Process Automation Coursework

Task 2

The system identification process aims to derive the transfer functions for the robot's right and left actuators, which are modeled as first-order systems. The process is implemented in the program through a series of steps, as described in the pseudocode provided in the Appendix. A part of this process is calculating the system parameters, which are the time constant and gain for each wheel. First, the system is subjected to an input PWM step signal of 1 for both actuators. The resulting angular velocity response data is collected and shown in Figure 1. The steady-state velocity is determined from the collected angular velocity data and the corresponding 63.2% of the steady-state velocity is calculated. Because the data is discrete, interpolation is done by the program to identify the precise time at which the angular velocity crosses the 63.2%. The time constant is obtained by subtracting the sampling time from the 63.2% crossing time. The gain is then calculated as the ratio of the steady-state velocity to the applied input signal.

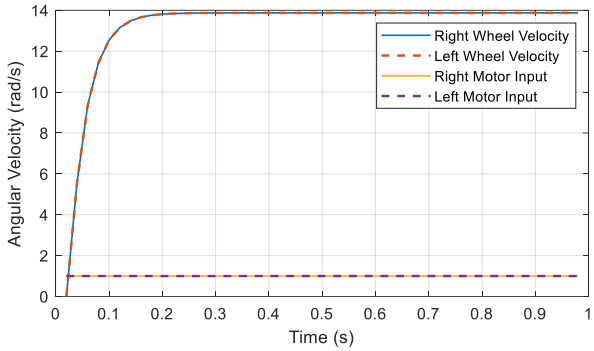


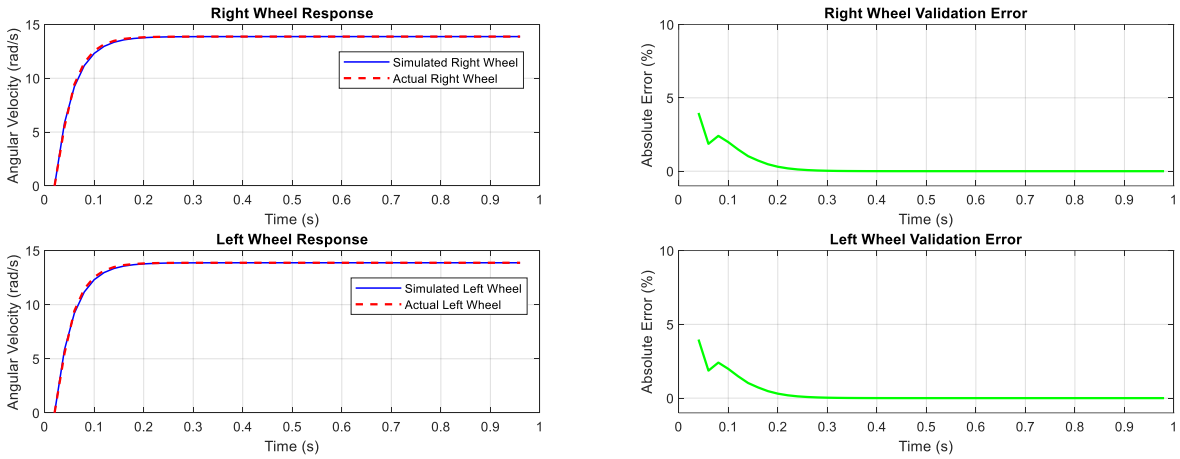
Figure 1 Step response of right and left actuators

Following the parameter determination, continuous-time transfer functions are generated for each wheel using the calculated time constant and gain. These transfer functions are then converted into discrete-time transfer functions using ZOH discretization with a sampling time of 0.02 seconds. This conversion aligns the model with the discrete sampling of the actual angular velocity data, thus ensuring direct comparison during validation. As seen from Figure 1, both the right and left wheel exhibits identical responses, thus the transfer function for both actuators are the same. The obtained continous and discrete transfer functions are shown in Equations (1) and (2).

$$G(s) = \frac{13.87}{0.03685s + 1} \qquad G(z) = \frac{5.811}{z - 0.5812} \tag{1) \& (2)}$$

To validate the model, the program simulates the system's response to a step input using the discrete transfer functions. These simulated responses are compared to the actual measured data for each wheel and is shown in Figure 2 (a). Additionally, the program computes the absolute error percentage between the simulated and actual responses as a quantitative measure of the model's performance, it is shown in Figure 2 (b). The absolute error percentage is calculated using Equation (3).

$$Absolute\ error\ \% = \left|\frac{Simulated\ velocity - Actual\ velocity}{Actual\ velocity}\right| \cdot 100\% \tag{3}$$



(a) (b)

Figure 2 (a) Model validation result (b) Model validation absolute error

The model validation results shows that the derived first-order transfer functions accurately represent the dynamic behavior of the right and left actuators. As illustrated in Figure 2 (a), the simulated angular velocity responses closely match the actual data, reflecting the validity of the identified system parameters, including the time constant and steady-state gain. Additionally, the absolute error plot in Figure 2 (b) indicates that the modeling error is minimal, peaking at 4% at the start of the simulation before rapidly decreasing and converges at near-zero values. However, the model is not perfect as some error is still observed. This persistent error is due to the exclusion of higher order dynamics which the first-order model does not capture. These dynamics cause deviations during the transient response and slight disrepancies in the steady-state response. Despite this, the model is sufficiently accurate as the higher order effects can be neglected overall.

Task 3

The velocity form of PID control is chosen for this task because of its compatibility with the robot's discrete-time simulation. This approach effectively prevents integral windup under actuator saturation, where the motor's PWM signal is constrained to the range [-1, 1]. After selecting the implementation method, the PID controller is tuned using the Ziegler-Nichols method. The tuning process is implemented through a series of steps, as detailed in the pseudocode provided in the Appendix. It starts with setting the controller on a proportional-only mode, where the integral time (Ti) is set to infinity and the derivative time (Td) is set to zero. The proportional gain (Kp) is then gradually increased until the system reaches marginal stability, indicated by sustained oscillations. To allow sufficient time to observe the marginally stable response, the simulation time is set to 20 seconds. The system's marginally stable response is shown in Figure 3. At this point, the ultimate gain (Ku) and oscillation period (Pu) are recorded as 0.342 and 0.04108 seconds, respectively. These values are used to calculate the controller settings for P, PI, and PID controllers and are summarized in Table 1.

Table 1 Controller settings with Ziegler-Nichols tuning

	P	PI	PID
Kp	0.1710	0.1539	0.2052
Ti	-	0.0342	0.0205
Td	-	-	0.0052

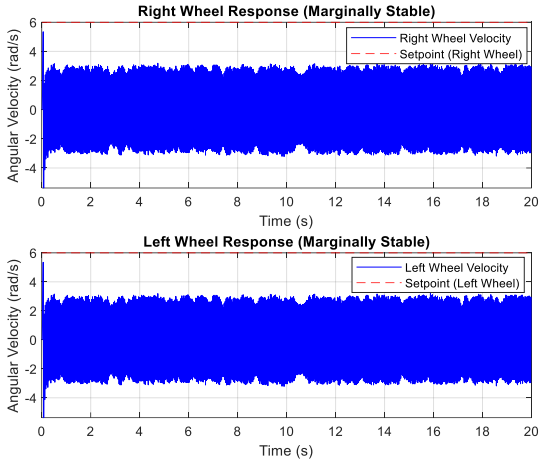


Figure 3 Marginally stable response for Ziegler-Nichols tuning

The controller settings shown in Table 1 are applied to the program, and the simulation is run at two setpoints: a low setpoint (3 rad/s) and a high setpoint (12 rad/s). A comparison of the performance of the P, PI, and PID controllers is presented in Figure 4 (a), (b), and (c). Starting with the P controller, shown in Figure 4 (a), it demonstrates fast initial responses for both setpoints but fails to reach the desired velocities due to the presence of steady-state error, which becomes more noticeable at higher setpoints. This limitation highlights the inherent drawback of the P controller, as it cannot compensate for the system's offset without integral action.

The PI controller, shown in Figure 4 (b), improves upon the P controller by eliminating the steady state error through the addition of integral action. For both set points, the controller achieves the desired steady state velocities with no offset. However, at the lower set point (3 rad/s), overshoot is observed during the transient phase, indicating that the controller temporarily exceeds the target value before settling. This occurs because the integral action accumulates error over time and introduces a delayed correction. At the lower setpoint, where the system requires less control effort to reach the target velocity, the integral term can dominate the control response, leading to a faster rise time but also causing overshoot. The proportional term, which reacts to the current error, cannot immediately counteract the accumulated action of the integral term, resulting in the observed overshoot.

(a) (b) (c)

Figure 4 (a) P, (b) PI, and (c) PID controller closed loop response with Ziegler-Nichols tuning

The PID controller, shown in Figure 4 (c), exhibits noisy oscillations at the lower setpoint (3 rad/s) and smaller but persistent fluctuations at the higher setpoint (12 rad/s). These oscillations are primarily caused by the derivative term amplifying high-frequency noise in the error signal, which is more significant at lower setpoints due to the smaller error magnitude and the relatively larger impact of noise. This sensitivity leads to significant control output fluctuations, particularly during steady-state operation.

The implementation of the IMC tuning process is detailed in the pseudocode provided in the Appendix. Since the system in this task is a discrete system, the sampling time is taken as the delay (td = 0.02 seconds). The tuning process begins with defining the desired closed-loop performance through a tuning parameter (λ), which determines the trade off between response speed and robustness. The value of λ was chosen was chosen based on a trial-and-error process, as shown in Figure 5, where different λ values were tested for both PI and PID IMC controllers to evaluate their impact on system performance. From Figure 5, it was concluded that λ = 0.025 provided a balance between fast response and robust stability, as shown by Figure 5 (b) for the PI implementation and Figure 5 (d) for the PID implementation of IMC. Both responses showed no overshoot or oscillations for low and high set points while achieving fast rise times.

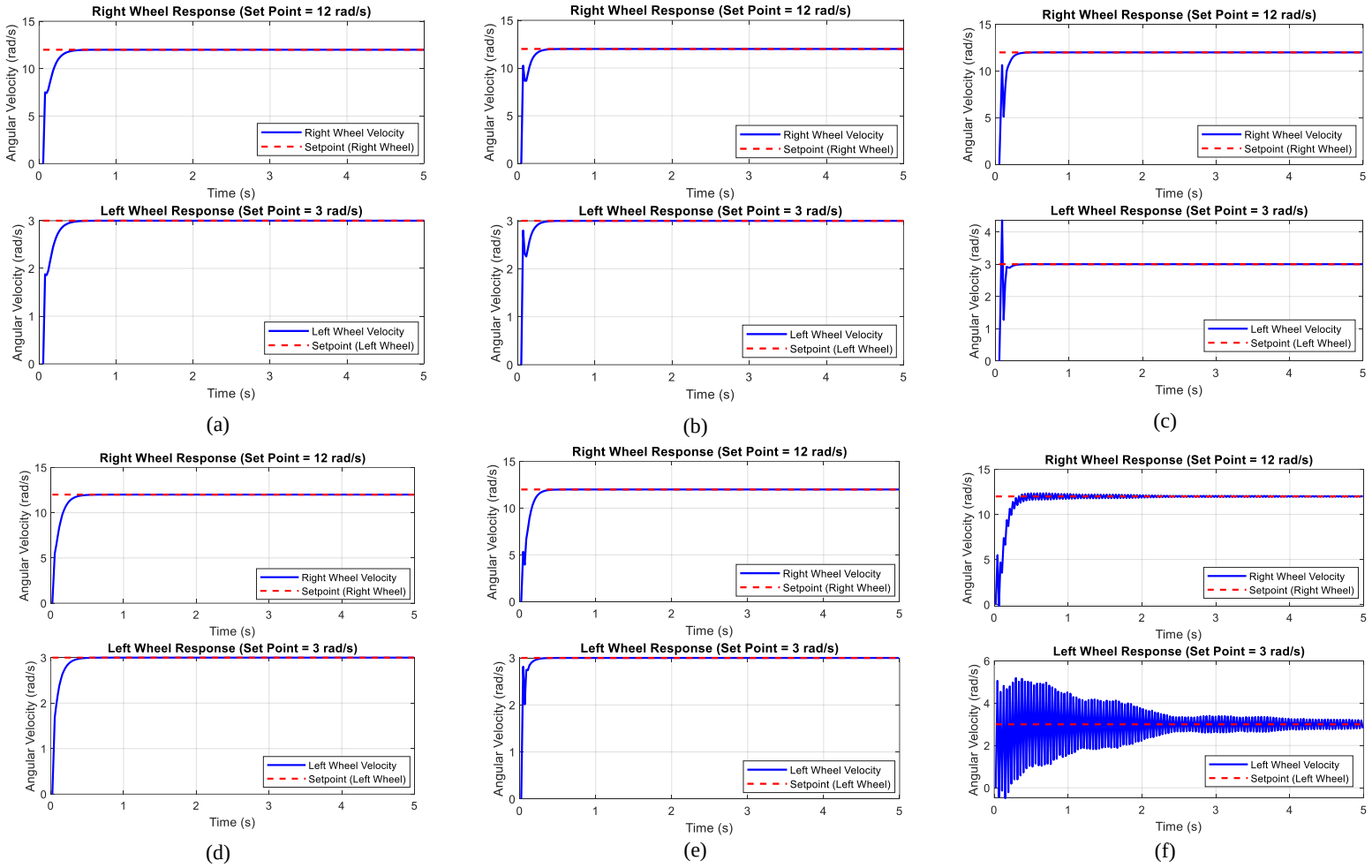


Figure 5 (a) PI IMC λ = 0.05, (b) PI IMC λ = 0.025, (c) PI IMC λ = 0.01, (d) PID IMC λ = 0.05, (e) PID IMC λ = 0.025, and (f) PID IMC λ = 0.01 controller closed loop response

The IMC tuning formulas used in this task are based on *Process Dynamics and Control* (Seborg et al., 2017) and are included in the Appendix. Using these formulas, the proportional gain, integral time, and derivative time are calculated based on the system's parameters identified in Task 2. The controller settings obtained with IMC tuning is summarized in Table 2.

Table 2 Controller settings with IMC tuning

	PI	PID
Kp	0.059	0.0965
Ti	0.0369	0.0469
Td	-	0.0079

The closed loop transfer function of the system in terms of Kp and Ki is derived using the Direct Synthesis method as follows:

$$C(s) = \frac{K_p s + K_i}{s} \tag{3}$$

Insert system model transfer function stated in Equation (1) and controller transfer function stated in Equation (3) into the following,

$$T(s) = \frac{C(s)G(s)}{1 + C(s)G(s)} = \frac{\frac{K_p s + K_i}{s}\frac{13.87}{0.03685s + 1}}{1 + \frac{K_p s + K_i}{s}\frac{13.87}{0.03685s + 1}} \tag{4}$$

Simplify to obtain the closed loop transfer function,

$$T(s) = \frac{13.87(K_p s + K_i)}{0.03685s^2 + (1 + 13.87K_p)s + 13.87K_i} \tag{5}$$

Values of Kp and Ki that will result in closed loop dynamics with a natural frequency of 20 rad/s and damping ratio of 0.85:

$$0.03685s^2 + (1 + 13.87K_p)s + 13.87K_i = s^2 + 2\zeta\omega_n s + \omega_n^2 \tag{5}$$

$$0.03685s^2 + (1 + 13.87K_p)s + 13.87K_i = s^2 + 2 \cdot 0.85 \cdot 20s + 20^2 \tag{6}$$

$$\frac{1 + 13.87K_p}{0.03685} = 2 \cdot 0.85 \cdot 20 \xrightarrow{yields} K_p = 0.0182 \tag{7}$$

$$\frac{13.87K_i}{0.03685} = 20^2 \xrightarrow{yields} K_i = 1.0627 \tag{8}$$

In determining the most suitable PID configuration for the actuator control, a comparison is done on the three tuning methods implemented in this task: Ziegler-Nichols (ZN), Internal Model Control (IMC), and Direct Synthesis (DS), shown in Figure 6. The DS method shows the slowest response in reaching the steady state of both set points. The IMC PI method offers a smoother response, with no overshoot and consistent performance across both set points, balancing stability and speed. The IMC PID method achieves faster rise times but introduces transient oscillations, which may compromise stability under noise or disturbances. The ZN PI method exhibits the fastest response but suffers from overshoot and noticeable oscillations, indicating overly aggressive tuning. Among these methods, the IMC PI method stands out as the best choice, providing a stable and fast response with no overshoot while effectively tracking both low and high set points.

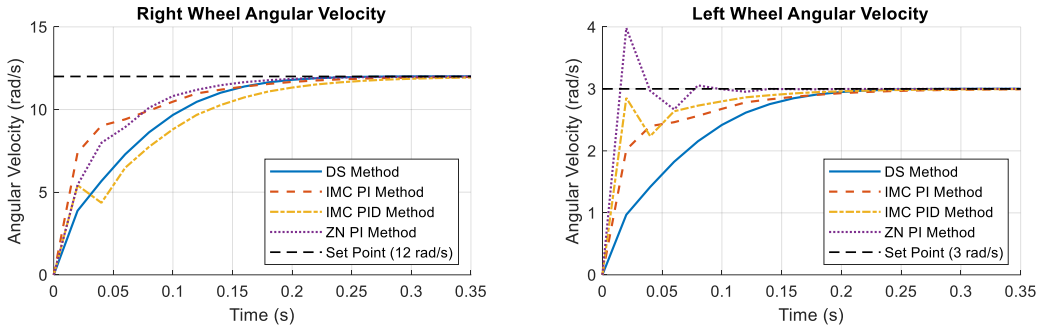
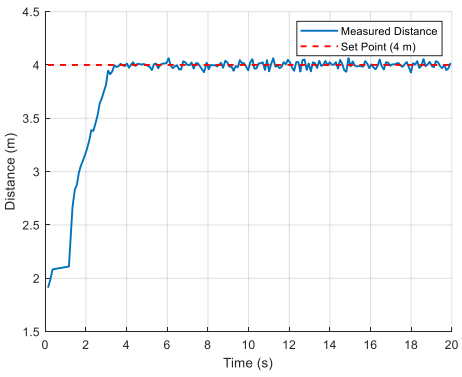


Figure 6 Comparison of PID tuning methods

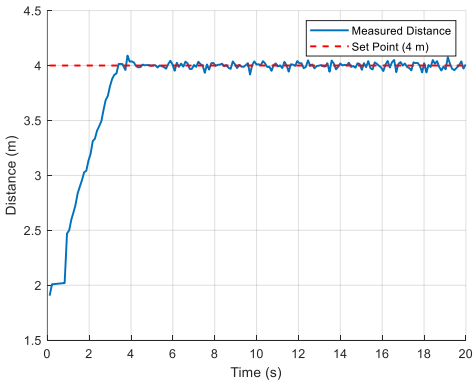
Task 4

The cascade control implementation in this system consists of an arrangement of two interconnected control loops: an outer loop for distance control and an inner loop for wheel velocity control. The outer loop is responsible for regulating the robot's distance from obstacles by adjusting the desired linear velocity based on sensor readings from the laser distance sensors. This desired linear velocity serves as the set point for the inner loop. The inner loop ensures control of the robot's wheel velocities by calculating the necessary motor PWM signals based on the wheel velocity errors. The detailed implementation of this cascade control is outlined in the pseudocode provided in the Appendix.

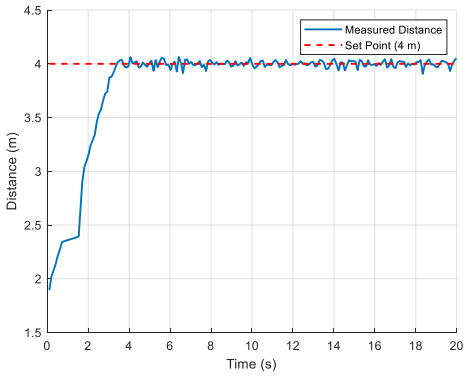
The tuning process for the cascade control was carried out systematically. The inner loop, already tuned in Task 3, implemented an IMC PI controller using parameters from Table 2. The outer loop was tuned iteratively by gradually increasing the proportional gain (Kp) and observing the system's response, with the integral time (Ti) set to infinity and the derivative time (Td) set to zero. The ideal proportional gain was determined to be Kp = 5, as shown in Figure 7. Figure 7 illustrates the system's response to a given distance setpoint with varying Kp values. At Kp = 4, the response shows no overshoot but a longer rise time. At Kp = 6, the system responds faster but exhibits overshoot and more aggressive control efforts done by the inner loop. In contrast, Kp = 5 provides a good balance with a fast rise time, minimal overshoot, and a less aggressive inner loop control signal. Using only a proportional controller for the outer loop was sufficient to meet the performance requirements, achieving the desired response with no steady-state error.



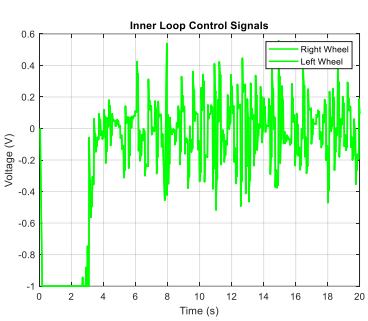
(a)



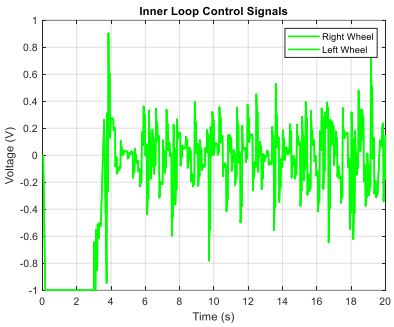
(b)



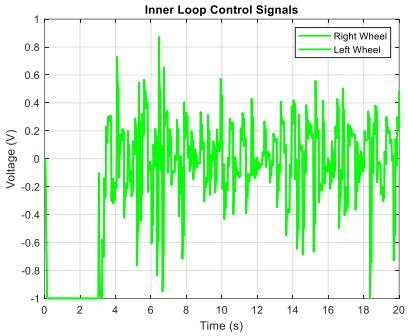
(c)



(d)



(e)



(f)

Figure 7 Outer loop output (a) Kp = 4, (b) Kp = 5, (c) Kp = 4, Inner loop control signal (d) Kp = 4, (e) Kp = 5, (f) Kp =6

Figure 8 demonstrates the performance of the leader-follower strategy. As seen in Figure 8, the robot exhibits accurate steady-state tracking, maintaining a minimal steady-state error across all intervals. During setpoint transitions (at 10, 20, and 30 seconds), the system responds smoothly with no oscillations and overshoots. The settling time for each transition is approximately 2–5 seconds, indicating an acceptable trade-off between speed and stability.

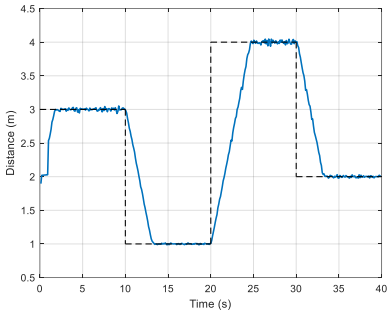
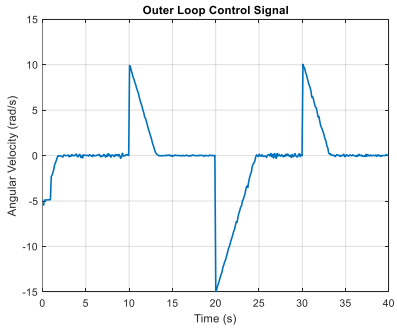
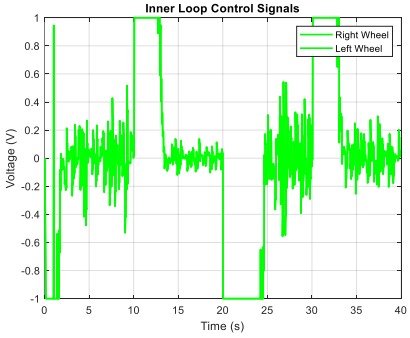


Figure 8 System response with leader-follower strategy

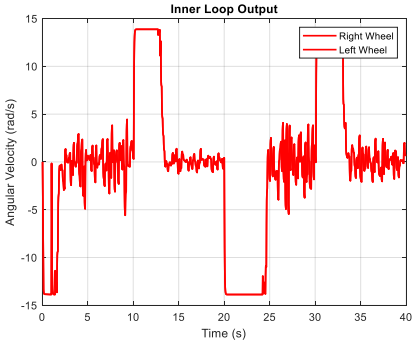
Figure 9 illustrates the leader-follower system's performance during dynamic set point transitions. In Figure 9 (a), the outer loop control signal shows significant peaks and dips around the set point transitions, reflecting the system's adaptation to sudden setpoint changes, with stabilization during steady-state intervals. Figure 9 (b) highlights the inner loop control signals, where large spikes and dips correspond to the outer loop's commands. Figure 9 (c) demonstrates the inner loop outputs, with angular velocities aligning closely with the outer loop commands, showing rapid transitions and oscillations in the steady-state.



(a)



(b)



(c)

Figure 9 Outer loop control signal, (b) Inner loop control signal, and (c) Inner loop output for leader-follower strategy

Appendix

Pseudocode for Task 2

```
Initialize:
  Clear all variables and close figures
  Restore default MATLAB path
  Add current directory and subdirectories to MATLAB path

Create Robot and World:
  Initialize robot object with configuration from 'robot_0002.json'
  Initialize world object with configuration from 'world_0002.json'

Set Parameters:
  Set total simulation duration (TotalTime)
  Set sampling time (t_sampling)

Initialize Variables:
  Set initial wheel control signals (uR, uL)
  Set initial wheel angular velocities (wR, wL)
  Create empty vectors to store input/output data for each wheel

Start Simulation Timer:
  Start total time (t_start) and loop timer (t_loop)

Begin Main Loop (while time elapsed < TotalTime):
  Calculate elapsed time (dt)
  If elapsed time (dt) >= sampling time:
    Reset loop timer (t_loop)

    Save current angular velocities and control signals to data vectors

    Update robot dynamics:
      Send control signals (uR, uL) to the robot actuators
      Update robot pose and sensors in simulation
      Retrieve new angular velocity readings from encoders (wR, wL)

    Pause briefly to ensure loop timing consistency

After Simulation Ends:
  Compute total number of samples and generate time vector for plotting

Plot Step Response:
  Plot wheel velocities (wR, wL) and motor inputs (uR, uL) against time

Calculate System Parameters (Time Constant and Gain):
  For each wheel (right and left):
    Determine steady-state velocity
    Calculate 63.2% of steady-state value
    Find time at which velocity crosses threshold using interpolation
    Calculate time constant (t63 - sampling time)
    Calculate gain (steady-state velocity / input signal)

Display Calculated Parameters:
  Output time constant and gain values for each wheel

Generate Transfer Functions:
  For each wheel:
    Create continuous-time transfer function (G)
```

Convert to discrete-time transfer function (G_disc) using zero-order hold

Simulate System Response:
Generate step input signal
Simulate output response for each wheel using discrete transfer function

Validate Model:
Compare simulated and actual responses for each wheel
Compute absolute error percentage between simulated and actual values

Plot Validation Results:
Plot simulated vs. actual responses for each wheel
Plot validation error as a percentage for each wheel

End Program

Pseudocode for Task 3 – Ziegler Nichols Tuning
Initialize:
Clear workspace, reset paths, and add necessary directories.
Create robot and world objects from JSON configuration files.
Define simulation parameters, including total duration (TotalTime) and sampling time (t_sampling).
Set PID controller gains (Kp, Ti, Td) based on desired control strategy (e.g., proportional-only or full PID).
Initialize variables for previous errors, control signals, and data storage (e.g., velocities, control inputs, setpoints, and time).

Start Simulation:

Start timers (t_start, t_loop) for total simulation time and loop execution.
While elapsed time < TotalTime:
If elapsed time ≥ t_sampling:
Reset loop timer.
Record current time and save it for later analysis.
Compute control errors for both wheels as the difference between setpoints and current velocities.
Use the velocity form of the PID formula to compute incremental changes in control signals (delta_uR, delta_uL).
Update control signals (uR, uL) by adding increments and saturating to the allowable range ([-1, 1]).
Shift and save previous errors and control signals for the next iteration.
Save current data for velocities, control signals, and setpoints.
Update robot dynamics, apply control signals, and retrieve new encoder readings for wheel velocities (wR, wL).
Pause briefly to maintain timing consistency.

Post-Simulation:

Use findpeaks to detect peaks in velocity data for both wheels.
Calculate time differences between consecutive peaks to determine oscillation periods.
Compute the average oscillation period for each wheel.

Display Results:

Output the calculated average oscillation periods for the right and left wheels.
Plot velocity responses for both wheels with setpoints.

End Program.

Pseudocode for Task 3 – IMC Tuning
Initialize:

Clear workspace, reset paths, and load external parameters (e.g., gain and time constants).
Create robot and world objects from JSON configuration files.

Define simulation parameters: total duration (TotalTime), sampling time (t_sampling), and desired closed-loop time constant (λ).
Set system delay to sampling time (td = t_sampling).

Calculate IMC PID Gains:

Compute gains (Kp, Ti, Td) using IMC formulas, accounting for delay and time constants.
Display the calculated gains.

Start Simulation:

Initialize previous errors and control signals for PID computation.
Set target wheel velocities (wR_set, wL_set) and initialize velocity data storage.
While elapsed time < TotalTime:
If elapsed time ≥ t_sampling:
Reset timer and record current time.
Compute control errors and use the IMC PID velocity formula to calculate control signal increments.
Update and saturate control signals to [-1, 1].
Update robot actuators, retrieve encoder readings, and store data for plotting.

Display Results:

Plot wheel velocities against setpoints to visualize performance.

End Program.

Pseudocode for Task 4 – Fixed Set Point
Initialization:

Clear all variables, close all figures, reset paths, and add the current working directory to the path.
Create a robot object (R1) and load its specifications from a JSON file.
Create a world object (W) and load its specifications from another JSON file.
Plot the world and robot's initial position.

Define Parameters:

Total simulation duration (TotalTime) and sampling times for the inner (sampling_inner) and outer loops (sampling_outer).
Initialize PID control parameters:
Outer Loop (Distance Control): Proportional, integral, and derivative gains.
Inner Loop (Wheel Velocity Control): Proportional, integral, and derivative gains.
Initialize variables for storing control signals, errors, and angular velocities for both loops.

Outer Loop Variables:

Initialize variables for distance control, such as previous control signal (u_outer_prev) and previous errors.

Inner Loop Variables:

Initialize variables for wheel velocity control, such as previous control signals for left and right wheels and their respective errors.

Set Desired Distance and Velocities:

Set the desired distance (d_set) from the obstacle.
Initialize wheel angular velocity setpoints (wR_set, wL_set) and actual wheel velocities (wR, wL).

Start Simulation Loop:

Run the simulation until the total simulation time is reached.

Outer Loop (Distance Control):

Execute if the elapsed time exceeds sampling_outer.
Read the robot's distance from an obstacle using laser sensors.
Compute the error between the desired and actual distances.
Calculate the change in the control signal (delta_u_outer) using the PID formula.
Update the control signal (u_outer) and convert it to a linear velocity (v_set).
Calculate wheel angular velocity setpoints (wR_set, wL_set).
Store control signals and distance data for plotting.
Update previous values for the next iteration.
Update the robot's plot and sensor readings.

Inner Loop (Wheel Velocity Control):

Execute if the elapsed time exceeds sampling_inner.
Compute errors for the right and left wheel velocities.
Calculate the change in control signals (delta_uR, delta_uL) for the right and left wheels using the PID formula.
Update control signals and saturate them to a PWM range of [-1, 1].
Update the robot's actuators and read wheel encoder velocities.
Store angular velocities and control signals for plotting.
Update previous values for the next iteration.

Stop the Robot:

Send zero control signals to the robot's actuators after the simulation loop ends.

Plot Results:

Outer loop control signals vs. time.
Inner loop control signals for both wheels vs. time.
Angular velocities for both wheels vs. time.
Distance to the obstacle vs. time

End Program.

Pseudocode for Task 4 – Leader Follower Navigation
Initialization:

Clear all variables, close all figures, reset paths, and add the current working directory to the path.
Create a robot object (R1) and load its specifications from a JSON file.
Create a world object (W) and load its specifications from another JSON file.
Plot the world and the robot's initial position.

Define Parameters:
Define the total simulation duration (TotalTime) and sampling times for the inner (sampling_inner) and outer loops (sampling_outer).
Define setpoint change intervals (setpoint_intervals) and corresponding leader distances (leader_setpoints).

Initialize PID control parameters:
Outer Loop (Distance Control): Proportional, integral, and derivative gains.
Inner Loop (Wheel Velocity Control): Proportional, integral, and derivative gains.
Initialize variables for storing control signals, errors, and angular velocities for both loops.

Outer Loop Variables:
Initialize variables for distance control:
Previous control signal (u_outer_prev).

Previous and second-previous distance errors (e_outer_prev1, e_outer_prev2).

Inner Loop Variables:
Initialize variables for wheel velocity control:
Previous control signals for left and right wheels (uR_prev, uL_prev).
Previous and second-previous velocity errors for both wheels (eR_prev1, eR_prev2, eL_prev1, eL_prev2).

Set Desired Distance and Velocities:
Set the initial desired distance (leader_distance) based on the first setpoint.
Initialize wheel angular velocity setpoints (wR_set, wL_set) and actual wheel velocities (wR, wL).

Start Simulation Loop:
Run the simulation until the total simulation time is reached.

Setpoint Update:
Divide TotalTime into intervals (setpoint_intervals) with corresponding distances (leader_setpoints).
During each simulation loop iteration:
Check if the elapsed simulation time has entered a new interval.
If the simulation time crosses the next interval:
Increment current_setpoint_index to the next interval.
Update leader_distance to the corresponding value in leader_setpoints.
If the simulation time exceeds the final interval:
Keep leader_distance constant at the last setpoint value.

Outer Loop (Distance Control):
Execute if the elapsed time exceeds sampling_outer.
Read the robot's distance from an obstacle using laser sensors.
Compute the error between the desired and actual distances.
Calculate the change in the control signal (delta_u_outer) using the PID formula.
Update the control signal (u_outer) and convert it to a desired linear velocity (v_set).
Calculate wheel angular velocity setpoints (wR_set, wL_set).
Store control signals and distance data for plotting.
Update previous values for the next iteration.
Update the robot's plot and sensor readings.

Inner Loop (Wheel Velocity Control):
Execute if the elapsed time exceeds sampling_inner.
Compute errors for the right and left wheel velocities.
Calculate the change in control signals (delta_uR, delta_uL) for the right and left wheels using the PID formula.
Update control signals and saturate them to a PWM range of [-1, 1].
Update the robot's actuators with the control signals.
Read the current wheel velocities from encoders (wR, wL).
Store angular velocities and control signals for plotting.
Update previous values for the next iteration.

Stop the Robot:
Send zero control signals to the robot's actuators after the simulation loop ends.

Plot Results:
Outer loop control signals vs. time.
Inner loop control signals for both wheels vs. time.
Angular velocities for both wheels vs. time.
Distance to the obstacle vs. time, including the setpoint as a dashed line.

End Program.