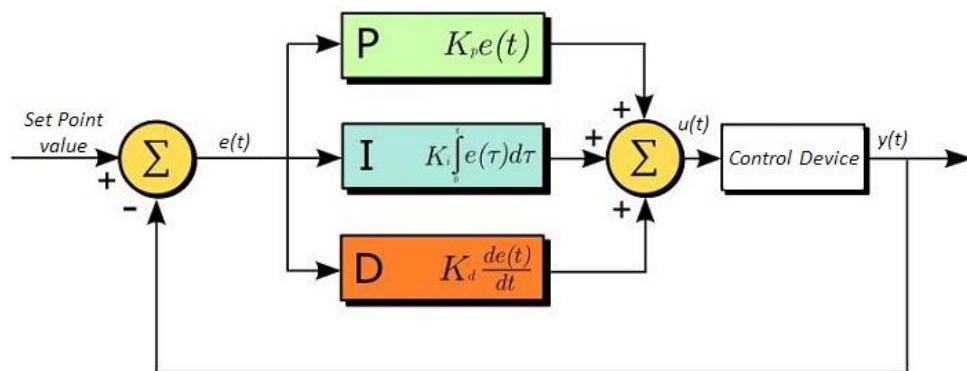


University of Wales Trinity Saint David

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

Shea Purnell-Whitehead

2212438

28/07/2025

Contents

1	Introduction	1
1.1	Aims	1
1.2	Objectives	1
2	Background	2
2.1	PID general background	2
2.2	Cruise control standards and regulations.....	2
3	PID tuning.....	3
3.1	Method	3
3.2	Results and analysis.....	3
3.2.1	Proportional gain (K_p)	3
3.2.2	Integral gain (K_i)	4
3.2.3	Derivative gain (K_d)	6
4	Time sample impact.....	7
4.1	Method	7
4.2	Results	8
5	Signal delay impact	9
5.1	Method	9
5.2	Results and analysis.....	9
6	Number of Bits impact	11
6.1	Method	11
6.2	Results and analysis.....	11
7	Conclusion.....	13
7.1	Velocity Conversion and Motion Analysis in Cruise Control Systems	14
8	Bibliography	16
9	Code	17
9.1	Control function.....	17
9.2	Cruise control script.....	17

1 Introduction

1.1 Aims

The primary aim of this report is to model, implement, and evaluate a digital PID control system for a vehicle cruise control application using MATLAB. This includes investigating the effects of signal encoding/decoding, sampling rate, controller tuning, and network delays on system performance, with the goal of achieving a stable and responsive closed-loop system

1.2 Objectives

To fulfil the above aim, the following objectives are established:

- To implement sensor, demand, and actuator signal encoding and decoding functions in MATLAB.
- To design and analyse a digital PID feedback controller that achieves zero steady-state error and acceptable transient performance.
- To evaluate the impact of varying bit resolutions on system performance and determine a minimum viable configuration.
- To assess the influence of different sampling times and identify the maximum sampling interval that maintains satisfactory control.
- To analyse the system's behaviour under different communication network delays and determine the maximum tolerable latency.
- To explore possible gain adjustments to compensate for delay and performance degradation.
- To demonstrate an in-depth understanding of practical considerations and limitations in digital control systems for automotive applications.

2 Background

2.1 PID general background

PID (Proportional–Integral–Derivative) controllers are widely used in engineering to regulate system behaviour by reducing the error between a desired setpoint and the actual output (Åström & Hägglund, 1995). The proportional term responds to current error, the integral term addresses accumulated past errors, and the derivative term predicts future trends. Together, these components enable fast, accurate, and stable control. PID controllers are popular due to their simplicity, robustness, and ease of implementation in digital systems like MATLAB.

2.2 Cruise control standards and regulations

Cruise control systems must comply with national and international safety standards. In the UK and EU, regulations such as UNECE R79 (UNECE, 2023) and R13-H (UNECE, 2008) address automated control and braking systems. Standards like ISO 15622-Adaptive Cruise Control (International Organization for Standardization, 2018) and ISO 26262-Functional Safety- (International Organization for Standardization, 2018) guide system reliability and performance. These regulations ensure safety, compatibility, and effective integration with other vehicle systems (Luo, et al., 2019)

3 PID tuning

3.1 Method

PID tuning involves selecting appropriate gain values for the proportional (K_p), integral (K_i), and derivative (K_d) terms to achieve desired system performance. Common methods include:

- **Manual Tuning:** Adjusting parameters based on system response. This is time-consuming but gives insight into system behaviour.
- **Ziegler–Nichols Method:** Involves setting K_i and K_d to zero, increasing K_p until the system reaches sustained oscillation, then using empirical formulas to set all gains (Ziegler & Nichols, 1993).
- **Cohen–Coon Method:** Suitable for systems with known delay and time constants, offering good setpoint tracking (Joseph & O, 2017).
- **Software-Based Tuning:** Tools like MATLAB's PID Tuner automate gain selection based on model analysis, allowing faster optimisation and testing through simulations (Mathworks, 2025).

For digital implementations, the tuned gains must also consider discretisation effects and sampling time.

3.2 Results and analysis

3.2.1 Proportional gain (K_p)

The proportional gain (K_p) defines how much the controller output responds in relation to the error signal. For instance, with an error magnitude of 20 and a (K_p) of 10, the resulting output would be 100. A higher (K_p) leads to a faster system response, resulting in more aggressive acceleration or deceleration. However, setting the proportional gain too high can cause the process variable to oscillate. If (K_p) continues to increase, these oscillations can grow in amplitude, potentially leading to system instability (Åström & Hägglund, 1995). Refer to Figure 1 for an illustration of this behaviour. In the simulation, a (K_p) value of 10 was applied.

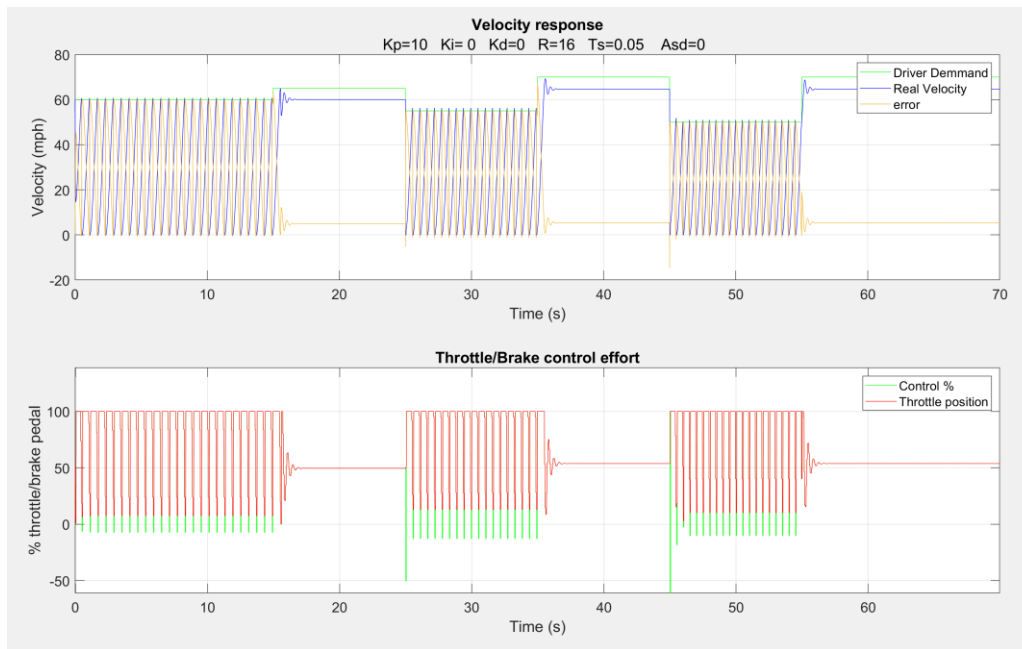


Figure 1-Matlab (K_p) set to 10 oscillations introduced (Purnell-Whitehead, 2025)

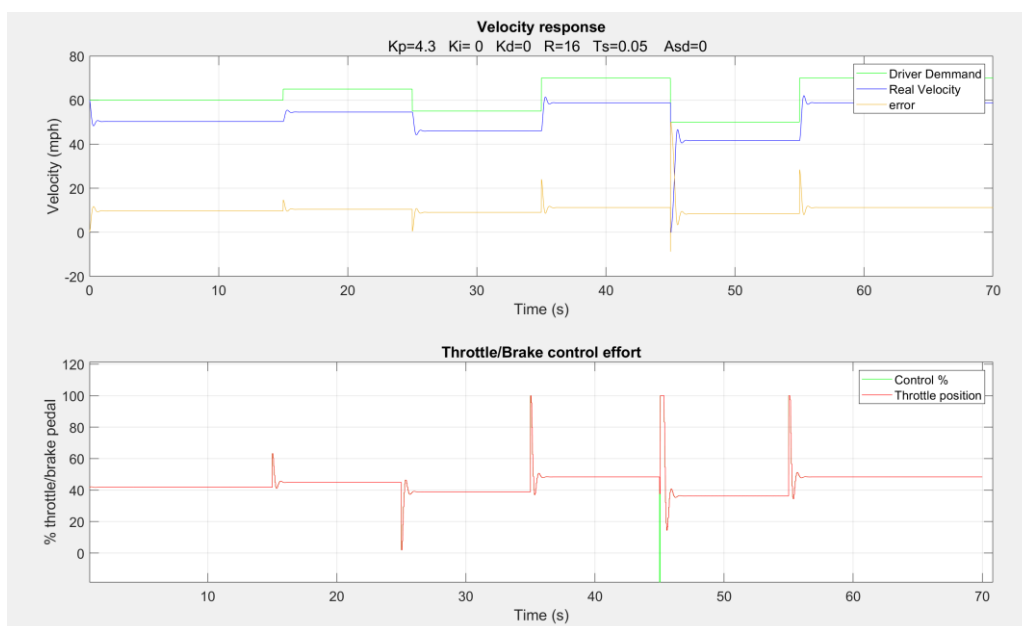


Figure 2-Matlab (K_p) Set to 4.3 is more stable with no overshoot (Purnell-Whitehead, 2025)

With proportional value set at 4.3 we can see real velocity and driver demand is now stable and a steady state error can be determined.

3.2.2 Integral gain (K_i)

Steady-state error refers to the remaining difference between the actual process variable and the target setpoint after the system has settled. The integral component addresses this by correcting any ongoing offset from the setpoint. Raising the integral gain (K_i) helps eliminate steady-state errors more quickly, but it can also lead to overshooting and potential instability. Conversely, a lower

(K_i) results in slower correction but ensures more stable and smoother control (Fernández de Cañete & Galindo, 2011).

Excessive values led integral wind up, shown in Figure 3.

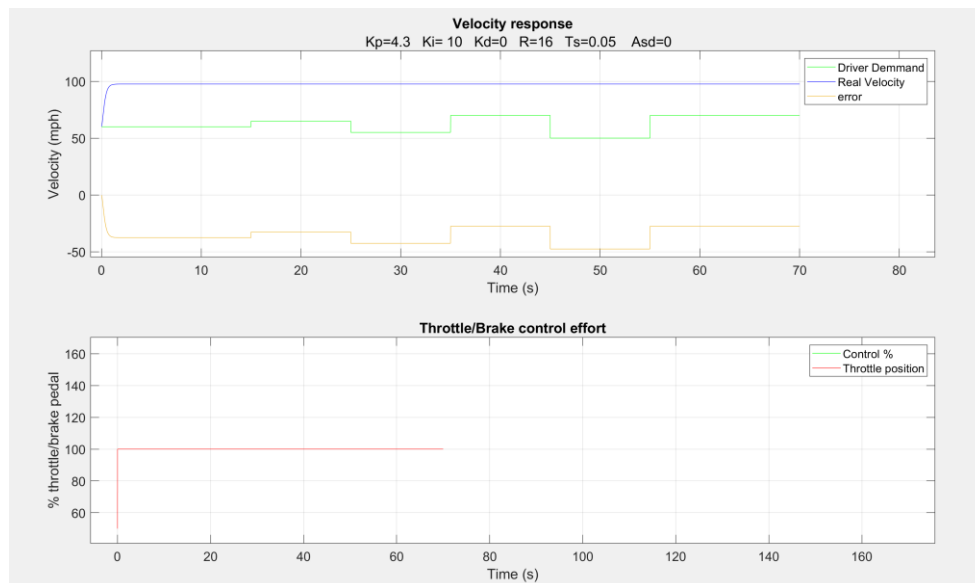


Figure 3-Matlab (K_i) set at 10 showing integral wind up

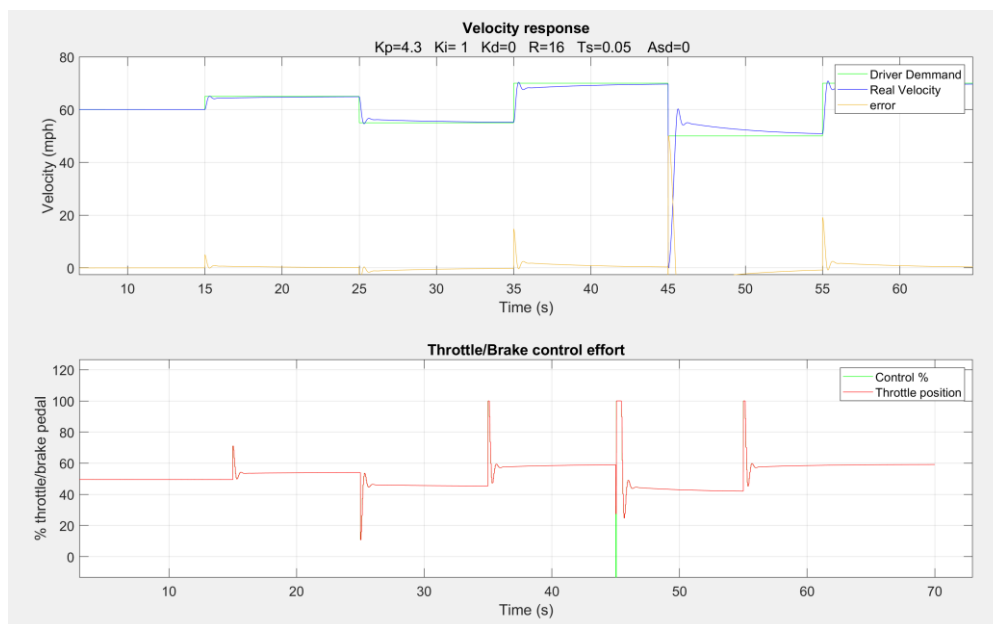


Figure 4-Matlab (K_i) set to 1 (Purnell-Whitehead, 2025)

With The (K_i) set to 1 as removed the steady state error allowing real velocity to follow demand more accurately. Functionally in a cruise control system, the speed set by the driver would match real velocity. This is the outcome of eliminating this steady state error.

3.2.3 Derivative gain (K_d)

Derivative control in a PID controller provides damping by responding to rapid changes in the error, which helps limit overshoot and suppress oscillations. Increasing the derivative gain (K_d) enhances system stability by reacting more strongly to the error's rate of change, but it can also amplify high-frequency sensor noise. Conversely, a lower (K_d) reduces sensitivity to quick fluctuations, resulting in a less responsive but smoother controller behaviour (Dorf & Bishop, 2011)

Figure 5 shows the amount of oscillating noise when adjusting (K_d).

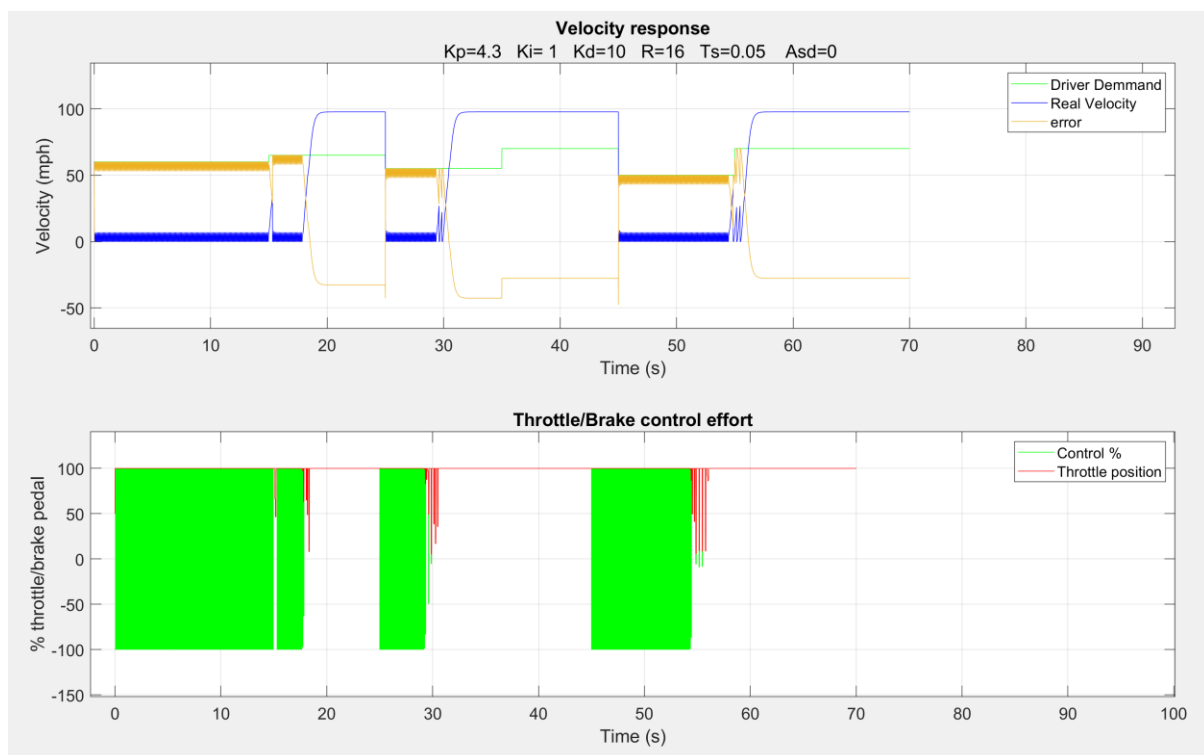


Figure 5-Matlab (K_d) set at 10 (Purnell-Whitehead, 2025)

4 Time sample impact

4.1 Method

The time sample in a digital control system plays a critical role in determining how accurately the controller responds to changes in the system. To investigate its effect, several simulation runs were performed in MATLAB using different sampling times. In each case, the same PID gains were used to isolate the impact of sampling time. The step response, control effort, and system stability were monitored to evaluate performance in rise time, settling time, and overshoot (Ismail & Hamza, 2015).

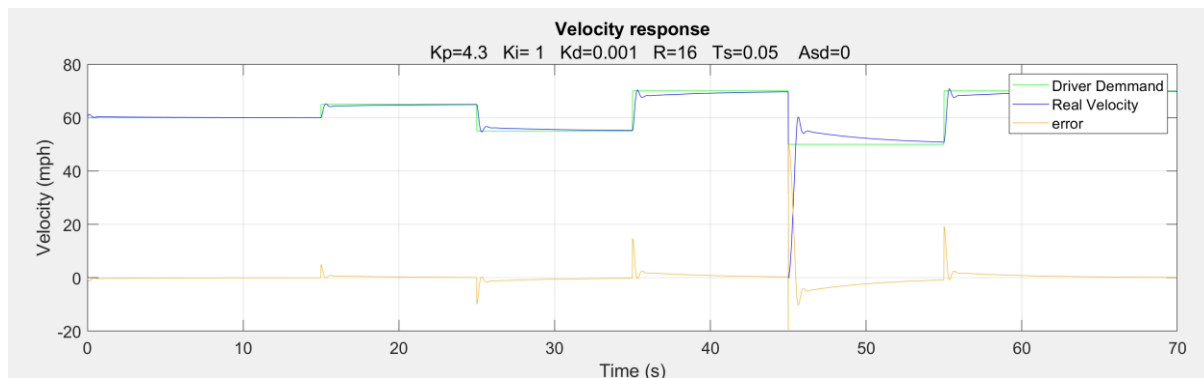


Figure 6-Time sample set to 0.05 (Purnell-Whitehead, 2025)

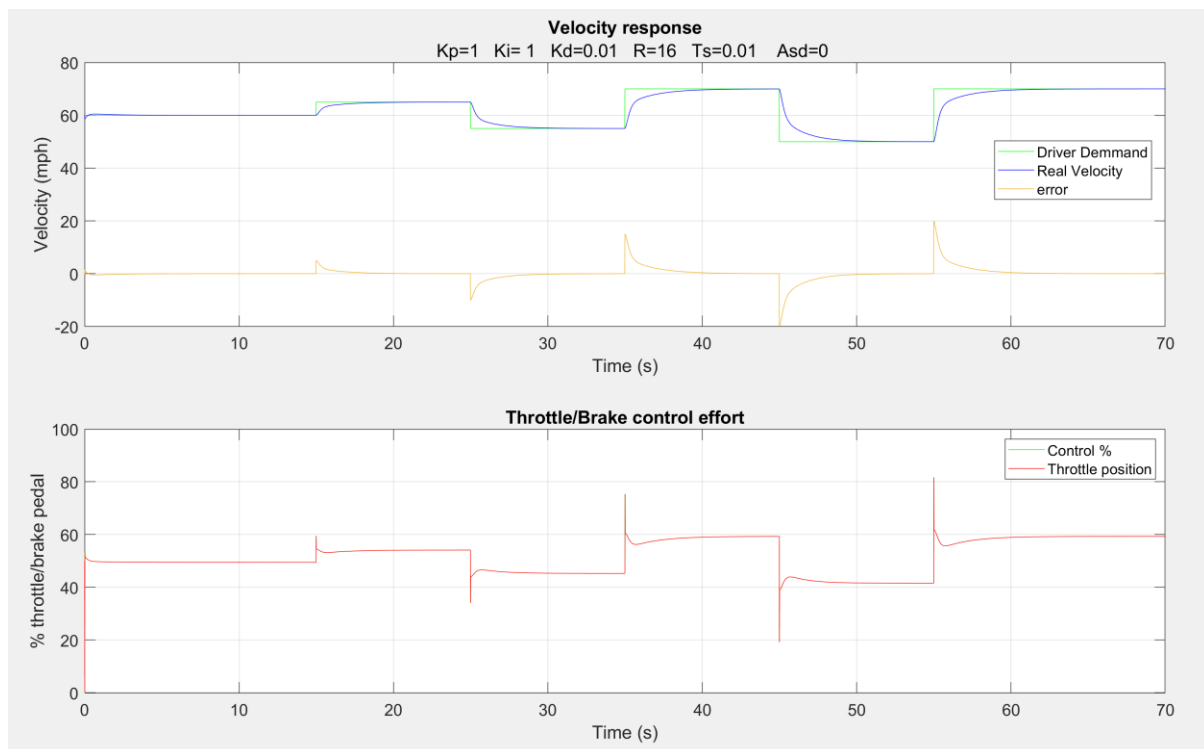


Figure 7-Matlab time sample set to 0.01 and PID re-tuned (Purnell-Whitehead, 2025)

This interval specifies how often the controller updates its calculations to maintain the desired system output. Selecting an appropriate sampling time is crucial, as it directly affects both the accuracy and responsiveness of the control system. It should be established prior to tuning the controller, since altering the sampling time afterward would necessitate retuning. Additionally, system cost must be considered—shorter sampling intervals demand more processing power, which can increase expenses or risk overloading a controller that lacks sufficient capability. Simulation results showed that sampling time had a significant impact on the cruise control system's accuracy and stability.

4.2 Results

The output graph (Figure 6) displays an anomaly where the speed briefly drops to 0 miles per hour (mph) when a lower speed is selected. To address this, the sampling time was increased, as the system does not require extremely fast responsiveness. With the revised sampling time set to 0.1 seconds, the system now demonstrates significantly improved stability. The issue of the temporary 0 mph reading has been eliminated, and both throttle and brake control efforts show reduced oscillations (Figure 7), indicating that the system is operating more efficiently. These adjusted parameters would be appropriate for a vehicle cruise control system. The real-world effect of the smoother rise and fall in real velocity would be a more comfortable/less harsh speed change.

5 Signal delay impact

5.1 Method

In digital control systems, especially those with distributed components (e.g., sensors, controllers, and actuators communicating over networks), signal delays are inevitable. These delays can arise from communication latency, processing time, or sensor/actuator lag (Rahnama, et al., 2016). To analyse the effect of such delays, artificial time lags were introduced into the feedback loop of the cruise control system in MATLAB.

The approach involved:

- Adding fixed delays (e.g 0.1 s, 0.5 s) in the simulation.
- Keeping the PID controller parameters constant to isolate the effect of the delay.
- Measuring the system's step response, overshoot, stability, and control effort at each delay level.

5.2 Results and analysis

The results showed that increasing delay significantly affects system performance:

0.1 s delay- Minimal effect on system behaviour. The cruise control maintained fast and stable tracking.

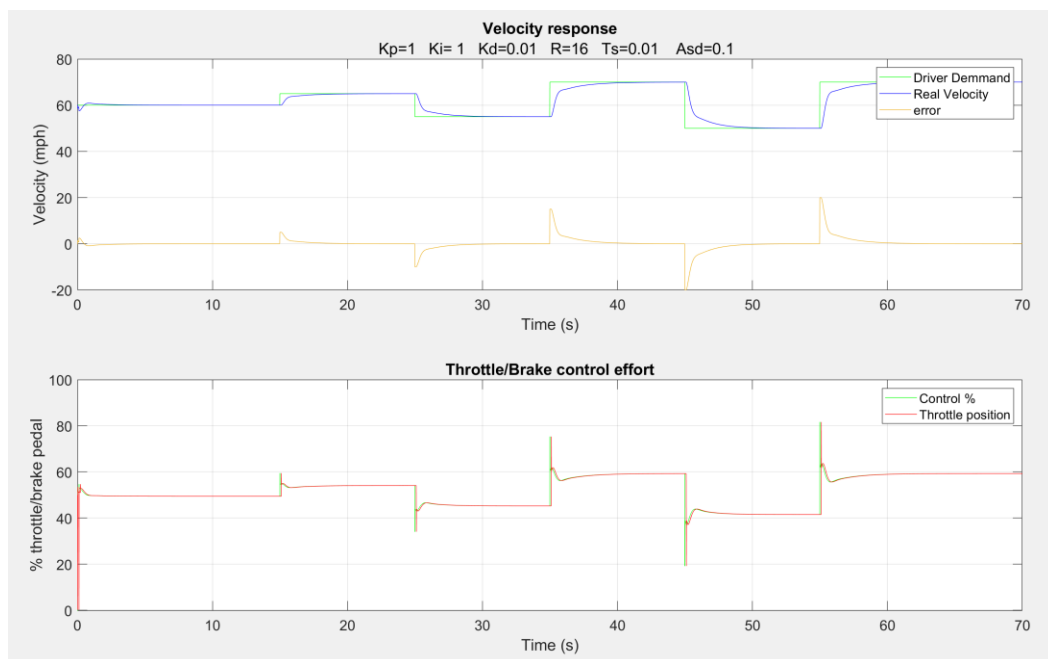


Figure 8-Matlab actuator signal delay set 0.1 (Purnell-Whitehead, 2025)

≥ 0.3 s delay- Caused degradation, including oscillations, overshoot, slow convergence, and in some cases, instability.

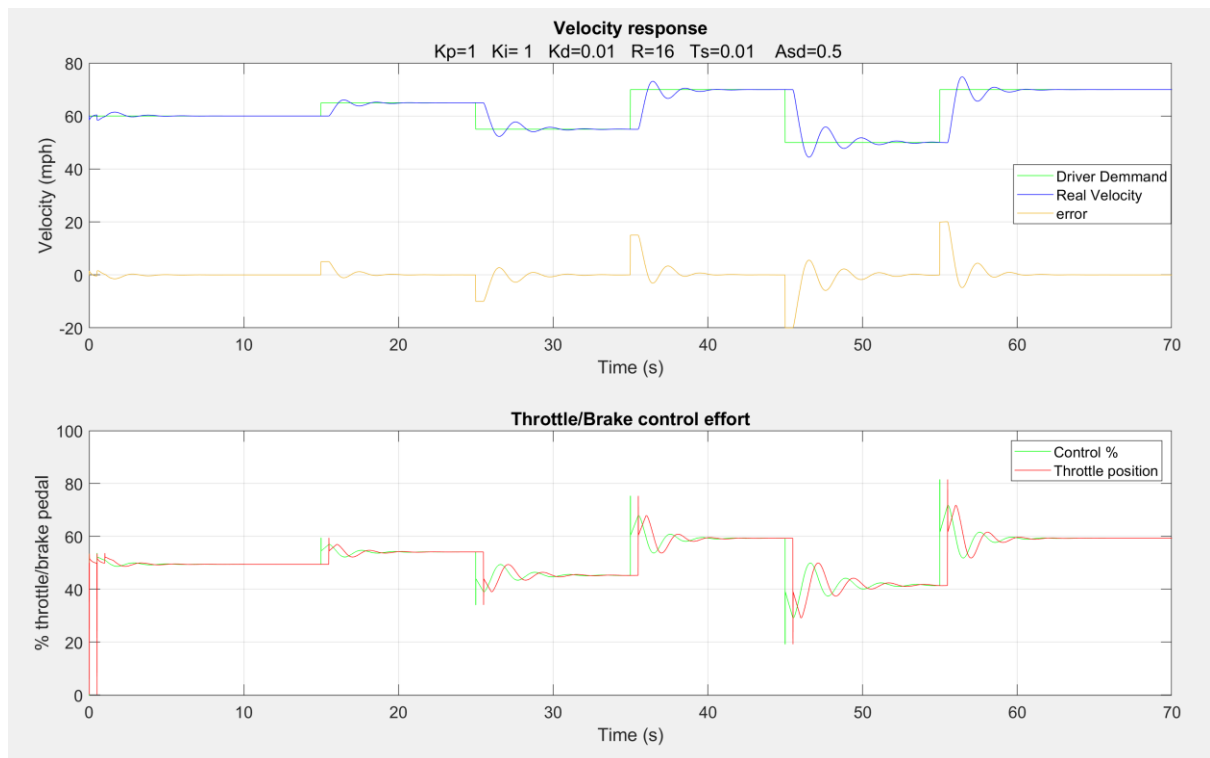


Figure 9-Matlab actuator signal delay set at 0.5 (Purnell-Whitehead, 2025)

6 Number of Bits impact

6.1 Method

To assess this, the following steps were taken:

- Encoding and decoding functions were implemented in MATLAB to simulate quantisation of the demand, sensor feedback, and control signal.
- Simulations were run using different bit resolutions: 4-bit, 8-bit, and 16-bit.
- All other parameters, including PID gains and sampling time, were kept constant.
- Key performance metrics (overshoot, steady-state error, rise time) were recorded to evaluate the effect of reduced resolution.

The aim was to identify the minimum number of bits that still maintains acceptable system performance.

6.2 Results and analysis

The simulation results demonstrated that bit resolution has a direct impact on control accuracy and system stability:

Higher resolutions (16 bits)

Provided smooth and precise control. The system exhibited minimal steady-state error and stable step responses.

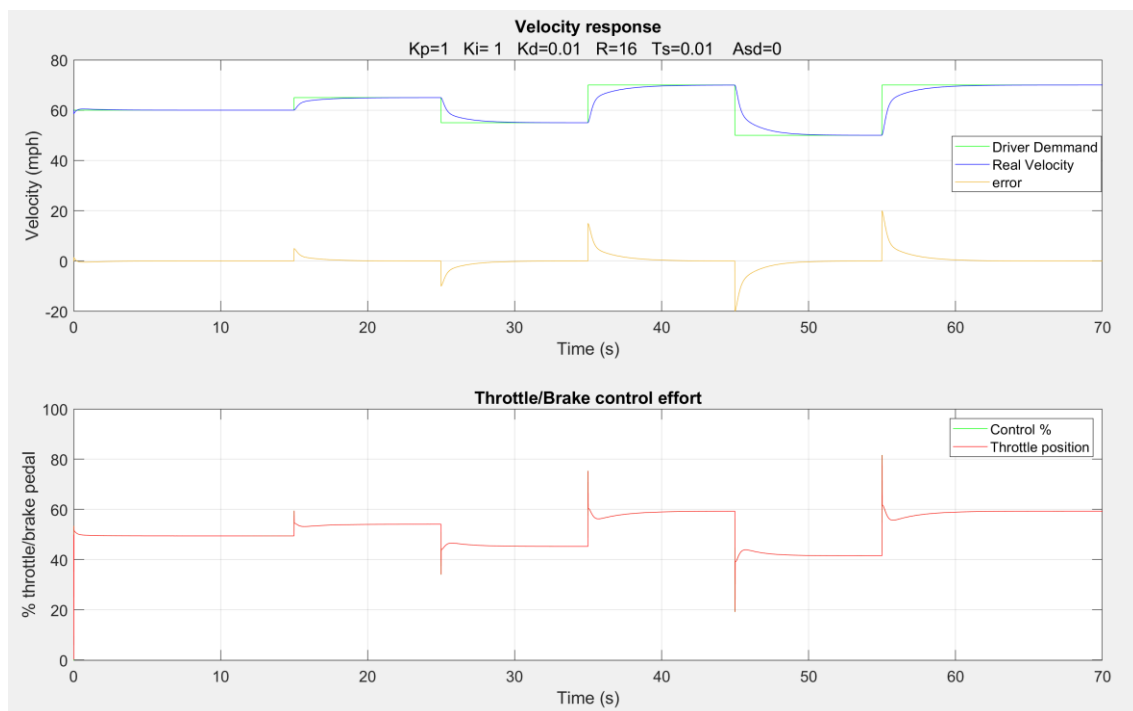


Figure 10-Matlab results shown using 16-bit resolution (Purnell-Whitehead, 2025)

Moderate resolutions (8 bits)

Showed slight degradation, with noticeable quantisation steps in control effort.

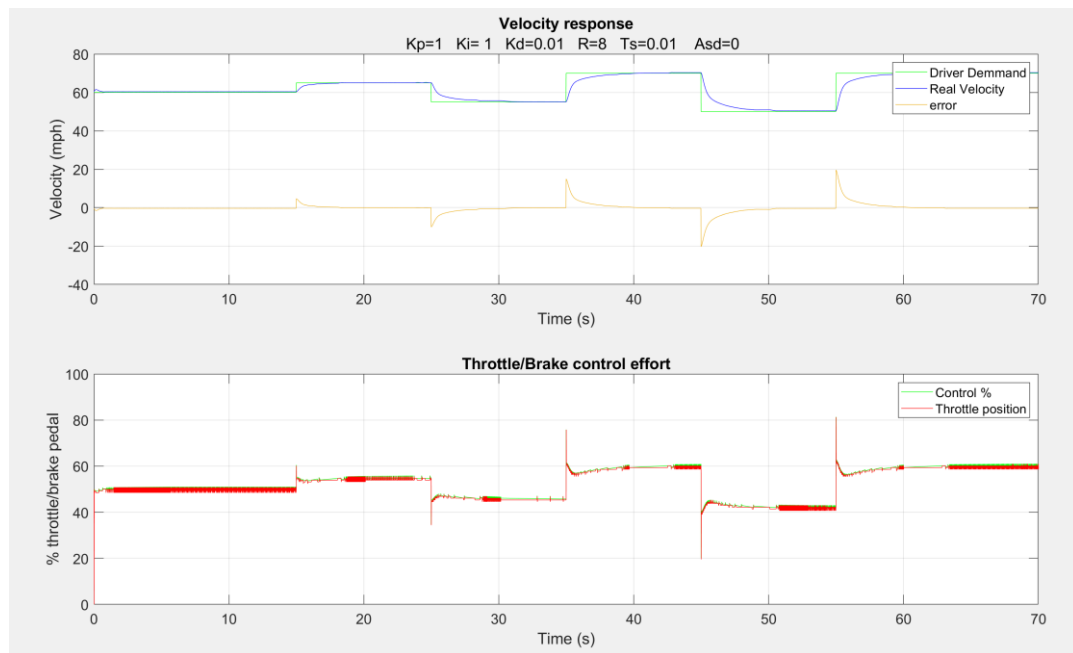


Figure 11-Matlab results shown using 8-bit resolution (Purnell-Whitehead, 2025)

Low resolutions (4 bits)

Caused significant quantisation noise, resulting in choppy control signals, increased steady-state error, and in some cases, oscillatory or unstable behaviour.

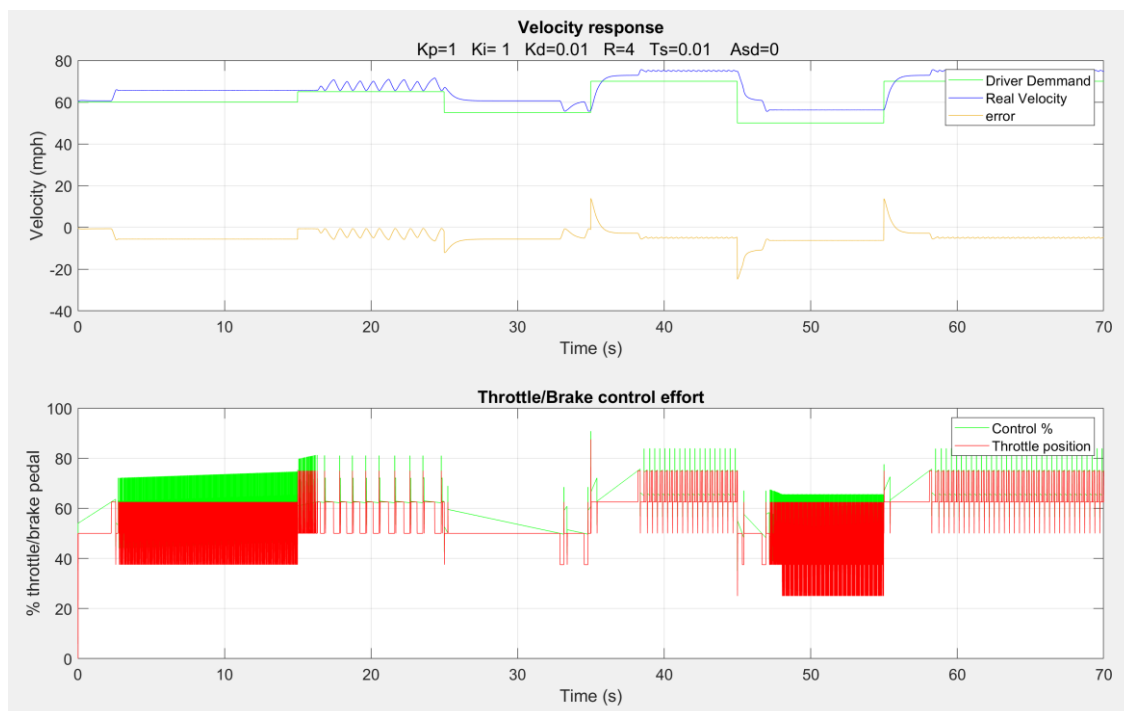


Figure 12-Matlab results shown using 4-bit resolution (Purnell-Whitehead, 2025)

7 Conclusion

This report has successfully modelled and evaluated a digital PID controller for a cruise control system using MATLAB. Through systematic analysis of PID gain tuning, sampling time, signal delay, and bit resolution, the study has demonstrated how each factor critically influences system performance, stability, and responsiveness. Key findings include the importance of balanced PID parameters, the need for appropriate sampling intervals to avoid instability, the significant performance degradation caused by excessive signal delay, and the detrimental effects of low-resolution signal encoding.

Optimal controller performance was achieved with carefully tuned PID gains ($K_p = 4.3$, $K_i = 1$, $K_d = 0.01$), a sampling interval of 0.01 seconds, a bit resolution of at least 16 bits and 0. The results underscore the necessity of considering real-world constraints such as processor capability and communication latency in the design of digital control systems. Overall, this project has reinforced the theoretical foundations of PID control and highlighted practical considerations for implementing such systems in automotive applications, particularly in achieving smooth, reliable, and safe cruise control functionality.

```

10  clc
11  clear
12  close all % House keeping
13  P = 1; % Proportional gain
14  I = 1; % Integral gain
15  D = 0.01; % Derivative gain
16  %%%
17  %% (2) set implementation parameters %%%
18
19  % sampling time must be an integer multiple of sim_step
20  Ts = 0.01;
21  % number of bits used to encode velocity sensor signal. velocity sensor
22  n_bits_sensor_signal = 16;
23  % range in mph [0,150]
24  m_velocityS = 0;
25  M_velocityS = 150;
26  % number of bits used to encode velocity demand signal. demand
27  % range in mph [30,110]
28  m_demand = 30; M_demand = 110;
29  n_bits_demand_signal = 16;
30  % number of bits used to encode throttle/brake actuator signal. actuator
31  % range in mph [-100,100] where [-100,0) is percentage of braking and
32  % (0,100] is percentage of throttle
33  n_bits_actuator_signal = 16;
34  R = 2.^n_bits_actuator_signal;
35  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
36  %% (3) set simulation parameters %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
37
38  sim_step = 0.01; % simulation time step in seconds
39  sim_time = 70; % simulation time length in seconds
40  T = 0:sim_step:sim_time; % simulation steps vector
41  % initialize vector where to store results for plots
42  [d_vec,u_vec ,y_vec,e_vec,Th_vec] = deal(nan(size(T)));
43  throttle_setting = 50; % initialize control action (throttle/brake setting)
44  % communication delay between controller and actuator in seconds.
45  % must be an integer multiple of sim_step
46  actuator_signal_delay = 0.0;

```

Figure 13-Matlab code with final settings (Purnell-Whitehead, 2025)

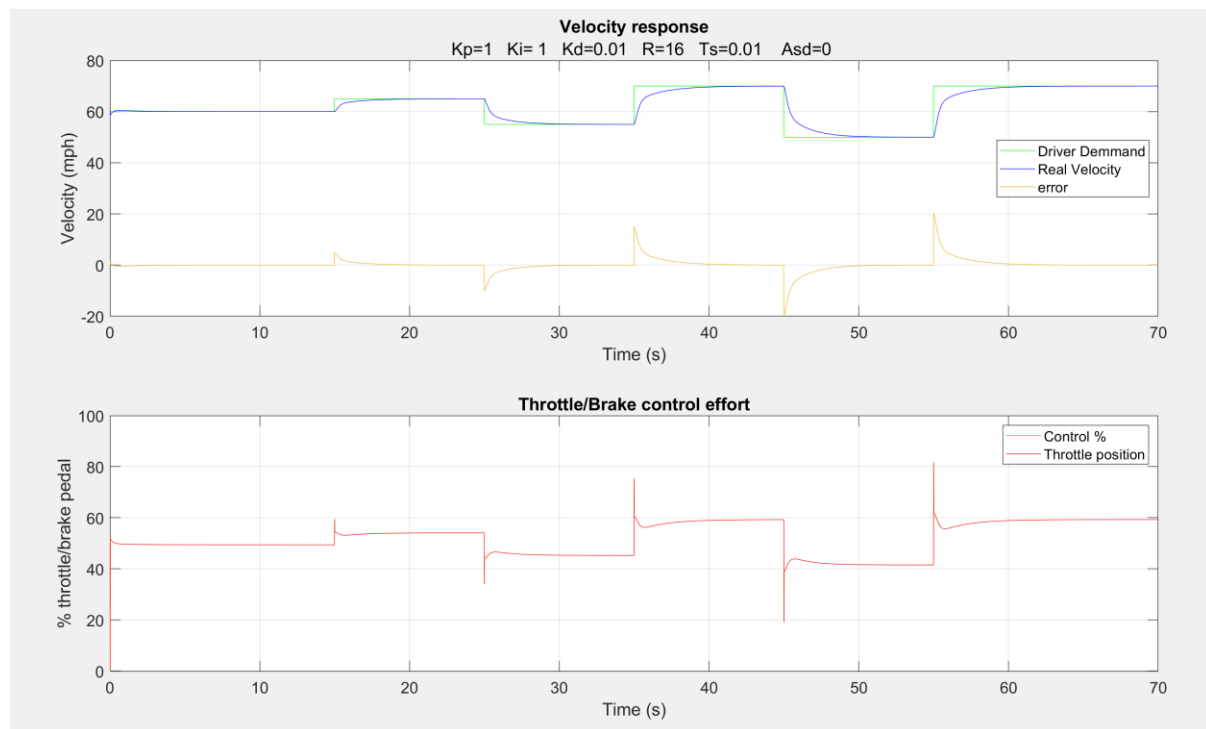


Figure 14-Matlab final settings (Purnell-Whitehead, 2025)

7.1 Velocity Conversion and Motion Analysis in Cruise Control Systems

In this study, vehicle speed is initially expressed in miles per hour (mph), which aligns with standard vehicle sensors and driver input. However, for accurate analysis in control engineering, speeds are converted to meters per second (m/s), the SI unit used in physics-based models and motion equations (e.g., $F = ma$). This ensures consistency when calculating acceleration and jerk, both of which depend on velocity changes over time.

```

89      %% Velocity m/s,Acceleration and Jerk%%
90
91      vms_vec = y_vec.*0.44704; % Converts mph to m/s
92      a_vec = diff(vms_vec)./sim_step; % Formula to calculate acceleration as derivative of velocity m/s^2
93      j_vec = diff(a_vec)./sim_step; %Formula to calculate jerk as a derivative off acceleration m/s^3
94      T_v = T; % Velocity time (same as original)
95      T_a = T(1:end-1); % Acceleration time vector (length of a_vec)
96      T_j = T(1:end-2); % Jerk time vector (length of j_vec)
97      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
98

```

Figure 15-Matlab code calculating mph to m/s, acceleration and jerk (Purnell-Whitehead, 2025)

Acceleration, defined as the rate of change of velocity, is a key indicator of system performance. It provides insight into how smoothly the vehicle speeds up or slows down, directly impacting passenger comfort and safety. Excessive or erratic acceleration may indicate poor controller tuning.

Jerk, the rate of change of acceleration, is equally important, particularly for evaluating ride comfort and mechanical wear. High jerk values often result in harsh, uncomfortable driving experiences and can stress vehicle components. Monitoring jerk helps identify overly aggressive control actions, often caused by excessive derivative gain in the PID controller.

Together, velocity (in m/s), acceleration, and jerk offer a comprehensive view of control system behaviour, supporting further refinement and validation of cruise control algorithms.

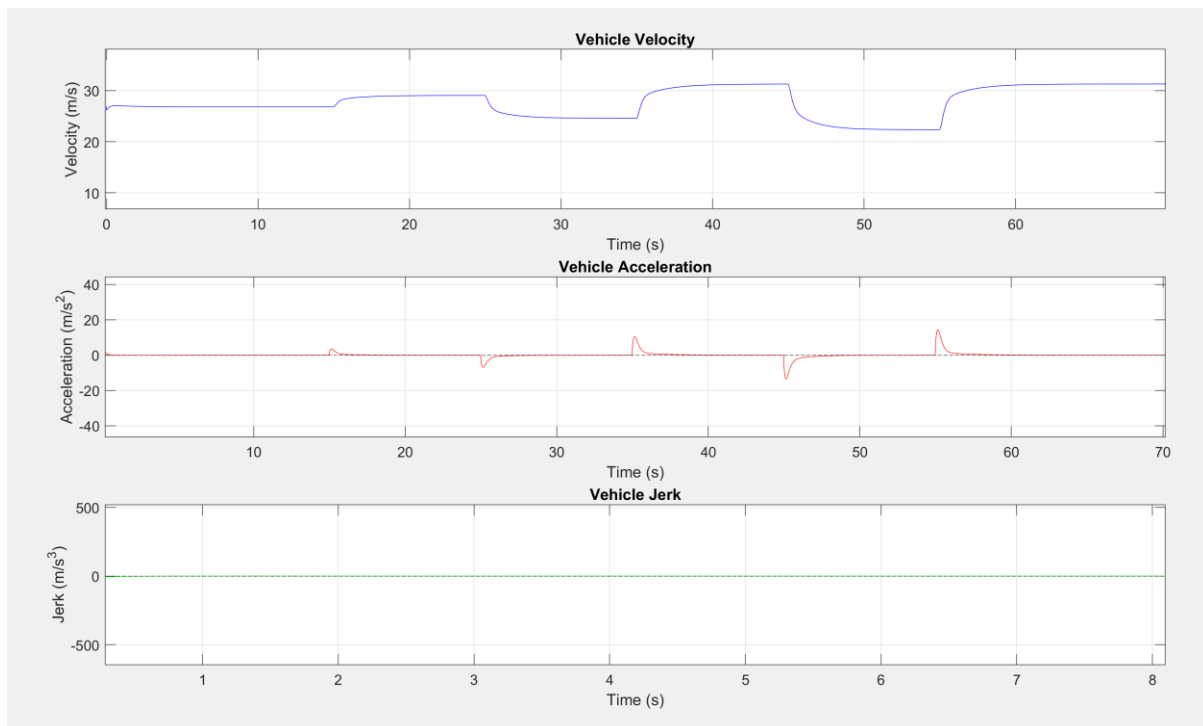


Figure 16-Matlab results of velocity(m/s), Vehicle acceleration and Vehicle jerk (Purnell-Whitehead, 2025)

8 Bibliography

- Åström, K. & Hägglund, T., 1995. *Advanced PID Control*. Research Triangle Park, NC: ISA – The Instrumentation, Systems, and Automation Society..
- Dorf, R. & Bishop, R. H., 2011. *Modern Control Systems*. New Jersey: Pearson.
- Fernández de Cañete, J. & Galindo, C., 2011. Introduction to Control Systems. In: *System Engineering and Automation*. Berlin: Springer, pp. 137-165.
- International Organization for Standardization, 2018. *ISO 15622:2018(en)*. Geneva: ISO.
- International Organization for Standardization, 2018. *ISO 26262-6:2018(en)*. Geneva: ISO.
- Ismail, A. & Hamza, M., 2015. *Effect of sampling time on PID controller design for a heat exchanger system*, s.l.: ICAST.
- Joseph, E. A. & O, O. O., 2017. Cohen-coon PID Tuning Method; A Better Option to Ziegler. *International Journal of Recent Engineering Research and Development (IJRERD)*, pp. 141-145.
- Luo, Y., Saberi, A. & Van den Brand, M., 2019. Safety-Driven Development and ISO 26262. In: *Automotive Systems and Software Engineering*. s.l.:Springer, pp. 225-254.
- Mathworks, 2025. *PID tuning*. [Online]
Available at: <https://www.mathworks.com/help/control/ref/pidtuner-app.html>
- Purnell-Whitehead, S., 2025. *Matlab PID tuning*. Vale of Glamorgan: UWTSD.
- Rahnama, A., Xia, M. & Antsaklis, P. J., 2016. *A Passivity-Based Design for Stability and Robustness in Event-Triggered Networked Control Systems with Communication Delays, Signal Quantizations and Packet Dropouts*, Notre Dame: University of Notre Dame.
- UNECE, 2008. *UN R13*. Geneva: UNECE.
- UNECE, 2023. *UN Regulation No. 79 Revision 5*. Geneva: UNECE.
- Ziegler, J. G. & Nichols, J. G., 1993. Optimum Settings for Automatic Controllers. *Dynamic systems, measurement and control*, pp. 759-765.

9 Code

9.1 Control function

```
function [controlout, error] = Controllerzz...
    (demand,velocity,e_vec,j,Ts,sim_step,control,P,I,D)
%This is a PID controller
% controlout = control signal
% error = difference between demand and velocity (phys (mph))
error = demand-velocity;
%% P part
P_part = error*P;
%% I part
persistent Prev_I_part
if j>1
    prev = e_vec(j-(Ts/sim_step));
else
    prev=control;
    Prev_I_part = prev;
end
I_part = (((error+prev)./2).*(Ts)+Prev_I_part).*I;
Prev_I_part = I_part; % Insert ControlOut Checks (max& min)
%% D part
D_part = ((error-prev)/Ts)*D;
%% output
controlout = P_part+I_part+D_part;
%% Limits
m_ControlSig = -100;
M_ControlSig = 100;
if controlout<m_ControlSig
    controlout=m_ControlSig;
end
if controlout>M_ControlSig
    controlout=M_ControlSig;
end
end
```

9.2 Cruise control script

```
% Shea Purnell-Whitehead (2212438) ACMA 6001-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 = 1; % Proportional gain
I = 1; % Integral gain
D = 0.01; % Derivative gain
%%%
%% (2) set implementation parameters %%
% sampling time must be an integer multiple of sim_step
Ts = 0.01;
% number of bits used to encode velocity sensor signal. velocity sensor
```

```

n_bits_sensor_signal = 16;
% range in mph [0,150]
m_velocityS = 0;
M_velocityS = 150;
% 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 = 0.0;
%% 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; % DECODING
FUNCTION

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

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

```

```

        %% 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

%% Velocity m/s,Acceleration and Jerk%%

vms_vec = y_vec.*0.44704; % Converts mph to m/s
a_vec = diff(vms_vec)./sim_step; % Formula to calculate acceleration as derivative
of velocity m/s^2
j_vec = diff(a_vec)./sim_step; %Formula to calculate jerk as a derivative off
acceleration m/s^3
T_v = T; % Velocity time (same as original)
T_a = T(1:end-1); % Acceleration time vector (length of a_vec)
T_j = T(1:end-2); % Jerk time vector (length of j_vec)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% (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), ' R=',num2str(n_bits_actuator_signal), ' Ts=',num2str(Ts), '
Asd=',num2str(actuator_signal_delay)]);
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');
supersize(1.5);

figure

% Velocity
subplot(3,1,1)
plot(T_v, vms_vec, 'b'); grid on; hold on

```

```
ylabel(0,'k--');
title('Vehicle Velocity')
ylabel('Velocity (m/s)')
xlabel('Time (s)')

% Acceleration
subplot(3,1,2)
plot(T_a, a_vec, 'r'); grid on; hold on
ylabel(0,'k--');
title('Vehicle Acceleration')
ylabel('Acceleration (m/s^2)')
xlabel('Time (s)')

% Jerk
subplot(3,1,3)
plot(T_j, j_vec, 'g'); grid on; hold on
ylabel(0,'k--');
title('Vehicle Jerk')
ylabel('Jerk (m/s^3)')
xlabel('Time (s)')
supersizeme(1.5);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```