mm
Final Value:     1.0008 mm
Peak Value:      1.3494 mm
--------------------------
Overshoot:       34.83 %
Rise Time:       8.3945 ms
```

Ramp responses

```
function vel_error = analyze_velocity_error(T, target_velocity, final_time_ms)
    % ANALYZE_VELOCITY_ERROR Calculates steady state velocity error up to a
specific time
    % Inputs:
    %   T: Table with columns 'time' (ms) and 'vel2' (units/s)
    %   target_velocity: The commanded constant speed (scalar)
    %   final_time_ms: The cut-off time (ms) to end analysis
```

```
    %% 1. Filter Data by Time
    % Find rows where time is less than or equal to the specified cut-off
    valid_indices = T.time <= final_time_ms;

    % Extract the velocity data only for the valid time window
    v_segment = T.vel2(valid_indices);
    t_segment = T.time(valid_indices);

    % Check if we have data remaining
    if isempty(v_segment)
        error('No data found within the specified time range (0 to %.2f ms).',
final_time_ms);
    end

    %% 2. Calculate Steady State Velocity
    % Average the last 10% of the *filtered* segment
    num_points = length(v_segment);
    ss_window = floor(0.10 * num_points);

    if ss_window < 1
        ss_window = 1;
    end

    % Compute average of the tail end of the segment
    v_steady_state = mean(v_segment(end-ss_window:end));

    %% 3. Calculate Error
    vel_error = target_velocity - v_steady_state;
    error_pct = (vel_error / target_velocity) * 100;

    %% 4. Print Results
    fprintf('--- Velocity Error Analysis ---\n');
    fprintf('Analysis Window:     0 to %.2f ms\n', t_segment(end));
    fprintf('Target Velocity:     %.4f\n', target_velocity);
    fprintf('Actual SS Velocity:  %.4f\n', v_steady_state);
    fprintf('Error (Abs):         %.4f\n', vel_error);
    fprintf('Error (%%):           %.2f%%\n', error_pct);
    fprintf('-------------------------------\n');
end

kper = analyze_velocity_error(kpramp, 5, 2000);
```

```
--- Velocity Error Analysis ---
Analysis Window:     0 to 2000.00 ms
Target Velocity:     5.0000
```

```
Actual SS Velocity:  4.8776
Error (Abs):         0.1224
Error (%):           2.45%
------------------------------
```

```
ller = analyze_velocity_error(llramp, 5, 2000);
```

```
--- Velocity Error Analysis ---
Analysis Window:     0 to 2000.00 ms
Target Velocity:     5.0000
Actual SS Velocity:  4.9742
Error (Abs):         0.0258
Error (%):           0.52%
------------------------------
```

```
llicer = analyze_velocity_error(llicramp, 5, 2000);
```

```
--- Velocity Error Analysis ---
Analysis Window:     0 to 2000.00 ms
Target Velocity:     5.0000
Actual SS Velocity:  4.9962
Error (Abs):         0.0038
Error (%):           0.08%
------------------------------
```