Dynamics and Control of Linear Systems

Part 1: Vibrations and Aeroelasticity

Ethan Sheehan
Student number: 2400720
Username: fc23871

Task 1: Single degree-of-freedom analysis

Results:

All relevant physical parameters of the airfoil were calculated using the equations provided below and can be seen in table 1. The torsional stiffness k, was calculated using Equation (1) and (2), where x_0 was found by taking the steady state amplitude of the Time-amplitude response of the 1 DoF system with constant force of 100N.[1]

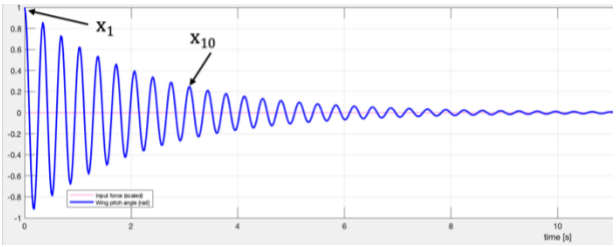


Figure 1, Time-amplitude plot of 1DOF system with initial angular displacement of 1rad

$$k = \frac{M}{x_0} = \frac{10}{0.0288}, \qquad M = Fb = 100 \times 0.1 \qquad (1,2)$$

$$I = \frac{k}{{\omega_n}^2} = \frac{346.8766}{18.3235^2}, \quad \omega_n = \frac{\omega_d}{\sqrt{1-\zeta^2}} = \frac{18.3178}{\sqrt{1-0.025^2}} \qquad (3,4)$$

$$\zeta = \frac{\Lambda}{2\pi} = \frac{0.1569}{2\pi}, \qquad \omega_d = \frac{2\pi}{T} = \frac{2\pi}{0.343} \qquad (5,6)$$

$$\Lambda = \frac{1}{N}\ln\left(\frac{x_1}{x_{10}}\right) = \frac{1}{9}\ln\left(\frac{1}{0.2436}\right) \qquad (7)$$

$$c = 2\zeta\sqrt{Ik} = 0.9456 \qquad (8)$$

The mass moment of inertia I was then calculated using Equation (3-7), first by modelling the 1 DoF system with an initial angular rotation of 1rad as seen in Figure 1. The 1st and 10th peaks were used to find the logarithmic decrement, Λ, and then the damping ratio, ζ.[2] Finally the torsional damping, c, was calculated using Equation(8).

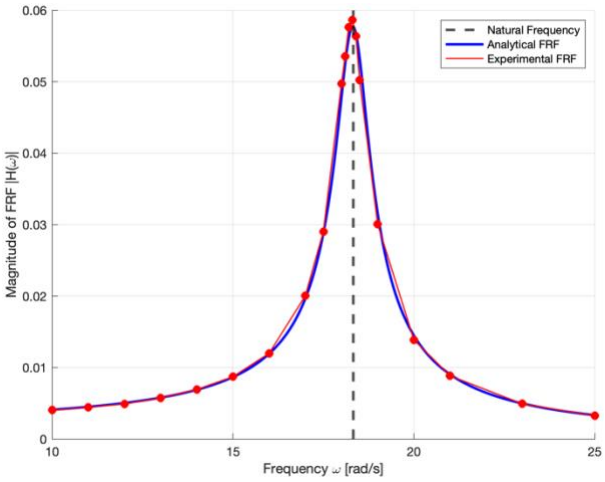


Table 1: Calculated 1DOF physical parameters

Physical Parameters	
Natural Frequency, ω_n [rad/s]	18.3235
Mass Moment of Inertia, I [kgm^2]	1.0331
Spring Torsional Stiffness, k [Nm/rad]	346.8766
Torsional Damping, c [Nms/rad]	0.9456

Figure 2, 1DOF FRF obtained using a sequence of a chosen number of harmonic excitation, overlayed with the FRF calculated analytically, and the 1DOF undamped natural frequency.[3]

Discussion:

The parameter with the highest level of uncertainty was the torsional damping. This is due to the fact that both other parameters (k and I) are used when calculating , c as seen in Equation (8) so its error will be a combination of the others, hence it will be highest.

NASA employed FRFs to validate their NASTRAN FEA model of the AH-64 airframe and find its parameters, similarly to what was done in this task.[4] The motivation in picking this report was it demostrates how experimental vibration data can be correllated with analytical simulations to ensure the accuracy of a model.

[1]Titurus, B. DCLS - V&A - Lecture 2, Stiffness, Damping, Inertia.
[2]Titurus, B. DCLS - V&A - Lecture 4, Free damped vibration.
[3]Titurus, B. DCLS - V&A - Lecture 8, Transfer Functions.
[4] Ferg, D. and Weisenburger, R. (1990). PLAN, EXECUTE, AND DISCUSS VIBRATION MEASUREMENTS AND CORRELATIONS TO EVALUATE A NASTRAN FINITE ELEMENT MODEL OF THE AH-64 HELICOPTER AIRFRAME. NASA Contractor Report 181973 - MCDONNELL DOUGLAS HELICOPTER COMPANY.

Task 2: Tuning the Tuned Vibration Absorber

Results:

Table 2: Calculated TVA physical parameters

TVA Parameters			
Natural Frequency, ω_n [rad/s]	Mass of TVA, m_{TVA} [kgm^2]	Stiffness of TVA, k_{TVA} [Nm/rad]	Damping of TVA, c_{TVA}, [Nms/rad]
18.3235	0.2551	85.6485	0.9348

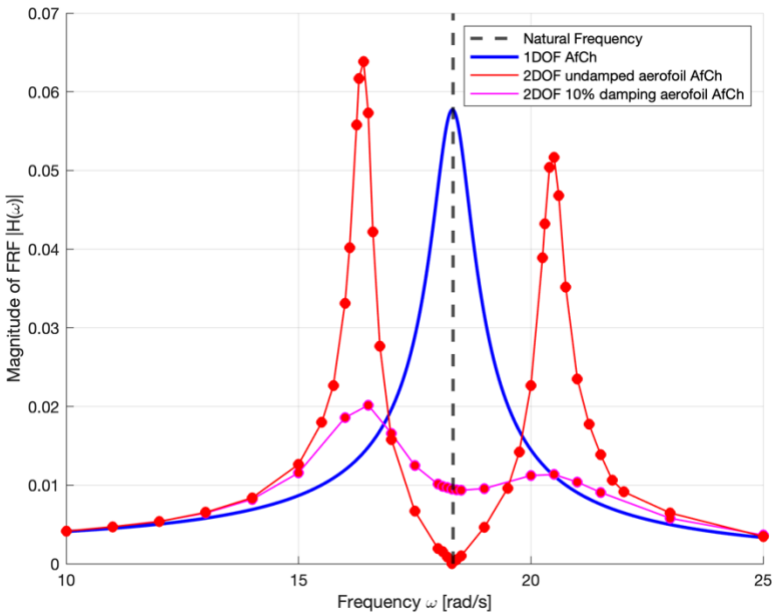


Figure 3, 1DOF AfCh overlayed with 2DOF aerofoil AfCh when TVA is undamped and 2DOF aerofoil AfCh when TVA damping is 10%

The TVA was tuned to absorb the resonance of the system using Equation (9), the tuning condition. This required the equivalent mass, m_{eq}, which was found using Equation (12), making use of the fact that the distance between the aerofoil hinge and the TVA attachment point is known, thus the radius of gyration is known. Finally, the TVA mass must not exceed 5% of the equivalent mass of the aerofoil, so Equation (10) was used to find the TVA mass and then Equation (11), a rearranged version of the tuning condition equation was used to find the the TVA stiffness.[5]

For the damped case, the damping ratio, ζ, was taken as 0.1 to fulfil the 10% damping requirement and the damping of the TVA, c_{TVA}, was found using Equation (13). [6]

$$\omega_n{}^2 = \frac{k}{m} = \frac{k_{TVA}}{m_{TVA}} = \omega_{TVA}{}^2, \qquad m_{TVA} = 0.05 m_{eq} = 0.05 \times 5.1019, \qquad k_{TVA} = \frac{m_{TVA}}{\omega_n{}^2} = \frac{0.2551}{18.3235^2} \qquad (9-11)$$

$$m_{eq} = \frac{I}{K^2} = \frac{1.0331}{0.45^2}, \qquad c_{TVA} = 2\zeta\sqrt{m_{TVA} k_{TVA}} = 2 \times 0.1\sqrt{0.2551 \times 85.6485} = 0.9348 \qquad (12, 13)$$

Discussion:

The undamped 2DOF AfCh has sharp, high peaks and a deep, sharp anti-resonance. The introduction of the 10% damping smooths the amplitude-frequency response by effectively dissipating the energy over a wider frequency range, reducing the peak amplitudes and extreme resonance at the cost of less anti-resonance as described by SJ Jang.[7]

NASA explored applying TVAs to mitigate vibrations in large space structures by optimizing their parameters to absorb resonant frequencies similar to how it was done in this task.[8] This paper was chosen since it shows these same principles apply at much larger scale, in much harsher environments, and far more complex aero contexts.

[5]Titurus, B. DCLS - V&A - Lecture 14, Tuned Vibration Absorbers.
[6]Titurus, B. DCLS - V&A - Lecture 15, Forced 2DOF systems with damping.
[7]Jang, S.J., Brennan, M.J., E. Rustighi and Jung, H.-J. (2012). A simple method for choosing the parameters of a two degree-of-freedom tuned vibration absorber. Journal of sound and vibration, 331(21).
[8] Card, M.F. and Peebles, S.W. (1982). PRELIMINARY SIZING OF VIBRATION ABSORBER FOR SPACEMAST STRUCTURE. NASA Technical Memorandum 8448.

Task 3: Equations of motion for 2 DOF system

Results:

Parameters	
Mass 1, [kg]	10
Mass 2, [kg]	0.5
Stiffness 1, [N/m]	100
Stiffness 2, [N/m]	150
Stiffness 3, [N/m]	500

Table 3: Parameters of 2DOF System

$$\boldsymbol{EoM} = \boldsymbol{M}\ddot{q} + \boldsymbol{K}q = \begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix}\begin{bmatrix} \ddot{x} \\ \ddot{x} \end{bmatrix} + \begin{bmatrix} k & bk_3 \\ bk_3 & b^2k_3 \end{bmatrix}\begin{bmatrix} x \\ x \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} \quad (14)$$

$$\mathcal{L} = E_K - E_P = \left[\tfrac{1}{2}M_1\dot{x}_1{}^2 + \tfrac{1}{2}M_2\dot{x}_2{}^2\right] - \left[\tfrac{1}{2}k_1x_1{}^2 + \tfrac{1}{2}k_2x_2{}^2 + \tfrac{1}{2}k_3x_1{}^2 + k_3bx_1x_2 + \tfrac{1}{2}k_3b^2x_1{}^2\right] \quad (15)$$

$$q_1 = x_1, \qquad q_2 = x_2, \qquad (\boldsymbol{K} - \omega^2\boldsymbol{M})A = 0 \quad (16-18)$$

$$b = \tfrac{a}{1-a}, \quad k = k_1 + k_2 + k_3, \quad \tfrac{d}{dt}\left(\tfrac{\partial \mathcal{L}}{\partial \dot{q}_i}\right) - \tfrac{\partial \mathcal{L}}{\partial x_i} = Q_i \quad (19-21)$$

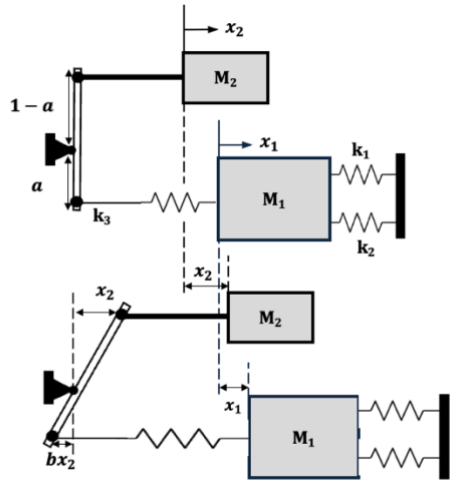


Figure 4, 2DOF system diagram

The system is described by generalized coordinates as seen in Equation (14) representing the displacements of masses M_1 and M_2. The system parameters include the mass values M_1 and M_2, the stiffness coefficients k_1, k_2, and k_3, the equivalent stiffness k and the constant a. The Lagrangian, given by Equation 15, consists of kinetic energy, E_K, representing the motion of the masses, and the potential energy, E_P, representing the elastic forces in the system, including individual stiffness contributions and coupling effects between x_1 and x_2. The lever induced displacement factor, b, was found using small angle aproximation and studying the movement of the lever, becoming simply the ratio fo the lever arms[9]

In order to find the natural frequency ω, the system was assumed to able to vibrate freely at a single frequency and model its motion as a function. Then this assumed motion function is used to find the free response of system, from which the vector of displacements and acceleration can be subsituted into the EOM, resulting in Equation (18): the linear matrix equation. A matlab function can be used to solve for the ω^2, the eigenvalues of the problem, which are the natural frequencies squared, and this can be done for each value of a from 0 to 1. [10]

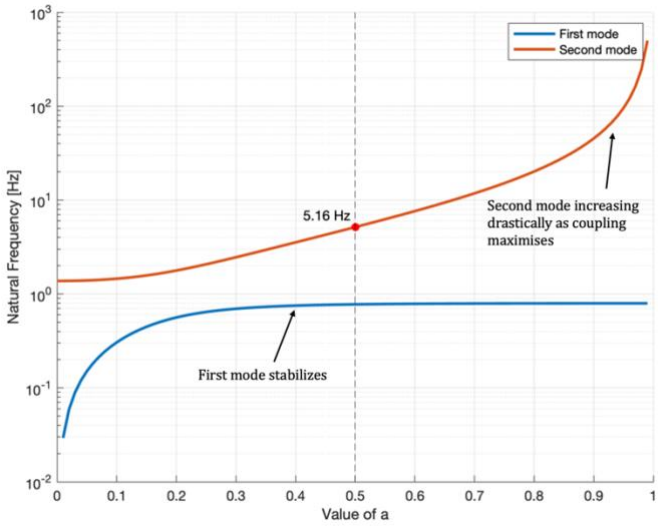


Figure 5, calculated changes in natural frequency of the 2DOF system with varying value for a

Discussion:

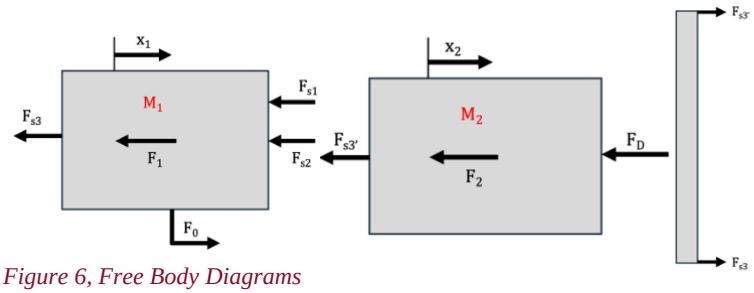
The results indicate that the first natural frequency stabilizes when $a > 0.3$ but the second mode increases sharply as a increases. M_2's motion on the system is amplified by the lever, introducing the nonlinear frequency growth due to the increased influence of lever-induced displacement, which enhances coupling effects as a increases and modifies stiffness distribution. [11]

Deployable space structures, such as solar arrray and antennas, can benefit from this sort of system arrangement as the natural frequency and stiffness can changed to dynamically mitigate vibrations and instabilities. A similar approach was used in NASA's ASTREX testbed, which studied lever-based mechanisms to enhance vibration control in space systems. [12]

[9]Titurus, B. DCLS - V&A - Lecture 16, Energy Methods.
[10]Titurus, B. DCLS - V&A - Lecture 11, Free Vibration Characteristics.
[11] Kocak, K. and Yilmaz, C. (2023). Design of a compliant lever-type passive vibration isolator with quasi-zero-stiffness mechanism. Journal of Sound and Vibration, 558.
[12] Carey, W.T. (1967). FINAL REPORT . LARGE SPACE STRUCTURE EXPERIMENTS FOR AAP . NASA.

Task 4: Damping in vibrating systems

Results:



F_1, F_2: Inertial forces

$F_{s1,s2,s3}$: Force from springs 1, 2, or 3

F_D: Force from damper

$x_{1,2}$: Displacement of masses 1 and 3

$F_{s3'}$: Force from spring 3 after the lever

Figure 6, Free Body Diagrams

$$F_0 - F_{s1} - F_{s2} - F_{s3} - F_1 = 0, \qquad M_1\ddot{x}_1 + k_1x_1 + k_2x_1 + k_3(x_1 + bx_2) = F_0 \tag{22,23}$$

Mass 1:

$$M_1\ddot{x}_1 + kx_1 + bk_3x_2 = F_0, \qquad b = \frac{a}{1-a}, \qquad k = k_1 + k_2 + k_3 \tag{24,25}$$

Mass 2:

$$F_2 - F_D - F_{s3'} = 0, \qquad M_2\ddot{x}_2 + c\dot{x}_2 + bk_3(x_1 + bx_2) = 0, \qquad M_2\ddot{x}_2 + c\dot{x}_2 + bk_3x_1 + b^2k_3x_2 = 0 \tag{26-28}$$

$$\boldsymbol{EoM} = \boldsymbol{M}\ddot{x} + \boldsymbol{C}\dot{x} + \boldsymbol{K}x = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}\begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & c \end{bmatrix}\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} + \begin{bmatrix} k & bk_3 \\ bk_3 & b^2k_3 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} F_0 \\ 0 \end{bmatrix} \tag{29}$$

The equations of motion above derived through Newton's approach matche the equations of motions derived in the previous task using Lagrange's approach, confirming their correctness. The same parameters were used as task 3, with the addition of a damping coefficient, c, of 0.01N, and a harmonic force, F_0, of 1kN. [13]

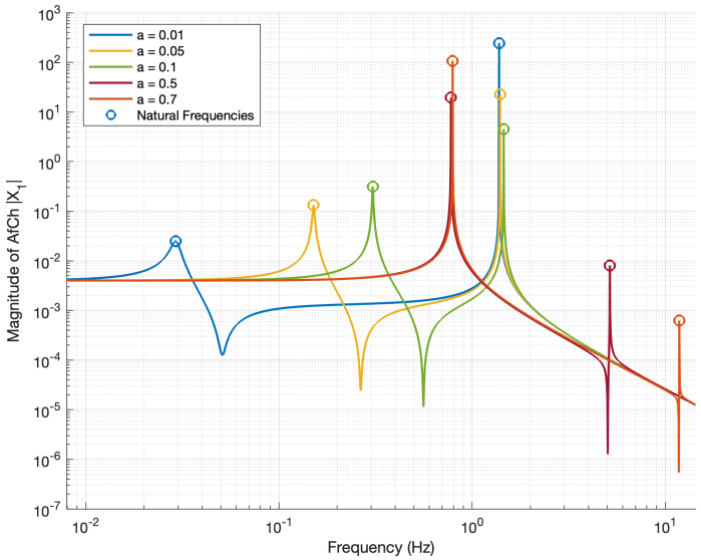


Figure 7, AfCh plots for the 2DOF system at various values of a on a log/log scale for visibility

Discussion

The damper reduces peak amplitudes in the frequency response, dissipating energy and mitigating excessive oscillations. For a selected AfCh, increased damping lowers resonance peaks and broadens the response, enhancing stability. As parameter a increases, damping effects diminish, leading to sharper peaks and increased vibrational energy retention. [14]

The ability to adjust damping properties, as demonstrated in NASA's Tension Element Damping (TED) system, is crucial for enhancing wind turbine stability. By tuning damping to specific vibrational modes, NASA's approach improves structural integrity and reduces unwanted oscillations. [15]

[13]Titurus, B. DCLS - V&A - Lecture 10, Intro to 2DOF systems.
[14]Titurus, B. DCLS - V&A - Lecture 13, Forced harmonic vibration.
[15] NASA (2025). Tension Element Damping (TED) With Hydraulics for Large Displacements | T2 Portal. Nasa.gov.

Task 5: Aeroelastic analysis of a 2DOF system

Results:

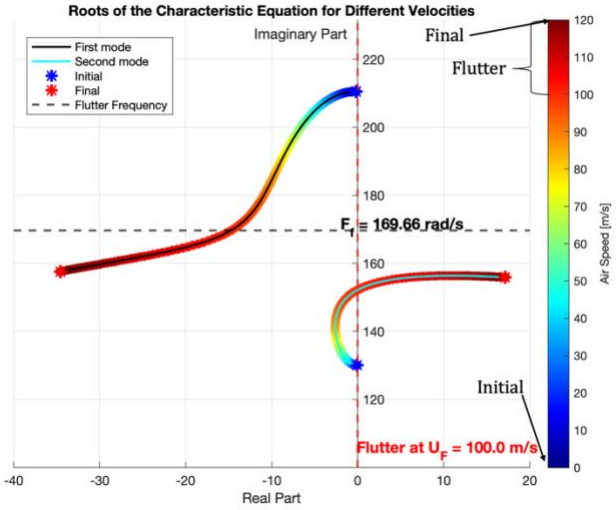


Figure 8, Roots of the characteristic equations at varying velocities.

Table 4: Chosen 2DOF parameters

Parameters	
Mass [kg]	10
Mass Moment of Inertia [kgm^2]	0.1
Distance to Aerodynamic Center [m]	-0.05
Distance to Center of Mass [m]	-0.075
Heaving stiffness [N/m]	2.5
Heaving damping [Ns/m]	0.1
Torsional stiffness [Nm/rad]	0.3
Torsional damping [Nm/rad]	0.05
Density [kg/m^2]	1.225
Chord length [m]	0.4
Wingspan [m]	1
Distance to Elastic Center [m]	-0.0625
Lift Curve Gradient	5
Rate of Change of Moment	-0.3
Flutter Speed [m/s]	100
Flutter Frequency [rad/s]	169.66
Vertical Harmonic Loading [kN]	1

The flutter speed, U_f, was determied by recording the speed at which the real part of the eigenvalue becomes positive. The flutter frequency, F_f, was found by computing the maximum imaginary part of the eigenvalue at the flutter speed. The uncertainty in the flutter speed is ±0.25 m/s or 0.5% total error. This estimate comes from the fact that velocties from 0 to 120 m/s were plotted with a 0.5 interval. The complex complex-valued steady state response of the system for the excitation frequency $0.99F_f$ and the air speed U_f, were found to be:

Heave : **- 0.0196 – 0.009l i** and Pitch: **- 0.2273 – 0.2128 i**

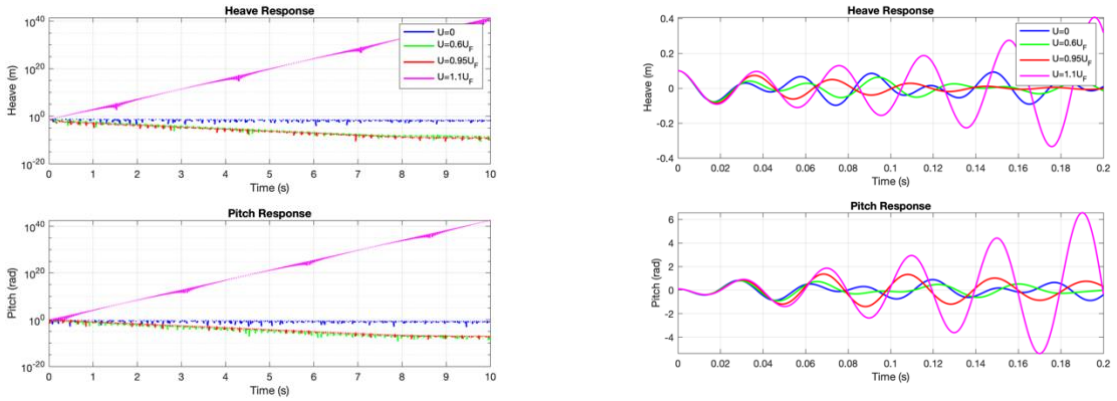


Figure 9, Free vibration response of pitch and heave of the the aeroelastic system at chosen airspeeds. Left is plotted on log scale, illustrating the diverging behavior of the response past the flutter speed and the rest being damped. The phase lag and amplitude difference of 0.6 U_F and 0.95 U_F for the pitch vs heave follows the corresponding mode shape and expected lag. Right shows the oscillatory nature of the flutter at a low timescale and better highlights the phase.

Discussion:

Structural damping helps delay flutter onset by dissipating vibrational energy, thus reducing oscillation amplitude as seen in Figure 9. The results show that at 100 m/s, instability occurs as eigenvalues turn positive, but damping influences the onset of this instability. Without sufficient damping, oscillations grow uncontrollably, increasing the risk of structural failure. [16]

Theodorsen's paper provides a fundamental understanding of flutter in aircraft structures, establishing classical 2-DOF models still relevant today. [17] NASA's report explores cutting-edge techniques, including computational simulations and active control strategies, to mitigate flutter in modern aircraft.[18] These reports were chosen as they highlight the shift from older analytical appraoches to newer computational simulations, both still being used to tackle comtemporary aerospace challenges.

[16]Titurus, B. DCLS - V&A - Lecture 19, Dynamic Aeroelasticity
[17] Theodorsen, T. (1949). General Theory of Aerodynamic Instability and the Mechanism of Flutter. NACA Report No. 496.
[18] Li, W., Geiselhart, K. and Robinson, J. (2022). Flutter Prediction for Aircraft Conceptual Design. NASA.

Appendix 1: V&A Task 5 Code

```
syms s;

%Parameters%%%%%%%%%%%%%
% Structural part
m = 10;
I = 0.1;

x_cm = -0.05;
x_ac = -0.02;

k_y = 2.5;
k_t = 0.03;

D_y = 0.1;
D_t = 0.05;

M_s = [m, x_cm*m; x_cm*m, I + m*x_cm^2];
D_s = [D_y, 0; 0, D_t];
K_s = 10^5 * [k_y, 0; 0, k_t];

% Aerodynamic parameters
rho = 1.225; % kg/m^3
c = 0.4; % m
S = 1; % m^2
ec = 0.025; % m
e = ec/c;
C_l = 5; % 1/rad
C_m = -0.3;

%Main Plot%%%%%%%%%%

%Setup
U_values = 0:0.5:120;
num_roots = 4;
real_v = nan(length(U_values), num_roots);
img_v = nan(length(U_values), num_roots);
flutter_found = false;
F_F = NaN;

%Main loop
for i = 1:length(U_values)
    U = U_values(i);

    % Aerodynamic matrices
    D_a = 0.5 * rho * U * [-s*c*C_l, 0; -s*c^2*e*C_l, s*c^3*C_m];
    K_a = 0.5 * rho * U^2 * [0, c*C_l; 0, c^2*e*C_l];

    % Combined matrices
    M = M_s * s^2;
    D = D_s * s - D_a;
    K = K_s - K_a;

    %Determinant
    A = M + D + K;
    det_A = det(A);

    %Polynomial's coefficients
    coeffs_s = coeffs(det_A, s, 'All');
    coeffs_numeric = double(coeffs_s);

    %Roots
    s_values = roots(coeffs_numeric);
    N = min(num_roots, length(s_values));
    real_v(i, 1:N) = real(s_values(1:N));
    img_v(i, 1:N) = imag(s_values(1:N));

    %Find flutter
    if any(real(s_values) > 0) && ~flutter_found
        U_F = U;
        F_F = max(abs(imag(s_values))); % Flutter frequency is the max
imaginary part
        flutter_found = true;
    end
end

%Plot
figure;
hold on;

%Main + scatter
colors = jet(length(U_values));
scatter(real_v(:, 1), img_v(:, 1), 50, U_values, 'filled');
h6 = plot(real_v(:, 1), img_v(:, 1), 'k','MarkerSize', 6, 'LineWidth', 1.5);


scatter(real_v(:, 3), img_v(:, 3), 50, U_values, 'filled');
h7 = plot(real_v(:, 3), img_v(:, 3), 'c', 'MarkerSize', 6, 'LineWidth', 1.5);

%Flutter plots
h8= xline(0, '--r', 'LineWidth', 1.5);
h9= yline(F_F, '--k', 'LineWidth', 1.5);
text(-3, F_F+1, sprintf('  F_f = %.2f rad/s', F_F), 'FontSize', 12, 'Color',
'k', 'FontWeight', 'bold');
text(-1.15, 105, sprintf('  Flutter at U_F = %.1f m/s', U_F), 'FontSize', 12,
'Color', 'r', 'FontWeight', 'bold');

% Initial + final
h1 = plot(real_v(1, 1), img_v(1, 1), 'b*', 'MarkerSize', 10, 'LineWidth', 2);
h2 = plot(real_v(end, 1), img_v(end, 1), 'r*', 'MarkerSize', 10,
'LineWidth', 2);
h3 = plot(real_v(1, 3), img_v(1, 3), 'b*', 'MarkerSize', 10, 'LineWidth', 2);
h4 = plot(real_v(end, 3), img_v(end, 3), 'r*', 'MarkerSize', 10,
'LineWidth', 2);
%h5 = plot(0, F_F, 'ok', 'MarkerSize', 8, 'LineWidth', 2);

%Format
grid on;
ylim([100, max(img_v(:)) * 1.1]); % Show only the upper half-plane
xlabel('Real Part');
ylabel('Imaginary Part');
title('Roots of the Characteristic Equation for Different Velocities');
%ylim([150 175])

%Colorbar
cb = colorbar;
colormap jet;
caxis([0 max(U)]);
set(cb, 'Ticks', 0:10:150);

%Format
ylabel(cb, 'Air Speed [m/s]');
legend([h6, h7, h1, h2, h9],'First mode', 'Second mode', 'Initial', 'Final',
'Flutter Frequency', 'Location', 'best');
ax = gca;
ax.XAxisLocation = 'origin';
ax.YAxisLocation = 'origin';
hold off;



% Free Vibration Response using Numerical
Integration%%%%%%%%%%%%%%%

%Setup
U_test = [0, 0.6*U_F, 0.95*U_F, 1.1*U_F];
tspan = linspace(0,0.2,1000);
y0 = [0.1; 0.1; 0; 0]; % Initial conditions [heave, heave_dot, pitch,
pitch_dot]

figure;
hold on;
colors = ['b', 'g', 'r', 'm'];

%Main loop
for j = 1:length(U_test)
    U_T = U_test(j);

    % Aerodynamic matrices
    D_a = 0.5 * rho * U_T * [-c*C_l, 0; -c^2*e*C_l, c^3*C_m];
    K_a = 0.5 * rho * U_T^2 * [0, c*C_l; 0, c^2*e*C_l];

    % Combined matrices
    M = M_s;
    D = D_s - D_a;
    K = K_s - K_a;

    % Convert second-order system to first-order
    A_sys = [zeros(2), eye(2); -M\K, -M\D];

    % Define system of ODEs
    odefun = @(t, y) A_sys * y;

    % Solve ODE
    [t, Y] = ode45(odefun, tspan, y0);

    % Plot
    subplot(2, 1, 1);
    plot(t, Y(:, 1), colors(j), 'LineWidth', 1.5); % Heave
    hold on;

    subplot(2, 1, 2);
    plot(t, Y(:, 2), colors(j), 'LineWidth', 1.5); % Pitch
    hold on;
end

%Plot

% Heave plot
subplot(2, 1, 1);
xlabel('Time (s)');
ylabel('Heave (m)');
%set(gca, 'YScale', 'log'); % Set log scale for y-axis
title('Heave Response');
legend('U=0', 'U=0.6U_F', 'U=0.95U_F', 'U=1.1U_F');
grid on;

% Pitch plot
subplot(2, 1, 2);
xlabel('Time (s)');
ylabel('Pitch (rad)');
%set(gca, 'YScale', 'log'); % Set log scale for y-axis
title('Pitch Response');
grid on;
hold off;



% Steady-State Response at Excitation Frequency 0.99 *
omega_F%%%%%%%%

% Excitation frequency
F_e = 0.99 * F_F;

% Aerodynamic matrices
D_a_ss = 0.5 * rho * U_F * [-c*C_l, 0; -c^2*e*C_l, c^3*C_m];
K_a_ss = 0.5 * rho * U_F^2 * [0, c*C_l; 0, c^2*e*C_l];

% Combined matrices
M_ss = M_s * F_e^2;
D_ss = (D_s - D_a)* F_e;
K_ss = K_s - K_a;

% Steady-State Response
A_ss = K_ss + 1i*D_ss - M_ss;
det_A_ss = det(A_ss);
A_ss_inv = inv(A_ss);

% Force
M_ac = 0;
L = 1000;

F = [L; x_ac*L + M_ac];

%F = [1000;0];

%Displacement
x_ss = (A_ss_inv * F)


%exportgraphics(figure(1), 'Task5_1.png', 'Resolution', 300)
%exportgraphics(figure(2), 'Task5_3.png', 'Resolution', 300)
```

Dynamics and Control of Linear Systems

Part 2: Signals, Systems and Control

Ethan Sheehan
Student number: 2400720
Username: fc23871

Academic integrity statement: OpenAI's ChatGPT-4 was used during code debugging in the form of error message explanations, also was used as a dictionary to search for MATLAB functions but not to generate code. No AI was used for data analysis or figure generation. The author has considered and included several suggested references and is fully accountable for the submitted work.

Task 1:

a)

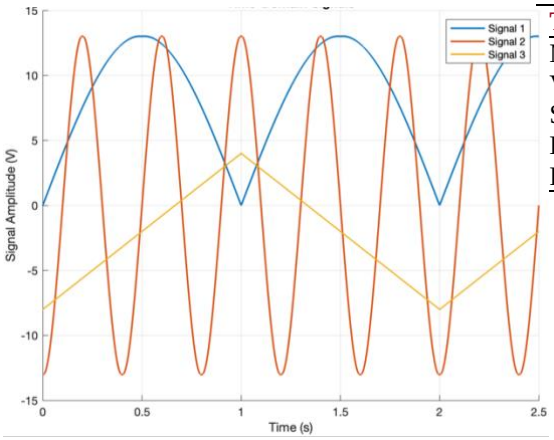


Figure 10, Time Domain plots for the 3 signals

Table 5: Time Domain Metrics Results

Time Domain Metrics	Signal 1	Signal 2	Signal 3
Mean [V]	8.2493	-0.0065	-2.0030
Variance [V2]	16.3050	84.4176	12.0180
Skewness	-0.4725	0.0007	0.0000
Kurtosis	1.9010	1.5122	1.8000
RMS [V]	9.1810	9.1856	4.0030

It is known that the 3 signals are triangular waveforms passed through an amplifier believed to be faulty. The first step taken in understanding the system's behavior was plotting the signal's amplitudes against time. From figure 10, the general shapes of the signals can be induced. At first glance, Signal 3 most resembles a triangular waveform, as Signal 2 and 3 have curved peaks. Signal 3 also has a noticeably lower amplitude than the others. Signal 2 has the highest frequency and Signal 1 seems to be reflected in the x axis. [19] The time domain metrics were then calculated and tabulated below, cutting off after a full period.

Table 6: Time Domain Metric Definitions

Metric	Description	Information gained	Still unknown
Mean	Indicates the signal's DC offset which reveals how the amplifier is shifting the baseline.	Signal 1 has the highest offset, Signal 2 has no offset and Signal 3 has negative offset.	Whether these offsets are intentional or a due to amplifier faults.
Variance	Describes the signal's power distribution and how much it deviates from the mean, revealing whether the amplifier is correctly aplifying the signal or introducing distortions such as clipping.	Signal 2 exhibits the highest variance possibly indicating clipping or a high frequency while Signal 3 has the lowest variance suggesting attenuation.	Whether the high varaiance results from intentional amplification or non-linear distortion and whether external noise is contributing to these fluctuations.
Skewness	Quantifies the symmetry of the waveform around its mean, for a perfect triangular wave this should be close to 0, with deviations suggesting distortions.	Signal 1 has negative skew indicating aymmetry likely caused by distortion or an unintened shift. Signal 2 and 3 maintains nearly 0 skewness suggesting proper amplification with symmetry preserved	Whether skewness varies with frequency and if the amplifier affects different parts of the signal's cycle differently
Kurtosis	Measures sharpness of signal's peaks. A perfect triangular wave should have a kurtosis of 1.8.	Signal 1 and Signal 3 both have a kurtosis slightly above the ideal value, suggesting minimal distortion but slightly sharper peaks than expected, Signal 2 has lower, indicating rounded peaks and potential clipping, attenuation, or non-linear distortion.	Whether the low kurtosis in Signal 1 is due to intentional signal shaping or an unintended amplifier effect
RMS	Determines the signal's true power and helps assess whether the amplifier is maintaining the correct amplitude (Root Mean Square)	Signals 1 and 2 have similar RMS values, indicating comparable overall power levels despite differences in DC offset and symmetry, Signal 3 has a significantly lower RMS, suggesting considerable power loss	Whether the amplifier's gain is nonlinear, potentially affecting different frequencies differently or if the lower RMS in Signal 3 by design

b)

Signal 1 has a high DC offset, indicating a significant shift to the baseline which could be how the signal actually is or an amplifier issue. This offset could also affect the amplifier's performance. Its moderatre variance suggests a stable power distribution with no sever clipping while its negative skewness indicates asymmetry likely due to distortion. The slightly elevated kurtosis suggests sharper peaks than expected but not to an extreme degree. Despite these distortions its RMS remains comparable to Signal 2 meaning its overall power level is maintained. These characteristics suggest that even though the signal experiences some amplifier-induced artifacts, it remains functional wihtout major power loss (assuming the generator outputs a constant power).

Signal 2 has by far the highest variance, indicating either very strong amplfification or potential clipping. The negligible skewness suggests symmetry is preserved but its low kurtosis indicate rounded peaks which could result from clipping. Its RMS remains similar to Signal 1, so the same conclusion can be drawn. While it is unclear if the distortion is intentional or due to amplifier limitations, the combination of high variance and low kurtosis suggests the possibility of non-linear effects.

Signal 3 has a negative DC offset and the lowest variance, implying attenuation rather than amplification. Its kurtosis is as expected meaning there is no severe peak distortion and it exhibits the shape of a triangular wave as it should. The significantly lower RMS suggests a large power loss, potentially due to attenuation or amplifier-related issues. To conclude, Signal 3 appears to be the most attenuated among the three with a shifted baseline and reduced power athough it retains a relatively undistorted waveform.

c)

Attempts can be made to diagnose the physical cause of the behaviors of the signals.

Since Signal 3 is the only signal to exhibit the expected triangular wave shape, this can be assumed to be the only signal that was correctly amplified so it can be concluded the DC offsets and different RMS values are by design. Since Signal 1 and 2 both have a higher RMS, this is most likely the root cause of the issue, explained by the signals exceeding the voltage limit of the amplfier leading to the flattening of the peaks.

The low kurtosis and symetric clipping occurs at both the positive and negative peaks of Signal 2 where the power is significanly greater than Signal 3. Signal 1 on the other hand only experiences asymetric clipping at its positive peaks, which, due to its positive DC offset, is the only place where it surpasses the power threshold. This is all explained by the theory that the amplifier isn't able to handle the higher power. This resolves all issues observed with Signal 2, asumming Signal 3 is perfect.

The other main discrepancy in the signals, the negative skew in Signal 1. A potential cause for this is a biasing issue in the amplifier not being properly centered at 0. Signal 1 is the only signal with a positive DC offset so the biased amplifier may be clipping the positive peaks earlier than the negative ones leading to a negative skew.

To conclude, the amplifier is not able to handle high powers, leading to clipping and it has a bias leading to skew for positive DC offsets. The amplifier must not have been specced correctly by spacecraft designers and is experiencing more power than it is built for. The only other explanations for this would require looking beyond the scope of the course and look at the inner working of amplifiers and finding an inner mechanical or electrical issue with it other than simply human design error, for example a mismatched component in the differential stage or a different common issue highlight by Aerotech in their fault troubleshooting documentation. [20][21]

---

[20] AEROTECH (2025). Amplifier Fault Troubleshooting. Aerotech.com.
[21] Analog Devices. Chapter 12: Differential amplifiers [Analog Devices Wiki].

Task 2:

a)

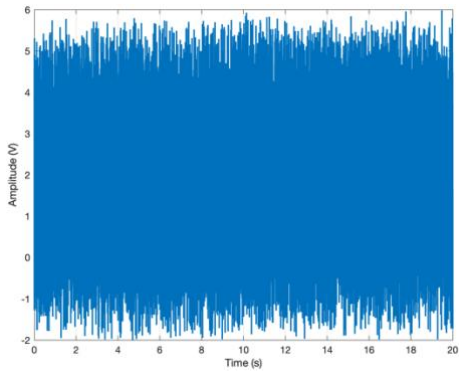


Figure 11, Time-amplitude plot of signal

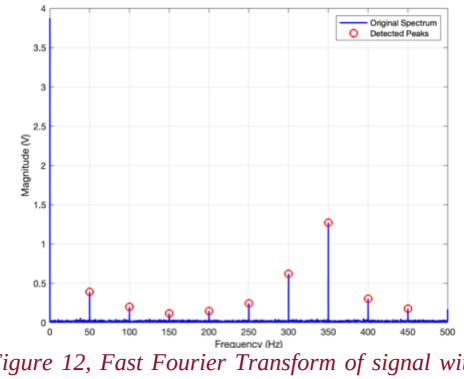


Figure 12, Fast Fourier Transform of signal with correct axis and dots on the identified peaks

The frequency axis of the FFT was scaled by first finding the sampling frequency which is the reciprocal of the time step. Then a sequence from 0 to N/2, where N is the total number of data point and it was halfed since real-valued signals are symmetric, was created and multiplied by the sampling frequcny to convert the values to Hz. Finally, this was divided by the N to normalize it. [22]

The magnitude axis of the FFT was scaled by first normalizing the FFT magnitudes by dividing the absolute values by N. Then, the magnitude was only taken from the positive prequency range, giving the single-sided spectrum and then doubled.

b)

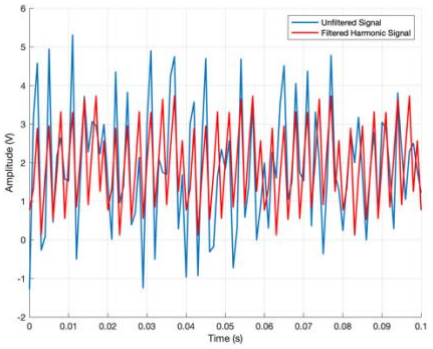
From the Time-Domain plot, it can be deduced that the signal has rapid fluctuations throughout. This could indicate the prescence of high frequency components in the signal. It can also be seen that the amplitude varies around from 6V to around -2V, centered around 2V, giving a DC offset of 2V. The signal is also spread quite consistently over the entire duration which suggests it is a stationary signal, not decaying or expanding over time. Finally there is lots of irregularity and variations throughout the signal. This looks like it could be caused by noise due to its seemingly random nature.

From the Magnitude Spectrum, it can be deduced that the strongest peak is at 350Hz which can be assumed to be the dominant frequency componenet. It can also be deduced that the signal has multiple frequency peaks at regular intervals which indicates harmonic components. The fact that these peaks are integer multiple of the fundamental frequency further displays harmonic properties. The detected strong peaks are noticeably higher than the baseline small peaks distrubuted relatively evenly across the spectrum. This indicates a low but continuous amount of noise.

c)

The unexpected harmonic components seen in the magnitude spectrum suggest the system likely picked up some noise and distortions during the test. The system is also experiencing a large amount of aliasing. As it's smapled at 1000Hz, only frequencies below 500Hz can be observed. The engineer is expecting to see the peak at the dominant frequency of the harmonic at 350Hz and then progressively lower magnitude peak at its integen multiples. Instead the peaks follow an pattern dictated by aliasing effects at the Nyquist frequency. [23]

Figure 13, Harmonic signal filtered by doing an inverse FFT after filtering out the noise and only keeping the peaks of the FFT



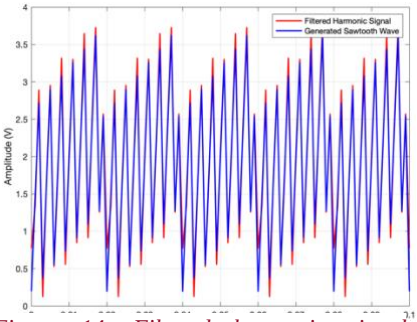


Figure 14, Filtered harmonic signal compared to a sawtooth wave matched to it with frequency of 350Hz. This sawtooth wave can be assumed to be test signal and what the engineers were epxecting to see.

[22]Burrow, S. DCLS - SS&C - Handout 1.6, Discrete Time-frequency transformations
[23]Burrow, S. DCLS - SS&C - Handout 1.5, Discrete signals and samping

Task 3:

a)

The bode plot was produced by first determining the transfer function of the system as seen in Equation 30.

$$Transfer\ Function = \frac{1}{ms^2 + cs} \qquad (30)$$

Where m is the mass of 20 grams and c is 9.4 Ns/m of damping of the linear bearing. This function was then plugged into MATLAB's "bode" function, which produced the bode plot. [24]

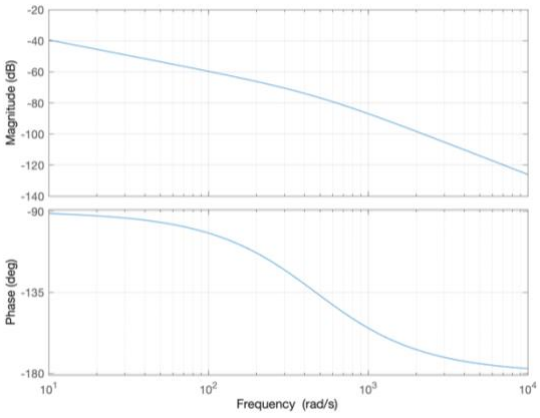


Figure 15, Bode Plot

b)

To determine the maximum excursion of the mirror, its dynamics were modelled as a second-order system with force input and position output. Given that the mirror has a mass and damping coefficient defined above, its equation of motion can be seen in Equation 31. The transfer function was evaluated, assuming zero initial conditions, as seen in Equation 32. This transfer function represents how the mirror's position responds to an applied force in the frequency domain. [25]

$$m\ddot{x} + c\dot{x} = F(t), \qquad H(s) = \frac{X(s)}{F(s)} = \frac{1}{-m\omega^2 + ic\omega}, \qquad \omega = 2\pi \times f, \qquad X = H \times F \times \frac{4}{\pi} = 0.000646 \quad (31-34)$$

Since the amplifier produces a 100 Hz square wave with amplitude ±5 V and force is directly proportional to voltage, the applied force is a square wave oscillating between ±5 N. The frequency was converted from Hz to radians using Equation 33 and this angular frequency was plugged back into the transfer function and its magnitude was found. Finally, this magnitude was multiplied by the force and multiplied by $\frac{4}{\pi}$ to convert back to a square wave (fourier series of a square has a factor of $\frac{4}{\pi}$ so need to multiply by this to convert the transfer function computed for a sine wave back to a square) as seen in Equation 34 and the maximum excursion was found to be 0.646mm. [26]

Task 4:

First step taken was building the 2 controllers in simulink since the goals and system definition have already been defined. Handout 3.2 was followed to build the Proportional with Rate Feedback (PR) and the Proportional-Derivative (PD) controllers. The basic simulink models can be seen below where K_p represents the proportional gain, K_d represents the derivative gain, and P(s) represents the plant, which for tuning is the simplified system but for evaluation is the original system. [27]

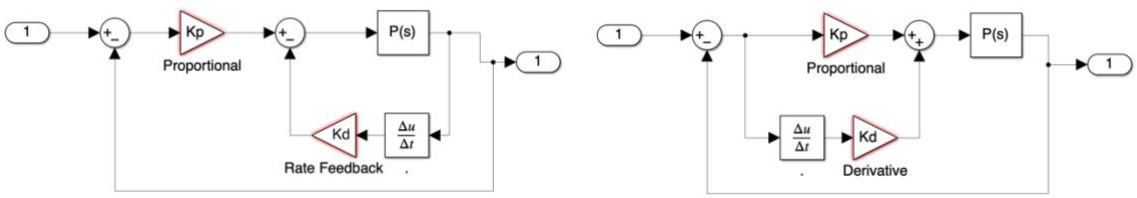


Figure 16, PR controller diagram (left) and PD controller diagram (right)

[24]Burrow, S. DCLS - SS&C - Handout 2.3, Poles, Zeros, and the Bode plot
[25]Burrow, S. DCLS - SS&C - Handout 1.4, Fourier and Laplace
[26]Stitt, R.M. (2000). SIMPLE FILTER TURNS SQUARE WAVES INTO SINE WAVES. Texas Instruments Incorporated.
[27]Burrow, S. DCLS - SS&C - Handout 3.2, Common Type of Controllers

The input was chosen to be a constant block outputting 1 to simulate the 1 rad reference input (desired setpoint) the system should track. A step input was also considered but ultimately gave near identical results, as the constant can be described as an instantaneous step, but the step made simulating small time steps more computationally heavy and error prone which hindered some simulations highlighted later in the discussion.

The second step, now that the controllers were set up and working, was picking the performance metrics to use to compare their performance. Four main metrics were chosen: Steady-state error, Overshoot, Settling time, and Rise time. Further metrics such as ITAE and ISE could have been considered but these 4 were considered the most important for this system. A brief description of each metric and how they were calculated is provided in the table below. [28]

Table 7: Controller performance metrics

Metric	Definition	Use	Calculation
Steady State Error (SSE)	The difference between the system's final output and the desired setpoint after transient effects have dissipated	Minimized (ideally zero) meaning final output is exactly the desired setpoint	$= Final\ Output - Setpoint$ (final output defined at T=500 but error will decrease at T increases)
Overshoot (OS)	The maximum deviation of the system output from the desired setpoint during the transient response as a %	Minimized (ideally zero) meaning maximum deviation is exactly the desiered setpoint	$= \frac{Peak\ Value - Setpoint}{100}$
Settling time (ST)	The time it takes for the system output to remain within a specified tolerance (2%) of the final value	Minimized meaning lower settling time indicate faster response	= when output first enters tolerance band and remains
Rise Time (RT)	The time it takes for the system to rise from 10% to 90% of its final value	Minimized meaning shorter rise time indicates quicker response to change	= time interval from 10% to 90%

These metrics were combined into a cost function as seen in Equation 35. A factor of 1 was applied to all except the SSE as this needs to be a significantly smaller value to be acceptable.

$$Cost = 1000 \times SSE + OS + ST + RT \tag{35}$$

The next step was tuning the gains of the controller. The task suggests a natural frequency for the closed loop system, ω_n of $0.06 \mathrm{rad/s}$, and a damping ratio, ζ, of 0.707. Because of this, the characteristic equation matching method was chosen to tune the gains. This method involves formulating a closed-loop transfer function using the controller structure and plant model (right side of Equation 36), then matching the coefficients (which turn out to the be gains, K_p and K_d) to the characteristic equation of the desired closed loop system (left side of Equation 36). The result ends up the same for both controllers as the transfer function for a PR and PD controller is the same.[29]

$$s^2 + 2\zeta\omega_n + {\omega_n}^2 = s^2 + \frac{K_d}{J}s + \frac{K_p}{J}, \qquad K_d = 2J\zeta\omega_n = 84.84, \qquad K_p = J{\omega_n}^2 = 3.6 \tag{36-38}$$

This tune was applied to both controllers as the equations work out to be the same and the results can be seen in figure 17 and tabulated in table 8. This was named the 'baseline' tune and will be used as a baseline for all other tunes. Objectively, the baseline tune isnt very good. It has very high rise and settles times for both the PR and PD controllers and a non-neglible overshoot. Other tuning methods were attempted in order to beat this tune.

The next tuning method was the "Manual" or "Trial and Error" method outlined in the handout. A matlab script was written to gradually increase the proportional gain from it's intial chosen value, K_{pi}, until the system began to oscillate at a constant amplitude. Then this proportional gain was halved and derivative gain was introduced and increased until overshoot was minimized. The handout suggests starting with a low K_{pi} value so 1 and 10 were chosen. Another parameter in this method was the step sizes for each gain iteration which was chosen to be 1. The Ziegler-Nichols closed loop method was also attempted. This method entails increasing K_p until a sustained oscillation is reached, similarly to the previous method, then determining the period of the oscillation and using this, in combination with the critical K_p found, to find K_d. This gave a good results for the PD controller at a high K_{pi} but resulted in an extremely high overshoot and settling time for the PR controller. Was tested at initial of 10 and 10,000.[29]

Finally, a Genetic Algorithm (GA) and a Particle Swarm Optimization MATLAB codes were written in an attempt to try the search/optimization tuning method. A lot of time went into selecting and fine tuning the parameters of these models to achieve good convergence and stable results. [30-33]

The table below shows the tuned gains and matrics for each method, with Manual 1 having initial condition of 1 and Manual 2 of 10. Ziegeler 1 of 10 and Zielgler 2 of 10000. GA 1 of 1 and GA 2 of 1000. PSO 1 of 10, PSO 2 of 100, and PSO 3 of 10000.

Table 8: Controller performance

Results	PR							PD						
	K_p	K_d	SSE	OS	ST	RT	Cost	K_p	K_d	SSE	OS	ST	RT	Cost
Baseline	3.6	84.84	2.3e-9	4.45	100.1	36.2	141	3.6	84.84	9.7e-7	4.05	97.1	36.2	137
Manual 1	1	60	7.8e-7	0.01	164.6	99.1	264	1	60	6.1e-7	0.00	168	99.1	267
Manual 2	5.5	146	2.3e-9	0.01	75.47	45.0	120	5.5	136	1.8e-8	0.01	67.1	40.6	108
Ziegler 1	6.6	50.16	4.6e-6	35.2	139.1	18.6	193	6.6	50.33	1.3e-6	35.7	137	18.0	191
Ziegler 2	6001	10800	0	13.1	28.55	7.49	49.1	6001	10800	2.7e-6	0.68	6.72	3.98	11.4
GA 1	6.33	85.8	2.7e-9	13.9	72.0	22.8	109	5.27	90.5	1.2e-6	7.99	81.0	26.7	116
GA 2	2288	6234	0	11.6	21.26	8.52	41.4	135.6	758.0	4e-16	1.83	25.2	10.2	37.2
PSO 1	38.99	409.9	2e-12	1.03	30.95	20.1	52.1	67.00	466.9	3.8e-7	0.47	14.3	11.2	25.9
PSO 2	318.5	2031	0	0.68	20.81	11.2	33.0	67.24	467.3	9.2e-6	0.55	23.7	11.2	35.4
PSO 3	3099 2421	56339 1	3.5e-5	0.38	0.01	0.01	0.44	7985 07	26922	4.3e-7	0.13	0.09	0.06	0.27

The results tabulated above show the best results after running each one for multiple times and the best was chosen. The colors from red (worst) to green (best) show a general overview of how each tune performed as a measure of their relative costs. The best and worst for each individual metric were also highlighted.

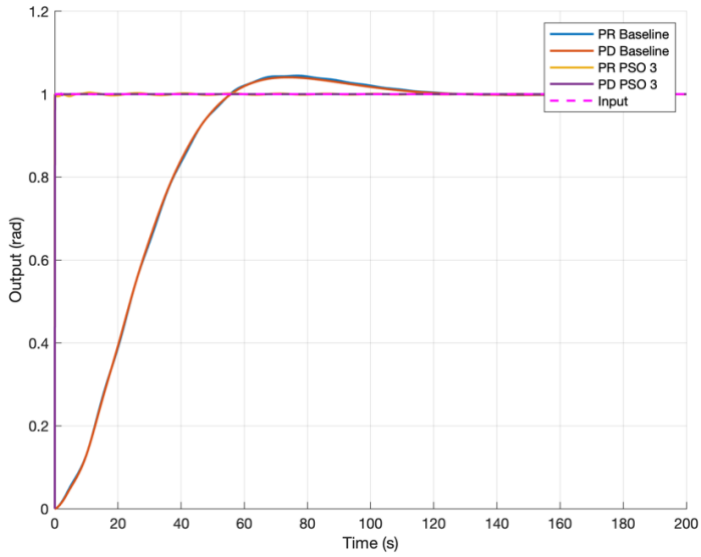


Figure 17, Control Response Diagram of Baseline vs PSO 3 tune. PSO 3 can be seen on the far left side.

Manual and Ziegels should have probably been run automatically through lots of numbers to give optimal results. It may have also been worth combining one of these with the PSO but the PSO was accurate enough it was deemed unnesesary.

The PSO output the best tunes, outperforming the GA mainly because it was able to converge much faster. PSOs are far less computationally taxing so far more gains were able to be tested in the same amount of time. If more computing power and time were available, even better tunes could have been achieved.

As seen in the table, for the baseline case, the PD controller slightly outperforms the PR in terms of OS, ST, and RT but not in SSE. This is to be expected as PR controllers specialize in removing SSE when the output is know but PD controllers perform better overall.

Looking at the applicability of these controllers for this system, the goal was to stabilize against slowly varying environmental disturbances and both controllers effectively do so with the baseline tune. After ~120 seconds, as seen in Figure 17, steady state is reached and they almost perfectly track the reference input indefinitely, ignoring the ever increasing disturbance.

As seen in figure 17, for the baseline case there is a sizable Rise Time and Settle time but this is simply not the case for PSO 3, which rises and settle practically instantaneously, almost imperceivable in the figure. The only downside of PSO 3 is the gains are rather high, resulting in high frequency oscillations as it settles. These oscillations die out and over a larger time period the SSE equalizes with the baseline but this could cause an issue if this oscillation excited the flexible modes of the apendages.

[30] Li, Z. (2016). The Optimization Design of PID Controller Parameters Based On Particle Swarm Optimization. North China Electric Power University.
[31] MATLAB (2015). Custom Data Type Optimization Using the Genetic Algorithm. Mathworks.com.
[32] Jayachitra, A. and Vinodha, R. (2014). Genetic Algorithm Based PID Controller Tuning Approach for Continuous Stirred Tank Reactor. Advances in Artificial Intelligence, 2014.
[33] Mirzal, A. (n.d.). PID Parameters Optimization by Using Genetic Algorithm. Hokkaido University.

Appendix 2: SS&C Code

Task 1:

```
% Load data
t = test_signals_part_a.time;
d1 = test_signals_part_a.data1;
d2 = test_signals_part_a.data2;
d3 = test_signals_part_a.data3;

%Time cut
idx = t <= 2;
t = t(idx);
d1 = d1(idx);
d2 = d2(idx);
d3= d3(idx);

% Metrics
metrics(d1, t, 'Signal 1');
metrics(d2, t, 'Signal 2');
metrics(d3, t, 'Signal 3');

% Plot
figure;
hold on;
plot(t, d1, 'LineWidth', 1.5);
plot(t, d2, 'LineWidth', 1.5);
plot(t, d3, 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Signal Amplitude (V)');
legend('Signal 1', 'Signal 2', 'Signal 3');
grid on;

%exportgraphics(figure(1), 'Task1Signal.png',
'Resolution', 300)

function metrics(signal, t, label)
    fprintf('\nMetrics for %s:\n', label);

    mean_v = mean(signal);
    variance_v = var(signal);
    skew_v = skewness(signal);
    kurtosis_v = kurtosis(signal);
    rms_v = mean(signal.^2)^0.5;

    fprintf('Mean: %.4f V\n', mean_v);
    fprintf('Variance: %.4f V^2\n', variance_v);
    fprintf('Skewness: %.4f\n', skew_v);
    fprintf('Kurtosis: %.4f\n', kurtosis_v);
    fprintf('RMS: %.4f V\n', rms_v);


end
```

Task 2:

```
% Load data
t = test_signal_part_b.time;
d = test_signal_part_b.data;

t_limit = 0.1;
idx = t <= t_limit;

%Plot original
figure;
plot(t, d, 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Amplitude (V)');
xlim([0 20])
grid on;

%FFT
N = length(d);
F_s = 1 / (t(2) - t(1));
f = F_s * (0:(N/2)) / N;
Y = fft(d);

% Magnitude Spectrum
Y_mag = abs(Y/N);
Y_mag = 2* Y_mag(1:N/2+1);

% Find peaks
[peak_values, peak_indices] = findpeaks(Y_mag,
'MinPeakHeight', 0.1);

%Filter
Y_filtered = zeros(size(Y));
Y_filtered(peak_indices) = Y(peak_indices);

%IFFT
d_filtered = ifft(Y_filtered); %,'symmetric

%Mean
mean_d = mean(d);
d_filtered = d_filtered + (mean_d - mean(d_filtered));

% Plots
figure;
plot(f, Y_mag, 'b', 'LineWidth', 1.5);
hold on;
plot(f(peak_indices), peak_values, 'ro', 'MarkerSize',
8, 'LineWidth', 1.5);
xlabel('Frequency (Hz)');
ylabel('Magnitude (V)');
grid on;
legend('Original Spectrum', 'Detected Peaks');

figure;
hold on;
plot(t(idx), d(idx), 'LineWidth', 1.5);
plot(t(idx), d_filtered(idx), 'r', 'LineWidth', 1.5);
legend('Unfiltered Signal' ,'Filtered Harmonic
Signal');
xlabel('Time (s)');
ylabel('Amplitude (V)');
grid on;

figure;
hold on;
%plot(t(indices), d(indices), 'LineWidth', 1.5);
plot(t(idx), d_filtered(idx), 'r', 'LineWidth', 1.5);
legend('Filtered Harmonic Signal');
xlabel('Time (s)');
ylabel('Amplitude (V)');
grid on;


%exportgraphics(figure(1), 'Task2Signal1.png',
'Resolution', 300)
%exportgraphics(figure(2), 'Task2Signal2.png',
'Resolution', 300)
%exportgraphics(figure(3), 'Task2Signal3.png',
'Resolution', 300)
%exportgraphics(figure(4), 'Task2Signal3.png',
'Resolution', 300)
```

Task 3:

```
% Parameters
m = 0.02; % kg
c = 9.4;  % Ns/m

% Transfer function
num = 1;
den = [m, c, 0];
sys = tf(num, den);

% Bode
figure;
bode(sys);
grid on;


% Input signal
f = 100;  % Hz
F_amp = 5; % N


w = 2 * pi * f;
H_jw = 1 ./ (-m * w.^2 + 1j * c * w);

% Displacement
X_amp = abs(H_jw) * F_amp* (4/pi);
fprintf('Maximum excursion of the mirror: %.6f
meters\n', X_amp);

%exportgraphics(figure(1), 'Task3Signal1.png',
'Resolution', 300)


%Sawtooth
F_f = f(peak_indices(7));
sawtooth_wave = sawtooth(2 * pi * F_f * t);

A_filtered = max(d_filtered) - min(d_filtered);
amp_sawtooth = max(sawtooth_wave) -
min(sawtooth_wave);
```

```
sawtooth_wave = sawtooth_wave * (A_filtered /
amp_sawtooth);
sawtooth_wave = sawtooth_wave +
mean(d_filtered) - mean(sawtooth_wave);


% Plot
figure;
plot(t(idx), d_filtered(idx), 'r', 'LineWidth', 1.5);
hold on;
plot(t(idx), sawtooth_wave(idx), 'b', 'LineWidth', 1.5);
legend('Filtered Harmonic Signal', 'Generated
Sawtooth Wave');
xlabel('Time (s)');
ylabel('Amplitude (V)');
grid on;
```

Task 4 Master:

```
% Parameters%%%%%%%%%%%%%
global T x ESS_w OS_w ST_w RT_w
x = 1; % Input of 1 rad (setpoint)
J = 10000; % Rigid body moment of inertia
T = 500; % Simulation run time

% First transfer function
N1 = 0.001;
D1 = [1, 0.06, 0.36];

% Second transfer function
N2 = 0.001;
D2 = [1, 0.6, 3.6];

%PR gains
Kp_PR = 3.6;
Kd_PR = 84.84;
% Kp_PR = 30992421.08;
% Kd_PR = 563391.84;

%PD gains
Kp_PD = 3.6;
Kd_PD = 84.84;
% Kp_PD = 798507.00;
% Kd_PD = 26922.04;

%Metric Weigts
ESS_w = 1000;
OS_w = 1;
ST_w = 1;
RT_w = 1;


%Getting Data%%%%%%%%%%%%%%%

%PR Cost function
PR_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 2);

Cost_PR = PR_output(1);
SSE_PR = PR_output(2);
OS_PR = PR_output(3);
ST_PR = PR_output(4);
RT_PR = PR_output(5);

PD_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 4);

Cost_PD = PD_output(1);
SSE_PD = PD_output(2);
OS_PD = PD_output(3);
ST_PD = PD_output(4);
RT_PD = PD_output(5);


% Run Simulink model for PR and PD controllers
sim('Task4_sim.slx');

% Take outputs from Simulink
t = ans.tout;
input = ans.simout(:, 3);
open = ans.simout(:, 1);
PR = ans.simout(:, 2);
PD = ans.simout(:, 4);


%Displating Data%%%%%%%%%%%%%

%Chosen gains
fprintf('Chosen PR Controller Gains:\n');
fprintf('Kp_PR = %.2f\n', Kp_PR);
fprintf('Kd_PR = %.2f\n', Kd_PR);

fprintf('Chosen PD Controller Gains:\n');
fprintf('Kp_PD = %.2f\n', Kp_PD);
fprintf('Kd_PD = %.2f\n', Kd_PD);


% Print the performance metrics PR
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', SSE_PR);
fprintf('OS: %.2f%%\n', OS_PR);
fprintf('ST: %.2f s\n', ST_PR);
fprintf('RT: %.2f s\n', RT_PR);
fprintf('Cost: %.2f \n', Cost_PR);

% Print the performance metrics PD
fprintf('\nPD Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', SSE_PD);
fprintf('OS: %.2f%%\n', OS_PD);
fprintf('ST: %.2f s\n', ST_PD);
fprintf('RT: %.2f s\n', RT_PD);
fprintf('Cost: %.2f\n', Cost_PD);

% Plot the results
figure;
hold on;
plot(t, PR, 'LineWidth', 1.5);
plot(t, PD, 'LineWidth', 1.5);
plot(t, input, 'm--', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Output (rad)');
legend('PR', 'PD', 'Input');
grid on;


%Functions%%%%%%%%%%
function cost = cost_function(Kp_PR, Kd_PR,
Kp_PD, Kd_PD, s) % s=2 is PR, s=4 is PD
    global T x ESS_w OS_w ST_w RT_w
    sim('Task4_sim.slx');


    t = ans.tout;
    P = ans.simout(:, s); %PR


    if s==2
        Kp = Kp_PR;
        Kd = Kp_PR;
    else
        Kp = Kp_PD;
        Kd = Kp_PD;
    end

    % Calculate metrics

    % Steady-state error
    SSE = abs(P(end) - x);

    % Overshoot
    max_P = max(P); % Maximum value
    OS = ((max_P - x) / x) * 100; % Overshoot
percentage

    % Settling time
    final = P(end); % Averaging last 10 points
    tolerance = 0.02 * abs(final); % 0.2% tolerance
    idx = find(abs(P - final) > tolerance, 1, 'last');
%finds last time is outside tolerance
    ST = t(idx);

    % Rise Time (from 10% to 90% of final value)
    RT_start_idx = find(P >= 0.1 * x, 1, 'first');
    RT_end_idx = find(P >= 0.9 * x, 1, 'first');
    RT_start = t(RT_start_idx);
    RT_end = t(RT_end_idx);
    RT = RT_end - RT_start;

    % If doesn't rise, RT is invalid, infinite cost
    if isnan(RT) || RT == 0
        cost = Inf;
        return;
    end

    % If doesn't settle, ST is invalid, infinite cost
    if ST > 0.9 * T
        cost = Inf;
        return;
    end


    % Combine all metrics into a cost function (no
printing here)
    cost = ESS_w * SSE + OS_w * abs(OS) + ST_w *
ST + RT_w * RT; % Scalar value

    cost = [cost, SSE, OS, ST, RT];
```

```
end
```

Task 4 PSO:

```
% Parameters%%%%%%%%%%%%%
global T x ESS_w OS_w ST_w RT_w
x = 1; % Input of 1 rad (setpoint)
J = 1000; % Rigid body moment of inertia
T = 500; % Simulation run time

% First transfer function
N1 = 0.001;
D1 = [1, 0.06, 0.36];

% Second transfer function
N2 = 0.001;
D2 = [1, 0.6, 3.6];

%Metric Weigts
ESS_w = 1000;
OS_w = 1;
ST_w = 1;
RT_w = 1;

%Initialize gains so never 0
Kp_PR = 1;
Kd_PR = 1;
Kp_PD = 1;
Kd_PD = 1;


% Set PSO parameters%%%%%%%%%%%%
num_particles = 50;
max_iterations = 500;
w = 0.7; %particle impusle
c1 = 1.5; %personal
c2 = 1.5; %social
convergence_threshold = 0.01;
convergence_window = 20;

%Initial
Kp_PR_i = 1;
Kd_PR_i = 1;
Kp_PD_i = 1;
Kd_PD_i = 1;


%Particle Swarm%%%%%%%%%%%

% Initialize particles and velocities PR
particles = rand(num_particles, 2) .* [Kp_PR_i,
Kd_PR_i];
velocities = zeros(num_particles, 2);
pbest = particles;
pbest_fitness = inf(num_particles, 1);
[gbest_fitness, gbest_index] = min(pbest_fitness);
gbest_PR = pbest(gbest_index, :);
fitness_history = zeros(max_iterations, 1);
stalled_iterations = 0;

for iter = 1:max_iterations
    for i = 1:num_particles
        Kp_PR = particles(i, 1);
        Kd_PR = particles(i, 2);
        PR_output = cost_function(Kp_PR, Kd_PR,
Kp_PD, Kd_PD, 2);
        fitness = PR_output(1);  % Cost from PR

        % Update personal best
        if fitness < pbest_fitness(i)
            pbest(i, :) = particles(i, :);
            pbest_fitness(i) = fitness;
        end
    end

    % Update global best fitness
    [new_gbest_fitness, gbest_index] =
min(pbest_fitness);
    if new_gbest_fitness < gbest_fitness
        gbest_fitness = new_gbest_fitness;
        gbest_PR = pbest(gbest_index, :);
        stalled_iterations = 0;
    else
        stalled_iterations = stalled_iterations + 1;
    end

    % Convergence check
    if stalled_iterations >= convergence_window &&
abs(fitness_history(max(1, iter-1)) - gbest_fitness) <
convergence_threshold
        fprintf('Converged at iteration %d\n', iter);
        break;
    end

    % Store global best fitness
    fitness_history(iter) = gbest_fitness;
    fprintf('Iteration %d: Best Kp = %.2f, Best Kd =
%.2f, Fitness = %.4f\n', ...
        iter, gbest_PR(1), gbest_PR(2), gbest_fitness);

    % Update velocities and positions
    for i = 1:num_particles
        r1 = rand();
        r2 = rand();
        velocities(i, :) = w * velocities(i, :) + ...
                    c1 * r1 * (pbest(i, :) - particles(i, :))
+ ...
                    c2 * r2 * (gbest_PR - particles(i, :));
        particles(i, :) = particles(i, :) + velocities(i, :);
    end
end

figure;
plot(1:max_iterations, fitness_history, 'LineWidth',
2);
xlabel('Iteration');
ylabel('Best Fitness Value');
title('PR Fitness History Over Iterations');
grid on;

% Optimized PR parameters
Kp_PR = gbest_PR(1);
Kd_PR = gbest_PR(2);


% Initialize particles and velocities PD
particles = rand(num_particles, 2) .* [Kp_PD_i,
Kd_PD_i];
velocities = zeros(num_particles, 2);
pbest = particles;
pbest_fitness = inf(num_particles, 1);
[gbest_fitness, gbest_index] = min(pbest_fitness);
gbest_PR = pbest(gbest_index, :);
fitness_history = zeros(max_iterations, 1);
stalled_iterations = 0;

for iter = 1:max_iterations
    for i = 1:num_particles
        Kp_PD = particles(i, 1);
        Kd_PD = particles(i, 2);
        PD_output = cost_function(Kp_PR, Kd_PR,
Kp_PD, Kd_PD, 4);
        fitness = PD_output(1);  % Cost from PD

        % Update personal best
        if fitness < pbest_fitness(i)
            pbest(i, :) = particles(i, :);
            pbest_fitness(i) = fitness;
        end
    end

    % Update global best fitness
    [new_gbest_fitness, gbest_index] =
min(pbest_fitness);
    if new_gbest_fitness < gbest_fitness
        gbest_fitness = new_gbest_fitness;
        gbest_PD = pbest(gbest_index, :);
        stalled_iterations = 0;
    else
        stalled_iterations = stalled_iterations + 1;
    end

    % Convergence check
    if stalled_iterations >= convergence_window &&
abs(fitness_history(max(1, iter-1)) - gbest_fitness) <
convergence_threshold
        fprintf('Converged at iteration %d\n', iter);
        break;
    end

    % Store global best fitness
    fitness_history(iter) = gbest_fitness;
    fprintf('Iteration %d: Best Kp = %.2f, Best Kd =
%.2f, Fitness = %.4f\n', ...
        iter, gbest_PD(1), gbest_PD(2), gbest_fitness);

    % Update velocities and positions
    for i = 1:num_particles
        r1 = rand();
        r2 = rand();
        velocities(i, :) = w * velocities(i, :) + ...
                    c1 * r1 * (pbest(i, :) - particles(i, :))
+ ...
                    c2 * r2 * (gbest_PD - particles(i, :));
```

```
        particles(i, :) = particles(i, :) + velocities(i, :);
    end
end



figure;
plot(1:max_iterations, fitness_history, 'LineWidth',
2);
xlabel('Iteration');
ylabel('Best Fitness Value');
title('PD Fitness History Over Iterations');
grid on;



% Optimized PD parameters
Kp_PD = gbest_PD(1);
Kd_PD = gbest_PD(2);



%Getting Data%%%%%%%%%%%%%%%%

%PR Cost function
PR_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 2);

Cost_PR = PR_output(1);
ESS_PR = PR_output(2);
OS_PR = PR_output(3);
ST_PR = PR_output(4);
RT_PR = PR_output(5);

PD_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 4);

Cost_PD = PD_output(1);
ESS_PD = PD_output(2);
OS_PD = PD_output(3);
ST_PD = PD_output(4);
RT_PD = PD_output(5);



% Run Simulink model for PR and PD controllers
sim('Task4_sim.slx');

% Take outputs from Simulink
t = ans.tout;
input = ans.simout(:, 3);
open = ans.simout(:, 1);
PR = ans.simout(:, 2);
PD = ans.simout(:, 4);



%Displating Data%%%%%%%%%%%%

%Chosen gains
fprintf('Chosen PR Controller Gains:\n');
fprintf('Kp_PR = %.2f\n', Kp_PR);
fprintf('Kd_PR = %.2f\n', Kd_PR);

fprintf('Chosen PD Controller Gains:\n');
fprintf('Kp_PD = %.2f\n', Kp_PD);
fprintf('Kd_PD = %.2f\n', Kd_PD);

% Print the performance metrics PR
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', ESS_PR);
fprintf('OS: %.2f%%\n', OS_PR);
fprintf('ST: %.2f s\n', ST_PR);
fprintf('RT: %.2f s\n', RT_PR);
fprintf('Cost: %.2f s\n', Cost_PR);

% Print the performance metrics PD
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', ESS_PD);
fprintf('OS: %.2f%%\n', OS_PD);
fprintf('ST: %.2f s\n', ST_PD);
fprintf('RT: %.2f s\n', RT_PD);
fprintf('Cost: %.2f\n', Cost_PD);

% Plot results
figure;
hold on;
plot(t, PR, 'LineWidth', 1.5);
plot(t, PD, 'LineWidth', 1.5);
plot(t, input, 'm--', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Output (rad)');
legend('PR', 'PD', 'Input');
grid on;



exportgraphics(figure(1), 'PSO2_1.png',
'Resolution', 300)
exportgraphics(figure(2), 'PSO2_2.png',
'Resolution', 300)
exportgraphics(figure(3), 'PSO2_3.png',
'Resolution', 300)

%Functions%%%%%%%%%%

function cost = cost_function(Kp_PR, Kd_PR,
Kp_PD, Kd_PD, s) % s=2 is PR, s=4 is PD
    global T x ESS_w OS_w ST_w RT_w
    sim('Task4_sim.slx');



    t = ans.tout;
    P = ans.simout(:, s); %PR



    if s==2
        Kp = Kp_PR;
        Kd = Kd_PR;
    else
        Kp = Kp_PD;
        Kd = Kd_PD;
    end

    % Calculate metrics

    % Steady-state error
    ESS = abs(P(end) - x);

    % Overshoot
    max_P = max(P); % Maximum value
    OS = ((max_P - x) / x) * 100; % Overshoot
percentage

    % Settling time
    final = P(end); % Averaging last 10 points
    tolerance = 0.02 * abs(final); % 0.2% tolerance
    idx = find(abs(P - final) > tolerance, 1, 'last');
%finds last time is outside tolerance
    ST = t(idx);

    % Rise Time (from 10% to 90% of final value)
    RT_start_idx = find(P >= 0.1 * x, 1, 'first');
    RT_end_idx = find(P >= 0.9 * x, 1, 'first');
    RT_start = t(RT_start_idx);
    RT_end = t(RT_end_idx);
    RT = RT_end - RT_start;

    % If doesn't rise, RT is invalid, infinite cost
    if any(isnan(RT))
        RT = 10000000;
    end
    if any(isnan(ESS))
        ESS = 1000000; %this is really messy took
forever to weed out all the errors
    end
    if any(isnan(OS))
        OS = 10000000;
    end
    if any(RT == 0)
      RT = 10000000;
    end
    if RT > 0.92 * T
        RT = 10000000;
    end

    % If doesn't settle, ST is invalid, infinite cost
    if ST > 0.92 * T
        ST = 10000000;
    end
    if  ST == 0
        ST = 10000000;
    end
    if any(isnan(ST))
        ST = 10000000;
    end

    % Combine all metrics into a cost function
    cost_sum = ESS_w * ESS + OS_w * abs(OS) +
ST_w * ST + RT_w * RT;

    if any(isnan(cost_sum))
        cost_sum = 80000000;
    end
    cost = [cost_sum, ESS, OS, ST, RT];
    if length(cost) ~= 5
        cost = [100000, 1000000, 1000000, 1000000,
1000000]; % Default values
    end
end
```

Task 4 GA:

```
% Parameters%%%%%%%%%%%%%
```

```
global T x ESS_w OS_w ST_w RT_w Kp_PR_i
Kd_PR_i Kp_PD_i Kd_PD_i
x = 1; % Input of 1 rad (setpoint)
J = 1000; % Rigid body moment of inertia
T = 500; % Simulation run time

% First transfer function
N1 = 0.001;
D1 = [1, 0.06, 0.36];

% Second transfer function
N2 = 0.001;
D2 = [1, 0.6, 3.6];

%Metric Weigts
ESS_w = 1000;
OS_w = 1;
ST_w = 1;
RT_w = 1;

%Initialize gains so never 0
Kp_PR = 1;
Kd_PR = 1;
Kp_PD = 1;
Kd_PD = 1;

%Inital
Kp_PR_i = 10;
Kd_PR_i = 100;
Kp_PD_i = 10;
Kd_PD_i = 100;

%Genetic
Algorithm%%%%%%%%%%%%%%%%%%%%

% Set GA parameters
population_size = 10; % Number of individuals per
generation
max_generations = 500; % Maximum number of
generations
mutation_rate = 0.4;
crossover_rate = 0.1;
convergence_threshold = 0.01;
convergence_count = 20;


% Initialization PR
population = rand(population_size, 2) .* [Kp_PR_i,
Kd_PR_i];  % Random values between 0 and 100
for Kp and Kd
previous_best_fitness = Inf;
convergence_counter = 0;
fitness_history = zeros(max_generations, 1);

% Main GA loop
for generation = 1:max_generations
    % Evaluate fitness of all individuals in the
population
    fitness = zeros(population_size, 1);
    for i = 1:population_size
        Kp_PR = population(i, 1);
        Kd_PR = population(i, 2);

        %Cost function calling
        current_fitness = cost_function(Kp_PR,
Kd_PR, Kp_PD, Kd_PD, 2);

        fitness(i) = current_fitness(1); % Assign the
fitness value
    end

    % Select parents using tournament selection
    parents = tournament_selection(population,
fitness, population_size);

    % Create offspring by crossover
    offspring = crossover(parents, crossover_rate,
population_size);

    % Apply mutation
    offspring = mutate(offspring, mutation_rate, 2);

    % Replace old population with next generation
    population = offspring;

    % Output the best solution of the current
generation
    [best_fitness_PR, best_index_PR] = min(fitness);
    best_Kp = population(best_index_PR, 1);
    best_Kd = population(best_index_PR, 2);
    fprintf('Generation %d: Best Kp = %.2f, Best Kd =
%.2f, Fitness = %.4f\n', generation, best_Kp,
best_Kd, best_fitness_PR);

    % Save the fitness value for the current
generation
    fitness_history(generation) = best_fitness_PR;

    % Convergence check
    if abs(previous_best_fitness - best_fitness_PR) <
convergence_threshold
        convergence_counter = convergence_counter
+ 1;
    else
        convergence_counter = 0;
    end

    if convergence_counter >= convergence_count
        fprintf('Convergence detected after %d
generations. Stopping early.\n', generation);
        break;
    end

    % Update previous best fitness
    previous_best_fitness = best_fitness_PR;
end

% Plot the fitness history
figure;
plot(1:generation, fitness_history(1:generation),
'LineWidth', 2);
xlabel('Generation');
ylabel('Best Fitness Value');
title('Fitness History Over Generations PR');
grid on;

% Best solution after all generations
[best_fitness_PR, best_index_PR] = min(fitness);
best_Kp_PR = population(best_index_PR, 1);
best_Kd_PR = population(best_index_PR, 2);

%Optimized PR parameters
Kp_PR = best_Kp_PR;
Kd_PR = best_Kd_PR;



% Initialization PD
population = rand(population_size, 2) .* [Kp_PD_i,
Kd_PD_i];  % Random values between 0 and 100
for Kp and Kd
previous_best_fitness = Inf;
convergence_counter = 0;
fitness_history = zeros(max_generations, 1);

% Main GA loop
for generation = 1:max_generations
    % Evaluate fitness of all individuals in the
population
    fitness = zeros(population_size, 1);
    for i = 1:population_size
        Kp_PD = population(i, 1);
        Kd_PD = population(i, 2);

        %Cost function calling
        current_fitness = cost_function(Kp_PR,
Kd_PR, Kp_PD, Kd_PD, 4);

        fitness(i) = current_fitness(1); % Assign the
fitness value
    end

    % Select parents using tournament selection
    parents = tournament_selection(population,
fitness, population_size);

    % Create offspring by crossover
    offspring = crossover(parents, crossover_rate,
population_size);

    % Apply mutation
    offspring = mutate(offspring, mutation_rate, 4);

    % Replace old population with next generation
    population = offspring;

    % Output the best solution of the current
generation
    [best_fitness, best_index] = min(fitness);
    best_Kp = population(best_index, 1);
    best_Kd = population(best_index, 2);
    fprintf('Generation %d: Best Kp = %.2f, Best Kd =
%.2f, Fitness = %.4f\n', generation, best_Kp,
best_Kd, best_fitness);

    % Save the fitness value for the current
generation
    fitness_history(generation) = best_fitness;

    % Convergence check
    if abs(previous_best_fitness - best_fitness) <
convergence_threshold
```

```
        convergence_counter = convergence_counter
+ 1;
    else
        convergence_counter = 0;
    end

    if convergence_counter >= convergence_count
        fprintf('Convergence detected after %d
generations. Stopping early.\n', generation);
        break;
    end

    % Update previous best fitness
    previous_best_fitness = best_fitness;
end

% Plot the fitness history
figure;
plot(1:generation, fitness_history(1:generation),
'LineWidth', 2);
xlabel('Generation');
ylabel('Best Fitness Value');
title('Fitness History Over Generations PD');
grid on;

% Best solution after all generations
[best_fitness, best_index] = min(fitness);
best_Kp = population(best_index, 1);
best_Kd = population(best_index, 2);

%Optimized PR parameters
Kp_PD = best_Kp;
Kd_PD = best_Kd;



%Getting Data%%%%%%%%%%%%%%%%

%PR Cost function
PR_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 2);

Cost_PR = PR_output(1);
ESS_PR = PR_output(2);
OS_PR = PR_output(3);
ST_PR = PR_output(4);
RT_PR = PR_output(5);

PD_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 4);

Cost_PD = PD_output(1);
ESS_PD = PD_output(2);
OS_PD = PD_output(3);
ST_PD = PD_output(4);
RT_PD = PD_output(5);



% Run Simulink model for PR and PD controllers
sim('Task4_sim.slx');

% Take outputs from Simulink
t = ans.tout;
input = ans.simout(:, 3);
open = ans.simout(:, 1);
PR = ans.simout(:, 2);
PD = ans.simout(:, 4);



%Displating Data%%%%%%%%%%%%

%Chosen gains
fprintf('Chosen PR Controller Gains:\n');
fprintf('Kp_PR = %.2f\n', Kp_PR);
fprintf('Kd_PR = %.2f\n', Kd_PR);

fprintf('Chosen PD Controller Gains:\n');
fprintf('Kp_PD = %.2f\n', Kp_PD);
fprintf('Kd_PD = %.2f\n', Kd_PD);



% Print the performance metrics PR
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', ESS_PR);
fprintf('OS: %.2f%%\n', OS_PR);
fprintf('ST: %.2f s\n', ST_PR);
fprintf('RT: %.2f s\n', RT_PR);
fprintf('Cost: %.2f s\n', Cost_PR);

% Print the performance metrics PD
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', ESS_PD);
fprintf('OS: %.2f%%\n', OS_PD);
fprintf('ST: %.2f s\n', ST_PD);
fprintf('RT: %.2f s\n', RT_PD);
fprintf('Cost: %.2f\n', Cost_PD);

% Plot results
figure;
hold on;
plot(t, PR, 'LineWidth', 1.5);
plot(t, PD, 'LineWidth', 1.5);
plot(t, input, 'm--', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Output (rad)');
legend('PR', 'PD', 'Input');
grid on;



exportgraphics(figure(1), 'GA4_1.png', 'Resolution',
300)
exportgraphics(figure(2), 'GA4_2.png', 'Resolution',
300)
exportgraphics(figure(3), 'GA4_3.png', 'Resolution',
300)



%Functions%%%%%%%%%%

function cost = cost_function(Kp_PR, Kd_PR,
Kp_PD, Kd_PD, s) % s=2 is PR, s=4 is PD
    global T x ESS_w OS_w ST_w RT_w
    sim('Task4_sim.slx');


    t = ans.tout;
    P = ans.simout(:, s); %PR
    if s==2
        Kp = Kp_PR;
        Kd = Kd_PR;
    else
        Kp = Kp_PD;
        Kd = Kd_PD;
    end

    % Calculate metrics

    % Steady-state error
    ESS = abs(P(end) - x);

    % Overshoot
    max_P = max(P); % Maximum value
    OS = ((max_P - x) / x) * 100; % Overshoot
percentage

    % Settling time
    final = P(end);
    tolerance = 0.02 * abs(final); % 0.2% tolerance
    idx = find(abs(P - final) > tolerance, 1, 'last');
    ST = t(idx);

    % Rise Time (from 10% to 90% of final value)
    RT_start_idx = find(P >= 0.1 * x, 1, 'first');
    RT_end_idx = find(P >= 0.9 * x, 1, 'first');
    RT_start = t(RT_start_idx);
    RT_end = t(RT_end_idx);
    RT = RT_end - RT_start;

    % If doesn't rise, RT is invalid, infinite cost
    if any(isnan(RT))
        RT = 10000000;
    end
    if any(isnan(ESS))
        ESS = 1000000; %this is really messy took
forever to weed out all the errors
    end
    if any(isnan(OS))
        OS = 10000000;
    end
    if any(RT == 0)
      RT = 10000000;
    end
    if RT > 0.92 * T
        RT = 10000000;
    end

    % If doesn't settle, ST is invalid, infinite cost
    if ST > 0.92 * T
        ST = 10000000;
    end
    if  ST == 0
        ST = 10000000;
    end
    if any(isnan(ST))
        ST = 10000000;
    end

    % Combine all metrics into a cost function
    cost_sum = ESS_w * ESS + OS_w * abs(OS) +
ST_w * ST + RT_w * RT;
```

```
    if any(isnan(cost_sum))
        cost_sum = 80000000;
    end
    cost = [cost_sum, ESS, OS, ST, RT];
    if length(cost) ~= 5
        cost = [100000, 1000000, 1000000, 1000000,
1000000]; % Default values
    end
end



% Tournament selection function
function parents = tournament_selection(population,
fitness, population_size)
    parents = zeros(population_size, 2);
    for i = 1:population_size
        % Randomly pick 2 individuals for tournament
selection
        candidates = randi([1, population_size], 2, 1);
        % Select the best candidate
        [~, winner] = min(fitness(candidates));
        parents(i, :) = population(candidates(winner),
:);
    end
end

% Crossover function
function offspring = crossover(parents,
crossover_rate, population_size)
    offspring = parents;
    for i = 1:2:population_size
        if rand() < crossover_rate
            crossover_point = randi([1, 2]);
            % Swap genetic material
            offspring(i, crossover_point:end) =
parents(i+1, crossover_point:end);
            offspring(i+1, crossover_point:end) =
parents(i, crossover_point:end);
        end
    end
end

% Mutation function
function offspring = mutate(offspring, mutation_rate,
s)
    global Kp_PR_i Kd_PR_i Kp_PD_i Kd_PD_i
    for i = 1:size(offspring, 1)
        if rand() < mutation_rate
            % Mutate both Kp and Kd
            if s == 2
                offspring(i, 1) = rand() * Kp_PR_i;
                offspring(i, 2) = rand() * Kd_PR_i;

            elseif s == 4
                offspring(i, 1) = rand() * Kp_PD_i;
                offspring(i, 2) = rand() * Kd_PD_i;
            end
        end
    end
end
```

Task 4 Comparer:

```
%%%%%%%%%%%%
global T x ESS_w OS_w ST_w RT_w
x = 1; % Input of 1 rad (setpoint)
J = 1000; % Rigid body moment of inertia
T = 200; % Simulation run time

% First transfer function
N1 = 0.001;
D1 = [1, 0.06, 0.36];

% Second transfer function
N2 = 0.001;
D2 = [1, 0.6, 3.6];

%PR gains
Kp_PR = 3.6;
Kd_PR = 84.84;
%Kp_PR = 30992421.08;
%Kd_PR = 563391.84;

%PD gains
Kp_PD = 3.6;
Kd_PD = 84.84;
%Kp_PD = 798507.00;
%Kd_PD = 26922.04;

%Metric Weigts
ESS_w = 1000;
OS_w = 1;
ST_w = 1;
RT_w = 1;



% Run Simulink model for PR and PD controllers
sim('Task4_sim.slx');

% Take outputs from Simulink
t = ans.tout;
input = ans.simout(:, 3);
open = ans.simout(:, 1);
PR = ans.simout(:, 2);
PD = ans.simout(:, 4);

e_PR = abs(PR-1);
e_PD = abs(PD-1);

e_diff = e_PR - e_PD;

% Plot the results
figure;
hold on;
plot(t, PR, 'LineWidth', 1.5);
plot(t, PD, 'LineWidth', 1.5);
legend('PR Baseline', 'PD Baseline', 'PR PSO 3',
'PD PSO 3', 'Input');
hold off

figure;
hold on;
plot(t, e_PR, 'LineWidth', 1.5);
plot(t, e_PD, 'LineWidth', 1.5);
legend('PR Baseline', 'PD Baseline', 'PR PSO 3',
'PD PSO 3', 'Input');
hold off


figure;
hold on;
plot(t, e_diff, 'LineWidth', 1.5);
%plot(t, e_diff, 'LineWidth', 1.5);
hold off



% Kp_PR = 30992421.08;
% Kd_PR = 563391.84;
% Kp_PD = 798507.00;
% Kd_PD = 26922.04;
%
% % Run Simulink model for PR and PD controllers
% sim('Task4_sim.slx');
%
% % Take outputs from Simulink
% t = ans.tout;
% input = ans.simout(:, 3);
% open = ans.simout(:, 1);
% PR = ans.simout(:, 2);
% PD = ans.simout(:, 4);
%
% plot(t, PR, 'LineWidth', 1);
% plot(t, PD, 'LineWidth', 1);
% plot(t, input, 'm--', 'LineWidth', 0.2);
% xlabel('Time (s)');
% ylabel('Output (rad)');
 % legend('PR Baseline', 'PD Baseline', 'PR PSO 3',
'PD PSO 3', 'Input');
% grid on;
%
% exportgraphics(figure(1), 'comparer_1.png',
'Resolution', 300)
```

Task 4 Manual:

```
% Parameters%%%%%%%%%
global T x ESS_w OS_w ST_w RT_w
x = 1;
J = 1000;
T = 500;

% First transfer function
N1 = 0.001;
D1 = [1, 0.06, 0.36];

% Second transfer function
N2 = 0.001;
D2 = [1, 0.6, 3.6];

%Metric Weigts
```

```
ESS_w = 1000;
OS_w = 1;
ST_w = 1;
RT_w = 1;


% Initial PR Gains
Kp_PR_i = 1;
Kd_PR_i = 0;

% PD Gains
Kp_PD_i = 1;
Kd_PD_i = 0;


%Increments
Kp_inc = 1;
Kd_inc = 1;

%Manual
Method%%%%%%%%%%%%%%%%

% Trial-and-Error method for PR controller tuning
Kp_PR = Kp_PR_i;
Kd_PR = Kd_PR_i;
Kp_PD = Kp_PD_i;
Kd_PD = Kd_PD_i;

% Increase Kp until loop oscillates at constant
amplitude
oscillating = false;
run_count = 0;
while ~oscillating
    run_count = run_count + 1;
    Kp_PR = Kp_PR + Kp_inc;
    sim('Task4_sim.slx');
    % Evaluate oscillation
    if check_oscillation(ans.simout(:, 2))
        oscillating = true;
    end
    fprintf('PR Tuning Run #%d: Kp = %.2f\n',
run_count, Kp_PR);
end

%Set PID gain to half of Kp (if oscillatory)
Kp_PR = Kp_PR / 2;

%Increase Kd until overshoot is minimized
OS_acceptable = false;
run_count = 0;
while ~OS_acceptable
    run_count = run_count + 1;
    Kd_PR = Kd_PR + Kd_inc;
    sim('Task4_sim.slx');
    % Check for overshoot condition
    if check_OS(ans.simout(:, 2))
        OS_acceptable = true;
    end
    fprintf('PR Tuning Run #%d: Kd = %.2f\n',
run_count, Kd_PR);
end

% Trial-and-Error method for PD controller tuning
Kp_PD = Kp_PD_i;
Kd_PD = Kd_PD_i;

% Increase Kp until loop oscillates at constant
amplitude
oscillating = false;
run_count = 0;
while ~oscillating
    run_count = run_count + 1;
    Kp_PD = Kp_PD + Kp_inc;
    sim('Task4_sim.slx');
    if check_oscillation(ans.simout(:, 4))
        oscillating = true;
    end
    fprintf('PD Tuning Run #%d: Kp = %.2f\n',
run_count, Kp_PD);
end

%Set PID gain to half of Kp (if oscillatory)
Kp_PD = Kp_PD / 2;
%Increase Kd until overshoot is minimized
OS_acceptable = false;
run_count = 0;
while ~OS_acceptable
    run_count = run_count + 1;
    Kd_PD = Kd_PD + Kd_inc;
    sim('Task4_sim.slx');
    % Check for overshoot condition
    if check_OS(ans.simout(:, 4))
        OS_acceptable = true;
    end
    fprintf('PD Tuning Run #%d: Kd = %.2f\n',
run_count, Kd_PD);
end


%Getting Data%%%%%%%%%%%%%%%%

%PR Cost function
PR_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 2);

Cost_PR = PR_output(1);
ESS_PR = PR_output(2);
OS_PR = PR_output(3);
ST_PR = PR_output(4);
RT_PR = PR_output(5);

PD_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 4);

Cost_PD = PD_output(1);
ESS_PD = PD_output(2);
OS_PD = PD_output(3);
ST_PD = PD_output(4);
RT_PD = PD_output(5);


sim('Task4_sim.slx');

t = ans.tout;
input = ans.simout(:, 3);
open = ans.simout(:, 1);
PR = ans.simout(:, 2);
PD = ans.simout(:, 4);


%Displating
Data%%%%%%%%%%%%%%%%%%

%Chosen gains
fprintf('Chosen PR Controller Gains:\n');
fprintf('Kp_PR = %.2f\n', Kp_PR);
fprintf('Kd_PR = %.2f\n', Kd_PR);

fprintf('Chosen PD Controller Gains:\n');
fprintf('Kp_PD = %.2f\n', Kp_PD);
fprintf('Kd_PD = %.2f\n', Kd_PD);

% Print the performance metrics PR
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', ESS_PR);
fprintf('OS: %.2f%%\n', OS_PR);
fprintf('ST: %.2f s\n', ST_PR);
fprintf('RT: %.2f s\n', RT_PR);
fprintf('Cost: %.2f s\n', Cost_PR);

% Print the performance metrics PD
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', ESS_PD);
fprintf('OS: %.2f%%\n', OS_PD);
fprintf('ST: %.2f s\n', ST_PD);
fprintf('RT: %.2f s\n', RT_PD);
fprintf('Cost: %.2f\n', Cost_PD);

% Plot the results
figure;
hold on;
plot(t, PR, 'LineWidth', 1.5);
plot(t, PD, 'LineWidth', 1.5);
plot(t, input, 'm--', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Output (rad)');
legend('PR', 'PD', 'Input');
grid on;


%Functions%%%%%%%%%%

function oscillating = check_oscillation(signal)
    oscillation_threshold = 0.0001;
    peak_to_peak = max(signal) - min(signal);
    oscillating = peak_to_peak >=
oscillation_threshold;
end

function overshoot = check_OS(signal)
    OS_threshold = 0.0001;
    max_value = max(signal);
    overshoot = (max_value - 1) <= OS_threshold;
end


function cost = cost_function(Kp_PR, Kd_PR,
Kp_PD, Kd_PD, s) % s=2 is PR, s=4 is PD
    global T x ESS_w OS_w ST_w RT_w
    sim('Task4_sim.slx');
```

```
    t = ans.tout;
    P = ans.simout(:, s); %PR


    if s==2
        Kp = Kp_PR;
        Kd = Kp_PR;
    else
        Kp = Kp_PD;
        Kd = Kp_PD;
    end

    % Calculate metrics

    % Steady-state error
    ESS = abs(P(end) - x);

    % Overshoot
    max_P = max(P); % Maximum value
    OS = ((max_P - x) / x) * 100; % Overshoot
percentage

    % Settling time
    final = P(end); % Averaging last 10 points
    tolerance = 0.02 * abs(final); % 0.2% tolerance
    idx = find(abs(P - final) > tolerance, 1, 'last');
%finds last time is outside tolerance
    ST = t(idx);

    % Rise Time (from 10% to 90% of final value)
    RT_start_idx = find(P >= 0.1 * x, 1, 'first');
    RT_end_idx = find(P >= 0.9 * x, 1, 'first');
    RT_start = t(RT_start_idx);
    RT_end = t(RT_end_idx);
    RT = RT_end - RT_start;

    % If doesn't rise, RT is invalid, infinite cost
    if isnan(RT) || RT == 0
        cost = Inf;
        return;
    end

    % If doesn't settle, ST is invalid, infinite cost
    if ST > 0.9 * T
        cost = Inf;
        return;
    end


    % Combine all metrics into a cost function (no
printing here)
    cost = ESS_w * ESS + OS_w * abs(OS) + ST_w *
ST + RT_w * RT; % Scalar value

    cost = [cost, ESS, OS, ST, RT];


end
```

Task 4: Ziegler

```
% Parameters%%%%%%%%%%%%%
global T x ESS_w OS_w ST_w RT_w
x = 1;
J = 1000;
T = 500;

% First transfer function
N1 = 0.001;
D1 = [1, 0.06, 0.36];

% Second transfer function
N2 = 0.001;
D2 = [1, 0.6, 3.6];

%Metric Weigts
ESS_w = 1000;
OS_w = 1;
ST_w = 1;
RT_w = 1;

% Initial PR Gains
Kp_PR_i = 10;
Kd_PR_i = 0;

% Initial PD Gains
Kp_PD_i = 10;
Kd_PD_i = 0;

%Increments
Kp_inc = 1;
Kd_inc = 1;


% Ziegler-
Nichols%%%%%%%%%%%%%%%%%

%PR
Kp_PR = Kp_PR_i;
Kd_PR = Kd_PR_i;

Kp_PD = Kp_PD_i;
Kd_PD = Kd_PD_i;

% Increase Kp until loop oscillates at constant
amplitude
oscillating = false;
run_count = 0;
while ~oscillating
    run_count = run_count + 1;
    Kp_PR = Kp_PR + Kp_inc;
    sim('Task4_sim.slx');
    if check_oscillation(ans.simout(:, 2))
        oscillating = true;
    end
    fprintf('PR Tuning Run #%d: Kp = %.2f\n',
run_count, Kp_PR);
end

Ku_PR = Kp_PR; %critical gain

t = ans.tout;
PR = ans.simout(:, 2);
Pu_PR = determine_oscillation_period(PR, t);

Kp_PR = 0.6 * Ku_PR;

Td_PR = 0.125 * Pu_PR;

Kd_PR = Td_PR * Kp_PR;


%PD
Kp_PD = Kp_PD_i;
Kd_PD = Kd_PD_i;

% Increase Kp until loop oscillates at constant
amplitude
oscillating = false;
run_count = 0;
while ~oscillating
    run_count = run_count + 1;
    Kp_PD = Kp_PD + Kp_inc;
    sim('Task4_sim.slx');
    % Evaluate oscillation
    if check_oscillation(ans.simout(:, 4))
        oscillating = true;
    end
    fprintf('PD Tuning Run #%d: Kp = %.2f\n',
run_count, Kp_PD);
end

Ku_PD = Kp_PD; %critical gain

t = ans.tout;
PD = ans.simout(:, 4);

Pu_PD = determine_oscillation_period(PD, t);

Kp_PD = 0.6 * Ku_PD;

Td_PD = 0.125 * Pu_PD;

Kd_PD = Td_PD * Kp_PD;


%Getting Data%%%%%%%%%%%%%%%%

%PR Cost function
PR_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 2);

Cost_PR = PR_output(1);
ESS_PR = PR_output(2);
OS_PR = PR_output(3);
ST_PR = PR_output(4);
RT_PR = PR_output(5);

PD_output = cost_function(Kp_PR, Kd_PR, Kp_PD,
Kd_PD, 4);

Cost_PD = PD_output(1);
ESS_PD = PD_output(2);
OS_PD = PD_output(3);
ST_PD = PD_output(4);
RT_PD = PD_output(5);
```

```
% Run Simulink model for PR and PD controllers
sim('Task4_sim.slx');

% Take outputs from Simulink
t = ans.tout;
input = ans.simout(:, 3);
open = ans.simout(:, 1);
PR = ans.simout(:, 2);
PD = ans.simout(:, 4);


%Displating Data%%%%%%%%%%%%%%%%%%

%Chosen gains
fprintf('Chosen PR Controller Gains:\n');
fprintf('Kp_PR = %.2f\n', Kp_PR);
fprintf('Kd_PR = %.2f\n', Kd_PR);

fprintf('Chosen PD Controller Gains:\n');
fprintf('Kp_PD = %.2f\n', Kp_PD);
fprintf('Kd_PD = %.2f\n', Kd_PD);

% Print the performance metrics PR
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', ESS_PR);
fprintf('OS: %.2f%%\n', OS_PR);
fprintf('ST: %.2f s\n', ST_PR);
fprintf('RT: %.2f s\n', RT_PR);
fprintf('Cost: %.2f s\n', Cost_PR);

% Print the performance metrics PD
fprintf('\nPR Controller Performance Metrics:\n');
fprintf('SSE: %.2e rad\n', ESS_PD);
fprintf('OS: %.2f%%\n', OS_PD);
fprintf('ST: %.2f s\n', ST_PD);
fprintf('RT: %.2f s\n', RT_PD);
fprintf('Cost: %.2f\n', Cost_PD);

% Plot the results
figure;
hold on;
plot(t, PR, 'LineWidth', 1.5);
plot(t, PD, 'LineWidth', 1.5);
plot(t, input, 'm--', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Output (rad)');
legend('PR', 'PD', 'Input');
grid on;



%Functions%%%%%%%%%%%%%

function P_u = determine_oscillation_period(signal,
time)
    [~, peak_indices] = findpeaks(signal,
'MinPeakDistance', round(length(time)/50));
    if length(peak_indices) < 2
        P_u = NaN;
        return;
    end
    peak_times = time(peak_indices);
    periods = diff(peak_times);
    P_u = mean(periods);
end


function oscillating = check_oscillation(signal)
    oscillation_threshold = 0.0001;
    peak_to_peak = max(signal) - min(signal);
    oscillating = peak_to_peak >=
oscillation_threshold;
end


function overshoot = check_OS(signal)
    OS_threshold = 0.0001;
    max_value = max(signal);
    overshoot = (max_value - 1) <= OS_threshold;
end


function cost = cost_function(Kp_PR, Kd_PR,
Kp_PD, Kd_PD, s) % s=2 is PR, s=4 is PD
    global T x ESS_w OS_w ST_w RT_w
    sim('Task4_sim.slx');


    t = ans.tout;
    P = ans.simout(:, s); %PR


    if s==2
        Kp = Kp_PR;
        Kd = Kp_PR;
    else
        Kp = Kp_PD;
        Kd = Kp_PD;
    end

    % Calculate metrics

    % Steady-state error
    ESS = abs(P(end) - x);

    % Overshoot
    max_P = max(P); % Maximum value
    OS = ((max_P - x) / x) * 100; % Overshoot
percentage

    % Settling time
    final = P(end); % Averaging last 10 points
    tolerance = 0.02 * abs(final); % 0.2% tolerance
    idx = find(abs(P - final) > tolerance, 1, 'last');
%finds last time is outside tolerance
    ST = t(idx);

    % Rise Time (from 10% to 90% of final value)
    RT_start_idx = find(P >= 0.1 * x, 1, 'first');
    RT_end_idx = find(P >= 0.9 * x, 1, 'first');
    RT_start = t(RT_start_idx);
    RT_end = t(RT_end_idx);
    RT = RT_end - RT_start;

    % If doesn't rise, RT is invalid, infinite cost
    % if isnan(RT) || RT == 0
    %     cost = Inf;
    %     return;
    % end

    % % If doesn't settle, ST is invalid, infinite cost
    % if ST > 0.9 * T
    %     cost = Inf;
    %     return;
    % end


    % Combine all metrics into a cost function (no
printing here)
    cost = ESS_w * ESS + OS_w * abs(OS) + ST_w *
ST + RT_w * RT; % Scalar value


    cost = [cost, ESS, OS, ST, RT];
```