Implementation of a Speed Controller

MTRN3020 – Modelling and Control of Mechatronic Systems

I verify that the contents of this report are my own work.

Nick Kapsanis
z5254990
Wednesday 25th June, 2025

1 Introduction

Often in mechatronic systems the variable of interest is physical and derivative or integral to our possible control inputs. As a consequence, it is often necessary to modify other state variables to hold the variable of interest constant. A typical example is a cruise control system. Highway cruise control would be ineffective if a positive road gradient reduced the velocity of the car. However, the torque of the engine and corresponding fuel consumption can be modified in order to maintain a desired constant velocity.

In this experiment we look to design a speed controller that can maintain a given velocity under sudden changes in load. To simulate this, two motors with identical characteristics are coupled as a generator system with the "load motor" having a bank of resistors available to it in order to change its applied loading characteristics. To validate the controller design and to verify experimental results, a model of the coupled motor speed-controller system is built and simulated in simulink.

2 Aim

The aim of this experiment is to design a discrete speed controller for a coupled motor system, such that the controller will operate with the specific sampling time of 9ms, ensuring zero steady state error with a response speed determined by the desired time constant of 36ms. The controller will then be experimentally verified by comparing the measured output speed against a simulated model of the experimental system with ideal characteristics.

3 Experimental Procedure

3.1 Apparatus

The apparatus consists of two identical coupled motors, one connected to a bank of 15 relays and resistors in parallel, each with 150Ω, this will henceforth be the "generator". The other motor in the system is the motor of interest, and has a rotary encoder on its shaft to track its state. The controlled motor has a voltage control input and the controller uses PWM to modify average power delivery. The controller is run virtually with conversion from PWM to voltage and ZOH applied physically in a dedicated amp for the motor.

For both experiments the controller calculated below is used at a sampling time of 9ms and the shaft position is recorded via encoder counts. The duration of the experiment is 1600 samples or 14.391 seconds.

3.2 Experiment A - Open Loop

In Experiment A, the coupled motor system is run in an 'open loop', that is, the generator has no resistance applied to its terminals and as such applies only frictional resistance (neglected in ideal model). An input is immediately applied for a set speed of 1000RPM and is stepped to 2000RPM at 2.07 seconds.

3.3 Experiment B - Closed Loop

In Experiment B, the coupled motor system is run in a 'closed loop', that is, the generator has a sequence of resistors applied to it. Specifically the 1600 samples are divided into 8 sections, for which the first and last section are no load, the same as in Experiment A, and the overall number of resistors connected in sequence is $[0, 5, 0, 2, 1, 54, 14, 0]$, each for 1.8 seconds.

4 Controller Design Calculation

In order to derive the controller for this system first the plant model must be found. Starting with an open loop diagram of the motor model and controller.

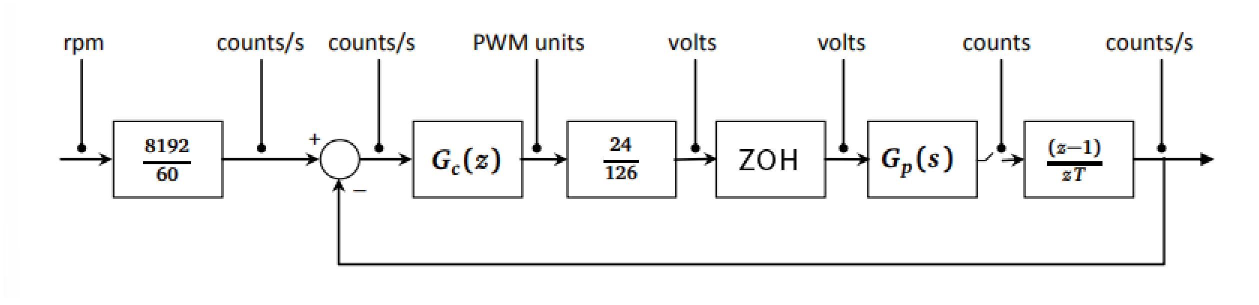

Figure 1: Speed Control Block Diagram

The motor shaft position is measured using an encoder and thus is represented by a transfer function of the form;

$$G'_p(s) = \frac{\Theta(s)}{V(s)} = \frac{A}{s(1+\tau_m s)} \tag{1}$$

Where $\Theta(s)$ is the Laplace transform of the motor shaft position $\theta(t)$, A is the gain with units counts/sec per volt and τ_m is the motor time constant in seconds. In continuous domain the integrator in the denominator can be refactored to make the transfer function;

$$G'(s) = \frac{\Omega(s)}{V(s)} = \frac{A}{(1+\tau_m s)} \tag{2}$$

The constants for the motor are found by curve fitting the expected relation to the data, the code is explicitly included below, but the terms are kept algebraic for now. The zero order hold can be included as;

$$G_p(s) = \frac{(1-e^{-sT})}{s}\frac{A}{s(1+\tau_m s)} \tag{3}$$

This is completely known after extracting motor data. A differentiator is needed to convert the units of this discrete transfer function from volts to counts to volts to counts per second. Taking the z transform of the standard difference equation gives us a differentiator in the z domain;

$$\omega(k) = \frac{\theta(k) - \theta(k-1)}{T} \tag{4}$$

$$\Omega(z) = \frac{\Theta(z) - z^{-1}\Theta(z)}{T} = \frac{(z-1)}{zT}\Theta(z) \tag{5}$$

This defines the overall plant transfer function as;

$$G_p(z) = \frac{\Theta(z)}{V(z)}\frac{\Omega(z)}{\Theta(z)} = G'_p(z)\frac{(z-1)}{zT} \tag{6}$$

Where $G'_p(z)$ is exactly the ZOH transformed transfer function in equation 3. Now that the plant is understood the Controller can be formed. By setting the poles using the standard pole placement approach. Inspecting said poles for stability and causality constraints. Enforcing the zero steady state error by introducing a variable b_0 to be the gain of the compensator $F(z)$. After this is done $G_p(z)$ and $F(z)$ are completely known and the controller can be obtained as;

$$G_c(z) = \frac{1}{G_p(z)}\frac{F(z)}{[1 - F(z)]} \tag{7}$$

The motor data can be found by determining the model of the response curve from the transfer function above and executing a curve fitting. This produces a plot and the motor parameters;

$$A = 31589.1679 \text{ count/s per volt}$$

$$\tau_m = 0.0371\text{s}$$

.

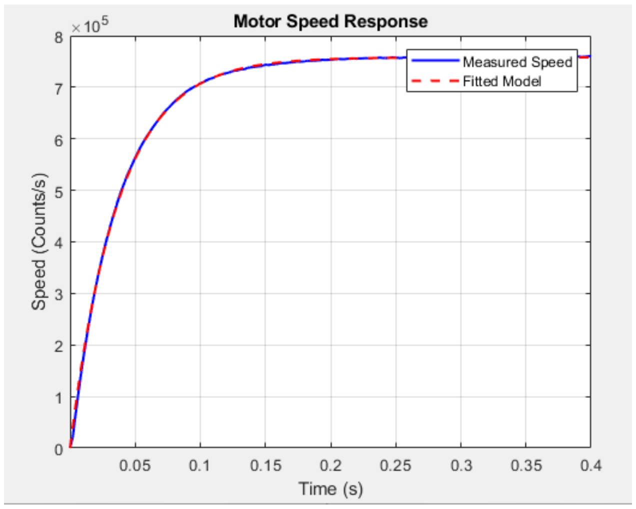

Figure 2: Motor Response at 24V

The MATLAB code shown in full in Section 9.2, explicitly calculates the above steps. The MATLAB code in Section 9.1, calls on the motor response curve fitting function above to calculate the controller transfer function according to the steps listed above, which leaves the actual transfer function as;

$$G_c(z) = \frac{0.0001705z^2 - 0.0001337z}{z^2 - 0.8939z - 0.1061} \tag{8}$$

. With a $b_0 = 0.1151$. These are the required experimental inputs.

5 Simulink Block Diagram

The above transfer function, along with an understanding of the units involved in the system can be used to produce a block diagram for simulation. Note that all the blocks are variable named, the variables are defined in the code attached in Appendix section 9.3.

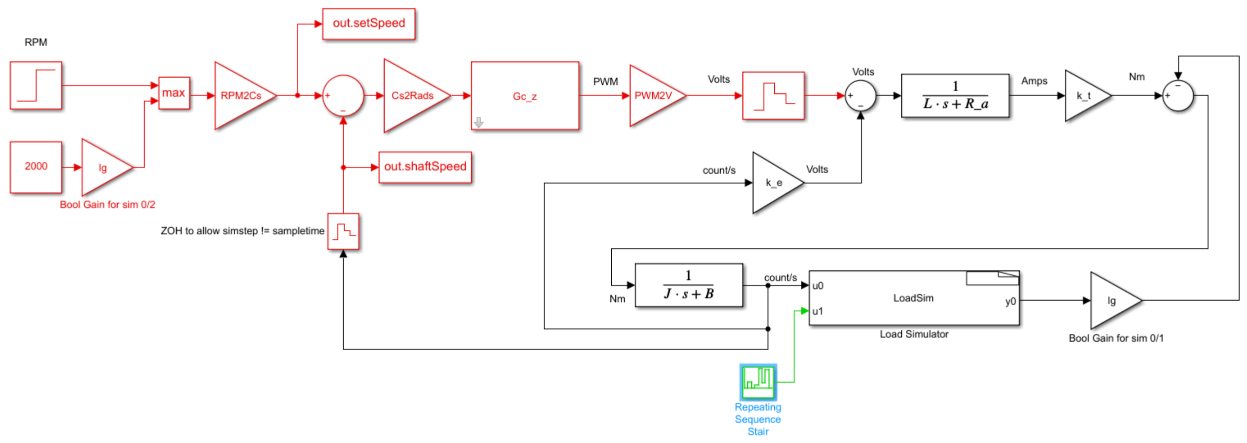

Figure 3: Simulink Block Diagram

Note two key deviations from the physical experiment here; the input logic and the load-sim output gain. These allow the l_g variable in the workspace to toggle the output of the load and the input functions, meaning that the same block diagram can be simulated for part A and B. Note also the colour, black is the continuous side and red is the discrete side, operating at the given time constant of $0.009s$. The green is a larger time constant equal to 1/8 of the total sim time per sample. The second ZOH block to bring the feedback loop into discrete time again is not strictly necessary in this simulation but it allows the use of a variable simulation time step while maintaining a fixed step for the discrete controller.

6 Simulation and Results

6.1 Part A

The complete code for the block diagram and both simulations is included in Appendix Section 9.3. The normalised data can be plotted over a restricted time scale with key values of τ plotted to visualise the step response settling time. In this case, each dataset for shaft speed is normalised against the set-speed input of 2000RPM. Additionally, the theoretical response represents a pure, first order system.

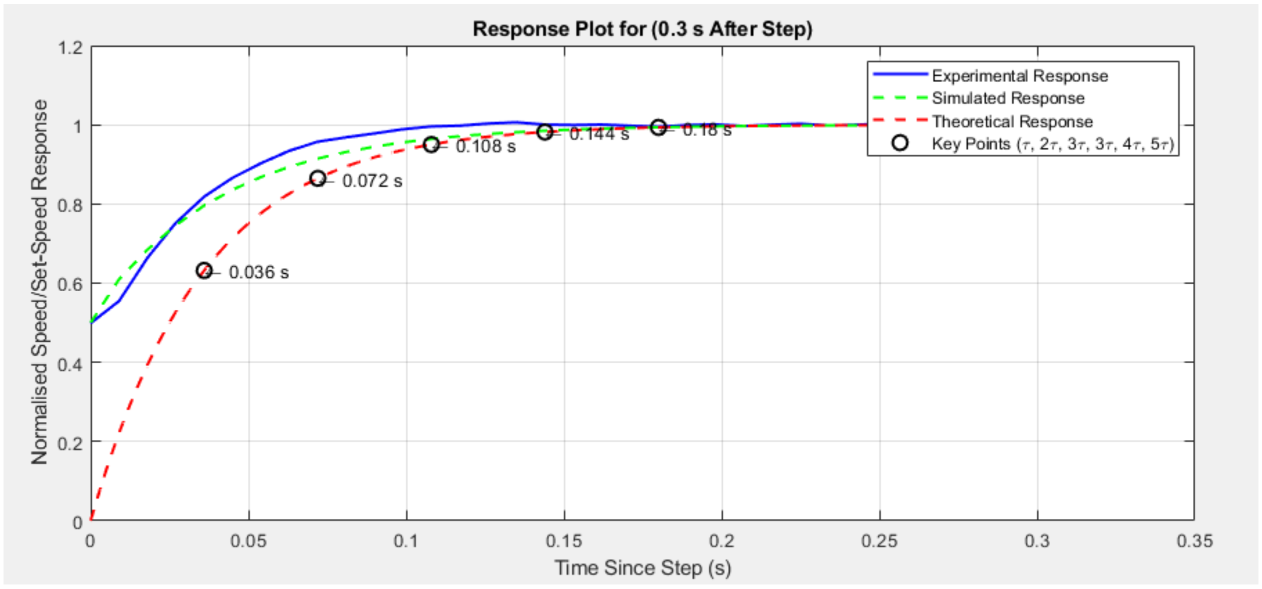

Figure 4: Normalised Step Response

This clearly demonstrates settling at approximately 5τ which verifies the controller design. The complete step response plot of experimental and simulated is also included;

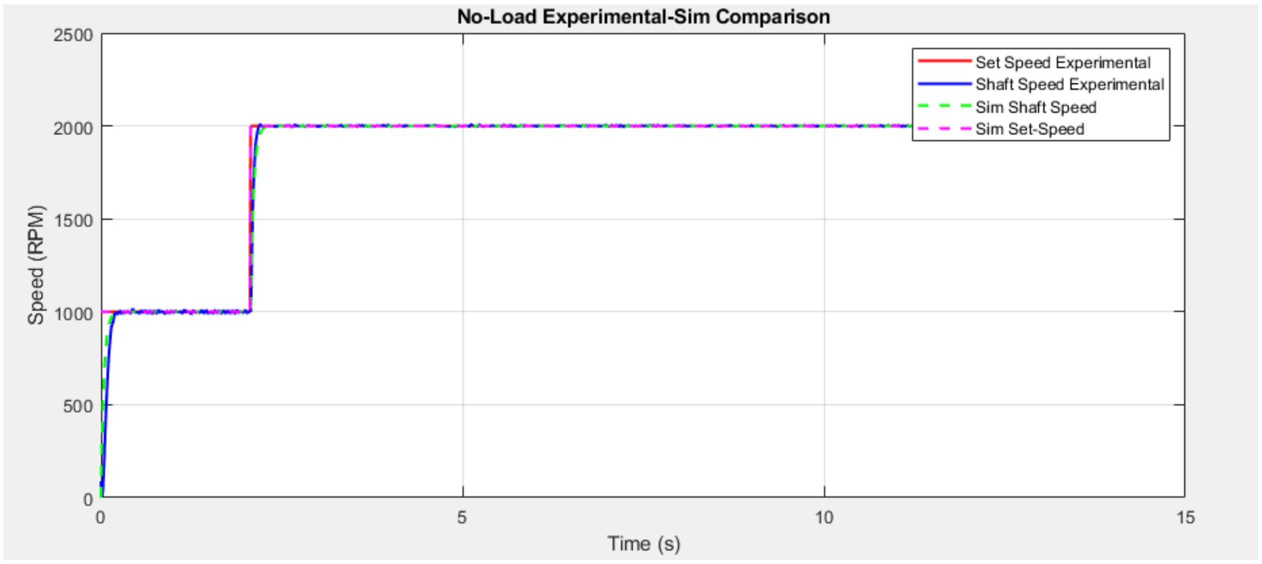

Figure 5: No Load Full Plot

A cutaway section demonstrates more clearly the dynamics of the experimental system over a step.

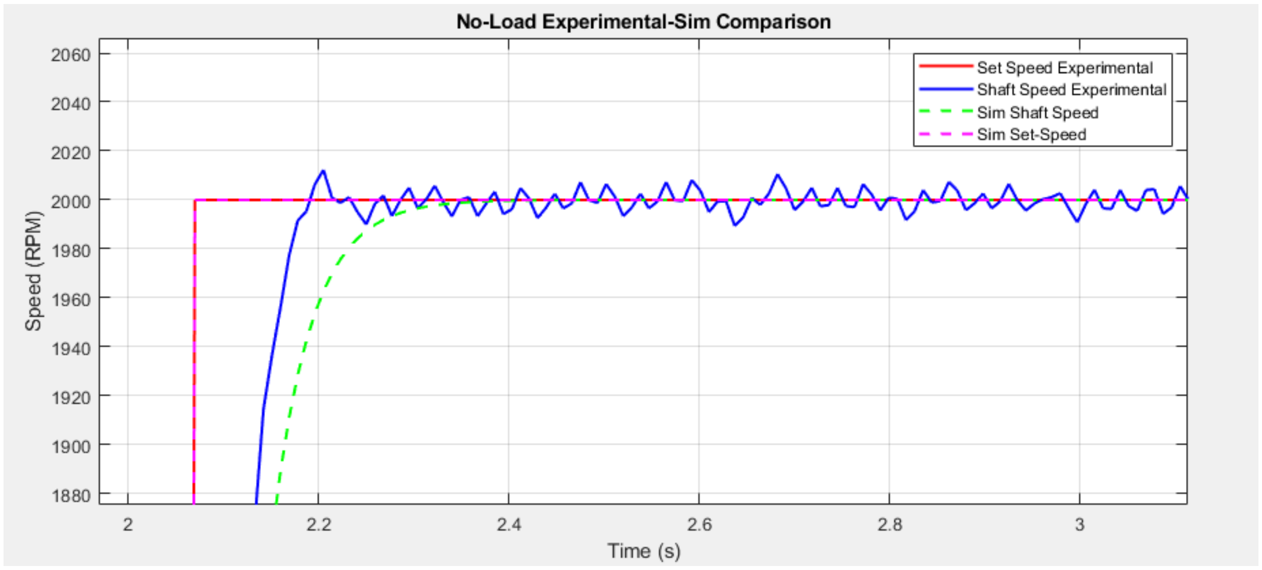

Figure 6: No Load - Section view

6.2 Part B

The complete code for the block diagram and both simulations is included in Appendix Section 9.3. The variable t_g is set in the MATLAB script to be 1, and thus engages both a constant input signal of 2000RPM and the feedback loop over the load-sim. This produces a response plot in full;

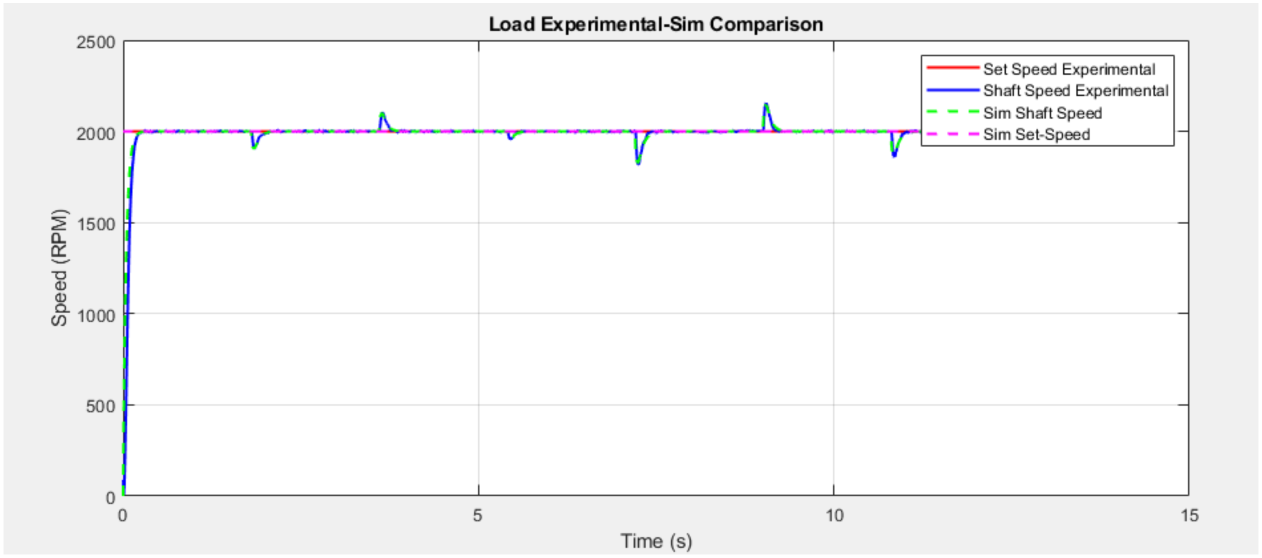

Figure 7: Load Plot

Again, this result is more telling over a step to a larger and smaller load.

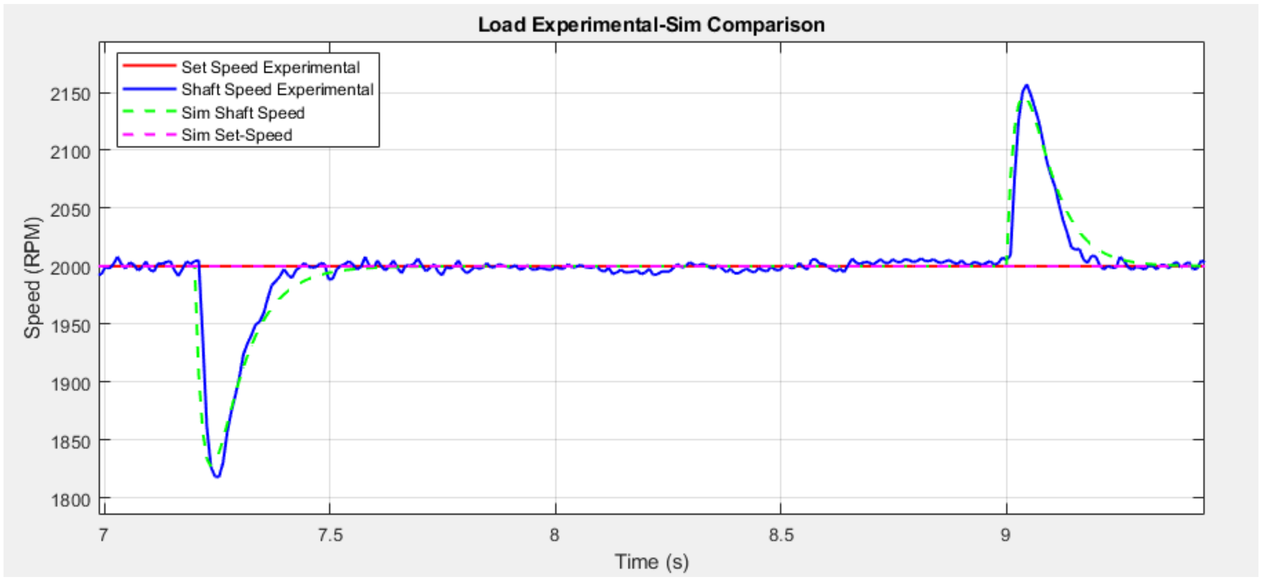

Figure 8: Load Plot - Section view

Clearly, the compatibility of the simulation and experimental results is high, with the experimental having some frictional dynamics or other errors that cause a slightly faster response time, larger overshoot and noisy data about the mean. However, as an average error both results are very comparable.

7 Discussion

The differences between experimental and ideal models can arise to many factors but they fall into six broad categories.

Model Simplifications By using motor gains and other conversion factors the dynamics of the system are essentially averaged. Depending on the type of lubricant used for instance, friction in the shaft could depend on RPM or applied lateral load, not that any is expected in this trial.

Parameter error Some of the known characteristics of the system are necessarily rounded for calculation and could cause minor discrepancies, or the voltage applied through the DC power supply could deviate from 24 V.

Sensor error The optical sensor that counts shaft position could skip counts or add counts (unlikely in this case).

Controller error If the computer running MS-DOS has any internal error it could cause the controller to skip steps or fail.

8 Other Uses

Quad Copter Motor Control Electronic speed controllers are widely used to power the props of drones and other electrical motors. In this case, instead of back EMF, the motor experiences drag forces which are proportional to the velocity of the prop squared. The power required is the product of velocity and force and is thus cubed proportional to velocity, as a consequence any changes in rotor speed have catastrophic effects on lift generation. Precise speed control is needed to maintain stable flight.

Temperature Control A less obvious parallel is the temperature control of an oven. In this case you only have control over the power delivery to a heating element, and changes to the oven such as opening the oven door or adding/removing objects at different temperatures to the set temp will require an adjustment in power delivery.

Cruise Control An obvious comparison is cruise control in a car, where the velocity should remain unchanged even as the gradient of the road changes, in this case motor characteristics are adjusted to increase torque and in an automatic car it can even change gears.

9 Appendix

9.1 Controller Transfer Function Code

The complete calculation of the controller is included

```
% z5254990
function Gc_z = Lab2_TransferFunction()
    tau_d = 36e-3; % Desired time constant in seconds
    T = 9e-3;      % Sampling time in seconds

    % first get the motor data from the experimental data
    [A, tau_m] = motorData(); % A is the motor gain, tau_m is the motor
        time constant

    % define the transfer function in continuous time
    s = tf('s');
    GP_s = (A / (s*(1 + tau_m * s)));
    GP_s = GP_s * 24/126;
    % convert to discrete time noting that this function is in volts->
        count
    GP_z_volttocount = c2d(GP_s,T, 'zoh');
    % and we need to convert to counts/sec
    z = tf('z',T); % define the discrete time z
    differentiator = (z-1)/(T*z);

    Gp_z = GP_z_volttocount * differentiator; % convert to counts/sec
    disp(Gp_z);

    % Pole PLacement based on desired characteristics.
    p_s = -1 / tau_d;          % Pole in the s-domain
    p_z = exp(p_s * T);         % Pole in the z-domain


    % Step 1: Check the poles and zeroes of Gp(z)
    plant_poles = pole(Gp_z);
    disp('Poles of Gp(z):');
    disp(plant_poles);

    plant_zeros = zero(Gp_z);
    disp('Zeros of Gp(z):');
    disp(plant_zeros);

    %nothing outside unit cirlce

    dominant_zero = plant_zeros(2);

    % solve for b0 to ensure steady state condition, F(1) = 1
    syms b0_sym
    eqn = b0_sym * (1 - dominant_zero)/((1 - p_z)*1) == 1;
    b0 = double(solve(eqn, b0_sym));
    disp('b0 : ');
    disp(b0);

    %define F(z)
    F_z = b0 * (z - dominant_zero) / (z * (z - p_z)); %sign?

    % find controller
    Gc_z = (1 / Gp_z) * (F_z / (1 - F_z));
    Gc_z = minreal(Gc_z, 0.001);

end
```

9.2 Motor Characteristics Code

```
function [A, tau_m] = motorData()
    % first find the motor gain A, and the motor time constant t_m
    run('C:\Users\nickk\OneDrive\Desktop\MTRN3020\LAB2_SpeedController\
        Laboratory Instructions-20250414\noload.m');

    time = TEST(:,1) / 1000; % time vector (ms) -> seconds
    speed = TEST(:,3); % speed vector (Counts/s)

    %plot the speed response
    figure;
    plot(time, speed, 'b-', 'LineWidth', 1.5);
    xlabel('Time (s)');
    ylabel('Speed (Counts/s)');
    title('Motor Speed Response');
    grid on;

    % we have the transfer function of the motor in the s-domain,
    % we want the step response of the motor in the s-domain
    model = @(params, t) params(1) * (1 - exp(-t / params(2))); % params
        (1) = A, params(2) = tau_m
    initial_guess = [max(speed), 0.1]; % Guess A as max speed, tau_m as
        0.1s
    params = lsqcurvefit(model, initial_guess, time, speed);
    A = params(1);
    tau_m = params(2);

    A = A / 24; %A is calculated here with a unit step response, but we
        actually have a 24V step response, so we need to divide by 24V

    % Display results
    fprintf('Motor Gain (A): %.4f counts/s per volt\n', A);
    fprintf('Motor Time Constant (tau_m): %.4f s\n', tau_m);

    % Plot fitted curve
    hold on;
    fitted_speed = model(params, time);
    plot(time, fitted_speed, 'r--', 'LineWidth', 1.5);
    legend('Measured Speed', 'Fitted Model');
end
```

9.3 Simulation and plotting Code

```
% function to define constants into workspace for use in simulink
%define all the physical constants
t_s = 0.009; %sampleing time
R_a = 4.89; %Ohm
L = 0.00042; %H
J = 0.0000109; % kg/m^2
B = 0.0000464; %Nm/(rad/s)
k_t = 0.0348; %Nm/A
k_e = 0.0348; %V/(rad/s)

%find sim time to run for
t = (1600-1)*t_s;

%define a few conversion factors
RPM2Cs = 8192/60;
Cs2RPM = 1/RPM2Cs;
Cs2Rads = 8192/(2*pi);
PWM2V = 24/126;

% get the transfer functions
Gc_z = Lab2_TransferFunction(); % this will create the Gp_z tf in the
    workspace

%run simulations
% load gain, if lg = 0, no load applied to sim, if lg = 1, step load
    applied to sim per zid.
lg = 0; % no load simulation
%run simulations
```

```matlab
simA = sim('NoLoadSim', 'StopTime', num2str(t)); %no load simulation

lg = 1;
simB = sim('NoLoadSim', 'StopTime', num2str(t)); % Stepped load simulation

% Extract simulation results for no-load (A)
simA_speed = simA.shaftSpeed; % speed in counts/s
simA_set = simA.setSpeed; % set speed in counts/s

% Extract simulation results for no-load (B)
simB_speed = simB.shaftSpeed; % speed in counts/s
simB_set = simB.setSpeed; % set speed in counts/s

% get experimental data for part A, no load, and B, stepped load
run('C:\Users\nickk\OneDrive\Desktop\MTRN3020\LAB2_SpeedController\LAB2\
    Data\A5254990.m');
%run('C:\Users\nickk\OneDrive\Desktop\MTRN3020\LAB2_SpeedController\Lab2\
    Data\A5254990.M');
ExpA = RUN;
run('C:\Users\nickk\OneDrive\Desktop\MTRN3020\LAB2_SpeedController\Lab2\
    Data\B5254990.m');
ExpB = RUN;
clear RUN;

% Extract experimental data for no-load (ExpA)
timeA = ExpA(:, 1) / 1000; % Convert time from ms to seconds
expA_speed = ExpA(:, 3); % Shaft speed in counts/s
expA_set = ExpA(:, 4);   % Set shaft speed in counts/s

% Extract experimental data for stepped load (ExpB)
timeB = ExpB(:, 1) / 1000; % Convert time from ms to seconds
expB_speed = ExpB(:, 3); % Shaft speed in counts/s
expB_set = ExpB(:, 4);   % Set shaft speed in counts/s

% Plot the no-load data (ExpA)
figure;
plot(timeA, expA_set*Cs2RPM, 'r', 'LineWidth', 1.5); hold on; %
    Experimental set_speed in red
plot(timeA, expA_speed*Cs2RPM, 'b', 'LineWidth', 1.5); % Experimental
    shaft speed in blue
plot(timeA, simA_speed*Cs2RPM, 'g--', 'LineWidth', 1.5); % Simulated shaft
     speed in green dashed line
plot(timeA, simA_set*Cs2RPM, 'm--', 'LineWidth', 1.5); % Simulated set
    speed in magenta dashed line
xlabel('Time (s)');
ylabel('Speed (RPM)');
title('No-Load Experimental-Sim Comparison');
legend('Set Speed Experimental', 'Shaft Speed Experimental', 'Sim Shaft
    Speed', 'Sim Set-Speed');
grid on;

% Plot the load data (ExpB)
figure;
```

```
plot(timeB, expB_set*Cs2RPM, 'r', 'LineWidth', 1.5); hold on; %
    Experimental set_speed in red
plot(timeB, expB_speed*Cs2RPM, 'b', 'LineWidth', 1.5); % Experimental
    shaft speed in blue
plot(timeB, simB_speed*Cs2RPM, 'g--', 'LineWidth', 1.5); % Simulated shaft
     speed in green dashed line
plot(timeB, simB_set*Cs2RPM, 'm--', 'LineWidth', 1.5); % Simulated set
    speed in magenta dashed line
xlabel('Time (s)');
ylabel('Speed (RPM)');
title('Load Experimental-Sim Comparison');
legend('Set Speed Experimental', 'Shaft Speed Experimental', 'Sim Shaft
    Speed', 'Sim Set-Speed');
grid on;

% now analysis to prove tau_d = 36ms is correct
% Define the step time and time range
step_time = 2.07; % Time when the step occurs
time_limit = 0.3; % Stop plotting 0.3 seconds after the step - arbitrary
    but makes a pretty plot
tau = 0.036; % Desired time constant (36 ms)

% indices for to the step time and time limit
start_index = find(timeA >= step_time, 1); % Start when the 1000->2000 RPM
     step happens
end_index = find(timeA >= step_time + time_limit, 1);

% get data
timeA_step = timeA(start_index:end_index) - step_time; % rel time so that
    plotting shows settling time better
expA_speed_step = expA_speed(start_index:end_index) ./ expA_set(
    start_index:end_index); % norm
simA_speed_step = simA_speed(start_index:end_index) ./ simA_set(
    start_index:end_index); % norm

% make ideal response data for tau = 36 ms
theoretical_time = linspace(0, time_limit, 1000);
theoretical_response = 1 - exp(-theoretical_time / tau); % ideal step
    response

% key times for visualisation
key_times = [tau, 2*tau, 3*tau, 4*tau, 5*tau];
key_values = 1 - exp(-key_times / tau); % ideal values at key times

% Plot
figure;
plot(timeA_step, expA_speed_step, 'b', 'LineWidth', 1.5); hold on; % Exp
    response
plot(timeA_step, simA_speed_step, 'g--', 'LineWidth', 1.5); % Sim response
plot(theoretical_time, theoretical_response_offset, 'r--', 'LineWidth',
    1.5); % Theory response

% key points on plot
```

```matlab
plot(key_times, key_values, 'ko', 'MarkerSize', 8, 'LineWidth', 1.5); % Key points
for i = 1:length(key_times)
    text(key_times(i), key_values(i), ...
        ['\leftarrow ', num2str(key_times(i)), ' s'], 'FontSize', 10, 'Color', 'k');
end

% Add labels, title, and legend
xlabel('Time Since Step (s)');
ylabel('Normalised Speed/Set-Speed Response');
title('Response Plot for (0.3 s After Step)');
legend('Experimental Response', 'Simulated Response', ...
       'Theoretical Response', 'Key Points (\tau, 2\tau, 3\tau, 3\tau, 4\tau, 5\tau)');
grid on;
```