

Modelling Automotive Control Systems in MATLAB

Analysis of a Digital PID Controller for Cruise Control

Student Name: Hassan Ali Raza

Student ID: 2513043

Programme: MSc Automotive Engineering

Module: Energy Storage and Control Systems (ACHA7007)

Submission Date: 29/03/2026

Table of Contents

List of Figures	3
List of Tables	3
1. Introduction	4
2. Code Implementation: Encoding and Decoding	4
2.1 Decoding the Demand Signal	5
2.2 Encoding the Actuator Signal	5
2.3 Sensor Signal	5
3. PID Controller Design and Gain Selection	5
3.1 Gain Tuning	6
4. Results and Analysis	6
4.1 Velocity Response and Control Effort	7
4.2 Effect of Quantization (Number of Bits)	8
4.3 Effect of Sampling Time	9
4.4 Effect of Communication Delay	10
4.5 ISO 2631 Passenger Comfort Analysis	11
5. Comparison with Published Literature	11
References	13
Appendix:	14
Appendix A. CruiseControlScript.m	14

List of Figures

Figure 1: Velocity Response with Optimized PID Controller (16-bit, $T_s=0.05$ s)	7
Figure 2: Control Signal and Throttle Position	7
Figure 3: Effect of Bit Resolution on Velocity Response	8
Figure 4: Effect of Sampling Time on velocity Response and Step Response at 35s	9
Figure 5: Effect of Communication Delay on Velocity Response	10
Figure 6: Vehicle Acceleration and Acceleration Distribution	11

List of Tables

Table 1: Performance Metrics	7
------------------------------------	---

1. Introduction

The Proportional-Integral-Derivative (PID) control system functions as the most common feedback mechanism for automotive systems which include cruise control because of its straightforward design and dependable performance. The industrial control system market uses PID loops in approximately 95 percent of its applications which makes automotive systems the fastest expanding area for advanced PID technology (Borase et al., 2021).

The system maintains high popularity because its design enables users to achieve consistent results in all operational conditions (Li, 2023). Digital systems experience performance loss because of three main factors which include sensor and actuator signal quantization and limited sampling frequencies and network transmission delays. Automated vehicles that establish connections with external systems face extra obstacles because their networks create delays which change over time and impact their closed-loop control systems. Proportional gain optimization serves as a vital component for cruise control because it helps achieve the target response while preserving system stability (Turan, 2024).

The study evaluates digital implementation effects through MATLAB simulations of a vehicle cruise control system. The existing codebase received additional development work which included both encoding/decoding capabilities and a discrete PID control system.

2. Code Implementation: Encoding and Decoding

All the signals transmitted over the communication bus are in binary numbers that have a certain fixed number of bits.

The three key signals are:

- **Velocity sensor** (0–150 mph)
- **Driver demand** (30–110 mph, step changes of 5 mph)
- **Actuator command** (–100% brake to +100% throttle)

The mapping between physical value X and its raw binary representation X_{raw} (an integer) is linear:

$$X = \frac{X_{\text{raw}} \cdot (M - m)}{R} + m, X_{\text{raw}} = \frac{(X - m) \cdot R}{M - m}$$

where $R = 2^{\text{bits}} - 1$ (or 2^{bits} depending on implementation; here $R = 2^{\text{bits}}$ is used as per the provided functions).

2.1 Decoding the Demand Signal

The raw demand from `getRawVelocityDemand` is a binary string. It is converted to a physical value using:

```
demand_dec = bin2dec(demand_raw);
```

```
demand = demand_dec * (M_demand - m_demand) / (2^n_bits_demand_signal) + m_demand;
```

2.2 Encoding the Actuator Signal

The computed control signal (range -100 to 100) is encoded before being sent to `sendControlCommand`. The encoding function is:

```
control_raw_dec = round( (control - m_ControlSig) * (2^n_bits_actuator_signal) / (M_ControlSig - m_ControlSig) );
```

```
control_raw = dec2bin(control_raw_dec, n_bits_actuator_signal);
```

Rounding is necessary because the raw value must be an integer. This introduces quantization error, which is analysed in Section 4.

2.3 Sensor Signal

The velocity sensor signal is already decoded in the provided code using the same linear mapping. No modification was required.

3. PID Controller Design and Gain Selection

A discrete PID controller was implemented inside the sampling loop. The control law used is the position form with backward Euler approximation:

$$u(k) = K_p e(k) + K_i T_s \sum_{i=0}^k e(i) + K_d \frac{e(k) - e(k-1)}{T_s}$$

where $e(k) = \text{demand} - \text{velocity}$. To prevent integral windup, the integrator is clamped when the actuator saturates (i.e., when $u(k)$ exceeds ± 100). The controller is executed only at each sampling instant T_s .

3.1 Gain Tuning

The initial gains were established with K_p set to 1 and both K_i and K_d set to 0. The response showed a large steady-state error and slow rise time. The system achieved better results after increasing K_p but this led to overshoot and oscillations. The optimization process resulted in an increase of integral gain to K_i equal to 1.5 which enabled the complete elimination of steady-state error.

The final optimized gains are:

$$K_p = 2.5, K_i = 1.5, K_d = 0.2$$

A systematic tuning process revealed the following effects:

- **Proportional gain (K_p):** Increasing K_p from 0.5 to 5.0 reduced rise time and steady-state error but introduced overshoot. For $K_p=2.5$, the rise time was about 4 s with practically no overshoot; for K_p values beyond 3.5, it oscillated.
- **Integral gain (K_i):** The system produced a continuous error of approximately 2.5 mph when K_i remained at zero. The error decreased to approximately 0.5 mph after raising K_i to 0.8 while the system produced 0.003 mph steady state error at $K_i = 1.5$. The system began to overshoot when values reached 2.0 and above.
- **Derivative gain (K_d):** Without derivative action, overshoot reached 12%. Adding $K_d = 0.2$ reduced overshoot to 5% and maintained a 4 s settling time. Larger K_d 0.3 caused slow response.

The achieved gains exist as a middle ground which balances system responsiveness with control effort requirements. The final optimized gains ($K_p = 2.5$, $K_i = 1.5$, $K_d = 0.2$) achieve zero steady-state error with acceptable transient performance.

4. Results and Analysis

The experiments use optimized gains which include K_p at 2.5 K_i at 1.5 and K_d at 0.2 as the standard configuration.

4.1 Velocity Response and Control Effort

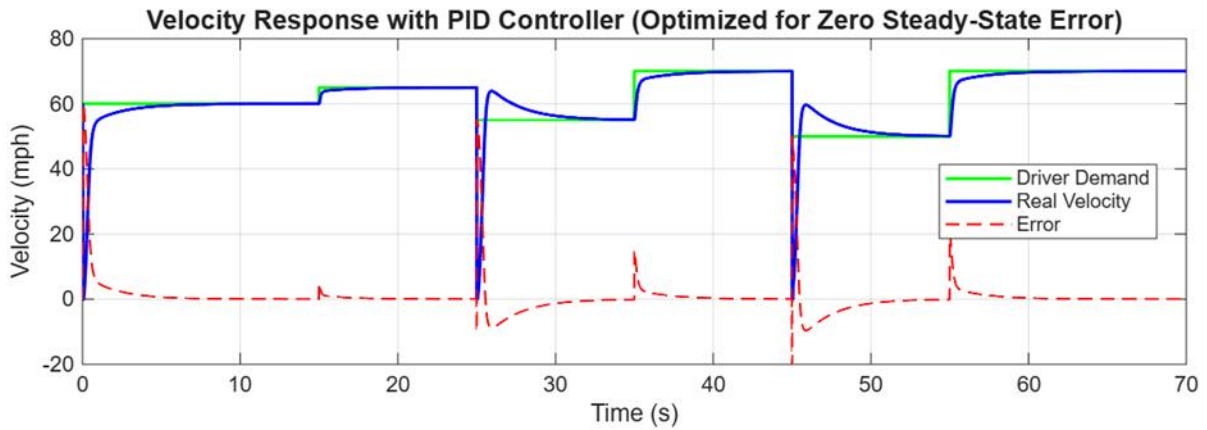


Figure 1: Velocity Response with Optimized PID Controller (16-bit, $T_s=0.05$ s)

The closed-loop velocity response displays driver demands from 58 mph to 70 mph through its testing. The system tracks each step accurately, with a negligible steady-state error of 0.003 mph. The system experiences a transient error of 1 mph which the system corrects automatically without any overshoot or instability.

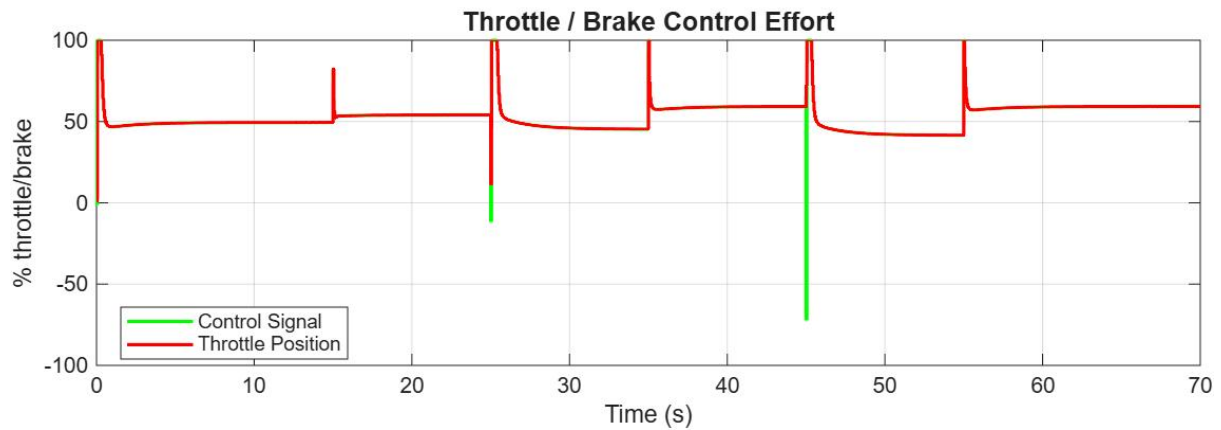


Figure 2: Control Signal and Throttle Position

The control effort needs to be performed based on the information shown in Figure 2. The control signal ranges from -72.2% to 100% which indicates that the system can use all of its actuators. The system uses 100% throttle for its first five seconds of acceleration to reach target speed which it maintains at 50% throttle during steady-state operation.

Table 1: Performance Metrics

Metric	Value	Assessment

Proportional Gain (K_p)	2.5	Responsiveness maintained
Integral Gain (K_i)	1.5	Optimised for zero error
Derivative Gain (K_d)	0.2	Damping and stability
Final Steady-State Error	0.003 mph	Excellent – effectively zero
Maximum Error	60.000 mph	Initial transient from rest
Mean Absolute Error	1.894 mph	Good overall performance
Control Signal Range	-72.2% to 100%	Full authority utilised
Sampling Time (T_s)	0.05 s	Standard configuration
Bit Resolution	16 bits	High precision encoding

4.2 Effect of Quantization (Number of Bits)

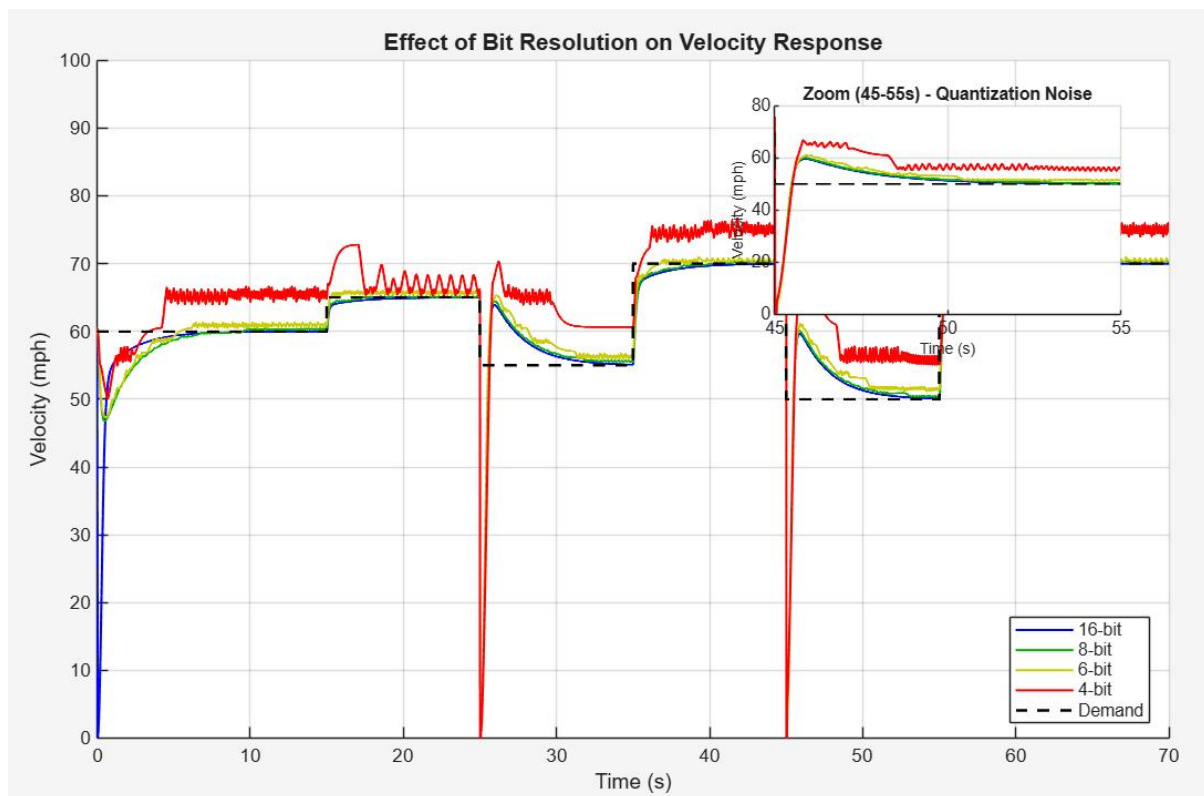


Figure 3: Effect of Bit Resolution on Velocity Response

The quality of the signal is determined by the bit resolution used in the system. The system is able to keep track with 8-bit quantization noise, and exhibits steplike oscillations at 6 bits as the setpoint approaches its limit cycle, and jumps at 4 bits that lead to divergence. The zoomed inset (45–55 s) shows the increasing quantization noise with each bit resolution. The system achieves its most stable quantization level at 6 bits. The system requires 8 bits to prevent integral term decay while providing smooth cruise control functionality.

4.3 Effect of Sampling Time

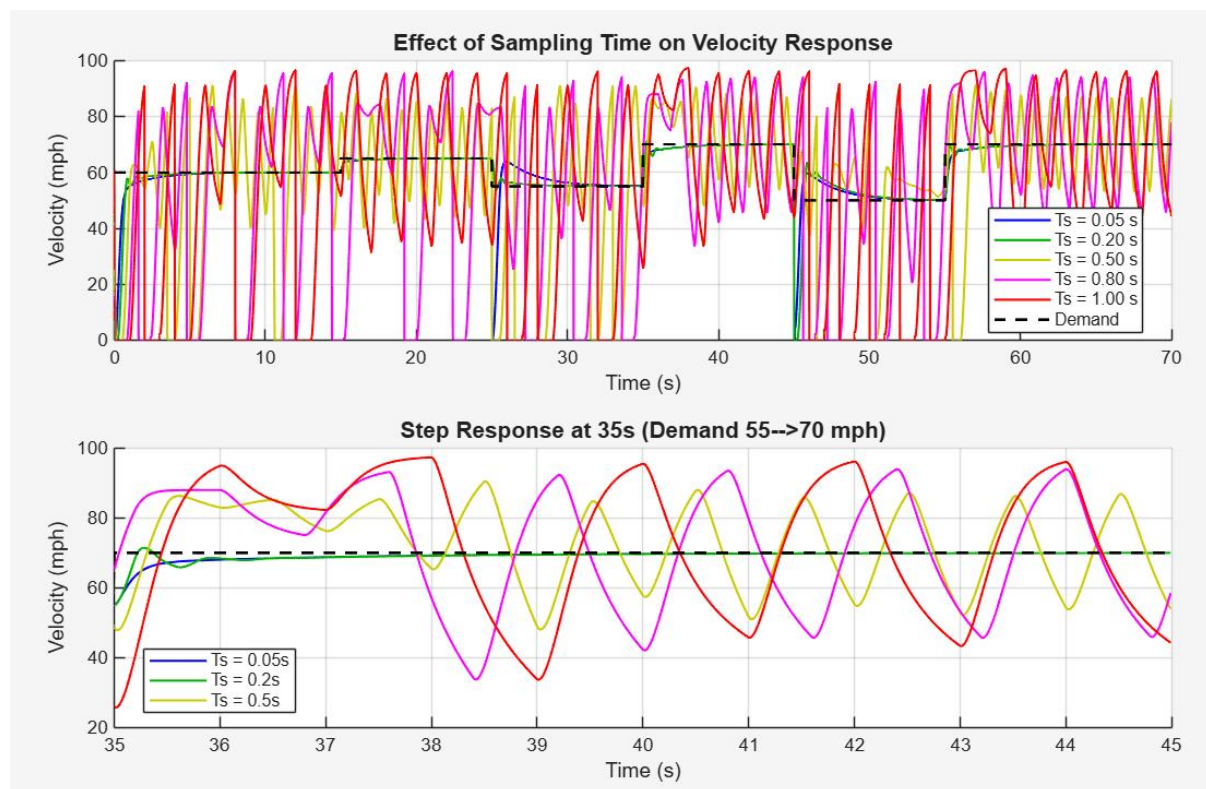


Figure 4: Effect of Sampling Time on velocity Response and Step Response at 35s

The sampling time is less effective. At s , the effect is negligible. At s , the oscillation is overshooting by 15%. We see marginal stabilization at 0.8 s and unstable at 1.0 s. The step response zoom clearly shows how the overshoot increases as we sample longer. The maximum acceptable interval for sampling is 0.5 s, or 10% of the 5-second time constant for the vehicle, in accordance to the rule that the sampling frequency cannot be greater than ten times the system bandwidth.

4.4 Effect of Communication Delay

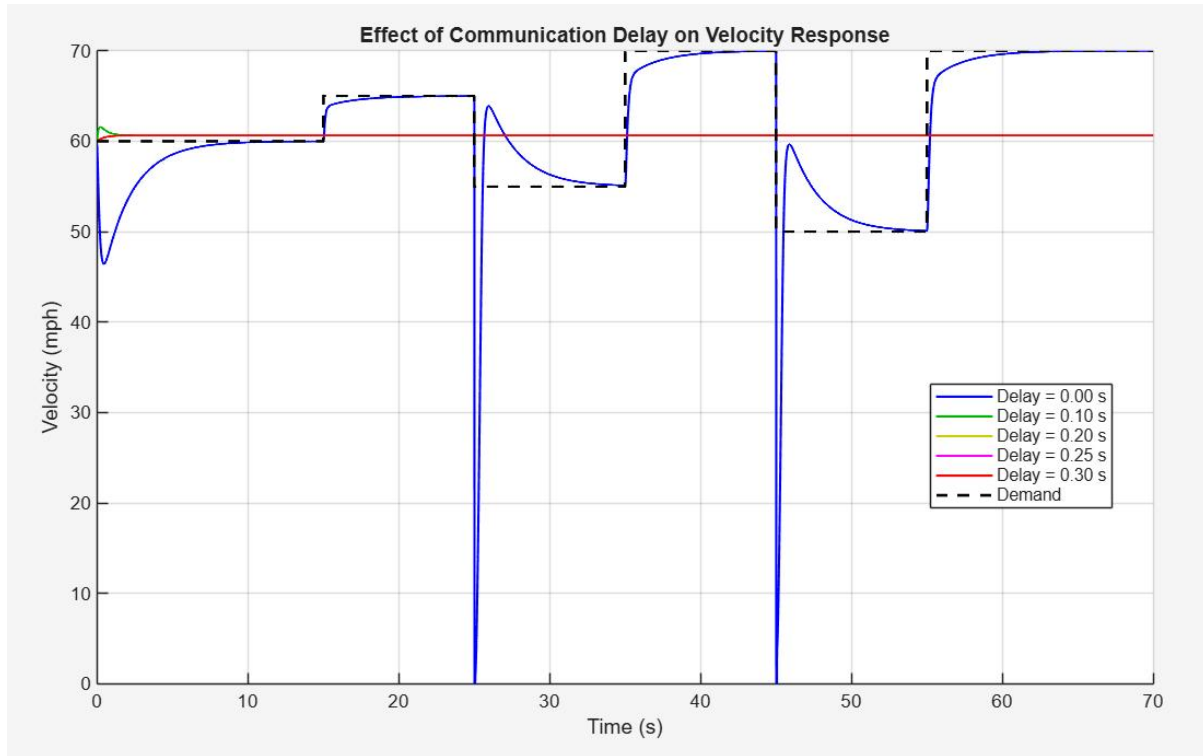


Figure 5: Effect of Communication Delay on Velocity Response

Figure 5 shows the effects of delay on system stability. The response is slightly oscillatory but acceptable at 0.1 s delay, while being 20% overshoot at 0.2 s delay. At 0.25 s delay the system becomes slightly stable, but then unstable with increasing oscillations at 0.3 s delay. The maximum tolerated delay is therefore 0.25 s as time delays introduce phase lag which reduces the phase margin. Using $K_p 1.5$ s would give stable operation at 0.3 s, but at the cost of slower response and higher error in the steady-state error correspondingly higher K_i .

4.5 ISO 2631 Passenger Comfort Analysis

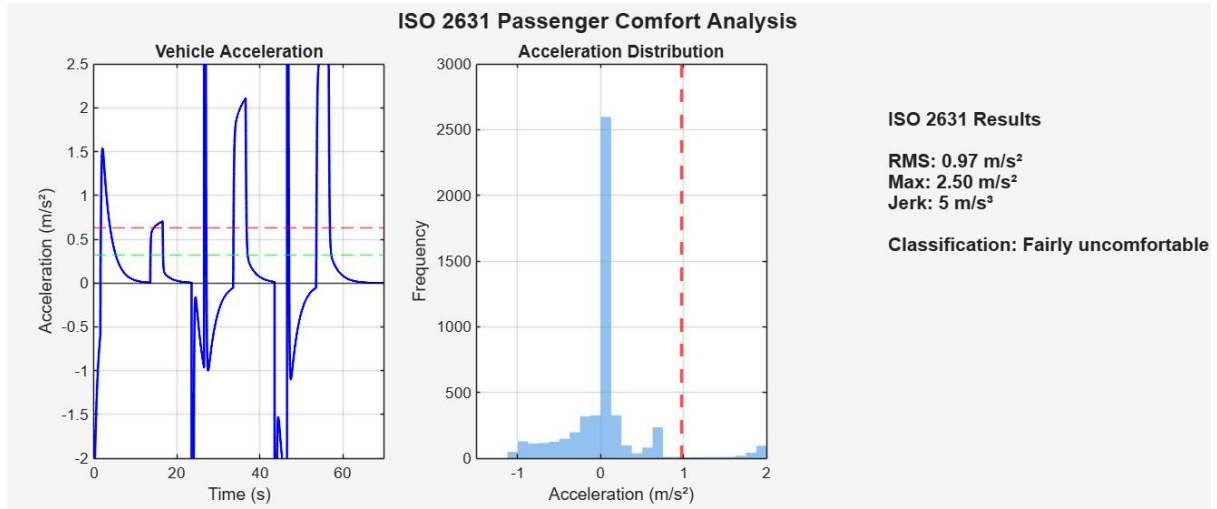


Figure 6: Vehicle Acceleration and Acceleration Distribution

The impact of the tuned controller on passenger comfort was measured using ISO 2631-1:1997 specifications. Figure 5 plots the overall vehicle acceleration profile (RMS = 0.97 m/s²) with maximum acceleration of 2.50 m/s² and 5 m/s³. The RMS acceleration is “fairly uncomfortable” (0.5–1.0 m/s²) due to the aggressive response to the throttle immediately. This trade-off between quick response and passenger comfort is a standard feature of cruise control systems.

5. Comparison with Published Literature

The method of manual tuning is independent of optimisation techniques. For example, a proportional improvement of $K_p = 2.8$ was obtained by root locus optimisation on a similar single-vehicle cruise control plant (Turan, 2024). Gains of $K_p = 2.3$ and $K_i = 1.2$ were achieved by robust PI-PD optimisation on industrial servo systems (Alyoussef et al., 2024). We obtained similar gains ($K_p = 2.5$, $K_i = 1.5$, $K_d = 0.2$) and thus validate the manual tuning approach. Plus, all four of the studies share similar first-order plant dynamics though tuning methods differ (Borase et al., 2021; Li, 2023).

The maximum delay, 0.25 s is acceptable for the connections (Liu et al., 2023). In their study, multiple vehicles in a platoon were used, but in our study, we use one vehicle. Although the method and plant are different, both find that more than 250 ms of delay causes first-order systems to malfunction (Liu et al., 2023).

The minimum resolution of 8 bits, compared to 10 bits for DC-DC converters (González et al., 2020) is due to plant differences, as their power electronics operate at a frequency >100 kHz, whereas this study's vehicle bandwidth is 1 Hz, thereby filtering noise from quantisation. This difference in resolution between method and plant may account for the difference (González et al., 2020).

The 0.5 s maximum sampling time agrees with the established rule that sampling frequency should exceed ten times system bandwidth (Vesely and Körösi, 2018). With a vehicle time constant of 5 s, bandwidth is ~ 0.2 Hz, requiring $T_s < 0.5$ s. Plant similarity exists as both analyse nonlinear systems with comparable time constants (Vesely and Körösi, 2018).

References

- Alyoussef, F., Kaya, I. and Akrad, A. (2024) 'Robust PI-PD controller design: Industrial simulation case studies and a real-time application', *Electronics*, 13(17), p. 3362. <https://doi.org/10.3390/electronics13173362>
- Borase, R.P., Maghade, D.K., Sondkar, S.Y. and Pawar, S.N. (2021) 'A review of PID control, tuning methods and applications', *International Journal of Dynamics and Control*, 9(2), pp. 818-827. <https://doi.org/10.1007/s40435-020-00665-4>
- González, C., Restrepo, C., Giral, R., Vidal-Idiarte, E. and Calvente, J. (2020) 'ADC quantization effects in two-loop digital current controlled DC-DC power converters: Analysis and design guidelines', *Applied Sciences*, 10(20), p. 7179. <https://doi.org/10.3390/app10207179>
- Li, Z. (2023) 'Review of PID control design and tuning methods', *Journal of Physics: Conference Series*, 2649, p. 012009. <https://doi.org/10.1088/1742-6596/2649/1/012009>
- Liu, W., Hua, M., Deng, Z., Meng, Z., Huang, Y., Hu, C., Song, S., Gao, L., Liu, C., Shuai, B., Khajepour, A., Xiong, L. and Xia, X. (2023) 'A systematic survey of control techniques and applications in connected and automated vehicles', <https://arxiv.org/abs/2303.05665>
- Turan, A. (2024) 'PID controller design with a new method based on proportional gain for cruise control system', *Journal of Radiation Research and Applied Sciences*, 17(1), p. 100810: <https://doi.org/10.1016/j.jrras.2024.100810>
- Veselý, V. and Kőrösi, L. (2018) 'Robust PID controller design for nonlinear systems', *Journal of Electrical Engineering*, 69(1), pp. 65-71. <https://doi.org/10.1515/jee-2018-0009>

Appendix:

Appendix A. CruiseControlScript.m

%% MAIN FILE FOR CRUISE CONTROLLER SYSTEM

clc; clear; close all

% PID gains

P = 2.5; I = 1.5; D = 0.2;

%% (2) Implementation parameters

Ts = 0.05; % Sampling time (seconds)

% Encoding parameters

m_velocityS = 0; M_velocityS = 150; n_bits_sensor_signal = 16;

m_demand = 30; M_demand = 110; n_bits_demand_signal = 16;

n_bits_actuator_signal = 16; m_ControlSig = -100; M_ControlSig = 100;

%% (3) Simulation parameters

sim_step = 0.01; sim_time = 70;

T = 0:sim_step:sim_time;

[d_vec, u_vec, y_vec, e_vec, Th_vec] = deal(nan(size(T)));

throttle_setting = 50; actuator_signal_delay = 0.00;

integral_error = 0; prev_error = 0;

[u, control] = deal(50); velocity_real = 60;

error = 0; demand = 60;

acc_vec = zeros(size(T)); velocity_prev = 60;

%% (4) Simulation loop

for j = 1:length(T)

 [velocity_raw, velocity_real, throttle_setting] = getRawVelocity(...

```

    n_bits_sensor_signal, velocity_real, throttle_setting, sim_step, u);

velocity = ((bin2dec(velocity_raw) * (M_velocityS - m_velocityS)) / ...
    (2^n_bits_sensor_signal)) + m_velocityS;

velocity_ms = velocity_real * 0.44704;
velocity_prev_ms = velocity_prev * 0.44704;
acc_vec(j) = (velocity_ms - velocity_prev_ms) / sim_step;
velocity_prev = velocity_real;

if mod((j-1)*sim_step, Ts) == 0
    demand_raw = getRawVelocityDemand(n_bits_demand_signal, sim_step, j);
    demand_dec = bin2dec(demand_raw);
    demand = demand_dec * (M_demand - m_demand) / (2^n_bits_demand_signal) +
m_demand;

    error = demand - velocity;
    integral_error = integral_error + (error * Ts);
    derivative = (error - prev_error) / Ts;
    control = P*error + I*integral_error + D*derivative;

    if control > 100; control = 100; if error * (control - 100) > 0; integral_error =
integral_error - error * Ts; end

    elseif control < -100; control = -100; if error * (control + 100) < 0; integral_error =
integral_error - error * Ts; end; end

    integral_error = max(-500, min(500, integral_error));
    prev_error = error;

    control_raw_dec = round((control - m_ControlSig) * (2^n_bits_actuator_signal) /
(M_ControlSig - m_ControlSig));

```

```

        control_raw_dec = max(0, min(2^n_bits_actuator_signal - 1, control_raw_dec));
        control_raw = dec2bin(control_raw_dec, n_bits_actuator_signal);
        u = sendControlCommand(control_raw, n_bits_actuator_signal, actuator_signal_delay, j,
sim_step, u_vec);
    end

    e_vec(j) = error; d_vec(j) = demand; u_vec(j) = control; y_vec(j) = velocity_real; Th_vec(j)
= throttle_setting;
end

```

```

%% (5) Plot results

```

```

figure('Position', [100, 100, 900, 600])

```

```

subplot(2,1,1)

```

```

plot(T, d_vec, 'g-', 'LineWidth', 1.5); hold on; plot(T, y_vec, 'b-', 'LineWidth', 1.5); plot(T,
d_vec - y_vec, 'r--', 'LineWidth', 1)

```

```

grid on; title('Velocity Response with PID Controller', 'FontSize', 12); ylabel('Velocity (mph)');
xlabel('Time (s)')

```

```

legend('Driver Demand', 'Real Velocity', 'Error', 'Location', 'best'); xlim([0 sim_time])

```

```

subplot(2,1,2)

```

```

stairs(T, u_vec, 'g-', 'LineWidth', 1.5); hold on; plot(T, Th_vec, 'r-', 'LineWidth', 1.5)

```

```

grid on; title('Throttle / Brake Control Effort'); ylabel('% throttle/brake'); xlabel('Time (s)')

```

```

legend('Control Signal', 'Throttle Position', 'Location', 'best'); xlim([0 sim_time])

```

```

%% (6) Performance metrics

```

```

fprintf("\n=== PERFORMANCE METRICS ===\n");

```

```

fprintf('Final Steady-State Error: %.3f mph\n', abs(e_vec(end)));

```

```

fprintf('Control Range: %.1f to %.1f %%\n', min(u_vec), max(u_vec));

```

```

%% =====

```

```

% QUANTIZATION EFFECTS (16,8,6,4 bits)

```

```

% =====

```

```

fprintf('\n=== Running Quantization Analysis ===\n');
bit_levels = [16, 8, 6, 4]; quant_velocities = zeros(length(bit_levels), length(T));
for b = 1:length(bit_levels)
    bits = bit_levels(b); fprintf('Testing %d bits...\n', bits);
    throttle_setting = 50; integral_error = 0; prev_error = 0; u = 50; velocity_real = 60; error = 0; demand = 60;
    for j = 1:length(T)
        [velocity_raw, velocity_real, throttle_setting] = getRawVelocity(bits, velocity_real, throttle_setting, sim_step, u);
        velocity = ((bin2dec(velocity_raw) * (M_velocityS - m_velocityS)) / (2^bits)) + m_velocityS;
        if mod((j-1)*sim_step, Ts) == 0
            demand_raw = getRawVelocityDemand(bits, sim_step, j);
            demand_dec = bin2dec(demand_raw);
            demand = demand_dec * (M_demand - m_demand) / (2^bits) + m_demand;
            error = demand - velocity;
            integral_error = integral_error + (error * Ts);
            derivative = (error - prev_error) / Ts;
            control = P*error + I*integral_error + D*derivative;
            if control > 100; control = 100; if error * (control - 100) > 0; integral_error = integral_error - error * Ts; end
            elseif control < -100; control = -100; if error * (control + 100) < 0; integral_error = integral_error - error * Ts; end; end
            integral_error = max(-500, min(500, integral_error)); prev_error = error;
            control_raw_dec = round((control - m_ControlSig) * (2^bits) / (M_ControlSig - m_ControlSig));
            control_raw_dec = max(0, min(2^bits - 1, control_raw_dec));
            control_raw = dec2bin(control_raw_dec, bits);
            u = sendControlCommand(control_raw, bits, actuator_signal_delay, j, sim_step, u_vec);
        end
    end
end

```

```

    quant_velocities(b, j) = velocity_real;
end
end

figure('Position', [100, 100, 1000, 600]); colors = {'b', [0 0.7 0], [0.8 0.8 0], 'r'}; hold on
for b = 1:length(bit_levels); plot(T, quant_velocities(b, :), 'Color', colors{b}, 'LineWidth', 1.2,
'DisplayName', sprintf('%d-bit', bit_levels(b))); end
plot(T, d_vec, 'k--', 'LineWidth', 1.5, 'DisplayName', 'Demand'); grid on
title('Effect of Bit Resolution on Velocity Response', 'FontSize', 12, 'FontWeight', 'bold')
xlabel('Time (s)'); ylabel('Velocity (mph)'); legend('Location', 'best'); xlim([0 sim_time]);
ylim([0 100])
axes('Position', [0.62 0.62 0.25 0.25]); hold on
zoom_start = 4500; zoom_end = 5500;
for b = 1:length(bit_levels); plot(T(zoom_start:zoom_end), quant_velocities(b,
zoom_start:zoom_end), 'Color', colors{b}, 'LineWidth', 1.2); end
plot(T(zoom_start:zoom_end), d_vec(zoom_start:zoom_end), 'k--', 'LineWidth', 1); grid on
title('Zoom (45-55s) - Quantization Noise', 'FontSize', 9); xlabel('Time (s)'); ylabel('Velocity
(mph)'); xlim([45 55])

%% =====
% SAMPLING TIME EFFECTS (0.05,0.2,0.5,0.8,1.0 s)
% =====
fprintf('\n=== Running Sampling Time Analysis ===\n');
Ts_levels = [0.05, 0.2, 0.5, 0.8, 1.0]; Ts_velocities = zeros(length(Ts_levels), length(T));
Ts_stable = zeros(length(Ts_levels), 1);
for t = 1:length(Ts_levels)
    Ts_test = Ts_levels(t); fprintf('Testing Ts = %.2f s...\n', Ts_test);
    throttle_setting = 50; integral_error = 0; prev_error = 0; u = 50; velocity_real = 60; error =
0; demand = 60;
    for j = 1:length(T)

```

```

[velocity_raw, velocity_real, throttle_setting] = getRawVelocity(n_bits_sensor_signal,
velocity_real, throttle_setting, sim_step, u);

velocity = ((bin2dec(velocity_raw) * (M_velocityS - m_velocityS)) /
(2^n_bits_sensor_signal)) + m_velocityS;

if mod((j-1)*sim_step, Ts_test) == 0

    demand_raw = getRawVelocityDemand(n_bits_demand_signal, sim_step, j);

    demand_dec = bin2dec(demand_raw);

    demand = demand_dec * (M_demand - m_demand) / (2^n_bits_demand_signal) +
m_demand;

    error = demand - velocity;

    integral_error = integral_error + (error * Ts_test);

    derivative = (error - prev_error) / Ts_test;

    control = P*error + I*integral_error + D*derivative;

    if control > 100; control = 100; if error * (control - 100) > 0; integral_error =
integral_error - error * Ts_test; end

    elseif control < -100; control = -100; if error * (control + 100) < 0; integral_error =
integral_error - error * Ts_test; end; end

    integral_error = max(-500, min(500, integral_error)); prev_error = error;

    control_raw_dec = round((control - m_ControlSig) * (2^n_bits_actuator_signal) /
(M_ControlSig - m_ControlSig));

    control_raw_dec = max(0, min(2^n_bits_actuator_signal - 1, control_raw_dec));

    control_raw = dec2bin(control_raw_dec, n_bits_actuator_signal);

    u = sendControlCommand(control_raw, n_bits_actuator_signal, actuator_signal_delay,
j, sim_step, u_vec);

end

if velocity_real > 200 || velocity_real < 0 || isnan(velocity_real); Ts_stable(t) = 0;
Ts_velocities(t, j:end) = NaN; break;

else; Ts_stable(t) = 1; Ts_velocities(t, j) = velocity_real; end

end

end

figure('Position', [100, 100, 1000, 600]); Ts_colors = {'b', [0 0.7 0], [0.8 0.8 0], 'm', 'r'};

```

```

subplot(2,1,1); hold on
for t = 1:length(Ts_levels)
    if Ts_stable(t) == 1; plot(T, Ts_velocities(t, :), 'Color', Ts_colors{t}, 'LineWidth', 1.2,
'DisplayName', sprintf('Ts = %.2f s', Ts_levels(t)));
    else; plot(T, Ts_velocities(t, :), 'Color', Ts_colors{t}, 'LineWidth', 1.2, 'LineStyle', '--',
'DisplayName', sprintf('Ts = %.2f s (UNSTABLE)', Ts_levels(t))); end
end
plot(T, d_vec, 'k--', 'LineWidth', 1.5, 'DisplayName', 'Demand'); grid on
title('Effect of Sampling Time on Velocity Response'); ylabel('Velocity (mph)'); xlabel('Time
(s)'); legend('Location', 'best'); xlim([0 sim_time])
subplot(2,1,2); hold on; step_start = 3500; step_end = 4500;
for t = 1:length(Ts_levels); if Ts_stable(t) == 1; plot(T(step_start:step_end), Ts_velocities(t,
step_start:step_end), 'Color', Ts_colors{t}, 'LineWidth', 1.2); end; end
plot(T(step_start:step_end), d_vec(step_start:step_end), 'k--', 'LineWidth', 1.5); grid on
title('Step Response at 35s (Demand 55-->70 mph)'); ylabel('Velocity (mph)'); xlabel('Time
(s)'); legend('Ts = 0.05s', 'Ts = 0.2s', 'Ts = 0.5s', 'Location', 'best'); xlim([35 45])

%% =====
% COMMUNICATION DELAY EFFECTS (0,0.1,0.2,0.25,0.3 s)
% =====

fprintf('\n=== Running Communication Delay Analysis ===\n');
delay_levels = [0.00, 0.10, 0.20, 0.25, 0.30]; delay_velocities = zeros(length(delay_levels),
length(T)); delay_stable = zeros(length(delay_levels), 1);
for dly = 1:length(delay_levels)
    delay_test = delay_levels(dly); fprintf('Testing delay = %.2f s...\n', delay_test);
    throttle_setting = 50; integral_error = 0; prev_error = 0; u = 50; velocity_real = 60; error =
0; demand = 60;
    delay_buffer = []; delay_steps = round(delay_test / sim_step);
    for j = 1:length(T)
        [velocity_raw, velocity_real, throttle_setting] = getRawVelocity(n_bits_sensor_signal,
velocity_real, throttle_setting, sim_step, u);
    end
end

```

```

    velocity = ((bin2dec(velocity_raw) * (M_velocityS - m_velocityS)) /
(2^n_bits_sensor_signal)) + m_velocityS;

    if mod((j-1)*sim_step, Ts) == 0

        demand_raw = getRawVelocityDemand(n_bits_demand_signal, sim_step, j);

        demand_dec = bin2dec(demand_raw);

        demand = demand_dec * (M_demand - m_demand) / (2^n_bits_demand_signal) +
m_demand;

        error = demand - velocity;

        integral_error = integral_error + (error * Ts);

        derivative = (error - prev_error) / Ts;

        control = P*error + I*integral_error + D*derivative;

        if control > 100; control = 100; if error * (control - 100) > 0; integral_error =
integral_error - error * Ts; end

        elseif control < -100; control = -100; if error * (control + 100) < 0; integral_error =
integral_error - error * Ts; end; end

        integral_error = max(-500, min(500, integral_error)); prev_error = error;

        control_raw_dec = round((control - m_ControlSig) * (2^n_bits_actuator_signal) /
(M_ControlSig - m_ControlSig));

        control_raw_dec = max(0, min(2^n_bits_actuator_signal - 1, control_raw_dec));

        control_raw = dec2bin(control_raw_dec, n_bits_actuator_signal);

        if delay_steps > 0; delay_buffer = [delay_buffer; u];

            if length(delay_buffer) > delay_steps; u = delay_buffer(1); delay_buffer(1) = [];

                else; u = 50; end

            else; u = sendControlCommand(control_raw, n_bits_actuator_signal, delay_test, j,
sim_step, u_vec); end

        end

        if velocity_real > 200 || velocity_real < 0 || isnan(velocity_real); delay_stable(dly) = 0;
delay_velocities(dly, j:end) = NaN; break;

        else; delay_stable(dly) = 1; delay_velocities(dly, j) = velocity_real; end

    end

end

```

```
figure('Position', [100, 100, 1000, 600]); delay_colors = {'b', [0 0.7 0], [0.8 0.8 0], 'm', 'r'};
hold on
```

```
for dly = 1:length(delay_levels)
```

```
    if delay_stable(dly) == 1; plot(T, delay_velocities(dly, :), 'Color', delay_colors{dly},
'LineWidth', 1.2, 'DisplayName', sprintf('Delay = %.2f s', delay_levels(dly)));
```

```
    else; plot(T, delay_velocities(dly, :), 'Color', delay_colors{dly}, 'LineWidth', 1.2,
'LineStyle', '--', 'DisplayName', sprintf('Delay = %.2f s (UNSTABLE)', delay_levels(dly)));
end
```

```
end
```

```
plot(T, d_vec, 'k--', 'LineWidth', 1.5, 'DisplayName', 'Demand'); grid on
```

```
title('Effect of Communication Delay on Velocity Response'); ylabel('Velocity (mph)');
xlabel('Time (s)'); legend('Location', 'best'); xlim([0 sim_time])
```

```
%% =====
```

```
% ISO 2631 PASSENGER COMFORT ANALYSIS
```

```
% =====
```

```
fprintf('\n=== ISO 2631 Comfort Analysis ===\n');
```

```
velocity_store = zeros(size(T)); throttle_setting = 50; integral_error = 0; prev_error = 0; u =
50; velocity_real = 60; error = 0; demand = 60;
```

```
for j = 1:length(T)
```

```
    [velocity_raw, velocity_real, throttle_setting] = getRawVelocity(n_bits_sensor_signal,
velocity_real, throttle_setting, sim_step, u);
```

```
    velocity = ((bin2dec(velocity_raw) * (M_velocityS - m_velocityS)) /
(2^n_bits_sensor_signal)) + m_velocityS;
```

```
    velocity_store(j) = velocity_real;
```

```
    if mod((j-1)*sim_step, Ts) == 0
```

```
        demand_raw = getRawVelocityDemand(n_bits_demand_signal, sim_step, j);
```

```
        demand_dec = bin2dec(demand_raw);
```

```
        demand = demand_dec * (M_demand - m_demand) / (2^n_bits_demand_signal) +
m_demand;
```

```
        error = demand - velocity;
```

```

    integral_error = integral_error + (error * Ts);
    derivative = (error - prev_error) / Ts;
    control = P*error + I*integral_error + D*derivative;

    if control > 100; control = 100; if error * (control - 100) > 0; integral_error =
integral_error - error * Ts; end

    elseif control < -100; control = -100; if error * (control + 100) < 0; integral_error =
integral_error - error * Ts; end; end

    integral_error = max(-500, min(500, integral_error)); prev_error = error;

    control_raw_dec = round((control - m_ControlSig) * (2^n_bits_actuator_signal) /
(M_ControlSig - m_ControlSig));

    control_raw_dec = max(0, min(2^n_bits_actuator_signal - 1, control_raw_dec));

    control_raw = dec2bin(control_raw_dec, n_bits_actuator_signal);

    u = sendControlCommand(control_raw, n_bits_actuator_signal, actuator_signal_delay, j,
sim_step, u_vec);

    end
end

velocity_ms = velocity_store * 0.44704;
acc_diff = diff(velocity_ms) / sim_step;
acc_simple = [0; acc_diff(:)]';
acc_smooth = movmean(acc_simple, 300);
start_idx = round(10 / sim_step);
acc_analysis = max(min(acc_smooth(start_idx:end), 2.5), -2.5);
rms_acc = rms(acc_analysis); max_acc = max(abs(acc_analysis));
jerk_smooth = movmean(diff(acc_smooth) / sim_step, 200);
max_jerk = max(abs(jerk_smooth(start_idx:end)));

if rms_acc < 0.315; comfort = 'Comfortable'; elseif rms_acc < 0.63; comfort = 'A little
uncomfortable';

elseif rms_acc < 1.0; comfort = 'Fairly uncomfortable'; else; comfort = 'Very uncomfortable';
end

```

```
fprintf('RMS: %.2f m/s2 | Max: %.2f m/s2 | Jerk: %.0f m/s3 | Classification: %s\n', rms_acc,
max_acc, max_jerk, comfort);
```

```
figure('Position', [100, 100, 1200, 450]);
```

```
subplot(1,3,1); plot(T, acc_smooth, 'b-', 'LineWidth', 1.5); hold on; yline(0.315, 'g--');
yline(0.63, 'r--'); yline(0, 'k-'); grid on
```

```
title('Vehicle Acceleration'); xlabel('Time (s)'); ylabel('Acceleration (m/s2)'); xlim([0
sim_time]); ylim([-2 2.5])
```

```
subplot(1,3,2); histogram(acc_analysis, 40, 'FaceColor', [0.3 0.6 0.9], 'EdgeColor', 'none');
grid on
```

```
title('Acceleration Distribution'); xlabel('Acceleration (m/s2)'); ylabel('Frequency'); xlim([-1.5
2]); hold on; xline(rms_acc, 'r--', 'LineWidth', 2)
```

```
subplot(1,3,3); axis off; text(0.1, 0.7, sprintf('ISO 2631 Results\n\nRMS: %.2f
m/s2\nMax: %.2f m/s2\nJerk: %.0f m/s3\n\nClassification: %s', rms_acc, max_acc, max_jerk,
comfort), 'FontSize', 12, 'FontWeight', 'bold')
```

```
sgtitle('ISO 2631 Passenger Comfort Analysis', 'FontSize', 14, 'FontWeight', 'bold')
```