

University of Wales Trinity Saint David
Wales Institute of Science & Art



Modelling Automotive Control Systems in MATLAB – Task 1

Hybrid Powertrain and Control ACEB7007

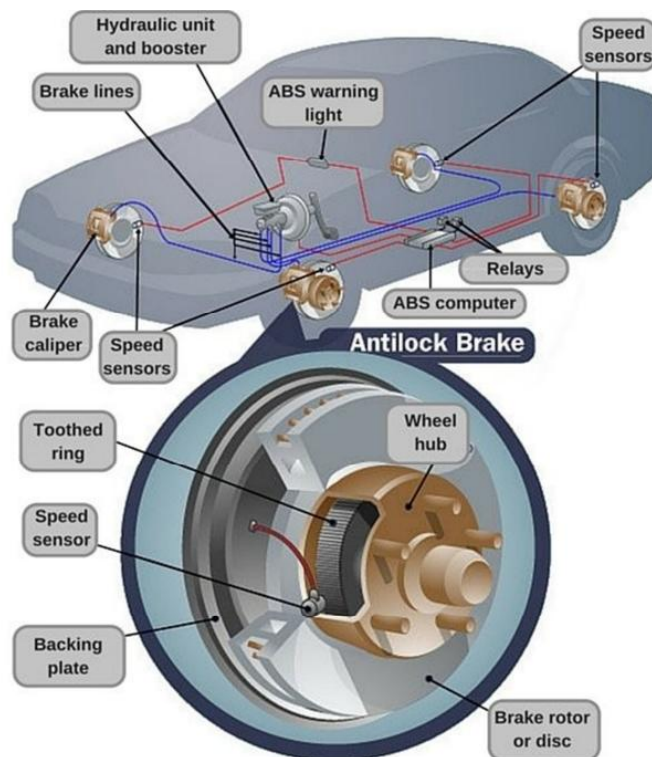


Figure 1 Complete overview of the Anti-lock Braking System (ABS) in a road car (Bhasin, 2019)

Diego Alexander Valencia Flores
2405760
March 30, 2025

Contents

1	Introduction	3
1.1	Selection of the control system	3
2	Anti-Lock Braking System.....	3
3	Algorithms and Communication Protocols	6
4	Different Software/Tools available for development	8
4.1	Possible Enhancements to the control system.....	10
5	Control System – PID Cruise Control.....	11
5.1	PID	11
5.2	Comfort Acceleration – ISO 15622	12
5.3	Control System – PID Cruise Control.....	13
5.3.1	Initialization	13
5.3.2	Set Implementation Parameters	14
5.3.3	Set Simulation Parameters	15
5.3.4	Simulation Loop	16
5.4	Conclusion– PID Cruise Control	21
5.	Reference list.....	22

1 Introduction

1.1 Selection of the control system

Anti-Lock Braking System (ABS) is a safety-critical control system in modern vehicles which prevents brakes from locking during braking situations while maintaining vehicle steerability (Bhasin, 2019).

2 Anti-Lock Braking System

The Anti-Lock Braking System (ABS) is highly relevant for research because it fundamentally improves vehicle safety and control under critical braking conditions. By modulating brake line pressure independently of the driver's pedal force, ABS ensures that wheel speed returns to an optimal slip range, preventing full wheel lock and minimizing the likelihood of dangerous skidding. This feature is especially critical on roads with low grip—such as those covered by rain, snow, or mud—because it allows the driver to maintain better steering control and stability (Jitesh, 2014).

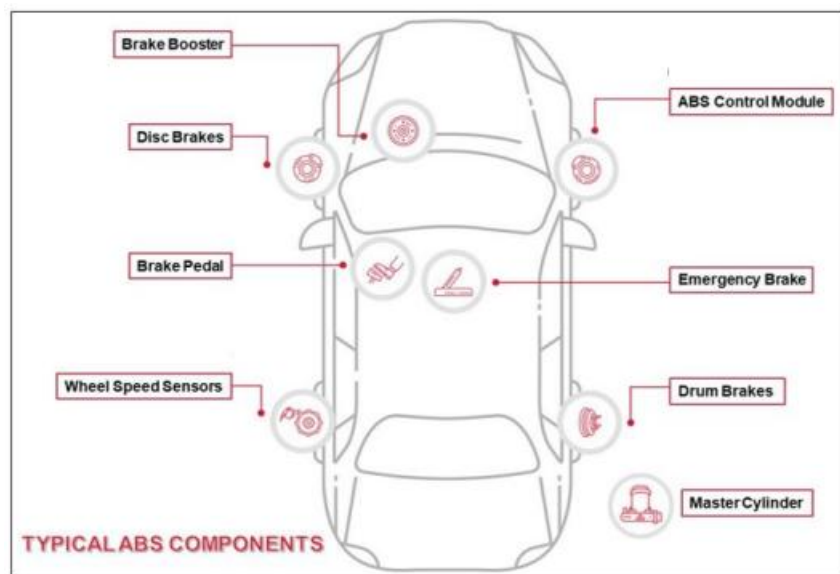


Figure 2 The main ABS Components (Abd El-Fatah et al., 2021)

To be precise one of the most critical inputs for ABS is the driver's braking command, typically gauged through the force applied to the brake pedal or, in some vehicles, through a dedicated brake pedal position sensor. This input is combined with continuous feedback from the vehicle's wheel speed sensors, which generate signals corresponding to each wheel's rotational velocity. These wheel speed signals are vital for detecting when a wheel is about to lock, as they allow the Electronic Control Unit (ECU) to calculate slip ratio—how fast the wheel is rotating compared to the vehicle's overall speed (Aly et al., 2011).

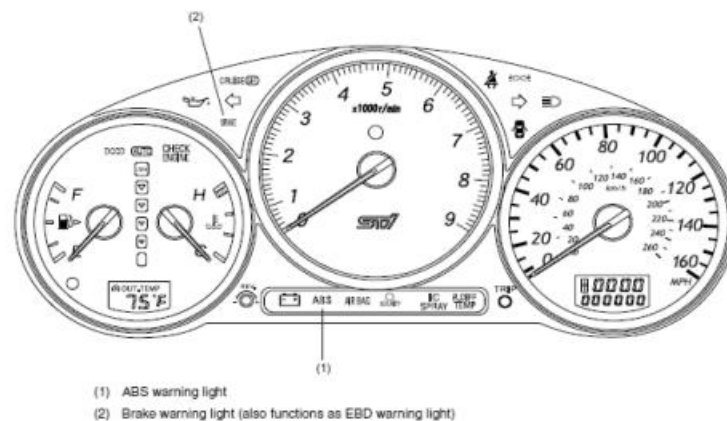
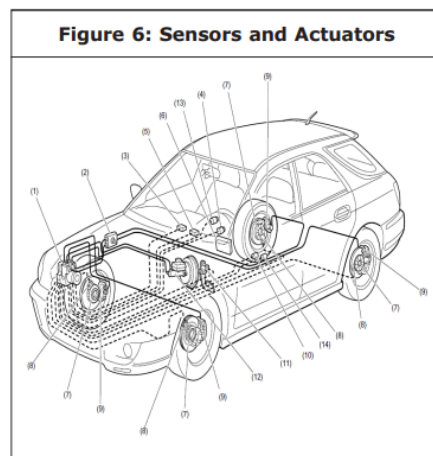


Figure 3 Car Dashboard Indicating ABS (Jitesh, 2014)

The main outputs of the ABS consist of commands to adjust and modulate brake fluid pressure in each wheel circuit. When the system detects excessive slip or an impending lock-up, it instructs the hydraulic modulator to reduce pressure. Once the wheel regains sufficient rotational speed, the ABS output shifts to reapply or hold brake pressure, stabilizing the vehicle and maintaining traction. In addition, there may be status outputs to alert the driver—such as warning lights in the instrument panel / dashboard—indicating when the ABS is actively regulating pressure or if a fault is detected in the system (Abd El-Fatah et al., 2021).

Actuators in the ABS are centered around the hydraulic modulator, which often contains multiple solenoid valves and an electric pump. These valves rapidly open and close in response to ECU commands, allowing brake fluid pressure to be “dumped” or held before being reapplied. The electric pump then replenishes pressure in the brake lines as needed. Together, these components provide fine control over how much braking force is exerted on each wheel, independent of the driver’s steady or forceful pedal input. This mechanism is essential for maintaining optimal slip levels and preventing lock-up under diverse road and driving conditions (Zhu et al., 2022).

FUNCTION OF SENSORS AND ACTUATORS



- (1) ABS control module and hydraulic control unit (ABSCM & H/U)
- (2) Two-way connector
- (3) Diagnosis connector
- (4) ABS warning light
- (5) Data link connector (for SUBARU select monitor)
- (6) Transmission control module (AT models only)
- (7) Tone wheels
- (8) ABS wheel speed sensor
- (9) Wheel cylinder
- (10) G sensor
- (11) Stop light switch
- (12) Master cylinder
- (13) Brake & EBD warning light
- (14) Lateral G sensor (STi)

Figure 4 Sensors and Actuators from a SUBARU (Jitesh, 2014)

Sensors form the backbone of the ABS’s closed-loop control strategy. Wheel speed sensors—commonly magnetic or Hall-effect types—generate electrical pulses directly proportional to wheel rotation, offering real-time data on when a wheel might be decelerating abnormally. Some advanced systems also integrate brake pressure sensors to measure the actual hydraulic pressure in each brake circuit, providing the ECU with additional insight for more refined pressure control. In certain vehicles, especially those that combine ABS with other stability features, the system may also incorporate yaw rate sensors and lateral accelerometers, allowing broader vehicle stability systems to coordinate braking across all four wheels in conjunction with steering inputs (Abd El-Fatah et al., 2021).

3 Algorithms and Communication Protocols

A more advanced ABS control strategy, as illustrated in Figure 7 from the paper “Surface Prediction and Control Algorithms for Anti-Lock Brake System,” involves two interlinked loops: a prediction loop and a control loop. The prediction loop’s primary goal is to converge on a desired slip ratio—often denoted λ_{max} —by gathering data under various road surface conditions. During simulations, this loop uses acceleration values obtained from a vehicle dynamics model to refine estimates of the optimal slip ratio. The system initially runs the prediction loop; if the slip ratio error falls below a predefined threshold, the algorithm switches to the control loop to regulate brake pressure around the newly predicted λ_{max} (Bhandari, Patil and Singh, 2012).

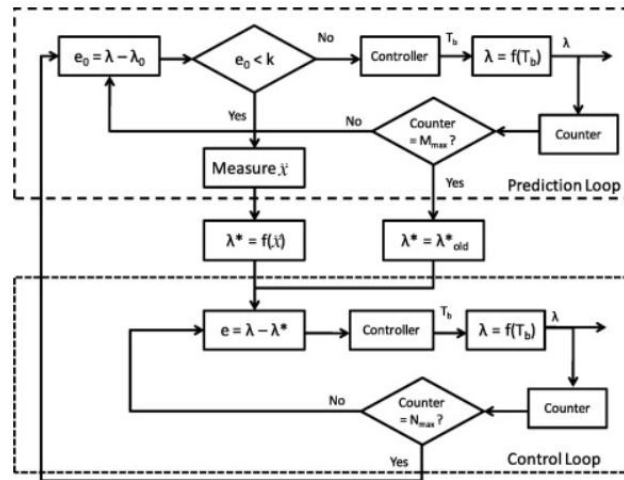


Figure 5 Prediction and control Loop (Bhandari, Patil and Singh, 2012)

Otherwise, the prediction loop continues for a set number of cycles to further refine its surface estimate (Bhandari, Patil and Singh, 2012). Once the surface is characterized, the algorithm updates λ_{max} and hands off to the control loop. In the control loop, the ABS continuously compares the measured slip ratio (λ) against this updated set point (λ_{max}), modulating brake pressure to maintain the slip within acceptable bounds and prevent lock-up. This two-stage (prediction and control) framework allows the ABS to adapt more quickly to changing road surfaces, thereby optimizing braking performance and improving overall vehicle stability (Anwar and Ashrafi, 2002).

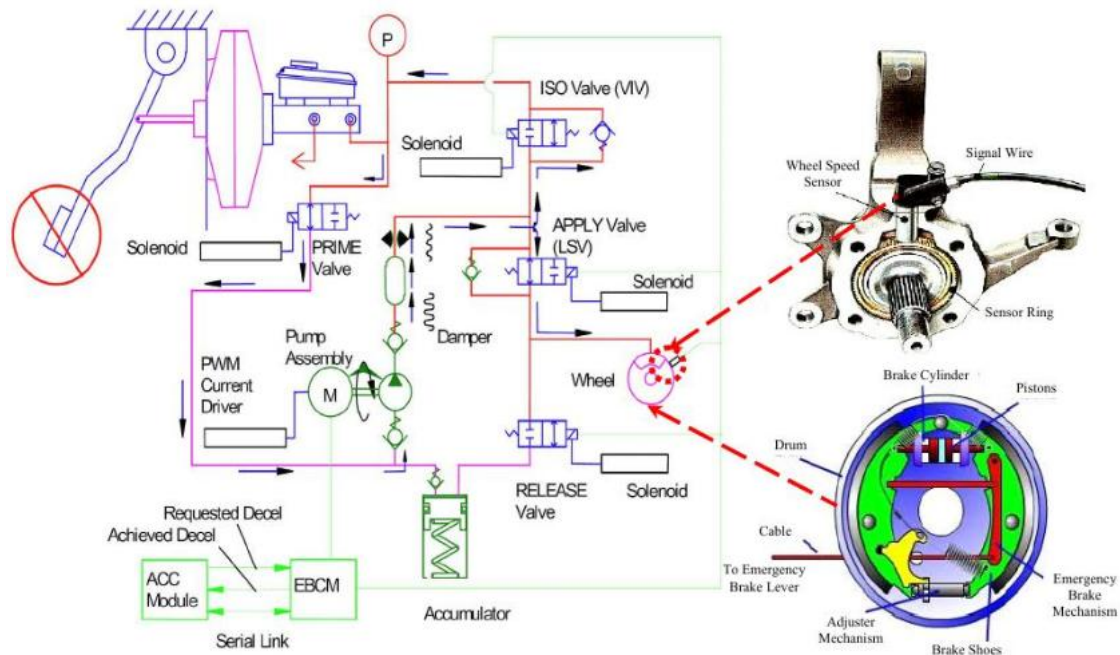


Figure 6 Anti-lock Braking System Communication Protocols (Aly et al., 2011)

In most modern vehicles, the Anti-Lock Braking System (ABS) communicates with other electronic control units (ECUs)—such as those managing traction control or electronic stability control (ESC)—through established automotive networks. Controller Area Network (CAN) is the de facto standard for transmitting critical real-time data, including wheel speed information, brake status, and fault codes. CAN's error-handling mechanisms and deterministic message scheduling make it especially suitable for safety-critical subsystems, ensuring minimal latency and robust fault tolerance. Some high-end or more advanced vehicles may use FlexRay or Automotive Ethernet to handle large volumes of data at higher transmission rates, though CAN remains most prevalent due to its maturity, widespread adoption, and proven reliability in ABS applications (Aly et al., 2011).

4 Different Software/Tools available for development

Modern anti-lock braking system (ABS) development relies on a suite of software tools that support model-based design, simulation, code implementation, and testing. At the core, MATLAB/Simulink is used to model the ABS control algorithms and vehicle dynamics. It is possible to build block diagrams of the braking control loop (wheel speed, slip calculation, brake pressure modulation) and can include logic using Stateflow state machines for events like wheel lock detection and valve control (Gamesh and Krishna, 2018).

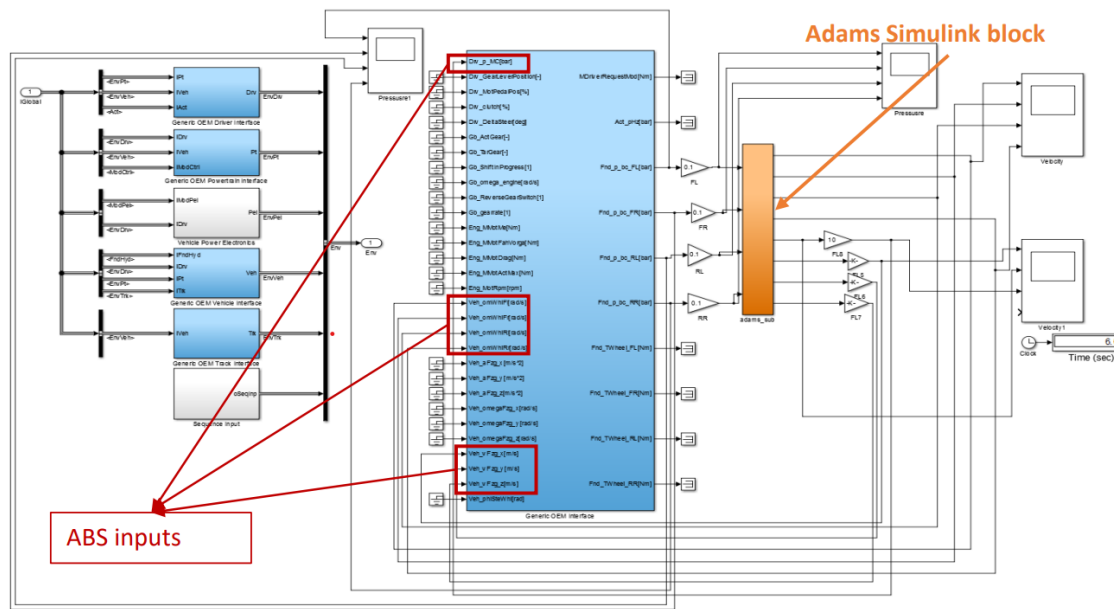


Figure 7 Co-Simulation Setup for ABS | Inputs from ADAMS/car | Output from Simulink ABS (Gamesh and Krishna, 2018)

Simulink models can be co-simulated with high-fidelity vehicle dynamics packages like CarSim or IPG CarMaker to provide realistic tire-road interaction. A co-simulation platform combining CarSim's vehicle model with a Simulink ABS controller has been shown to accurately reproduce ABS behavior across different speeds and road surfaces, validating that the control logic will perform as expected in a variety of scenarios (Geleta, Gopal and Lemi, 2023).

Once the control logic is refined in simulation—often through Model-in-the-Loop (MIL) testing—developers move to implementation. Using Simulink’s auto-code generation or embedded C/C++ coding, they convert the model into ECU software, frequently adhering to AUTOSAR, which provides standardized interfaces for seamless integration with other vehicle systems (SRM Tech, 2024). Tools like MATLAB/Simulink support direct AUTOSAR exports; alternatively, developers incorporate generated C code into a larger brake controller ECU.

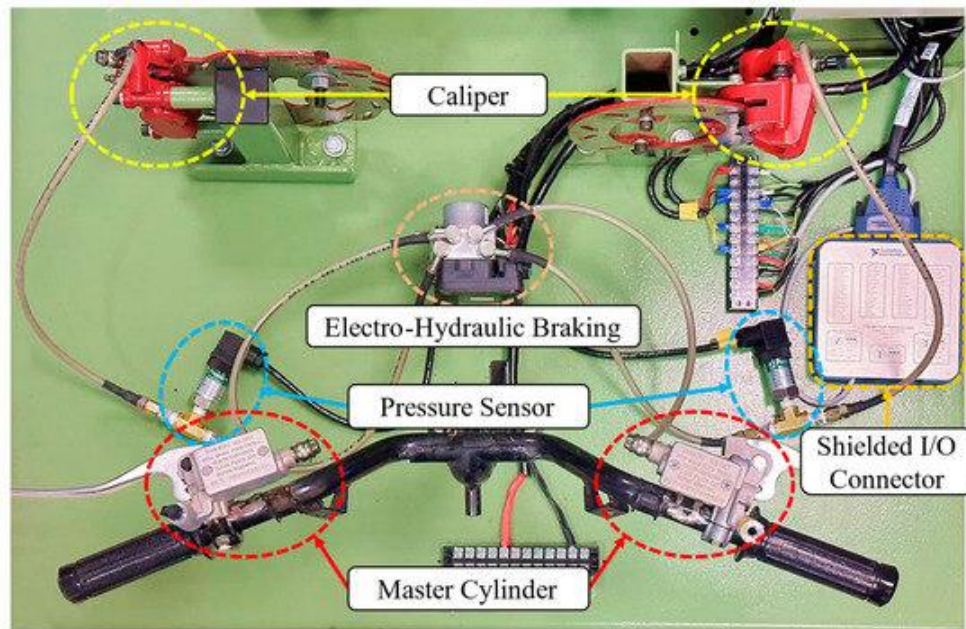


Figure 8 Motorcycle Hardware-in-the-loop (HIL) braking experimentation platform (Hsu, Chen and Zhang, 2024)

Next, Hardware-in-the-Loop (HIL) testing validates the software in real time. dSPACE or NI platforms run a detailed vehicle model (e.g., CarSim) that sends realistic wheel-speed and sensor signals to the ABS controller, while monitoring the controller’s actuator outputs (valves, pump) (dSPACE, 2025). High-fidelity benches may include physical brake components and motors to replicate actual loads. Throughout development, data logging and calibration tools (Vector CANape, ETAS INCA) capture key signals and enable parameter tuning in real time. This workflow ensures confidence in ABS performance and safety, refining the system until it reliably prevents lock-up and maintains stability under varied conditions (Hu, 2025).

4.1 Possible Enhancements to the control system

Predictive or adaptive ABS strategies aim to anticipate road conditions and dynamically adjust braking torque before traction is lost. Unlike conventional threshold-based approaches, these methods integrate real-time surface estimation models that gauge friction coefficients under varied circumstances—dry asphalt, ice, wet cobblestone. By identifying surface characteristics early, the controller modifies wheel slip targets and brake pressure to maximize deceleration safely.

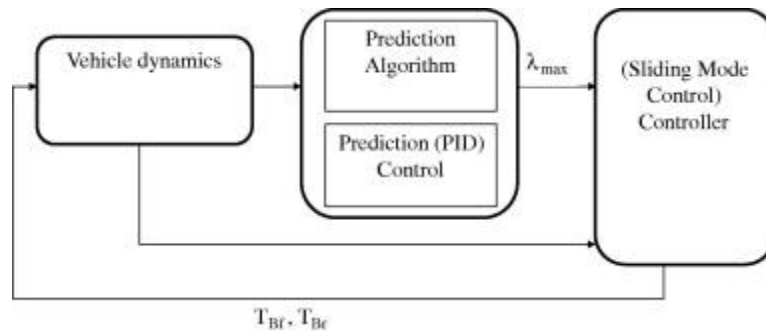


Figure 9 Flow chart of representing prediction and control loop (Bhandari, Patil and Singh, 2012).

Such solutions often couple a half car model for high-speed braking with slip ratio prediction, ensuring minimal stopping distance while maintaining directional control. Comparative studies have shown that when surface identification is combined with sliding mode or PID-based control, predicted slip ratios yield improved braking force utilization, thus reducing stopping distances and preserving stability across diverse surfaces (Bhandari, Patil and Singh, 2012).

5 Control System – PID Cruise Control

5.1 PID

Proportional-Integral-Derivative (PID) control remains the most commonly implemented strategy in modern control systems. It is estimated that over 90% of industrial control loops utilize some form of PID control. Throughout the last five decades, extensive academic and industrial research has been devoted to enhancing PID control—particularly in areas such as tuning methodologies, system identification techniques, and adaptive control strategies (Knospe, 2006).

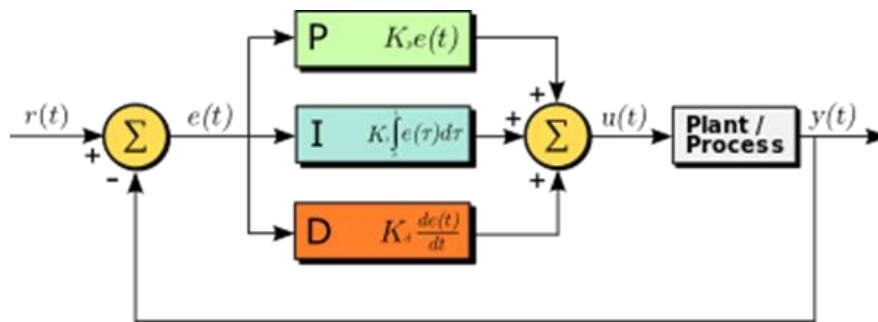


Figure 10 PID close loop representation (info.erdosmiller.com, n.d.)

The effectiveness of PID control stems from the distinct roles of its three terms, which collectively address the core requirements of most control applications. The integral term is crucial for eliminating steady-state error when tracking a constant setpoint, a principle underpinned by the final value theorem and the internal model principle. It also plays a key role in rejecting constant disturbances. However, integral action can be sluggish in responding to real-time error and may introduce lag. In contrast, the proportional term provides an immediate response to the current error, making it effective for rapid correction, although it often cannot fully eliminate steady-state error on its own without using excessively high gain values. The derivative term, when included, contributes predictive action by responding to the rate of change of the error, thereby improving system stability and damping (Li, Kiam Heong Ang and Chong, 2006).

5.2 Comfort Acceleration – ISO 15622

Cruise control systems, especially in the context of passenger vehicles, ride comfort becomes a primary factor influencing the user's driving experience and satisfaction. Unlike race cars, where maximum performance typically takes precedence over comfort, leading to aggressive accelerations, sudden decelerations, and significant vibrations—road vehicles must prioritize smoothness and ride quality.

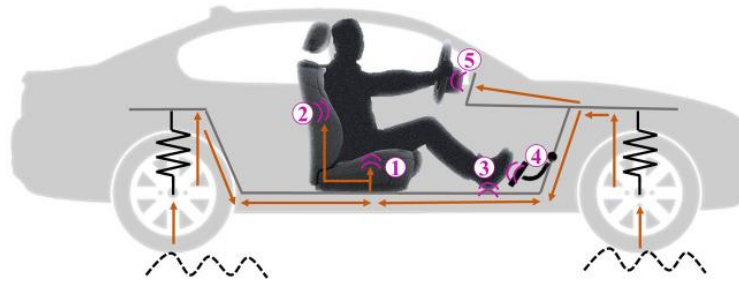


Figure 11 Induced vibrations due to bumpy road - Comfortability aspect (Deubel, Ernst and Prokop, 2023)

A comfortable ride ensures passengers feel relaxed, secure, and content, especially during longer journeys (Gebhard, 1970). For instance, the ISO standard 15622, which covers Full Speed Range Adaptive Cruise Control systems, suggests longitudinal acceleration limits typically ranging from $+4 \text{ m/s}^2$ (approximately $+0.41 \text{ g}$) to -5 m/s^2 (approximately -0.51 g), however recent years it changed to preferred comfort levels closer to $+2 \text{ m/s}^2$ (approximately $+0.20 \text{ g}$) to -3.5 m/s^2 (approximately -0.36 g) (de Winkel et al., 2023).

These acceleration thresholds aim to ensure passenger comfort and safety during automated driving operations as well as looking for vehicle stability (Arvind Desikan and Kalaichelvi V, 2015).

5.3 Control System – PID Cruise Control

5.3.1 Initialization

The first section of the MATLAB code serves as housekeeping, preparing a clean workspace to ensure consistent and error-free execution of the cruise control system simulation.

```
clc;  
clear;  
close all;  
  
%% (1) Initialization  
% PID gains and sampling time  
P = 0.05;  
I = 10;  
D = 0.001;  
Ts = 0.005; % Sampling time for control update [s]
```

Figure 12 House Keeping and P, I, D and Ts variables stated.

Subsequently, the critical variables for the PID controller—specifically the proportional gain (P), integral gain (I), derivative gain (D), and the sampling time (Ts)—are initialized. The proportional gain determines how aggressively the controller responds to current errors, while the integral gain ensures zero steady-state error by integrating past errors. The derivative gain adds predictive capability, smoothing the response by considering the error rate. The sampling time (Ts) is set to 0.005 seconds, defining the frequency at which the PID controller updates its output commands to the actuator which would be the engine itself.

5.3.2 Set Implementation Parameters

These parameters include the simulation time step (sim_step), which is set to 0.01 seconds, ensuring sufficient temporal resolution for observing the dynamic response of the system clearly and accurately.

```
%% (2) Set Implementation Parameters
sim_step = 0.01;      % Simulation time step [s]
m_velocityS = 0;      % Minimum velocity sensor reading (mph)
M_velocityS = 150;    % Maximum velocity sensor reading (mph)
n_bits_sensor_signal = 16;
m_demand = 30;        % Minimum driver demand (mph)
M_demand = 110;       % Maximum driver demand (mph)
n_bits_demand_signal = 16;
n_bits_actuator_signal = 16; % Bits for actuator signal
R = 2^n_bits_actuator_signal; % Actuator scaling factor
```

Figure 13 Implementation Parameters

Additionally, the minimum (m_velocityS) and maximum (M_velocityS) velocity sensor readings are established, ranging from 0 mph (representing a stationary vehicle) to 150 mph (capturing typical upper operational limits of standard road vehicles). These values determine the operational range and resolution for encoding and decoding velocity sensor signals. Similarly, the range for the driver's demand (m_demand and M_demand) is set between 30 mph and 110 mph, reflecting realistic cruising speeds typically encountered on highways.

The code also specifies the resolution for digital signals by assigning the number of bits used for encoding sensor (n_bits_sensor_signal), driver demand (n_bits_demand_signal), and actuator signals (n_bits_actuator_signal), each configured at 16 bits. However, there are not going to be changes to these parameters as they do not improve the model, so staying at 16 bits would be the ideal scenario. Lower bit counts risk quantization errors, undermining control accuracy, while higher bit counts inflate complexity without notable gains. (Srivastava and Pandit, 2016).

5.3.3 Set Simulation Parameters

The overall conditions for running the cruise control simulation covers the total simulation duration (sim_time) which is set to 70 seconds, which provides a suitable interval for observing the system's dynamic behaviour. Additionally, the simulation time vector (T) ranges from 0 to 70 seconds, defined by sim_step, allowing initial conditions to stabilize before analysis.

```
%% (3) Set Simulation Parameters
sim_time = 70;           % Total simulation time [s]
T = -10:sim_step:sim_time; % Simulation time vector
Tt = 0:sim_step:sim_time;
% Preallocate arrays for storing simulation data
[d_vec, u_vec, y_vec, e_vec, Th_vec] = deal(nan(size(T)));
throttle_setting = 50;    % Initial throttle/brake setting
actuator_signal_delay = 00; % Communication delay [s]
m_ControlSig = -100;      % Minimum control signal value
M_ControlSig = 100;       % Maximum control signal value
[u, control] = deal(50);
```

Arrays for storing critical simulation data, including driver demand, control signals, vehicle velocity, errors, and throttle positions, are preallocated to enhance computational efficiency. The initial throttle position (throttle_setting) is set at 50%, and the control signals (u and control) start at 50% as well. Control signal limits are defined with a minimum (m_ControlSig) of -100 and a maximum (M_ControlSig) of 100, representing full braking to full throttle, respectively. Additionally, a communication delay (actuator_signal_delay) is initialized at zero seconds, ready for further experimentation and analysis. However adding a delay to the system would not be beneficial as the cruise control code would be at a mismatched with the current situation leading to unwanted acceleration therefore this variable won't change.

5.3.4 Simulation Loop

With the simulation parameters defined, the simulation loop begins by initializing the vehicle's velocity at 60 mph.

```
%% (4) Simulation Loop
integral_error = 0;
prev_error = 0;
% Scaling factors for decoding/encoding signals
R_sensor = 2.^n_bits_sensor_signal;
R_demand = 2.^n_bits_demand_signal;
R_actuator = 2.^n_bits_actuator_signal;

velocity_real = 60; % Initial vehicle velocity (mph)
```

Figure 14 Primary variables set for the simulation loop to run

In each time step, the PID control equation updates the velocity based on the driver's demand, directing throttle or brake actions to move from the current speed toward the target speed. Specifically, when the driver demand is lower than the actual velocity, the loop calculates a negative control signal, leading to braking until the car decelerates to the desired speed. Conversely, if the demanded speed is higher, the PID controller computes a positive control output, increasing throttle to accelerate until the setpoint is reached.

```
for j = 1:length(T)
% Get raw velocity measurement from sensor (assumed defined externally)
[velocity_raw, velocity_real, throttle_setting] = ...
    getRawVelocity(n_bits_sensor_signal, velocity_real, throttle_setting, sim_step, u);

% Decode the raw binary velocity signal into mph
velocity = ((bin2dec(velocity_raw) * (R_velocity5 - m_velocity5)) / (R_sensor - 1)) + m_velocity5;

% Update control every Ts seconds
if mod((j-1)/sim_step, Ts) == 0
% Get and decode driver's velocity demand signal (assumed defined externally)
demand_raw = getRawVelocityDemand(n_bits_demand_signal, sim_step, j);
demand = ((bin2dec(demand_raw) * (R_demand - m_demand)) / (R_demand - 1)) + m_demand;

% Compute PID error terms
error = demand - velocity;
integral_error = integral_error + error * Ts;
if j == 1
    derivative = 0;
else
    derivative = (error - prev_error) / Ts;
end

% Compute control output (with feed-forward term)
control = P * (error + I * integral_error + D * derivative) + demand;

% Clamp control signal to the valid range [-100, 100]
control = max(min(control, M_ControlSig), m_ControlSig);
prev_error = error;

% Encode control signal into binary string for the actuator
control_norm = round((control - m_ControlSig) / (M_ControlSig - m_ControlSig)) * (R_actuator - 1);
control_raw = dec2bin(control_norm, n_bits_actuator_signal);

% Send control command to actuator (assumed defined externally)
u = sendControlCommand(control_raw, n_bits_actuator_signal, actuator_signal_delay, j, sim_step, u_vec);
end

% Store simulation data for later analysis/plotting
e_vec(j) = error;
d_vec(j) = demand;
u_vec(j) = control;
y_vec(j) = velocity_real;
Th_vec(j) = throttle_setting;
end
```

Figure 15 Loop which run the entire simulation using the control equation

$$I_{out} = K_P \left(e + K_I \int e \, dt + K_D \frac{de}{dt} \right) + I_0$$

Figure 16 PID Controller: Combination of 3 control modes which enables a controller to have no offset error and reduces tendency for output oscillations, Also known as 3-term controller

To assess control accuracy, along with the resulting acceleration in g-forces experienced by the occupant. Initially, the PID gains were selected based on an educated guess: $P=0.05$, $I=10$, and $D=0.001$, with a sampling time of $T_s=0.005$.

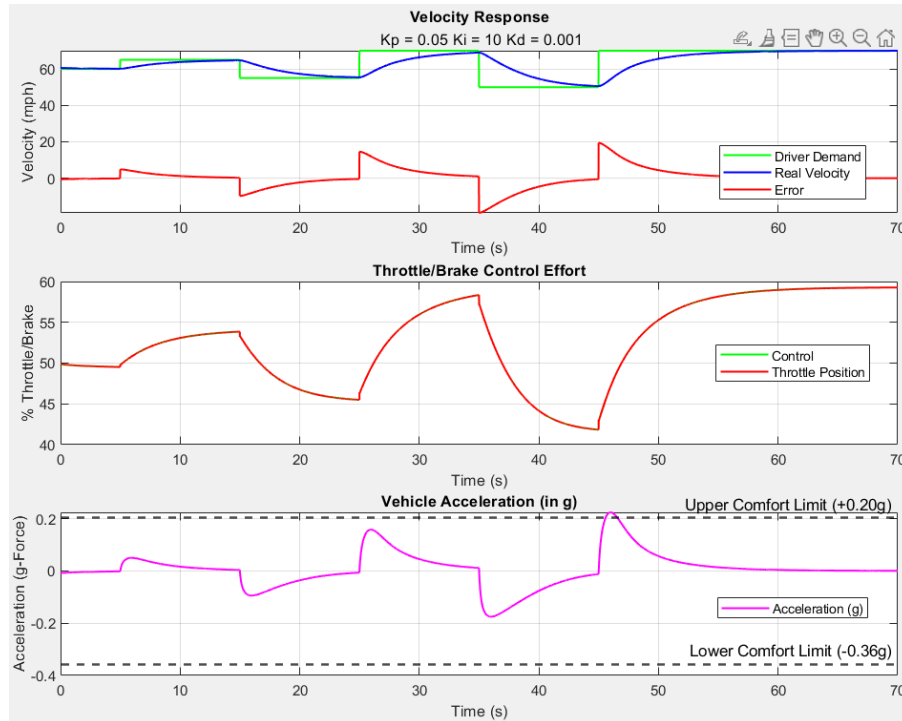


Figure 17 $P=0.05$; $I=10$; $D=0.001$; $T_s=0.005$

As shown in the graph above, this preliminary setup was also tested against different sampling times to identify a suitable operating point that both makes physical sense and delivers a smooth ride. In this initial trial, the graph illustrates the driver demand, the actual vehicle velocity, and their difference (the error), as well as the throttle/brake control effort. By comparing the resulting acceleration g-forces with ISO 15622 limits (+0.20 g for accelerating and -0.36 g for braking), it appears that the initial gain selection, is slightly above the ideal comfort threshold, still produces acceptable performance for a standard road vehicle.

To refine these gains, it was conducted a systematic sweep of over 10,000 (P,I,D) combinations at the fixed sampling time of $T_s=0.005$. Logging a crucial performance indicators the peak acceleration experienced in g-force. Any combination exceeding

ISO 15622 limits was deemed less suitable for ensuring passenger comfort, while combinations with large errors were similarly ruled out.

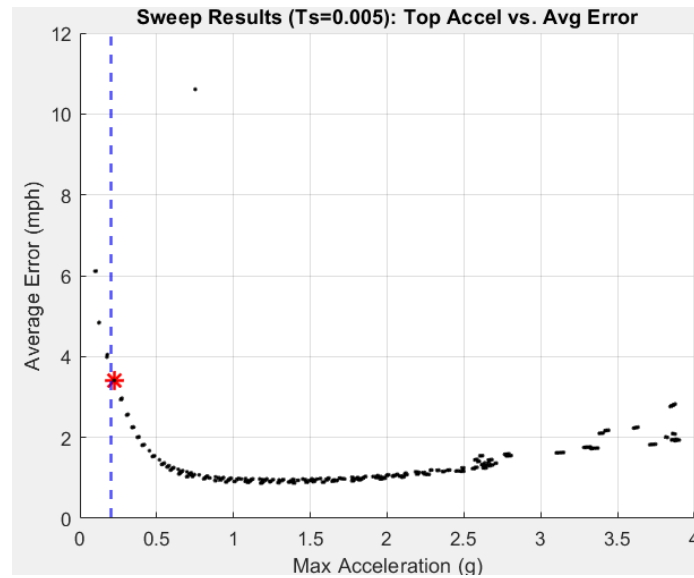


Figure 18 $T_s=0.005$; 10,000 runs, where the X in red is the value of g force in the first try

By comparing the trade-off between low error and low g-forces, it can be possible to pinpoint an optimal controller design, striking a balance between responsiveness and ride comfort that is why a line showing the limit at which the acceleration was higher than 0.2 g.

Top 10 Combinations with Lowest Max Acceleration (g):

P	I	D	MaxAccel_g
0.02	10	0.002	0.098365
0.02	10	0.004	0.098404
0.02	10	0.003	0.09841
0.02	10	0.005	0.098449
0.02	10	0.006	0.098485
0.02	10	0.007	0.098494
0.02	10	0.008	0.098503
0.02	10	0.001	0.098525
0.02	10	0.009	0.098562
0.02	10	0.01	0.098572

Figure 19 Top 10 best possible combinations of PID values with a $T_s=0.005$ (fixed sampling time); 10,000 runs

The previous table was generated listing the top 10 best PID combinations (based on minimal acceleration and error). However, changing the sampling time itself also has a clear influence on the controller's performance.

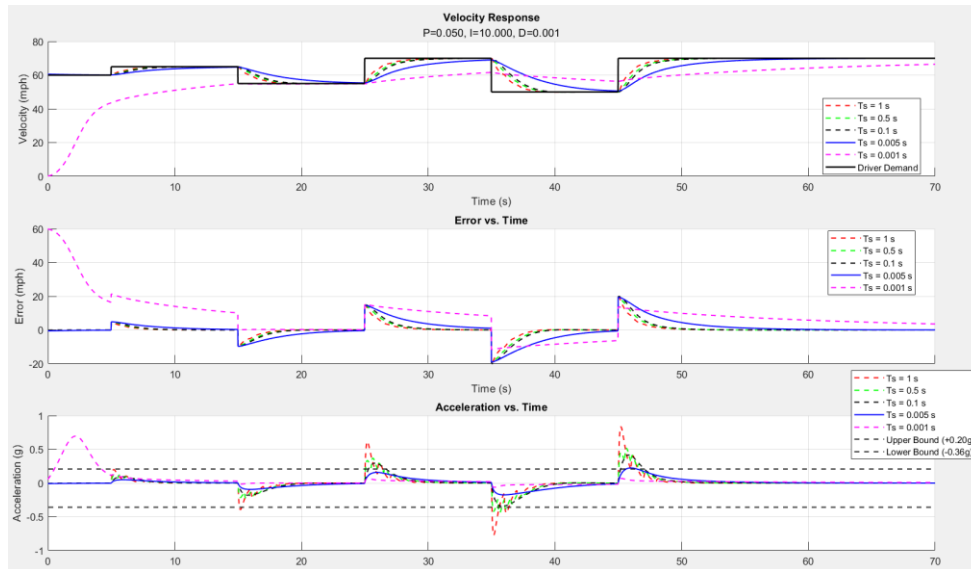


Figure 20 Test of different sampling times showing the best apparent sampling time

The same PID gains applied at different sampling times lead to distinct velocity responses, error profiles, and acceleration patterns. By comparing these scenarios, it becomes evident that the optimal sampling time is $T_s=0.005$ as shown in figure 20.

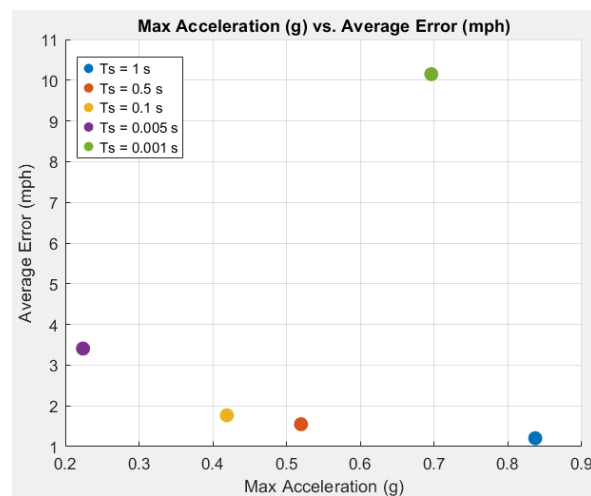


Figure 21 $P=0.05$; $I=10$; $D=0.001$

Also to a more direct comparison of the sampling times in (Average error vs G-force) a scatter plot showing it as evidence that the best scenario would be $T_s=0.005$.

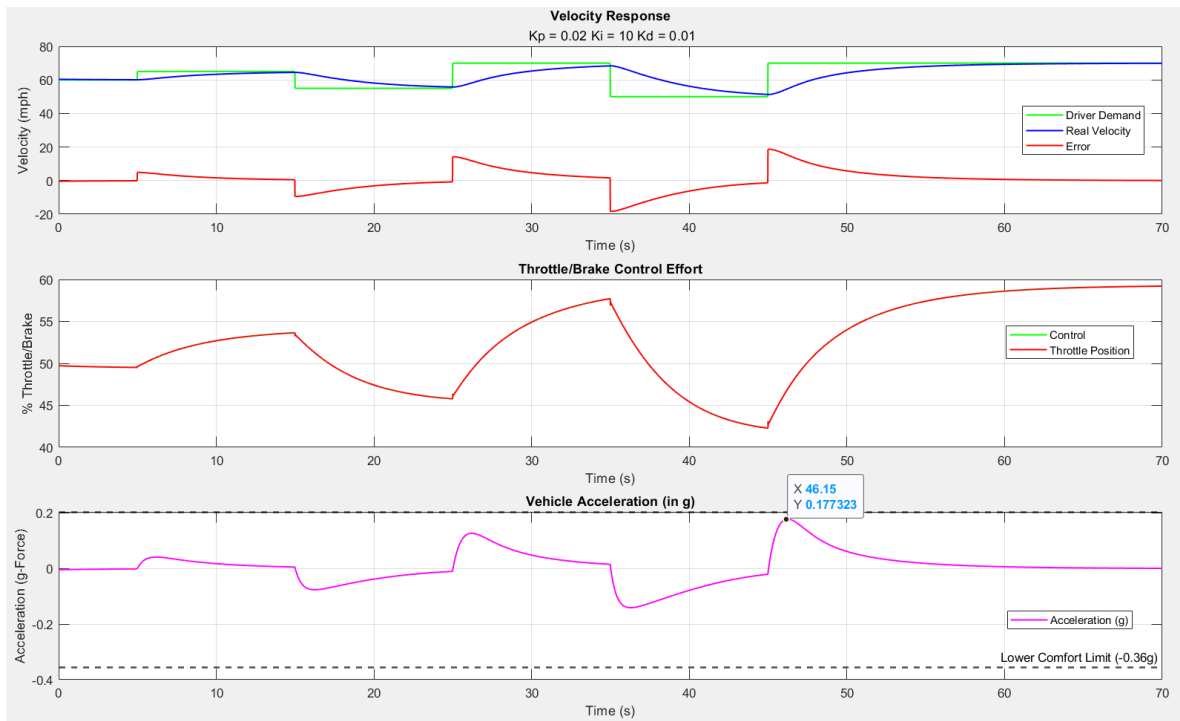


Figure 22 Final Deployment of the cruise control using the top combination of values for the PID using $T_s=0.005$ for the sampling time

Using one of the top PID combinations identified in the table, the resulting plots confirm that the acceleration g-forces remain within the specified comfort range. This outcome indicates that the chosen controller gains successfully balance responsive speed regulation and passenger comfort, making it a strong candidate for final deployment in a production cruise control system.

5.4 Conclusion– PID Cruise Control

With carefully tuned PID gains, the final cruise control configuration consistently stays within the ISO 15622 acceleration thresholds, delivering both smooth velocity tracking and passenger comfort. Adjusting the proportional, integral, and derivative gains—along with varying the sampling time—demonstrated that even minor modifications can significantly influence system responsiveness and ride quality, underscoring the challenges inherent to cruise control optimization.

Future advancements could involve integrating a more systematic approach, such as a Design of Experiments (DOE) method (e.g., Box-Behnken) or Response Surface Methodology (RSM), to automate parameter exploration as to understand the direct impact of each variable. Streamlined code efficiency and added robustness to external disturbances would further enhance real-world deployment.

5. Reference list

Abd El-Fatah, S., Sharkawy, A.-N., Ghazaly, N. and Moaaz, A. (2021). A Comparative Study of Different Control Methods for Anti-Lock Braking System (ABS). *SVU-International Journal of Engineering Sciences and Applications*, 2(1), pp.27–34. doi:<https://doi.org/10.21608/svusrc.2021.65855.1007>.

Aly, A.A., Zeidan, E.-S., Hamed, A. and Salem, F. (2011). An Antilock-Braking Systems (ABS) Control: A Technical Review. *Intelligent Control and Automation*, [online] 02(03), pp.186–195. doi:<https://doi.org/10.4236/ica.2011.23023>.

Anwar, S. and Ashrafi, B. (2002). A Predictive Control Algorithm for an AntiLock Braking System. *SAE Transactions*, [online] 111, pp.484–490. doi:<https://doi.org/10.2307/44719225>.

Arvind Desikan and Kalaichelvi V (2015). Design for a Semi-Active Preview Suspension system using Fuzzy-logic control and Image Processing techniques. *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*. [online] doi:<https://doi.org/10.1109/CYBER.2015.7287933>.

Bhandari, R., Patil, S. and Singh, R.K. (2012). Surface prediction and control algorithms for anti-lock brake system. *Transportation Research Part C: Emerging Technologies*, 21(1), pp.181–195. doi:<https://doi.org/10.1016/j.trc.2011.09.004>.

Bhasin, K. (2019). A review paper on anti-lock braking system (ABS) and its future scope. *Int. J. Res. Appl. Sci. Eng. Technol*, 7(8), pp.372-375.

Chaturvedi, S. and Kumar, N. (2021). Design and Implementation of an Optimized PID Controller for the Adaptive Cruise Control System. *IETE Journal of Research*, [online] pp.1–8. doi:<https://doi.org/10.1080/03772063.2021.2012282>.

de Winkel, K.N., Irmak, T., Happee, R. and Shyrokau, B. (2023). Standards for passenger comfort in automated vehicles: Acceleration and jerk. *Applied Ergonomics*, [online] 106, p.103881. doi:<https://doi.org/10.1016/j.apergo.2022.103881>.

Deubel, C., Ernst, S. and Prokop, G. (2023). Objective evaluation methods of vehicle ride comfort—A literature review. *Journal of Sound and Vibration*, [online] 548, p.117515. doi:<https://doi.org/10.1016/j.jsv.2022.117515>.

dSPACE (2025). *Braking Test Benches*. [online] Dspace.com. Available at: https://www.dspace.com/en/inc/home/products/hw/test_benches/braking.cfm [Accessed 24 Mar. 2025].

Gamesh, L.D.C. and Krishna, Dr.B. (2018). *Simulink and ADAMS Co Simulation for ABS & ESC testing and Validation with Physical Test*. [online] MAHINDRA. Available at: <https://www.matlabexpo.com/content/dam/mathworks/mathworks-dot-com/images/events/matlabexpo/in/2018/simulink-and-adams-co-simulation-for-abs-and-esc-testing-and-validation-with-physical-test.pdf#:~:text=Introduction%3A,vehicle%20level%20changes%20and%20its> [Accessed 24 Mar. 2025].

Gebhard, J.W. (1970). ACCELERATION AND COMFORT IN PUBLIC GROUND TRANSPORTATION. *trid.trb.org*. [online] Available at: <https://trid.trb.org/view/10990> [Accessed 29 Mar. 2025].

Geleta, A.F., Gopal, M. and Lemi, M.T. (2023). Vehicle anti-lock brake system - dynamic modeling and simulation based on MATLAB Simulink and CarSim. *Research on Engineering Structures and Materials*. [online] doi:<https://doi.org/10.17515/resm2023.22me0601rs>.

Horn, S., Rossner, P., Madigan, R., Bieg, H.-J. and Merat, N. (2024). User evaluation of comfortable deceleration profiles for highly automated driving: Findings from a test

track study. *Transportation Research Part F Traffic Psychology and Behaviour*, [online] 105(3), pp.206–221. doi:<https://doi.org/10.1016/j.trf.2024.05.025>.

Hsu, J.-A., Chen, C.-K. and Zhang, X.-D. (2024). *Hardware-in-the-Loop Simulations and Experiments of Anti-Lock Braking System for Cornering Motorcycles*. [online] MDPI. Available at: <https://www.mdpi.com/2079-9292/13/21/4171> [Accessed 24 Mar. 2025].

Hu, J. (2025). *Hardware-in-the-Loop (HIL) with VehicleSim*. [online] Carsim.com. Available at: <https://www.carsim.com/products/supporting/realtime/index.php> [Accessed 24 Mar. 2025].

info.erdosmiller.com. (n.d.). *Introduction to PID Control Loops*. [online] Available at: <https://info.erdosmiller.com/blog/introduction-to-pid-control-loops> [Accessed 30 Mar. 2025].

ISO - International Organization for Standardization (2018). *ISO 15622:2018*. [online] ISO. Available at: <https://www.iso.org/standard/71515.html>.

Jitesh, S. (2014). ANTILOCK BRAKING SYSTEM (ABS). *Int. J. Mech. Eng. & Rob. Res*, [online] 3(4). Available at: <https://www.ijmerr.com/uploadfile/2015/0407/20150407041958451.pdf> [Accessed 24 Mar. 2025].

Kiam Heong Ang, Chong, G. and Yun Li (2005). PID control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, [online] 13(4), pp.559–576. doi:<https://doi.org/10.1109/tcst.2005.847331>.

Knospe, C. (2006). PID control. *IEEE Control Systems*, 26(1), pp.30–31. doi:<https://doi.org/10.1109/mcs.2006.1580151>.

Li, Y., Kiam Heong Ang and Chong, G.C.Y. (2006). PID control system analysis and design. *IEEE Control Systems Magazine*, [online] 26(1), pp.32–41. doi:<https://doi.org/10.1109/mcs.2006.1580152>.

Srivastava, S. and Pandit, V.S. (2016). A PI/PID controller for time delay systems with desired closed loop time response and guaranteed gain and phase margins. *Journal of Process Control*, [online] 37, pp.70–77. doi:<https://doi.org/10.1016/j.jprocont.2015.11.001>.

SRM Tech (2024). *AUTOSAR In Automotives | Architecture and Applications*. [online] SRM Technologies - Global Partner for Digital, Embedded and Product Engineering Services. Available at: <https://www.srmtech.com/knowledge-base/blogs/autosar-in-automotive-systems-an-in-depth-exploration/> [Accessed 24 Mar. 2025].

www.sciencedirect.com. (n.d.). *Response Surface Methodology - an overview | ScienceDirect Topics*. [online] Available at: <https://www.sciencedirect.com/topics/agricultural-and-biological-sciences/response-surface-methodology> [Accessed 30 Mar. 2025].

Zhu, S., Fan, X., Qi, G. and Wang, P. (2022). Review on Control Algorithms of Vehicle Anti-lock Braking System. *Recent Patents on Engineering*, [online] 16. doi:<https://doi.org/10.2174/1872212116666220324154143>.

6 Appendix

Note: When running the code – the extra files (functions) are necessary.

6.1 Optimal Code to Run the Cruise Control System

```
clc;

clear;

close all;

%% (1) Initialization

% PID gains and sampling time

P = 0.02;

I = 10;

D = 0.01;

Ts = 0.01; % Sampling time for control update [s]

%% (2) Set Implementation Parameters

sim_step = 0.01; % Simulation time step [s]

m_velocityS = 0; % Minimum velocity sensor reading (mph)

M_velocityS = 150; % Maximum velocity sensor reading (mph)

n_bits_sensor_signal = 16;

m_demand = 30; % Minimum driver demand (mph)

M_demand = 110; % Maximum driver demand (mph)

n_bits_demand_signal = 16;

n_bits_actuator_signal = 16; % Bits for actuator signal

R = 2^n_bits_actuator_signal; % Actuator scaling factor

%% (3) Set Simulation Parameters
```

```

sim_time = 70;    % Total simulation time [s]

T = -10:sim_step:sim_time; % Simulation time vector

Tt=0:sim_step:sim_time;

% Preallocate arrays for storing simulation data
[d_vec, u_vec, y_vec, e_vec, Th_vec] = deal(nan(size(T)));

throttle_setting = 50;    % Initial throttle/brake setting

actuator_signal_delay = 00; % Communication delay [s]

m_ControlSig = -100;    % Minimum control signal value

M_ControlSig = 100;    % Maximum control signal value

[u, control] = deal(50);

%% (4) Simulation Loop

integral_error = 0;

prev_error = 0;

% Scaling factors for decoding/encoding signals

R_sensor = 2.^n_bits_sensor_signal;

R_demand = 2.^n_bits_demand_signal;

R_actuator = 2.^n_bits_actuator_signal;

velocity_real = 60; % Initial vehicle velocity (mph)

for j = 1:length(T)

    % Get raw velocity measurement from sensor (assumed defined externally)

    [velocity_raw, velocity_real, throttle_setting] = ...

        getRawVelocity(n_bits_sensor_signal, velocity_real, throttle_setting, sim_step, u);

```

```

% Decode the raw binary velocity signal into mph

velocity = ((bin2dec(velocity_raw) * (M_velocityS - m_velocityS)) / (R_sensor - 1)) +
m_velocityS;

% Update control every Ts seconds

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

    % Get and decode driver's velocity demand signal (assumed defined externally)

    demand_raw = getRawVelocityDemand(n_bits_demand_signal, sim_step, j);

    demand = ((bin2dec(demand_raw) * (M_demand - m_demand)) / (R_demand - 1))
+ m_demand;

% Compute PID error terms

error = demand - velocity;

integral_error = integral_error + error * Ts;

if j == 1

    derivative = 0;

else

    derivative = (error - prev_error) / Ts;

end

% Compute control output (with feed-forward term)

control = P * (error + I * integral_error + D * derivative); % + demand;

% Clamp control signal to the valid range [-100, 100]

control = max(min(control, M_ControlSig), m_ControlSig);

prev_error = error;

```

```

    % Encode control signal into binary string for the actuator

    control_norm = round(((control - m_ControlSig) / (M_ControlSig - m_ControlSig)) *
(R_actuator - 1));

    control_raw = dec2bin(control_norm, n_bits_actuator_signal);

    % Send control command to actuator (assumed defined externally)

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

    end

    % Store simulation data for later analysis/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;

% Subplot 1: Velocity Response (Driver Demand, Actual Velocity, and Error)

subplot(3,1,1);

plot(T, d_vec, 'g-', 'LineWidth',1.2); hold on;

xlim([0 70])

plot(T, y_vec, 'b-', 'LineWidth',1.2);

```

```

xlim([0 70])

plot(T, d_vec - y_vec, 'r-', 'LineWidth',1.2);

xlim([0 70])

grid on;

title('Velocity Response');

subtitle([' Kp = ', num2str(P),' Ki = ', num2str(I),' Kd = ', num2str(D)]);

xlabel('Time (s)'); ylabel('Velocity (mph)');

legend('Driver Demand', 'Real Velocity', 'Error', 'Location', 'Best');

```

```

% Subplot 2: Throttle/Brake Control Effort

subplot(3,1,2);

stairs(T, u_vec, 'g-', 'LineWidth',1.2); hold on;

xlim([0 70])

plot(T, Th_vec, 'r-', 'LineWidth',1.2);

xlim([0 70])

grid on;

title('Throttle/Brake Control Effort');

xlabel('Time (s)'); ylabel('% Throttle/Brake');

legend('Control', 'Throttle Position', 'Location', 'Best');

```

```

% Subplot 3: Vehicle Acceleration in g

% Calculate acceleration (in mph/s) as the derivative of velocity

acceleration_mph = diff(y_vec) / sim_step;

% Convert acceleration to m/s^2 (1 mph/s ~ 0.44704 m/s^2), then to g (1 g ~ 9.81 m/s^2)

acceleration_g = (acceleration_mph * 0.44704) / 9.81;

```

```

T_acc = T(2:end);

subplot(3,1,3);

plot(T_acc, acceleration_g, 'm-', 'LineWidth',1.2);

% Add upper and lower boundary lines

yline(0.204, 'k--', 'LineWidth', 1.5, 'Label', 'Upper Comfort Limit (+0.20g)');

yline(-0.357, 'k--', 'LineWidth', 1.5, 'Label', 'Lower Comfort Limit (-0.36g)');


xlim([0 70])

grid on;

title('Vehicle Acceleration (in g)');

xlabel('Time (s)'); ylabel('Acceleration (g-Force)');

legend('Acceleration (g)','Location','Best');

```

6.2 Code to Run the Cruise Control System sweep of different PID values

%This updated version only run 2000 values but the user can change the range of which the code is going to run.

```

clc;

clear;

close all;


%% Define parameter ranges

Ts = 0.005; % Fixed sampling time

P_range = 0.01:0.01:0.1;

I_range = 10:10:200;

D_range = 0.001:0.001:0.01;


% Special (P,I,D) combination

```

```

specialP = 0.05;

specialI = 10;

specialD = 0.001;


%% Simulation parameters

sim_step = 0.01;

sim_time = 70;

T = -10:sim_step:sim_time;


% We'll store results in [P, I, D, avgError, maxAccel]

PID_Results = [];

PID_Labels = {};


% Count total runs for progress bar

nP = length(P_range);

nI = length(I_range);

nD = length(D_range);

total_runs = nP * nI * nD;

current_run = 0;


% Create a waitbar for progress

h_wait = waitbar(0, 'Starting parameter sweep...');


for pVal = P_range
    for iVal = I_range
        for dVal = D_range

```

```

current_run = current_run + 1;

% Update the waitbar
waitbar(current_run/total_runs, h_wait, ...
    sprintf('Run %d of %d\nP=%.4f, I=%.2f, D=%.4f', ...
        current_run, total_runs, pVal, iVal, dVal));

% Initialize state variables for each run
integral_error = 0;
prev_error = 0;
velocity_real = 60;
throttle_setting = 50;
u = 50;

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

for j = 1:length(T)
    [velocity_raw, velocity_real, throttle_setting] = ...
        getRawVelocity(16, velocity_real, throttle_setting, sim_step, u);

    % Decode velocity in mph
    velocity = (bin2dec(velocity_raw)*150)/(2^16 -1);

    % Control update every Ts
    if mod((j-1)*sim_step, Ts) == 0
        demand_raw = getRawVelocityDemand(16, sim_step, j);
    end
end

```

```

demand = ((bin2dec(demand_raw)*(110-30)) / (2^16 -1)) + 30;

error = demand - velocity;

if j == 1
    derivative = 0;
else
    derivative = (error - prev_error)/Ts;
end

integral_error = integral_error + error*Ts;
prev_error = error;

% Pure PID control (no feed-forward)
control = pVal*(error + iVal*integral_error + dVal*derivative);
control = max(min(control,100),-100);

control_norm = round(((control+100)/200)*(2^16 -1));
control_raw = dec2bin(control_norm,16);

u = sendControlCommand(control_raw,16,0,j,sim_step,control);
end

d_vec(j) = demand;
y_vec(j) = velocity_real;
e_vec(j) = d_vec(j) - y_vec(j);

```

```

end

% Compute average absolute error from [0..70]
timeMask = (T >= 0 & T <= 70);
avgError = mean(abs(e_vec(timeMask)));

% Compute acceleration in g
dt = diff(y_vec)/sim_step;
accel_g = (dt*0.44704)/9.81;
T_acc = T(2:end);
accMask = (T_acc>=0 & T_acc<=70);
if isempty(accMask) || all(isnan(accel_g))
    topAccel = 0;
else
    topAccel = max(accel_g(accMask));
end

% Store results
PID_Results = [PID_Results; pVal, iVal, dVal, avgError, topAccel];
PID_Labels{size(PID_Results,1)} = ...
    sprintf('P=%.4f, I=%.2f, D=%.4f', pVal, iVal, dVal);
end
end
end

% Close the waitbar

```

```

close(h_wait);

%% Scatter Plot of maxAccel (x-axis) vs avgError (y-axis)

figure;

hold on; grid on;

title(sprintf('Sweep Results (Ts=%.3f): Top Accel vs. Avg Error', Ts));

xlabel('Max Acceleration (g)');

ylabel('Average Error (mph)');

% Add vertical dashed line at x=0.2g
refLine = xline(0.2, 'b--', 'LineWidth',1.5, ...
    'DisplayName','Reference 0.2 g');

for iRow = 1:size(PID_Results,1)
    pTest = PID_Results(iRow,1);
    iTest = PID_Results(iRow,2);
    dTest = PID_Results(iRow,3);
    avgE = PID_Results(iRow,4);
    maxA = PID_Results(iRow,5);

    % Check if it's the special combination
    if abs(pTest - specialP)<1e-12 && abs(iTest - specialI)<1e-12 && abs(dTest -
specialD)<1e-12
        % Mark it as a star
        scatter(maxA, avgE, 100, '*', 'r', 'LineWidth',1.5, ...
            'DisplayName', sprintf('%s (Special)', PID_Labels{iRow}));
    end
end

```

```

else
    % Normal point = ''
    scatter(maxA, avgE, 25, 'k', 'DisplayName', PID_Labels{iRow});
end
end

end

%legend('Location','Best');

%% After the parameter sweep completes and PID_Results is populated:
%
% PID_Results has rows: [P, I, D, avgError, maxAccel]

% 1) Sort by maxAccel (5th column) ascending:
[~, idx_sort] = sort(PID_Results(:,5), 'ascend');

% 2) Select the top 10 rows (lowest g-forces):
lowest10Idx = idx_sort(1:10);

% 3) Extract only columns [1,2,3,5] => P, I, D, maxAccel
lowest10Data = PID_Results(lowest10Idx, [1,2,3,5]);

% 4) Convert to a table:
lowest10Table = array2table(lowest10Data,...
    'VariableNames', {'P','I','D', 'MaxAccel_g'});

% 5) Display it in the command window:

```

```
disp('Top 10 Combinations with Lowest Max Acceleration (g):');
disp(lowest10Table);
```

6.3 Code to run the sampling time range to know which Sampling Time would be the optimal

```
clc;
clear;
close all;

%% PID gains
P = 0.05;
I = 10;
D = 0.001;

%% Sampling times for comparison
Ts_values = [1, 0.5, 0.1, 0.005, 0.001];
colors = {'r--','g--','k--','b-','m--'};
labels = {'Ts = 1 s','Ts = 0.5 s','Ts = 0.1 s','Ts = 0.005 s','Ts = 0.001 s'};

%% Simulation parameters
sim_step = 0.01;
sim_time = 70;
T = -10:sim_step:sim_time;

% We'll capture the final run's demand for plotting
d_vec_final = [];

% Create a figure with three subplots (velocity, error, acceleration)
figure;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SUBPLOT 1: Velocity Response %%%%%%%%%%%%%
subplot(3,1,1);
hold on; grid on;
title('Velocity Response');
subtitle(sprintf('P=%0.3f, I=%0.3f, D=%0.3f', P, I, D)); % Show PID gains in subtitle
xlabel('Time (s)');
ylabel('Velocity (mph)');
xlim([0 70]);

% We'll store error, acceleration, average error, and max acceleration for each Ts
allErrors = {};
```

```

allAccels = {};
allLabels = {};
avgErrors = nan(1, length(Ts_values));
maxAccVals = nan(1, length(Ts_values)); % top boundary of acceleration (g)

% Time mask for the interval [0..70] to compute average error
timeMask = (T >= 0 & T <= 70);

for idx = 1:length(Ts_values)
    Ts = Ts_values(idx);

    integral_error = 0;
    prev_error = 0;
    velocity_real = 60;
    throttle_setting = 50;
    u = 50;

    [d_vec, y_vec, e_tmp, accel_tmp] = deal(nan(size(T)));

    for j = 1:length(T)
        % Read velocity
        [velocity_raw, velocity_real, throttle_setting] = ...
            getRawVelocity(16, velocity_real, throttle_setting, sim_step, u);

        % Decode velocity in mph
        velocity = (bin2dec(velocity_raw)*150)/(2^16 - 1);

        % Update control every Ts
        if mod((j-1)*sim_step, Ts) == 0
            demand_raw = getRawVelocityDemand(16, sim_step, j);
            demand = ((bin2dec(demand_raw)*(110 - 30)) / (2^16 - 1)) + 30;

            error = demand - velocity;

            if j == 1
                derivative = 0;
            else
                derivative = (error - prev_error) / Ts;
            end

            integral_error = integral_error + error * Ts;
            prev_error = error;

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

```

```

    control = max(min(control,100),-100);

    control_norm = round(((control + 100)/200)*(2^16 - 1));
    control_raw = dec2bin(control_norm,16);

    u = sendControlCommand(control_raw,16,0,j,sim_step,control);
end

    d_vec(j) = demand;
    y_vec(j) = velocity_real;
    e_tmp(j) = d_vec(j) - y_vec(j);
end

% Plot velocity for this sampling time
plot(T, y_vec, colors{idx}, 'LineWidth', 1.2, 'DisplayName', labels{idx});

% If this is the last Ts, store its demand for final plotting
if idx == length(Ts_values)
    d_vec_final = d_vec;
end

% Compute acceleration in g
dt = diff(y_vec)/sim_step;
accel_g = (dt * 0.44704) / 9.81;
accel_tmp(1) = 0;
accel_tmp(2:end) = accel_g;

allErrors{idx} = e_tmp;
allAccels{idx} = accel_tmp;
allLabels{idx} = labels{idx};

% Compute average absolute error in [0..70]
avgErrors(idx) = mean(abs(e_tmp(timeMask)));

% Create a time array for acceleration that matches T(2:end)
T_acc = T(2:end);
accMask = (T_acc >= 0 & T_acc <= 70);
% Find top boundary of acceleration (the maximum positive acceleration in g)
maxAccVals(idx) = max(accel_tmp(accMask));
end

% Plot the final run's demand in black
plot(T, d_vec_final, 'k-', 'LineWidth', 1.5, 'DisplayName', 'Driver Demand');
legend('Location','Best');

```

```

%%%%%%%%%%%%% SUBPLOT 2: Error for Each Sampling Time
%%%%%%%%%%%%%
subplot(3,1,2);
hold on; grid on;
title('Error vs. Time');
xlabel('Time (s)');
ylabel('Error (mph)');
xlim([0 70]);

for iErr = 1:length(allLabels)
    plot(T, allErrors{iErr}, colors{iErr}, 'LineWidth', 1.2, 'DisplayName', allLabels{iErr});
end
legend('Location', 'Best');

%%%%%%%%%%%%% SUBPLOT 3: Acceleration for Each Sampling Time
%%%%%%%%%%%%%
subplot(3,1,3);
hold on; grid on;
title('Acceleration vs. Time');
xlabel('Time (s)');
ylabel('Acceleration (g)');
xlim([0 70]);

for iAcc = 1:length(allLabels)
    plot(T, allAccels{iAcc}, colors{iAcc}, 'LineWidth', 1.2, 'DisplayName',
allLabels{iAcc});
end

% Add upper/lower dashed lines
accUpper = yline(2/9.81, 'k--', 'LineWidth', 1.5, ...
'DisplayName', 'Upper Bound (+0.20g)');
accLower = yline(-3.5/9.81, 'k--', 'LineWidth', 1.5, ...
'DisplayName', 'Lower Bound (-0.36g)');
legend('Location', 'Best');

%% Scatter plot: top boundary of acceleration vs. average error
figure; hold on; grid on;
title('Max Acceleration (g) vs. Average Error (mph)');
xlabel('Max Acceleration (g)');
ylabel('Average Error (mph)');

for idx = 1:length(Ts_values)
    scatter(maxAccVals(idx), avgErrors(idx), 80, 'filled', ...

```

```

    'DisplayName', sprintf('%s', labels{idx}));
end
legend('Location','Best');

%% =====

%% ADDITIONAL: Testing Multiple (P,I,D) Combinations
%% WITHOUT changing the main code above
%% =====

disp('Now testing multiple (P,I,D) combinations...');

% Define sets of P, I, D to test:
P_array = [0.01, 0.05, 0.1];
I_array = [5, 10];
D_array = [0, 0.001];

% We'll fix the sampling time to your chosen best (Ts=0.005).
Ts_PIDtest = 0.01;

% We'll store results in the format:
% [P_value, I_value, D_value, avgError, topAccel]
PID_Results = [];
PID_Labels = {};

% Special combination to highlight in scatter:
specialP = 0.05;
specialI = 10;
specialD = 0.001;

```

```
% For referencing the time interval [0..70] in average error & acceleration
```

```
timeMaskPID = (T >= 0 & T <= 70);
```

```
idxComb = 0;
```

```
for P_test = P_array
```

```
    for I_test = I_array
```

```
        for D_test = D_array
```

```
            idxComb = idxComb + 1;
```

```
            % Initialize for each run
```

```
            integral_error = 0;
```

```
            prev_error = 0;
```

```
            velocity_real = 60;
```

```
            throttle_setting = 50;
```

```
            u = 50;
```

```
            % Preallocate vectors for storing data
```

```
[d_vec_PID, y_vec_PID, e_vec_PID] = deal(nan(size(T)));
```

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

```
    [velocity_raw, velocity_real, throttle_setting] = ...
```

```
        getRawVelocity(16, velocity_real, throttle_setting, sim_step, u);
```

```
    % Decode sensor velocity in mph
```

```
    velocity = (bin2dec(velocity_raw)*150)/(2^16 - 1);
```

```

% Update control logic every Ts_PIDtest
if mod((j-1)*sim_step, Ts_PIDtest) == 0
    demand_raw = getRawVelocityDemand(16, sim_step, j);
    demand = ((bin2dec(demand_raw)*(110-30)) / (2^16 - 1)) + 30;

    error = demand - velocity;

    if j == 1
        derivative = 0;
    else
        derivative = (error - prev_error) / Ts_PIDtest;
    end

    integral_error = integral_error + error*Ts_PIDtest;
    prev_error = error;

    % Pure PID (no feed-forward):
    controlVal = P_test*(error + I_test*integral_error + D_test*derivative);
    controlVal = max(min(controlVal, 100), -100);

    control_norm = round((((controlVal + 100)/200)*(2^16-1)));
    control_raw = dec2bin(control_norm,16);

    u = sendControlCommand(control_raw,16,0,j,sim_step,controlVal);
end

```

```

% Store data

d_vec_PID(j) = demand;

y_vec_PID(j) = velocity_real;

e_vec_PID(j) = d_vec_PID(j) - y_vec_PID(j);

end

% Compute average absolute error in [0..70]

avgError = mean(abs(e_vec_PID(timeMaskPID)));

% Compute acceleration in g

dt_PID = diff(y_vec_PID)/sim_step;

accel_g_PID = (dt_PID * 0.44704)/9.81;

T_accPID = T(2:end);

accMaskPID = (T_accPID >= 0 & T_accPID <= 70);

if isempty(accMaskPID) || all(isnan(accel_g_PID))

    topAccel = 0;

else

    topAccel = max(accel_g_PID(accMaskPID));

end

% Store results

PID_Results(idxComb,:) = [P_test, I_test, D_test, avgError, topAccel];

PID_Labels{idxComb} = sprintf('P=%.3f, I=%.3f, D=%.3f', P_test, I_test, D_test);

```

```

        end
    end
end

% Now create a scatter plot of topAccel (x-axis) vs avgError (y-axis)
figure;
hold on; grid on;
title('PID Gains: Max Acceleration (g) vs. Average Error (mph)');
xlabel('Max Acceleration (g)');
ylabel('Average Error (mph)');

for iRow = 1:size(PID_Results,1)
    Pval = PID_Results(iRow,1);
    lval = PID_Results(iRow,2);
    Dval = PID_Results(iRow,3);

    avgE = PID_Results(iRow,4);
    maxA = PID_Results(iRow,5);

    % Check if this is the special combination
    if abs(Pval - specialP) < 1e-9 && abs(lval - specialI) < 1e-9 && abs(Dval - specialD) <
1e-9
        % Highlight this point differently (e.g., red diamond)
        scatter(maxA, avgE, 120, 'rd', 'filled', ...
'DisplayName', sprintf('%s (Special)', PID_Labels{iRow}));
    else

```

```
        scatter(maxA, avgE, 70, 'filled', 'DisplayName', PID_Labels{iRow});  
    end  
end  
legend('Location','Best');
```