

University of Wales Trinity Saint David

Wales Institute of Science and Art



Prifysgol Cymru

Y Drindod Dewi Sant

University of Wales

Trinity Saint David

Modelling Industrial Control Systems in MATLAB

Control Technology

ACMA6001

Jordan Hessey

2315713

03/08/2026

Contents

Table of Tables	2
Table of Figures.....	2
1 Introduction	3
1.1 Aims	3
1.2 Objectives	3
2 Background	4
2.1 PID general background	4
2.2 Cruise control standards.....	4
3 Encoding and Decoding Signals.....	5
3.1 Encoding code	5
3.2 Decoding code	5
4 PID tuning.....	7
4.1 Method	7
4.2 Results and analysis.....	7
5 Time step impact.....	14
5.1 Method	14
5.2 Results and analysis.....	14
6 Signal delay impact	16
6.1 Method	16
6.2 Results and analysis.....	16
7 Number of Bits impact	18
7.1 Method	18
7.2 Results and analysis.....	18
8 Conclusion.....	20
9 References	21
10 Appendices.....	22
10.1 Controller function: Controllerrrss.m.....	22
10.2 Simulation script: CruiseControlScript_pract.m	23
10.3 Vehicle model and sensor: getRawVelocity.m	26
10.4 Demand encoder: getRawVelocityDemand.m	27
10.5 Actuator decoder: sendControlCommand.m	27

Table of Tables

Table 1 - Overview of controller results for set gains	7
--	---

Table of Figures

Figure 1 - Script conversion control output from decimal to binary	5
Figure 2 - Script conversion from raw binary into decimal (Mph).....	5
Figure 3 - Testing graph outputs are correct before adding PID controls	6
Figure 4 - Proportional run 1, Kp set to 4	8
Figure 5 - Proportional run 2, Kp set to 8	8
Figure 6 - Proportional run 3, Kp set to 100	9
Figure 7 - Proportional & Integral run 1, Kp set to 4, Ki set to 1.....	10
Figure 8 - Proportional & Integral run 2, Kp set to 2, Ki set to 0.9.....	10
Figure 9 - Proportional & Integral run 2, Kp set to 4, Ki set to 10.....	11
Figure 10 - Proportional, Integral & Derivative run 1, Kp set to 4, Ki set to 1, D set to 0.3.....	12
Figure 11 - Proportional, Integral & Derivative run 2, Kp set to 4, Ki set to 1, D set to 2.....	12
Figure 12 - Proportional, Integral & Derivative run 3, Kp set to 4, Ki set to 1, D set to 5.....	13
Figure 13 - Sampling Time variable baseline	14
Figure 14 - Sampling Time variable set to 0.2 seconds.....	15
Figure 15 - Sampling Time variable set to 0.4 seconds.....	15
Figure 16 - Signal delay variable baseline	16
Figure 17 - Signal delay variable set to 0.25 seconds	17
Figure 18 - Signal delay variable set to 0.5 seconds	17
Figure 19 - Number of bits variable baseline	18
Figure 20 - Number of bits variable set to 8	19
Figure 21 - Number of bits variable set to 4	19

1 Introduction

1.1 Aims

This report aims to develop and evaluate a MATLAB model of a closed-loop cruise-control system. A digital proportional-integral-derivative (PID) controller regulates vehicle speed while signal encoding, sampling interval, network delay and numerical resolution are investigated. The design is assessed using explicit engineering criteria and the trade-off between velocity response and control effort.

1.2 Objectives

- Implement the supplied vehicle model and PID function
- Encode and decode each digital signal
- Tune Proportional (K_p), Integral (K_i) and Derivative (K_d) systematically
- Compare velocity response with actuator demand
- Determine how sampling time, delay and bit depth affect performance
- Test whether retuning increases delay tolerance
- Identify practical implementation limits against ISO standards

2 Background

2.1 PID general background

PID controllers are widely used in industrial process', motor and manufacturing systems because they are simple, computationally efficient, easily implemented and capable of providing robust performance without requiring a highly detailed system model (Åström and Hägglund 2001; National Instruments 2025).

PID control combines proportional, integral and derivative responses to the error between demanded and measured output. Proportional action responds to present error, integral action removes steady-state offset, and derivative action improves damping. Depending on the application, P, PI, PD or full PID configurations may be selected.

2.2 Cruise control standards

Cruise control systems must comply with multiple ISO standards, but ISO 15622 specifies functionality, diagnostics and performance tests (ISO 2018a; ISO 2018b; ISO 2022). Key aspects within this standard defines acceptable limits for the likes of acceleration, jerk and driver override behaviour to ensure passenger comfort and safety. (Mustafar et al. 2020) evaluated PI-controlled adaptive cruise control against ISO acceleration, deceleration and jerk limits, demonstrating that gains and controller outputs must be constrained to maintain compliant tracking performance and passenger comfort.

3 Encoding and Decoding Signals

Before being able to use the raw data for sensors, demands and actuator signals within the vehicle model, the data must be firstly processed into a manageable data set.

- Velocity sensor 0-150 mph
- Driver demand 30-110 mph in steps of 5mph
- Actuator
 - Braking -100% (m)
 - Throttle +100% (M).
- R (16 bits) $2^{16} = 65,536$

3.1 Encoding code

With the controller outputting throttle/brake through the communication bus in binary numbers, it's required to encode the signal into a binary number. This results in the values turning from -100 to +100 into 0 to 200. This is achieved by using the below formula and using dec2bin within the script to convert from decimal to binary. See Figure 1.

$$X_{Raw} = \frac{(X_{Physical} - m) \times R}{M - m}$$

```
%% encode control signal into x-bit binary number
control_raw = dec2bin((control-m_ControlSig).*R./(M_ControlSig-m_ControlSig));
```

Figure 1 - Script conversion control output from decimal to binary

3.2 Decoding code

The driver demand signal is obtained from the communication bus as a raw binary string. This in turn represents a value in the range 30 to 110 mph, which is encoded across $2^{16} = 65,536$ quantisation levels. Converting to a usable physical value is achieved within MATLAB using bin2dec function, binary to decimal integer. Moving on to being multiplied by the signal span ($110 - 30 = 80$ mph), then, divided by 65,536 levels. Finally, the range minimum added for true demanded speed. See Figure 2 for equation.

$$X_{Physical} = \frac{X_{Raw} \times (M - m)}{R} + m$$

```
%% decode raw signal (from x-bit binary to mph)
demand = (bin2dec(demand_raw).*(M_demand-m_demand)./R)+m_demand;
```

Figure 2 - Script conversion from raw binary into decimal (Mph)

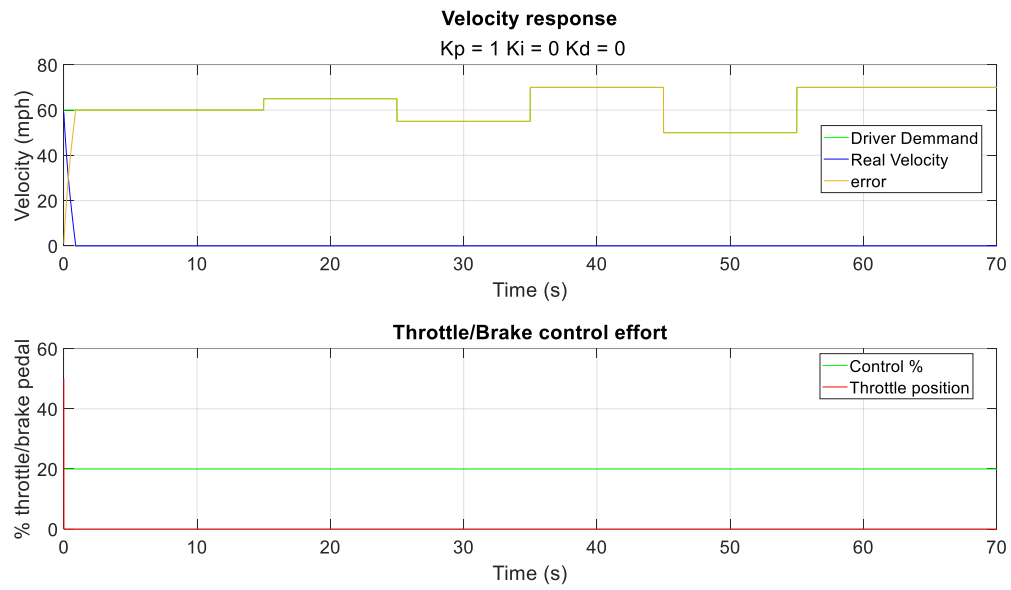


Figure 3 - Testing graph outputs are correct before adding PID controls

4 PID tuning

4.1 Method

A staged testing approach was adopted, proportional term firstly evaluated independently, followed by a proportional–integral controller and, finally, the complete proportional–integral–derivative controller. This approach enables the influence of each control term on the closed-loop response to be identified clearly.

Each of the PID gains were tuned manually against a stepped demand profile of 5mph increments over 70 seconds using 16-bit signals with a sampling time of $T_s = 0.05$ and zero delay.

Further testing will investigate how changes in sampling time, communication delay and signal bit depth affect the overall performance and robustness of the system.

4.2 Results and analysis

Table 1 gives an overall summary of the results seen from the tests carried out for the various configurations used. Each test run was against the full 70 second demand and assessed based on steady-state error, settling time and peak effort.

Table 1 - Overview of controller results for set gains

Controller	Run	Controller gains (Kp, Ki, Kd)	Steady-state error (mph)	Settling time (s)	Peak effort (%)
P	1	4, 0, 0	~ 10 mph	~ 1 s	100%
P	2	8, 0, 0	~ 5 mph when stable	Did not settle	100%
P	3	100, 0, 0	No valid steady state	Did not settle	100%
PI	1	4, 1, 0	~ 0–1 mph	~ 1 s	100%
PI	2	2, 0.9, 0	~ 2–5 mph	~ 1–2 s	~ 75%
PI	3	4, 10, 0	~ 30–50 mph	Did not settle	100%
PID	1	4, 1, 0.3	~ 0–1 mph	Less than 1 s	100%
PID	2	4, 1, 2	~ 0–1 mph	~ 2–3 s	100%
PID	3	4, 1, 5	No valid steady state	Did not settle	100%

4.2.1 Proportional

The first proportional run (Figure 4) remained stable and followed each demand change but retained an approximately 10 mph steady-state error. Increasing the proportional gain reduced this offset and accelerated the response, however, the second run (Figure 5) developed sustained oscillations and repeated actuator saturation during lower-demand intervals. The highest gain run (Figure 6) was unstable throughout, with rapid velocity cycling and control switching between its limits.

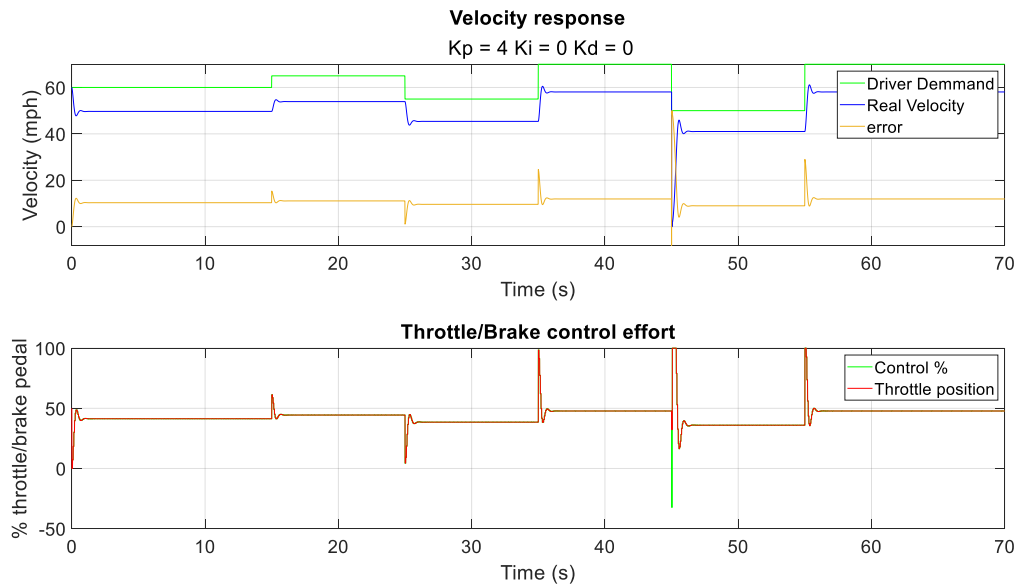


Figure 4 - Proportional run 1, K_p set to 4

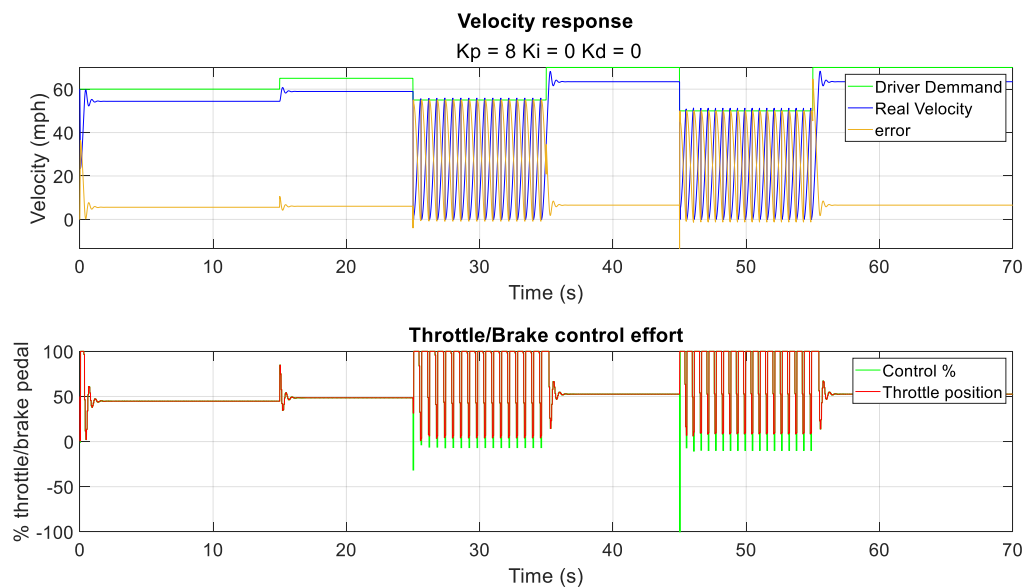


Figure 5 - Proportional run 2, K_p set to 8

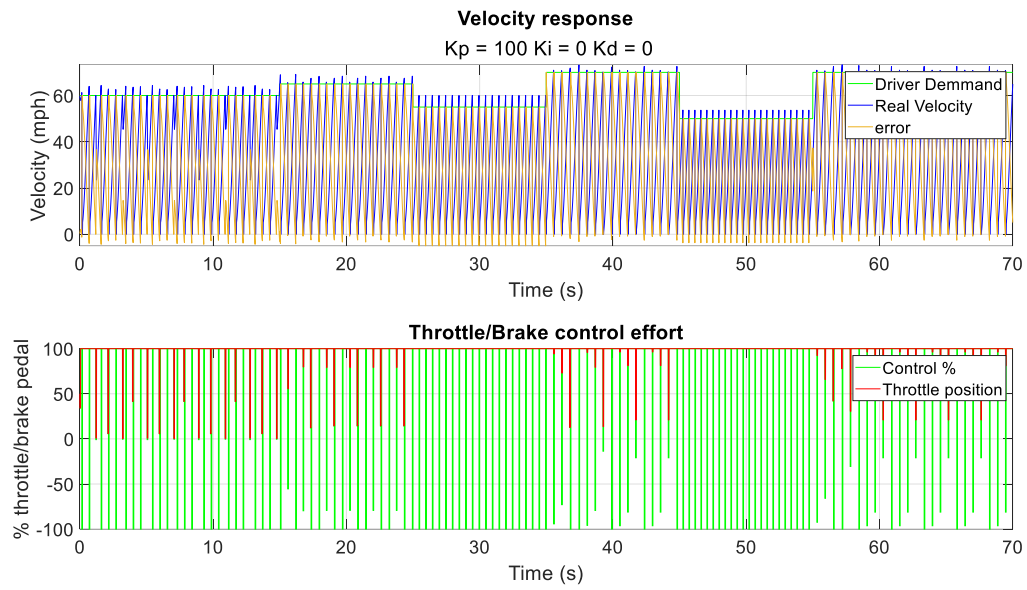


Figure 6 - Proportional run 3, K_p set to 100

4.2.2 Integral

The first PI run (Figure 7) tracked each demand level and largely removed the proportional controller's steady state offset, although step changes produced brief error and actuator spikes. The second run (Figure 8) remained stable and smoother but retained a small tracking error between demand changes, suggesting insufficient integral correction. In the final run (Figure 9), excessive integral action caused wind-up, actuator saturation and loss of tracking.

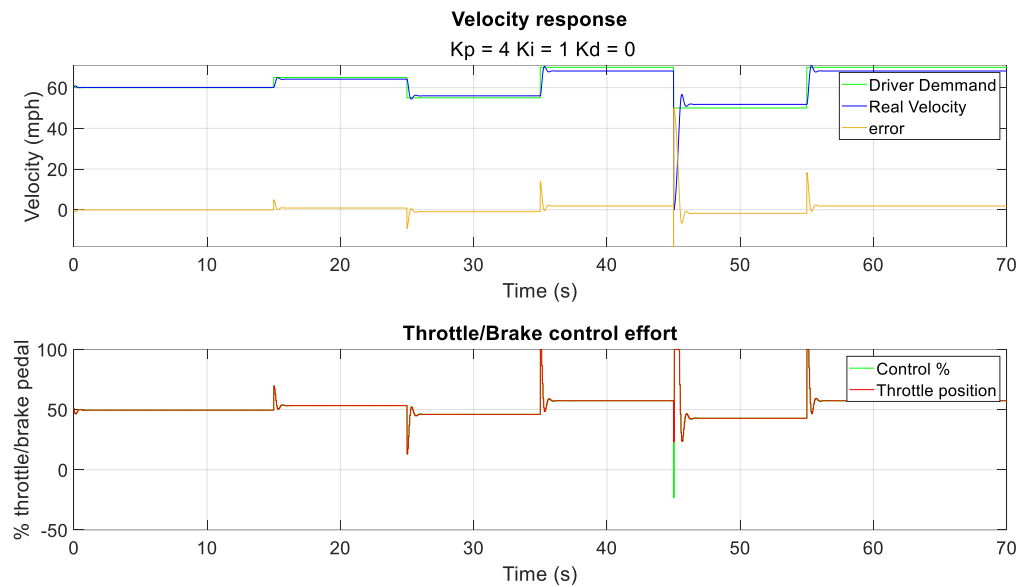


Figure 7 - Proportional & Integral run 1, K_p set to 4, K_i set to 1

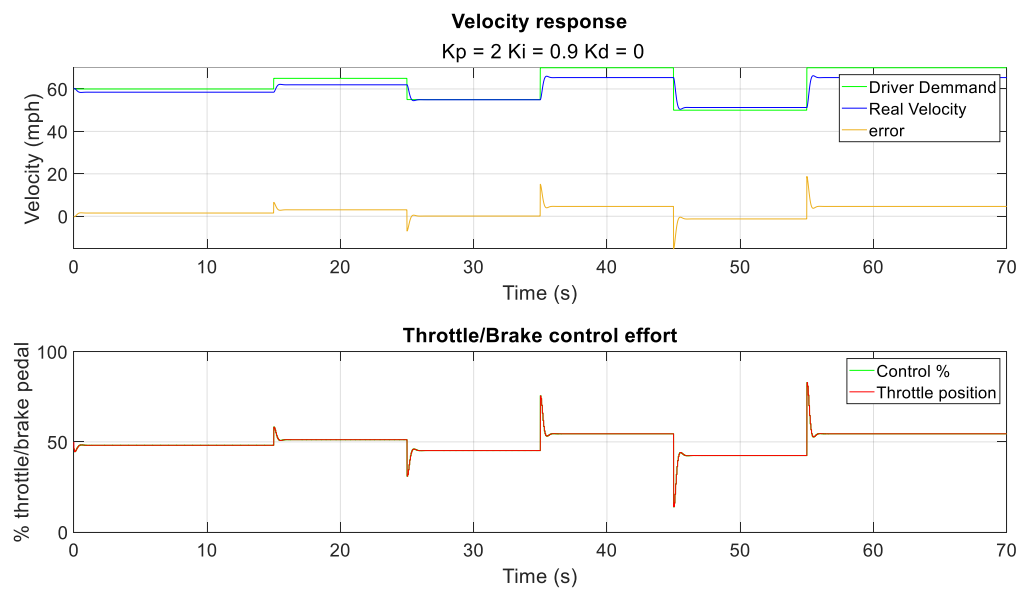


Figure 8 - Proportional & Integral run 2, K_p set to 2, K_i set to 0.9

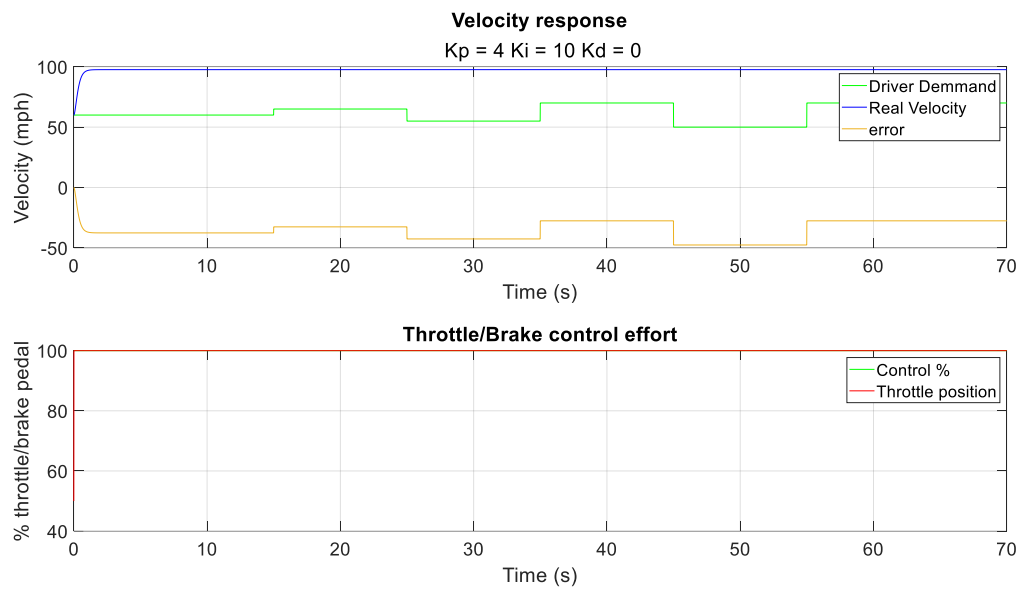


Figure 9 - Proportional & Integral run 2, K_p set to 4, K_i set to 10

4.2.3 Derivative

The first PID run (Figure 10) achieved rapid tracking and negligible steady-state error, but demand changes briefly saturated the actuator. The second run (Figure 11) was more damped, although slower rise and settling increased transient error and produced short control oscillations. Excessive derivative gain in the final run (Figure 12) amplified numerical noise, causing continuous actuator switching, severe saturation and complete loss of velocity tracking.

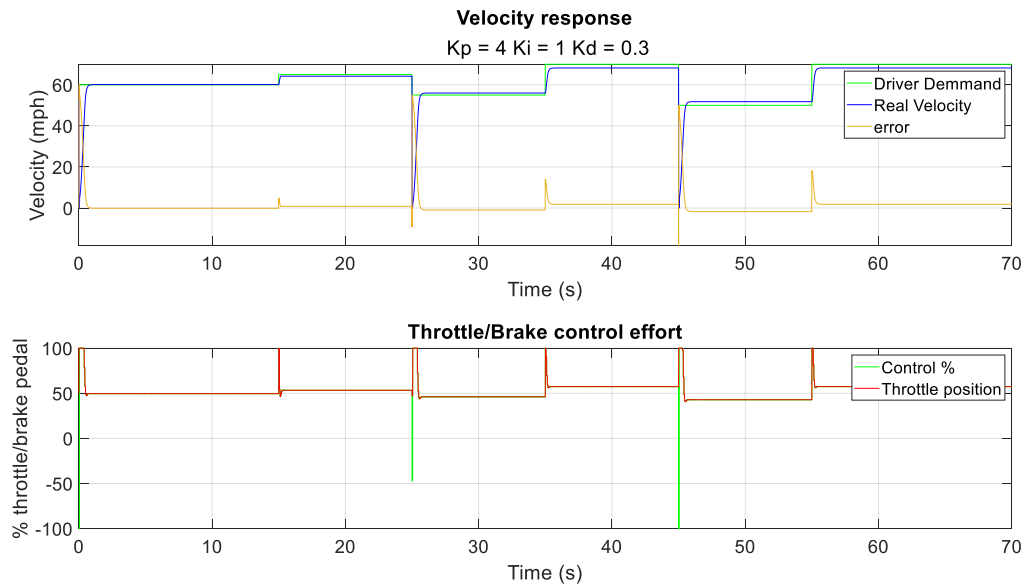


Figure 10 - Proportional, Integral & Derivative run 1, K_p set to 4, K_i set to 1, D set to 0.3

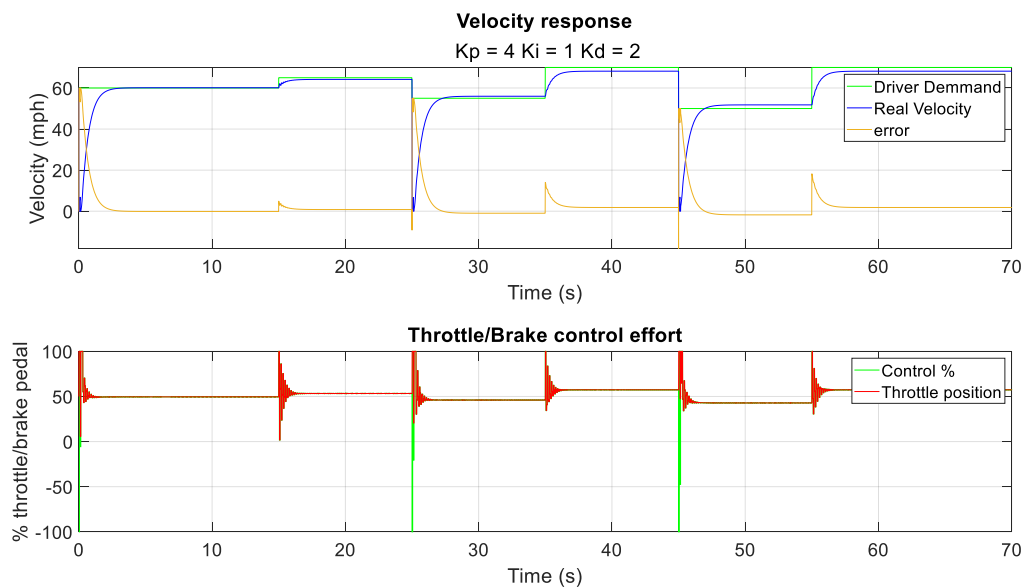


Figure 11 - Proportional, Integral & Derivative run 2, K_p set to 4, K_i set to 1, D set to 2

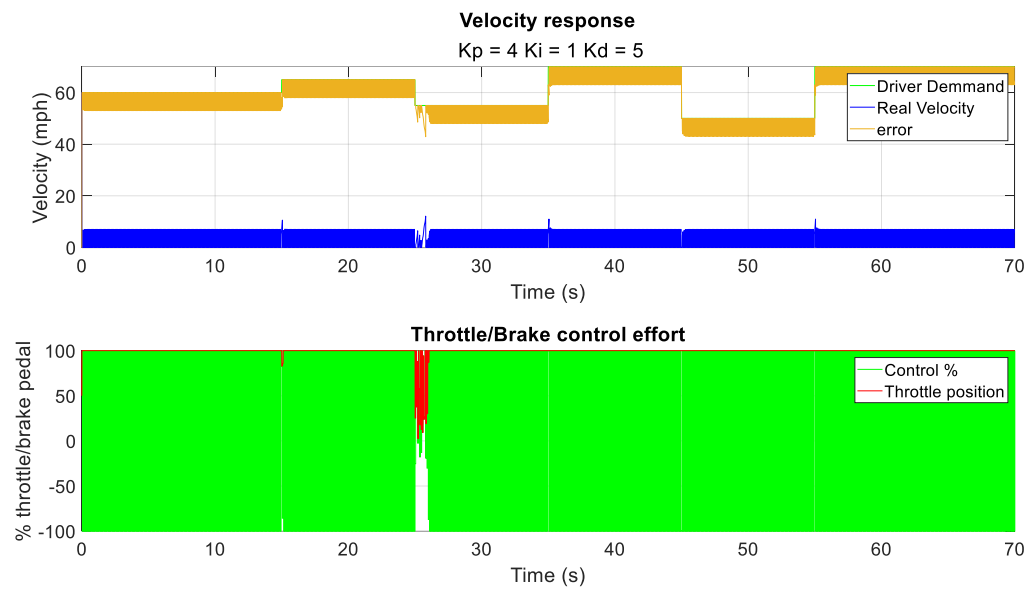


Figure 12 - Proportional, Integral & Derivative run 3, K_p set to 4, K_i set to 1, D set to 5

4.2.4 Overall tuning, meeting standards, automation

Progressive testing showed proportional action responded rapidly but retained steady-state error, while excessive integral or derivative gain caused saturation and instability. PI settings of ($K_p=2$) and ($K_i=0.9$) provided the best compromise.

However, steep velocity changes, error spikes, actuator peaks approaching 80% and residual tracking error prevented demonstration of ISO 15622 compliance. Cohen–Coon tuning could automate gain selection from process gain, time constant and delay, reducing transient peaks while preserving stability (Cohen and Coon 1953; ISO 2018).

5 Time step impact

5.1 Method

For testing the time step impact on the controller, the gains for PID were fixed. With the gains fixed, the sampling time was tested in increments of 0.1s from baseline until the response became oscillatory or unstable.

5.2 Results and analysis

At the baseline sampling time of 0.05 s, the PI controller produced a stable and smooth response, with limited overshoot and brief actuator peaks following each demand change. Increasing the sampling time to 0.15 s introduced greater overshoot, minor control oscillations and a longer settling period, although the response remained adequately damped. At 0.25 s, pronounced undershoot, overshoot and repeated oscillations were observed, while the control effort became increasingly abrupt. This trend supports MathWorks' finding that increasing controller sampling time can produce a more oscillatory closed-loop response and may require the controller to be redesigned (MathWorks 2026). A longer sampling interval causes corrective actions to use older measurements, increasing effective delay and reducing the stability margin. Therefore, 0.15 s represented the longest reasonable sampling time for maintaining a smooth, non-oscillatory response.

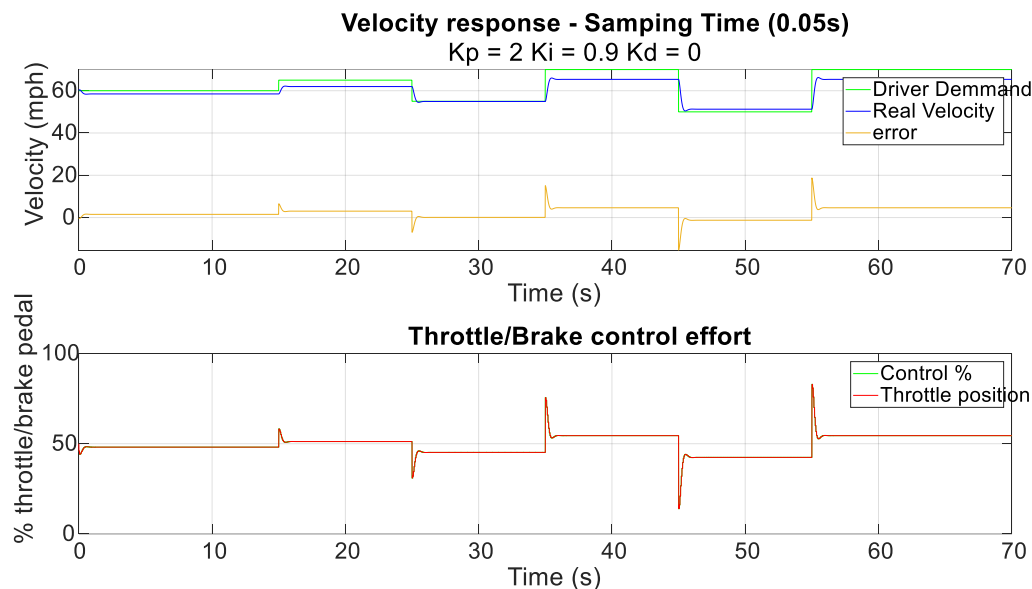


Figure 13 - Sampling Time variable baseline

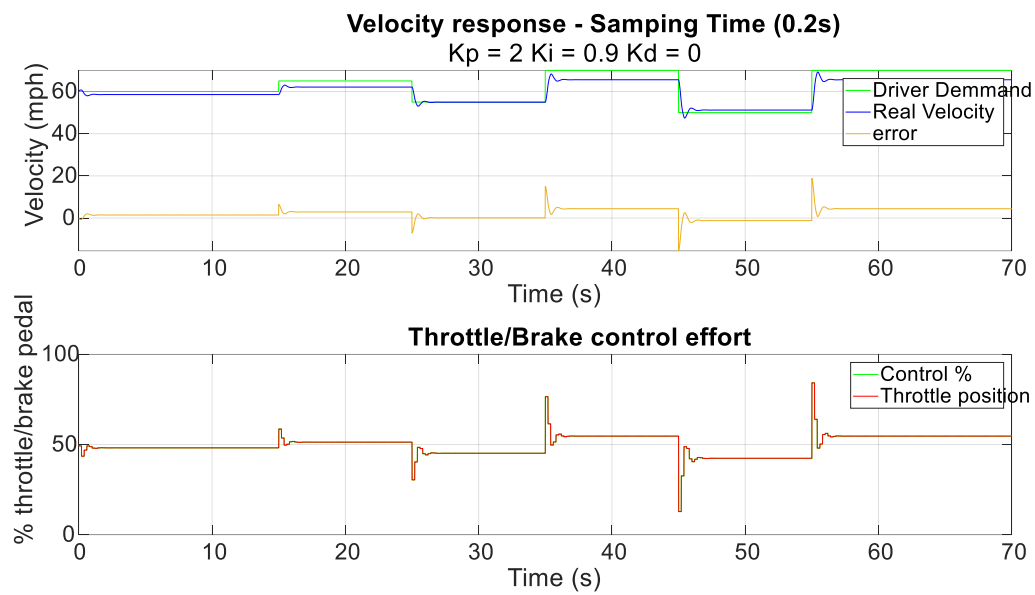


Figure 14 - Sampling Time variable set to 0.2 seconds

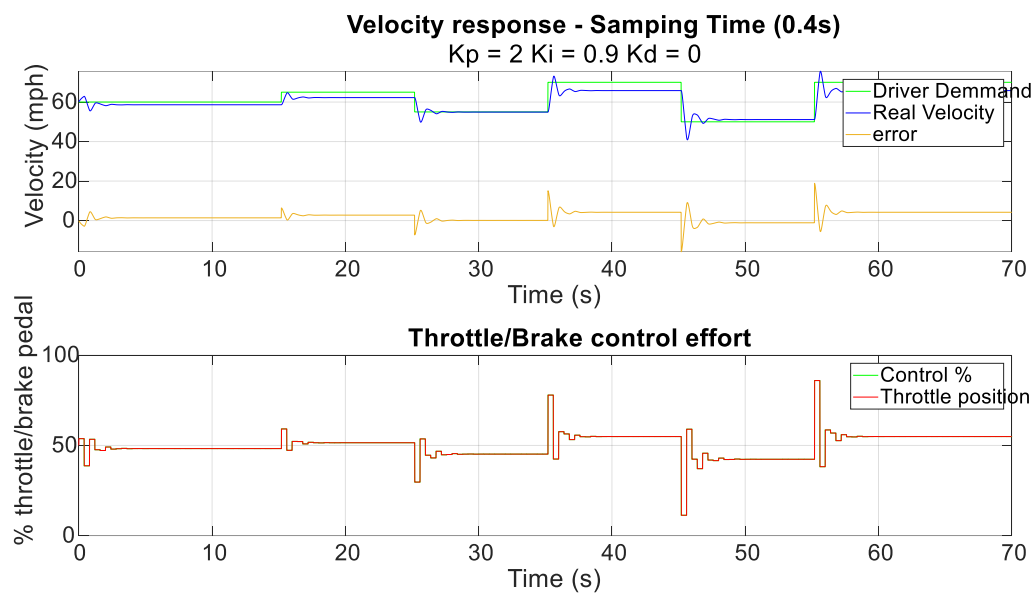


Figure 15 - Sampling Time variable set to 0.4 seconds

6 Signal delay impact

6.1 Method

The influence of actuator communication delay was evaluated while maintaining set PID gain values. Simulations were conducted using delays of 0 s, 0.25 s and 0.5 s. The velocity response and control effort were compared to determine how increasing latency affected tracking accuracy, oscillatory behaviour, settling performance and the onset of closed-loop instability.

6.2 Results and analysis

With zero communication delay, the PI controller remained stable and tracked each demand change with limited overshoot and relatively smooth control effort. Introducing a 0.25 s delay increased overshoot, settling time and damped oscillations, although the response remained stable. At 0.5 s, sustained oscillations, actuator saturation and large tracking errors demonstrated closed-loop instability. These results support Wang, Sun and Bai (Wang et al. 2021) who found that communication delay can degrade control performance and potentially destabilise networked control systems. Delay introduces phase lag because commands are calculated from outdated vehicle conditions. Therefore, 0.25 s was the largest tested delay that retained stability with the fixed gains; reducing the gains could increase delay tolerance but would produce a slower nominal response.

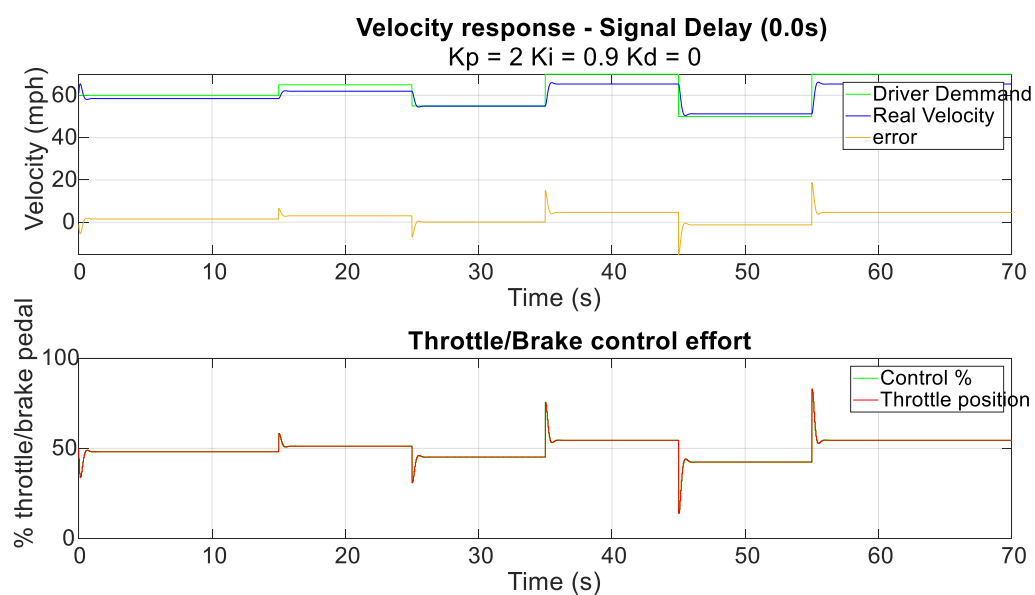


Figure 16 - Signal delay variable baseline

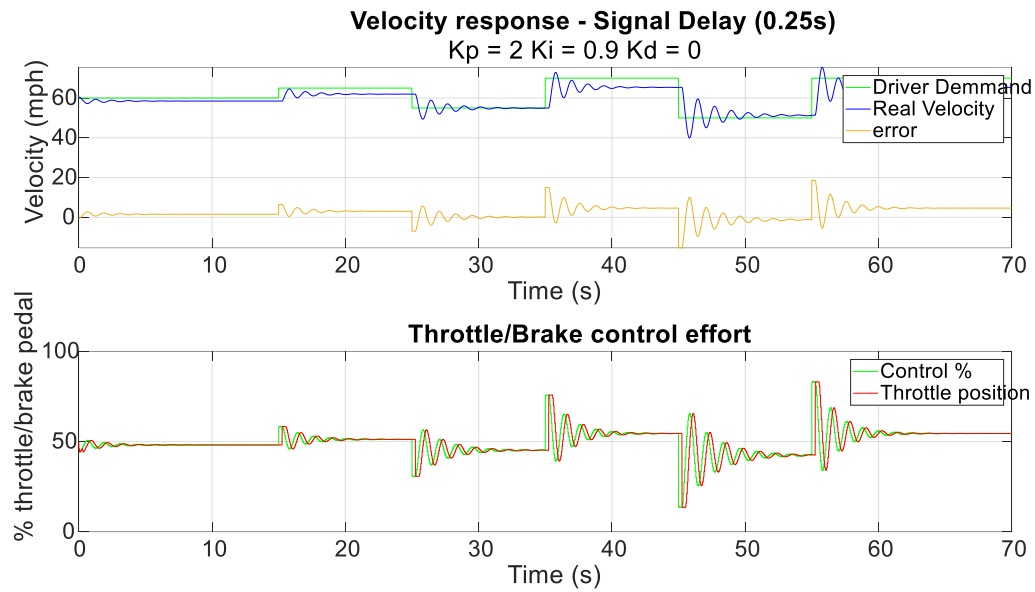


Figure 17 - Signal delay variable set to 0.25 seconds

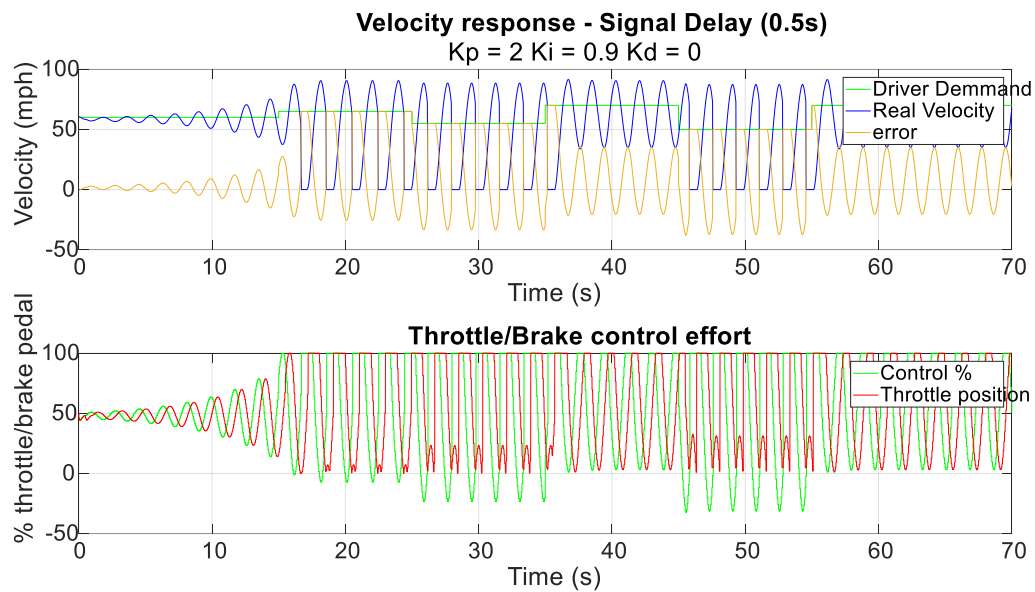


Figure 18 - Signal delay variable set to 0.5 seconds

7 Number of Bits impact

7.1 Method

The effect of signal reduction was investigated while maintaining constant PID controller gains. The bit depth of the sensor, demand and actuator communication channels was reduced simultaneously from the 16-bit baseline to 8 bits and subsequently to 4 bits. The simulation was repeated at each resolution, and the resulting changes in velocity tracking, steady-state accuracy behaviour were evaluated.

7.2 Results and analysis

At 16 bits, the PI controller produced a smooth and stable response, with limited transient error and consistent velocity tracking. Reducing the signals to 8 bits introduced minor quantisation ripple; however, overall tracking remained comparable to the baseline. At 4 bits, coarse resolution caused persistent velocity oscillations and rapid actuator fluctuations. This supports Lim and Loh (Lim and Loh 2014) who demonstrated that quantised feedback could produce self-oscillating limit cycles through switching between discrete signal levels. Although demand was broadly followed, the repeated acceleration and deceleration would reduce ride comfort. Consequently, 8 bits was the minimum tested resolution that maintained acceptable performance, whereas 4 bits provided insufficiently smooth and accurate control.

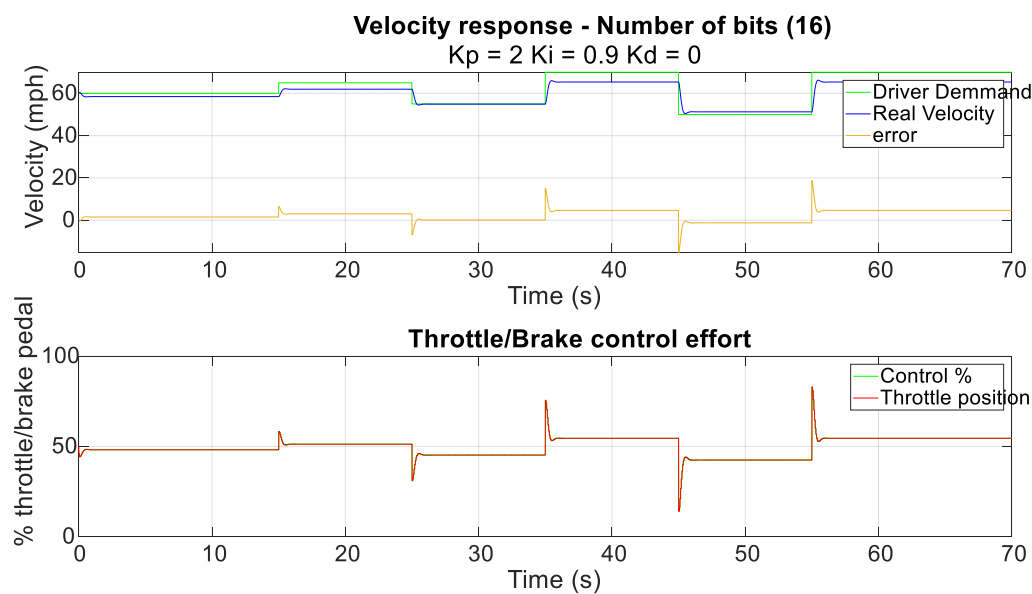


Figure 19 - Number of bits variable baseline

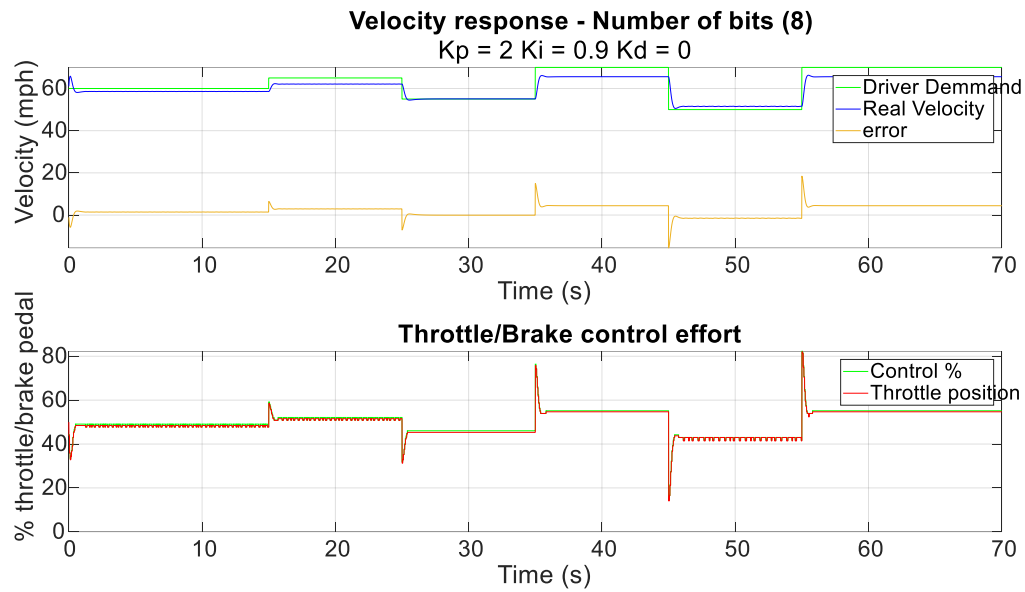


Figure 20 - Number of bits variable set to 8

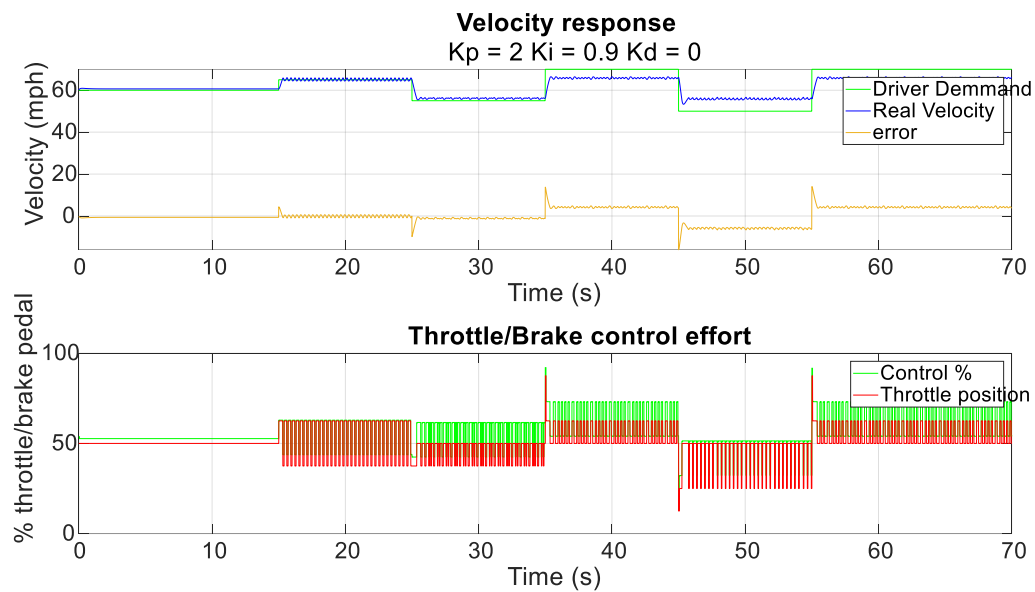


Figure 21 - Number of bits variable set to 4

8 Conclusion

This report developed and evaluated a digitally communicated, closed-loop cruise-control model using MATLAB. Progressive testing demonstrated that proportional control provided a rapid response but retained steady-state error, while integral action improved tracking accuracy. Excessive integral and derivative gains caused wind-up, actuator saturation, noise amplification and instability, reflecting the recognised trade-offs associated with PID tuning (Somefun et al. 2021). Overall, the PI controller with ($K_p=2$), ($K_i=0.9$) and ($K_d=0$) provided the best compromise between stability, tracking accuracy and control effort.

Digital implementation factors significantly influenced performance. The 0.05 s baseline sampling time produced the smoothest response, while 0.15 s represented the longest reasonable interval tested, supporting evidence that slower sampling can increase closed-loop oscillation (MathWorks 2026). A 0.25 s communication delay was the largest tested value that retained stability, whereas 0.5 s caused sustained instability, consistent with the recognised effects of network-induced delay (Wang et al. 2021). Signal resolution could be reduced to 8 bits with limited performance loss, while 4-bit communication caused quantisation-induced oscillation and unacceptable tracking behaviour (Lim and Loh 2014).

The results demonstrate the importance of considering controller tuning alongside sampling, latency and signal resolution. Although the selected configuration supported stable longitudinal control, compliance with ISO 15622 was not demonstrated because acceleration, deceleration, jerk, driver override, diagnostics and fault responses were not fully validated (ISO 2018). Further work should implement output-rate limiting, systematic fault testing and anti-wind-up protection to improve recovery from actuator saturation (MathWorks 2026a; MathWorks 2026b).

9 References

- Åström, K.J. and Häggglund, T. 2001. The future of PID control. *Control Engineering Practice* 9(11), pp. 1163–1175. doi: 10.1016/S0967-0661(01)00062-4.
- Cohen, G.H. and Coon, G.A. 1953. Theoretical consideration of retarded control. *Transactions of the ASME* 75(5), pp. 827–834. Available at: <https://asmedigitalcollection.asme.org/fluidsengineering/article/75/5/827/1145041/Theoretical-Consideration-of-Retarded-Control>.
- ISO. 2018a. ISO 15622:2018 Intelligent transport systems—Adaptive cruise control systems—Performance requirements and test procedures. Available at: <https://www.iso.org/standard/71515.html>.
- ISO. 2018b. ISO 26262-1:2018 Road vehicles—Functional safety—Part 1: Vocabulary. Available at: <https://www.iso.org/standard/68383.html>.
- ISO. 2022. ISO 21448:2022 Road vehicles—Safety of the intended functionality. Available at: <https://www.iso.org/standard/77490.html>.
- Lim, L.H.I. and Loh, A.P. 2014. Limit cycles in high-resolution quantized feedback systems. *International Journal of Simulation Systems, Science and Technology* 15(6), pp. 1–7. doi: 10.5013/IJSSST.a.15.06.01.
- MathWorks. 2026a. Anti-windup control using PID Controller block. Available at: <https://www.mathworks.com/help/simulink/slref/anti-windup-control-using-a-pid-controller.html>.
- MathWorks. 2026b. Model computational delay and sampling effects. Available at: <https://www.mathworks.com/help/slcontrol/ug/modeling-computational-delays-and-sampling-effects.html>.
- MathWorks. 2026c. stepinfo—Rise time, settling time, and other step-response characteristics. Available at: <https://www.mathworks.com/help/control/ref/dynamicsystem.stepinfo.html>.
- Mustafar, M.Z.C., Abu Bakar, S.A., Zainuddin, Z., Mansor, M.S.F., Ismail, N.H.F. and Rani, M.R. 2020. Optimal design of an adaptive cruise control system for driving comfort and fuel economy. *Journal of the Society of Automotive Engineers Malaysia* 4(3), pp. 279–287. doi: 10.56381/jsaem.v4i3.23.
- National Instruments. 2025. The PID controller and theory explained. Available at: <https://www.ni.com/en/shop/labview/pid-theory-explained.html>.
- Somefun, O.A., Akingbade, K. and Dahunsi, F. 2021. The dilemma of PID tuning. *Annual Reviews in Control* 52, pp. 65–74. doi: 10.1016/j.arcontrol.2021.05.002.
- Wang, Z., Sun, J. and Bai, Y. 2021. Stability analysis of event-triggered networked control systems with time-varying delay and packet loss. *Journal of Systems Science and Complexity* 34(1), pp. 265–280. doi: 10.1007/s11424-020-9299-1.

10 Appendices

10.1 Controller function: Controllerrrss.m

This function calculates the sampled PID command and applies the actuator limits.

```
function [ControlOut,errorOut] = Controllerrrss...
    (demand,velocity,j,e_vec,Ts,sim_step,control,P,I,D)
%% Difference of current vehicle speed
errorOut=demand-velocity;
%% Propotional
P_part = errorOut.*P;
%% Persistent I Part
persistent Prev_I_part
if j>1
    prev = e_vec(j-Ts/sim_step);
else
    prev = control;
    Prev_I_part = prev; % Store the current integral part for the next iteration
end
%% Integral
I_part = ((errorOut+prev)./2.*(Ts)+Prev_I_part).*I;
%% Derivative
D_part = ((errorOut-prev)./Ts).*D;
%% Control output
ControlOut = P_part+I_part+D_part; % Example control law, can be adjusted as
needed
%% Limits output
ControlOut(ControlOut>100)=100;
ControlOut(ControlOut<-100)=-100;
end
```

10.2 Simulation script: CruiseControlScript_pract.m

This main script sets the baseline parameters, runs the vehicle loop, stores the response data and plots velocity tracking against throttle/brake effort.

```

%% MAIN FILE FOR CRUISE CONTROLLER SYSTEM %%%%%%%%%%
% (1) initialize
% (2) set implementation parameters
% (3) set simulation parameters
% (4) simulation loop
% (5) plot results
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% (1) Initialize %%%%%%%%%%
clc
clear
close all % house keeping
P = 4;
I = 1;
D = 0.3;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% (2) set implementation parameters %%%%%%%%%%

% sampling time must be an integer multiple of sim_step
Ts = 0.05;
% number of bits used to encode velocity sensor signal. velocity sensor
% range in mph [0,150]
m_velocityS = 0;
M_velocityS = 150;
n_bits_sensor_signal = 16;
% number of bits used to encode velocity demand signal. demand
% range in mph [30,110]
m_demand = 30; M_demand = 110;
n_bits_demand_signal = 16;
% number of bits used to encode throttle/brake actuator signal. actuator
% range in mph [-100,100] where [-100,0) is percentage of braking and
% (0,100] is percentage of throttle
n_bits_actuator_signal = 16;
R = 2.^n_bits_actuator_signal;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% (3) set simulation parameters %%%%%%%%%%

sim_step = 0.01; % simulation time step in seconds
sim_time = 70; % simulation time length in seconds
T = 0:sim_step:sim_time; % simulation steps vector
% initialize vector where to store results for plots
[d_vec,u_vec ,y_vec,e_vec,Th_vec] = deal(nan(size(T)));
throttle_setting = 50; % initialize control action (throttle/brake setting)
% communication delay between controller and actuator in seconds.
% must be an integer multiple of sim_step
actuator_signal_delay = 0.00;
%% remove when in function
m_ControlSig = -100;
M_ControlSig = 100;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[u, control] = deal(50);
%% (4) simulation loop %%%%%%%%%%
velocity_real = 60;

```

```

for j = 1:length(T) % for every simulation step

    loops_remaining = length(T)-j; % display progression
    disp(loops_remaining)
    % get vehicle raw velocity sensor measurement from the communication bus
    [velocity_raw, velocity_real,throttle_setting] =
    getRawVelocity(n_bits_sensor_signal,velocity_real,throttle_setting, sim_step,u);

    % decode raw bus signal (from x-bit binary to mph)
    velocity = ((bin2dec(velocity_raw).*(M_velocityS-m_velocityS))/R)+m_velocityS;

    if mod((j-1)*sim_step,Ts) == 0 % every sampling time

        % get driver's velocity demand raw signal from the communication bus
        demand_raw = getRawVelocityDemand(n_bits_sensor_signal,sim_step,j);

        %% decode raw signal (from x-bit binary to mph)
        demand = (bin2dec(demand_raw).*(M_demand-m_demand)./R)+m_demand; %
        INCORRECT DECODING! INSERT YOUR DECODING FUNCTION HERE

        %% calculate control signal and error (range of control [-100,100])
        [control, error] =
        Controllerrrss(demand,velocity,j,e_vec,Ts,sim_step,control,P,I,D); % INCORRECT
        CONTROL! INSERT YOUR CONTROL ALGORITHM HERE

        %% encode control signal into x-bit binary number
        control_raw = dec2bin((control-m_ControlSig)./(M_ControlSig-
        m_ControlSig).*R); % INCORRECT ENCODING! INSERT YOUR ENCODING FUNCTION HERE

        %% send control command to throttle/brake actuator via communication
        % bus (record the control signal instance as u)
        u =
        sendControlCommand(control_raw,n_bits_actuator_signal,actuator_signal_delay,j,
        sim_step, u_vec);
    end

    % store values for plotting
    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

% plot velocity response
subplot(2,1,1)
plot(T,d_vec,'g-');hold on
plot(T,y_vec,'b-');
plot(T,d_vec-y_vec); grid;
title('Velocity response');
subtitle([' Kp = ', num2str(P), ' Ki = ', num2str(I), ' Kd = ',...
        num2str(D)]);
ylabel('Velocity (mph)'); xlabel('Time (s)');

```

```
legend('Driver Demmand','Real Velocity','error')

% plot throttle/brake actuator effort
subplot(2,1,2)
stairs(T,u_vec,'g-');
hold on;
plot(T,Th_vec,'r-'); grid;
title('Throttle/Brake control effort');
ylabel('% throttle/brake pedal'); xlabel('Time (s)');
legend('Control %','Throttle position')
supersizeme(2.5)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

10.3 Vehicle model and sensor: getRawVelocity.m

This helper advances the vehicle model, applies throttle or braking limits and encodes the resulting velocity measurement.

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% u initial throttle setting as whole percentage

h = sim_step;
throttle_setting = u;% (u/300)+ throttle_setting;
persistent d v_dot x_v v_dot_last x_v_old

if isempty (d) % this just Initializes
    d = 1;
    %    x_v = 0.25;
    x_v_old = 0.25;
    v_dot_last = 0;
end

% parameters
gear = 1.2;
final_drive = 2.82;
rs_to_rpm = 9.549; % rad/s to rpm
mps_to_mph = 2.2369;
radius = 0.3;
cd = 0.29;
roh = 1.23;
area = 2;
mass = 1854;

if abs(throttle_setting) > 100 % limit throttle to +/-100%
    throttle_setting = 100;
end

if throttle_setting < 0
    brake = abs(throttle_setting);%P;
    throttle_setting = 0;
else
    brake = 0;
end

wheel_speed = (old_velocity / mps_to_mph) / radius; % rad/s
engSpeedRPM = wheel_speed * rs_to_rpm * gear * final_drive;

x_v_dot = (-10 * x_v_old + 0.05 * (throttle_setting)); %internal dynamics of
throttle actuator
x_v = x_v_old + x_v_dot*h; % numerical integration
x_v_old = x_v;

engSpeedRPM = 528.7 + (engSpeedRPM * 0.152) + (-0.0000217 * engSpeedRPM^2);
engine_torque = x_v * engSpeedRPM*gear * final_drive;

%v_dot = ((engine_torque / radius) - (cd * roh * area* 0.5 * velocity^2) - (0.05 *
mass * 9.81) -(brake * mass * 9.81)) / mass;
% N wheel force % - aero force N
v_dot = ((engine_torque / radius) - (cd * roh * area* 0.5 * old_velocity^2) ...
- (0.05 * mass * 9.81) -(brake * mass * 9.81)) / mass;

acc = v_dot + (h / 2) * (v_dot - v_dot_last);
```

```

v_dot_last = v_dot;
velocity = old_velocity + acc;
% old_velocity = velocity;

if velocity < 0 || isnan(velocity)
    velocity = 0;
end
if velocity > 150
    velocity = 150;
end
throttle_setting = throttle_setting + brake; % gives negative output to plot if
braking
% convert into binary signal
velocity_raw = dec2bin((2^x*(velocity))/150);
velocity_real = velocity;

```

10.4 Demand encoder: getRawVelocityDemand.m

This helper produces the time-varying driver demand as an encoded binary string.

```

function [d]=getRawVelocityDemand(x,sim_step,j)
%% simulate driver's velocity demand (in mph)

% driver velocity demand can go from 30 to 110 mph at steps of 5 mph.
% Only x bits are used to encode this signal. 00...0 corresponds to 30
% mph and 11...1 to 110 mph

% global sim_step
% global j

if j*sim_step >= 0 && j*sim_step < 15
    d=dec2bin((2^x*(60-30))/(110-30));
elseif j*sim_step >= 15 && j*sim_step < 25
    d=dec2bin((2^x*(65-30))/(110-30));
elseif j*sim_step >= 25 && j*sim_step < 35
    d=dec2bin((2^x*(55-30))/(110-30));
elseif j*sim_step >= 35 && j*sim_step < 45
    d=dec2bin((2^x*(70-30))/(110-30));
elseif j*sim_step >= 45 && j*sim_step < 55
    d=dec2bin((2^x*(50-30))/(110-30));
elseif j*sim_step >=55
    d=dec2bin((2^x*(70-30))/(110-30));
end

```

10.5 Actuator decoder: sendControlCommand.m

This helper decodes the signed actuator command and applies the configured controller-to-actuator delay.

```

function u = sendControlCommand(u_raw,x,actuator_signal_delay,j, sim_step, u_vec)
u = bin2dec(u_raw);
u = (u*(100+100))/(2^x) - 100;

if actuator_signal_delay > 0 && actuator_signal_delay/sim_step < j
    u = u_vec(j-actuator_signal_delay/sim_step);
end

```