

Control Technology ACMA 6001

Modelling Industrial Control Systems in MATLAB

BEng Manufacturing Systems

Chris Beddoe (2114310)

Date: 22nd July 2024

ACMA6001 Control Technology

Modelling Industrial Control Systems in MATLAB

Chris Beddoe

University of Wales Trinity Saint David

22nd July 2024

Executive Summary

This project focuses on the analysis and optimisation of an industrial control system within Aston Martin's St Athan production facility, specifically the glazing cell and process. The primary objectives include evaluating sensors, actuators, control algorithms, sampling times, and communication protocols, alongside proposing enhancements to improve performance and reliability.

The system currently operates using a Siemens S7 PLC, with various proximity, pressure, and temperature sensors, as well as pneumatic actuators. Analysis revealed that while the system is effective, certain enhancements could improve efficiency and reduce maintenance. Recommendations include replacing existing physical guards with light curtains, which would optimise cycle times and reduce wear.

Additionally, the PID control simulation for a cruise control system was developed in MATLAB, using trial-and-error tuning methods. The final parameters ($K_p = 1$, $K_i = 1$, $K_d = 0.1$ with a reduced sample time of 0.1 seconds) produced a stable output with minimal oscillation and overshoot. However, the simulation does not fully account for real-world vehicle dynamics, suggesting that further refinements may be necessary for real-life applications.

The project underscores the need for both precise control tuning and robust system architecture to achieve optimal performance in automated industrial settings.

Table of Contents

Executive Summary	1
1.0 Introduction.....	1
2.0 Task 1	1
2.1 System Overview	1
2.2 Sensor and Actuator Evaluation.....	2
2.2.1 Sensors.....	2
2.2.1 Actuators.....	4
2.3 Control Algorithm Analysis	4
2.4 Sampling Time Optimisation	5
2.5 Communication Protocols Assessment.....	8
3.0 System Enhancement Proposals	8
4.0 Task 2	9
4.1 Code description	9
4.2 PID tuning methods	16
4.2.1 Trial and error	16
4.2.2 method-based tuning	16
4.2.3 model-based tuning	16
5.0 Tuning	17
6.0 Conclusion	19
7.0 Bibliography	20
8.0 Appendix A.....	22
7.1 Appendix B.....	26

1.0 Introduction

The aims of this project are to identify, analyse and enhance an industrial control system within Aston Martin. Once determined, a focus will be set on optimising its performance, reliability, and efficiency. This project will involve identifying the system's inputs and outputs, evaluating sensors and actuators, understanding and improving the control algorithm if possible. Determining suitable sampling times, assessing communication protocols, exploring relevant software and tools, and proposing potential system enhancements will also be carried out.

2.0 Task 1

2.1 System Overview

Glazing adhesive application is a precise, and essential operation within the Aston Martin St Athan production facility. To carry out this process, an automated PLC control system is utilised, in conjunction with an adhesive application robot. Within this cell there are numerous sensors and actuators controlled via Siemens S7 PLC. A set program is running on the PLC that is constantly monitoring system inputs from various types of sensors or switches, which in turn trigger programmed actions via pneumatic actuators.



Figure 1 – Image of glazing cell

2.2 Sensor and Actuator Evaluation

Within the glazing cell there are examples of different types of sensors and actuators being used. Below is a list of them with examples of how they can be used within an automated production environment.

2.2.1 Sensors

Proximity sensors – These sensors detect the presence of objects without physical contact. Types include inductive, capacitive, and photoelectric sensors. Inductive proximity sensors can be used to detect the position of metal objects (Johnson, 2019). In this cell they are used to detect the positions of the moving safety barriers and clamp positions.



Figure 2 – Proximity sensor for gate detection



Figure 3 – Proximity sensors detecting clamp positions

Fibre Optic Sensors - These sensors use optical fibres to detect changes in light transmission, which can indicate the presence of objects, changes in temperature, pressure, or other environmental factors. They are highly sensitive and immune to electromagnetic interference. Gupta & Kumar (2018) state that fibre optic sensors can be used to detect the presence of glass, the exact reason they are utilised within the glazing cell.

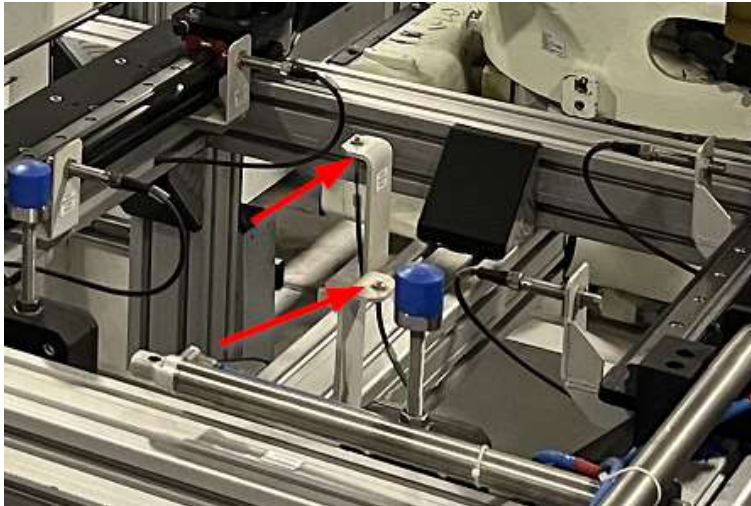


Figure 4 – Arrows highlighting fibre optic sensors in glazing cell

Other sensors used include temperature sensors to detect the temperature of the adhesive in the robot application head, pipes and hopper. Pressure sensors are utilised to detect the overall adhesive system pressure and the presence of air pressure in the pneumatic system. (Walker & Hill, 2021)

Figure 5, adjacent, illustrates an air pressure switch using both the normally closed (NC) and normally open (NO) contact to detect both high and low system pressure. Figure 5 is taken from the glazing cell electrical drawings.

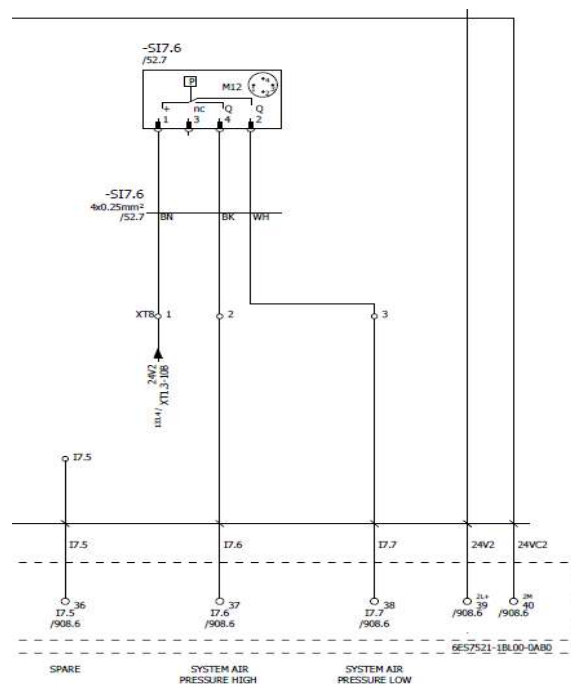


Figure 5 – Schematic drawing of system air pressure detection via pressure switch

2.2.1 Actuators

Solenoid valves – Within the glazing cell these actuators are being used to control the flow of pneumatic system pressure, which in turn controls the movements of clamps and guards using cylinders. Solenoid valves are commonly used in fluid power pneumatic and hydraulic systems, meaning inputs from sensors can be used to enable outputs to control the flow of fluid or air. (Ahmed, 2016)

Pneumatic actuators - These actuators use compressed air to produce motion. They are used within the glazing cell for opening and closing guards and clamping via cylinders but are also widely used within industry within various material handling systems. (Patel, 2018)

2.3 Control Algorithm Analysis

As stated within the system overview, the glazing cell is controlled using a Siemens S7 1500 series. Siemens S7 PLCs use a diverse range of control algorithms to manage various industrial processes. This PLC program itself is written using Siemens SIMATIC S7 software. Other PLC manufacturers include Allen Bradley, Mitubishi and Omron, all utilising their own logic programming software. Allen Bradley PLCs use Studio 5000 Logix Designer (Rockwell Automation, 2024), Mitubishi utilises MELSEC (Mitsubishi Electric, 2024) and Omron using CX-Programmer (OMRON, 2024).

The glazing cell Siemens PLC utilises ladder logic, as shown in figure 6, to control the overall process of the cell and is uploaded to the PLC once complete. Figure 6

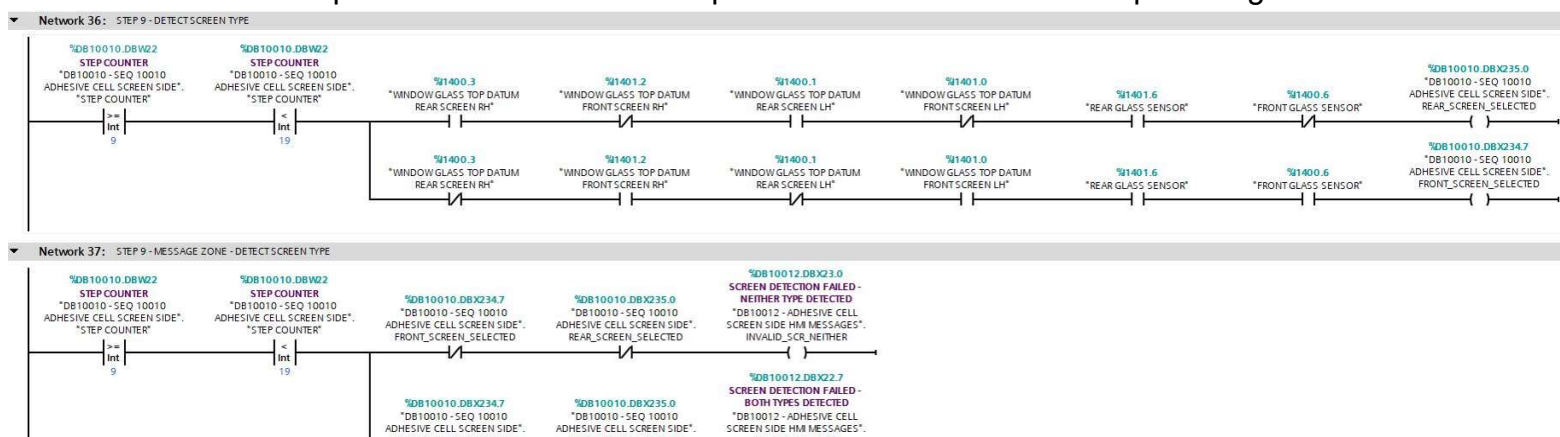


Figure 6 – Ladder logic for screen type detection in glazing cell (Premier Automation, 2018)

illustrates a section of the logic that is used to detect the screen type that has been placed in the glazing cell using inputs from proximity and fibre optic sensors. The top line of logic in this example illustrates the AND and NOT conditions that must be made to trigger an internal logic flag that indicates that a rear screen has been placed into the cell, thus triggering the next step of the overall PLC program.

Siemens S7 PLCs can use a diverse range of control algorithms to manage various industrial processes. From basic logical operations to advanced PID control, algorithms provide the flexibility and power needed to handle complex automation tasks. By utilising built-in functions, function blocks, and custom programming capabilities, Siemens S7 PLCs can be tailored to meet the specific needs of virtually any industrial application.

2.4 Sampling Time Optimisation

Sampling time is a critical parameter in any PLC system, as it determines the frequency at which the controller reads input data, executes control algorithms, and updates outputs. Setting the optimal sampling time ensures that a system responds promptly to changes in the process while avoiding unnecessary computational load and resource utilisation. If the sampling time is too short, the PLC might perform excessive computations, leading to increased CPU load and potential delays in processing other tasks. On the other hand, if the sampling time is too long, the system may respond sluggishly to process variations, causing performance degradation and instability (Bolton, 2015). In this situation, that may mean inputs from sensors are not detected. Therefore, finding the optimal sampling time involves balancing responsiveness and computational efficiency. Properly optimised sampling times enhance the accuracy and reliability of the control system, ensuring efficient and effective process management (Johnson, 2013).

Within this system a minimum cycle time is set at 1ms, and a maximum cycle time set to 150ms as per figure 7, page 6. This has been deemed optimal for this system

by the Aston Martin controls team due to the fact it is quite a small cell with less inputs and outputs than other areas of the factory.

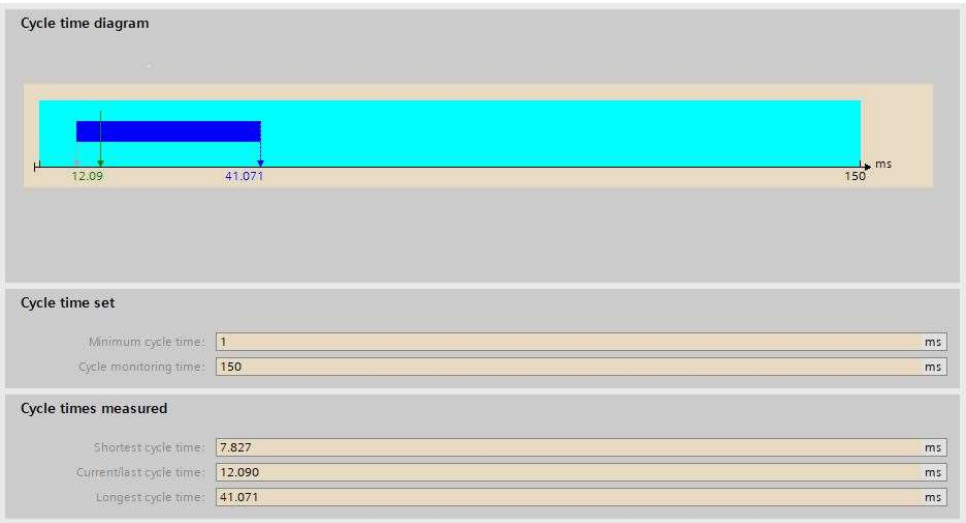


Figure 7 – Cycle time set for glazing cell PLC

For example, in the body shop the sample time has been set to a minimum of 20ms as a lower sampling time resulted in PLC failures due to too much load on the system. Figure 8 illustrates this with the top graph providing sampling times, versus the bottom graph showing percentage load of the PLC, peak loading being thirty eight percent.

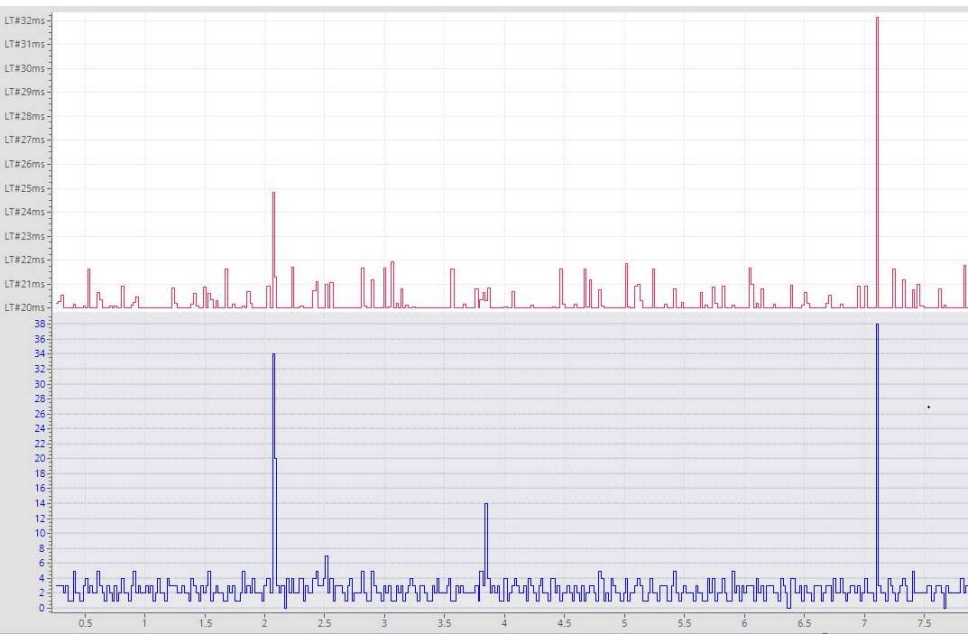


Figure 8 – Image showing sample time of 20ms minimum and load percentage of PLC

The glazing cell incorporates various safety systems from guard position proximity switches, emergency stop buttons, safe edge sensors and mechanic safety interlocks with integrated switches. A sample of the safety program is depicted in figure 9, whilst the sampling time of 1ms for the safety switches is shown in figure 10. A short cycle time is essential for the safety switches due to the nature of what they are attempting to achieve. A lag of detection of a safety switch could mean the difference between saving an employee from harm or not.

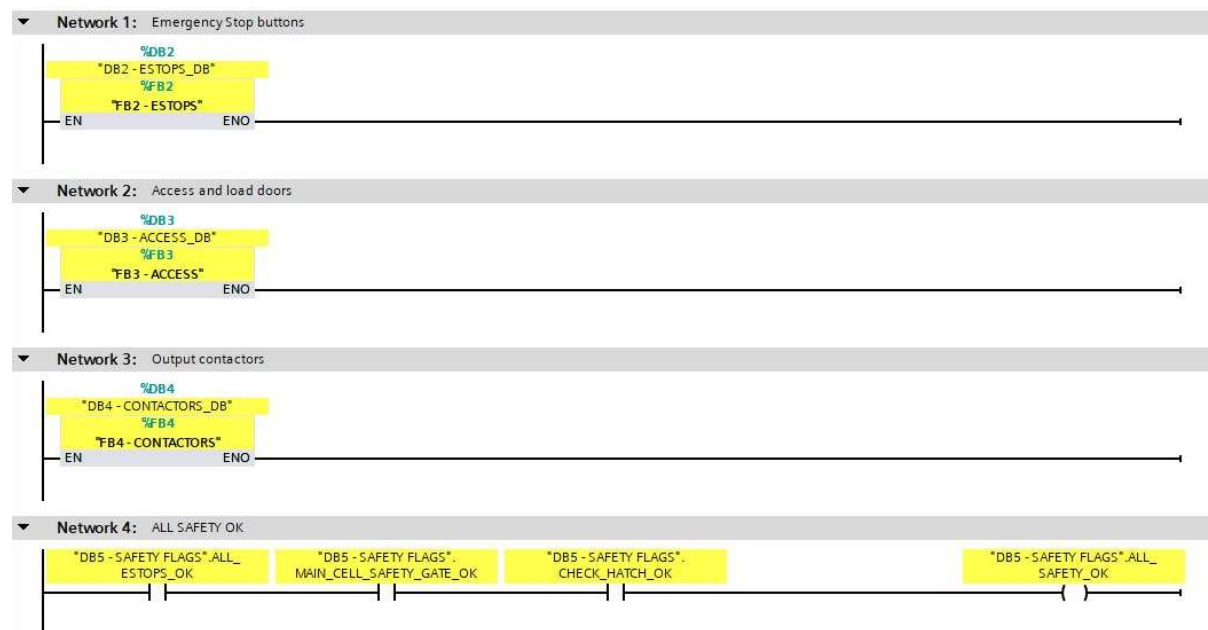


Figure 9 – Ladder logic of main safety program for glazing cell

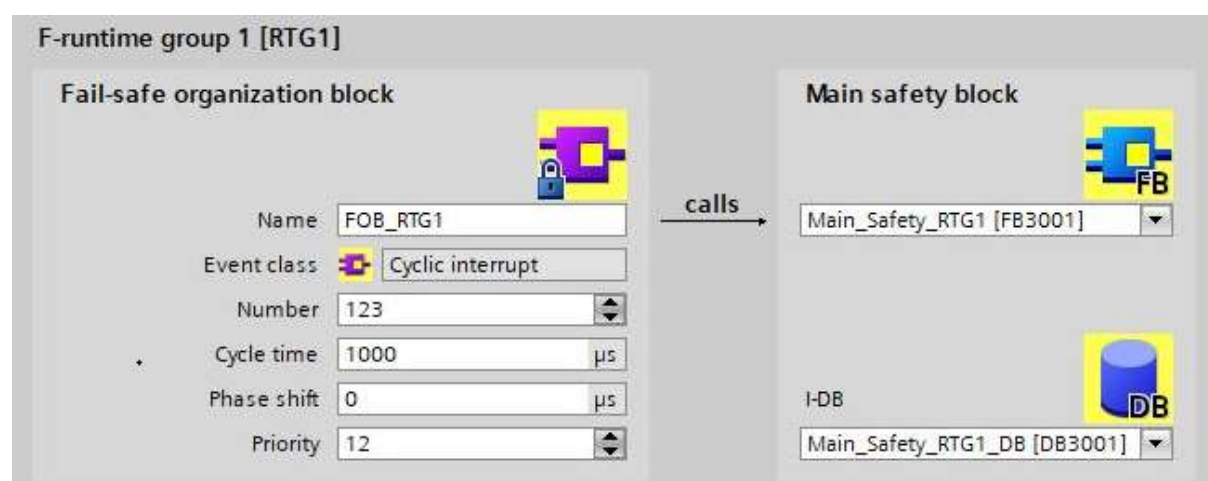


Figure 10 – Cycle time of 1m set for safety system cycle time

2.5 Communication Protocols Assessment

There are numerous communication systems within industrial PLC systems. Within Aston Martin PROFINET and PROFIBUS are both utilised, with the glazing cell using PROFINET.

PROFINET is a widely used industrial Ethernet standard developed by Siemens for automation applications due to its high-speed performance and real-time communication which is suited to complex automation tasks. It has a higher initial investment cost and requires specialist knowledge to configure and maintain but can handle many devices and extensive network topologies (Siemens , 2024).

PROFIBUS and AS-I are also Siemens based communications but are not as capable. PROFIBUS is determined to be reliable but has limited bandwidth and scalability constraints meaning it may not be as effective for large or complex networks. AS-i is more cost effective than PROFINET and PROFIBUS, is simple to install and use, but is not suited to high-speed or complex communications (Siemens , 2024).

When analysing the glazing cell, a PROFIBUS system could have been used due to the scale of the cell. In keeping with the rest of the factory, PROFINET was deemed the standard to be used within Aston Martin to reduce complexity for controls engineers maintaining the system.

3.0 System Enhancement Proposals

Analysis of the current glazing cell system highlights some potential for improvements. For example, the current guarding that opens and closes during cycles for placement of glazing is a Perspex guard, lifted and lowered using pneumatic cylinders. A safe edge is incorporated along with both top and bottom position proximity switches that feed into the PLC logic for detection of obstructions such as a technician's arm, or an assistor lifting the glass into the cell. This guarding could be replaced by a light curtain from manufacturer such as SICK, see Appendix figure B for examples. Not only could this result reduced cycle time due to the

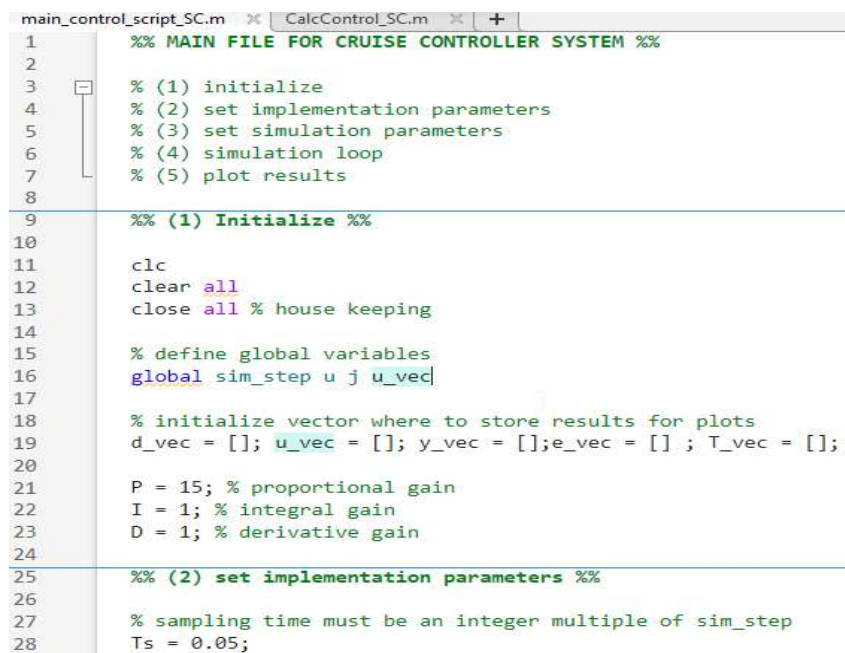
removal of time taken for guarding to open and close, but maintenance costs could also be reduced as there would be less equipment to wear, fail and to be maintained.

4.0 Task 2

4.1 Code description

MATLAB is a software which can be utilised for data analysis, development of algorithms and create models. In this report MATLAB is to be used to simulate a cruise control system in a vehicle. The simulation will incorporate calculations that will simulate a PID controlled system used to maintain a steady speed for a driver. Alterations will be made to the proportional, integral and derivative gains, and the effects of these alterations will be studied in graph format and analysed. The outcome of this report is to find and recommend optimum settings for the cruise control system which are to provide a smooth system for the driver.

The MATLAB code is split into various sections, the main script (see appendix A for full main script) and numerous function codes. The main script will run the simulation and be used to plot the graphs, the separate function codes are then used to perform operations supplementary to the main script, but their results are incorporated within the main script. See figure 11 below for a sample of the main script.



```
main_control_script_SC.m  CalcControl_SC.m  +
1      %% MAIN FILE FOR CRUISE CONTROLLER SYSTEM %%
2
3      % (1) initialize
4      % (2) set implementation parameters
5      % (3) set simulation parameters
6      % (4) simulation loop
7      % (5) plot results
8
9      %% (1) Initialize %%
10
11      clc
12      clear all
13      close all % house keeping
14
15      % define global variables
16      global sim_step u j u_vec
17
18      % initialize vector where to store results for plots
19      d_vec = []; u_vec = []; y_vec = []; e_vec = [] ; T_vec = [];
20
21      P = 15; % proportional gain
22      I = 1; % integral gain
23      D = 1; % derivative gain
24
25      %% (2) set implementation parameters %%
26
27      % sampling time must be an integer multiple of sim_step
28      Ts = 0.05;
```

Figure 11 – MATLAB main control script MATLAB (2024)

The main script itself is split into five sections. Initialisation, setting of implementation parameters, setting simulation parameters, simulation loop and plot results.

Initialisation, is used to clear the workspace, close all figures, and clear the command window (Mathworks, 2024). Global variables are defined as the variables, `sim_step`, `U`, `J` and `u_vec`. These variables will later be populated using figures from the other function codes running in parallel with the main script. The following vectors, `d_vec`, `u_vec`, `e_vec` and `T_vec` are used to plot the graphs that will be created when running the simulation. The proportional, integral and derivative gains are then set, these are the values that will be altered, and outcomes evaluated during of the PID simulation.

The proportional gain (K_p) determines the ratio of output response to the error signal. For example, if the error term has a magnitude of 20, a proportional gain of 10 would produce a proportional response of 100. This means that increasing the proportional gain will increase the speed of the control system response, meaning more harsh acceleration or deceleration. However, if the proportional gain is excessive, the process variable will begin to oscillate. If K_p is increased further, the oscillations will become larger, and the system will become unstable (Hägglund & Åström, 2006). See figure 12 below for an example of this. To simulate this, a K_p value of 100 was used.

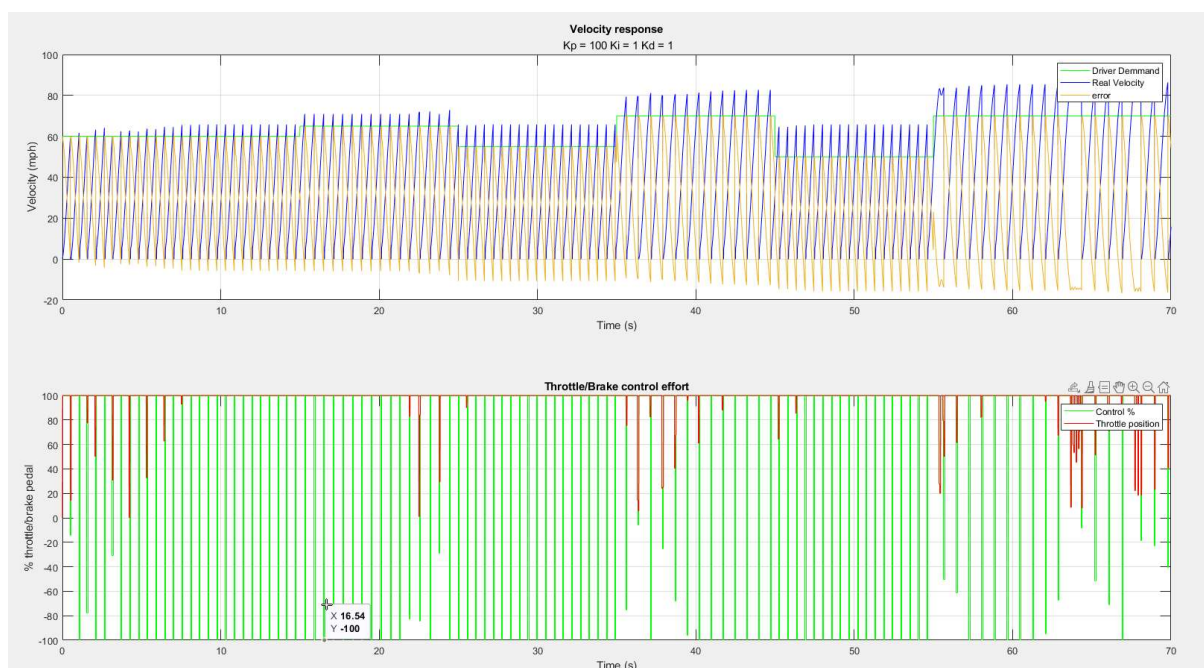


Figure 12 – Simulation output graph to show oscillation when K_p is set too high, MATLAB (2024)

Steady state error is the final difference between the measured process variable and the desired set point. Integral action removes steady-state error by compensating for any persistent deviation from the setpoint. Increasing the integral gain (K_i) can result in a faster elimination of steady state errors but may cause the system to overshoot the set point and become unstable. A lower K_i value slows down error correction but provides smoother control (Seborg, et al., 2016) . With reference to figure 13 below, see the steady state error.

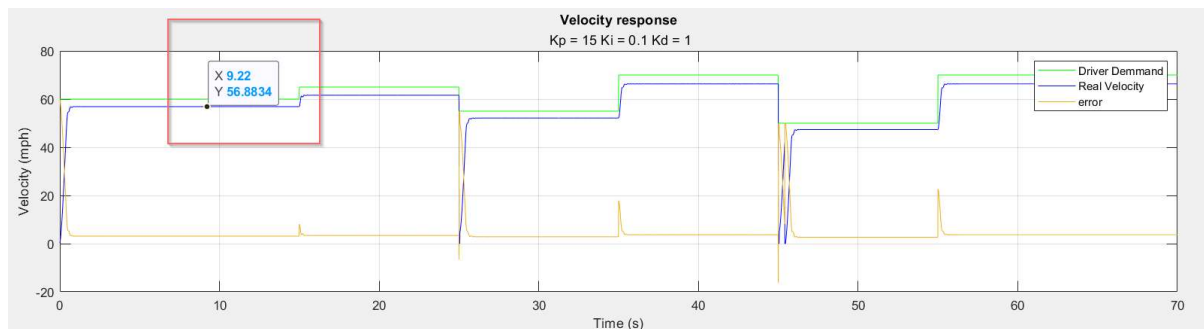


Figure 13 - Simulation output graph to show steady state error, MATLAB (2024)

In the simulation above a system response of 65.9mph is observed from a set point of 60mph. That is a steady state error of 4.1mph. To eliminate this the K_i can be adjusted. See result of setting K_i to 1 in figure 14 below. The error has been reduced although a small amount of overshoot is observed at around 45 seconds.

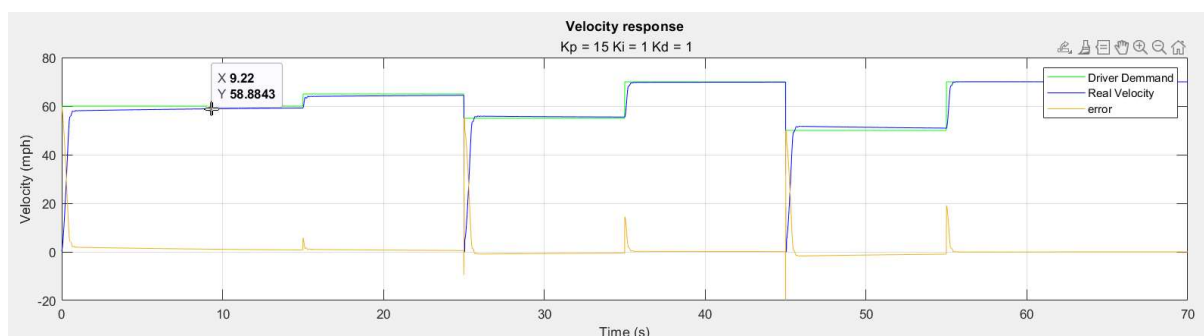
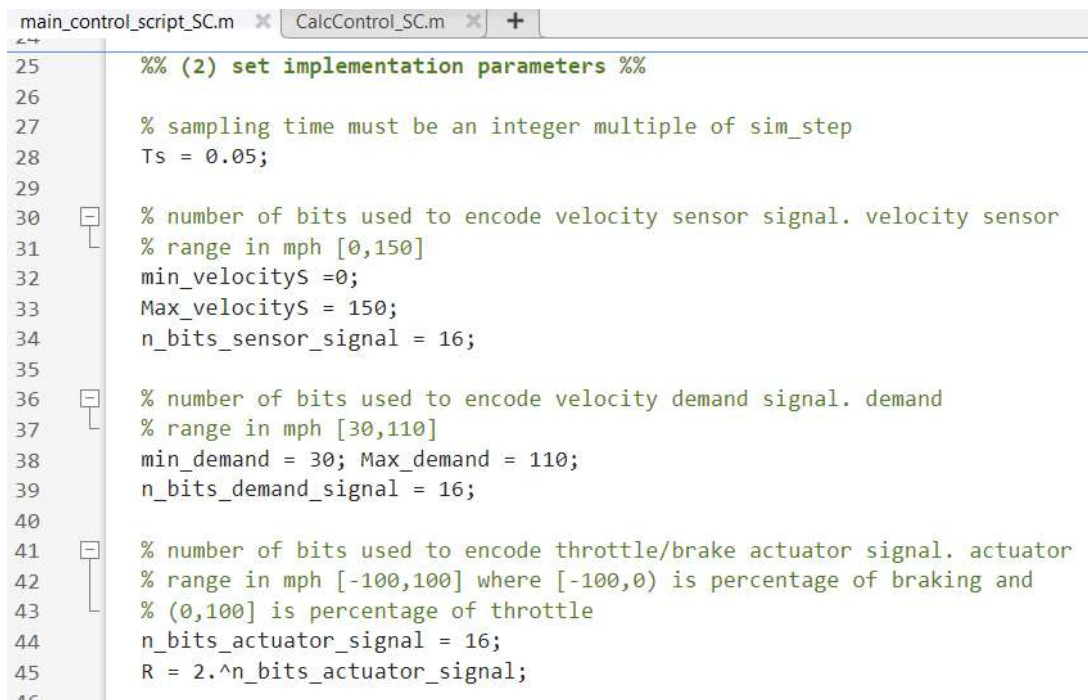


Figure 14 – Simulation output showing result of adjusting K_i to reduce steady state error, MATLAB (2024)

Derivative action dampens the system's response by counteracting sudden changes, reducing overshoot and oscillations as stated by Dorf and Bishop (2010). A higher derivative gain (K_d) increases system stability by reacting to error rate changes but can introduce noise sensitivity. A lower K_d makes the system less responsive to rapid changes but avoids amplifying high-frequency noise.

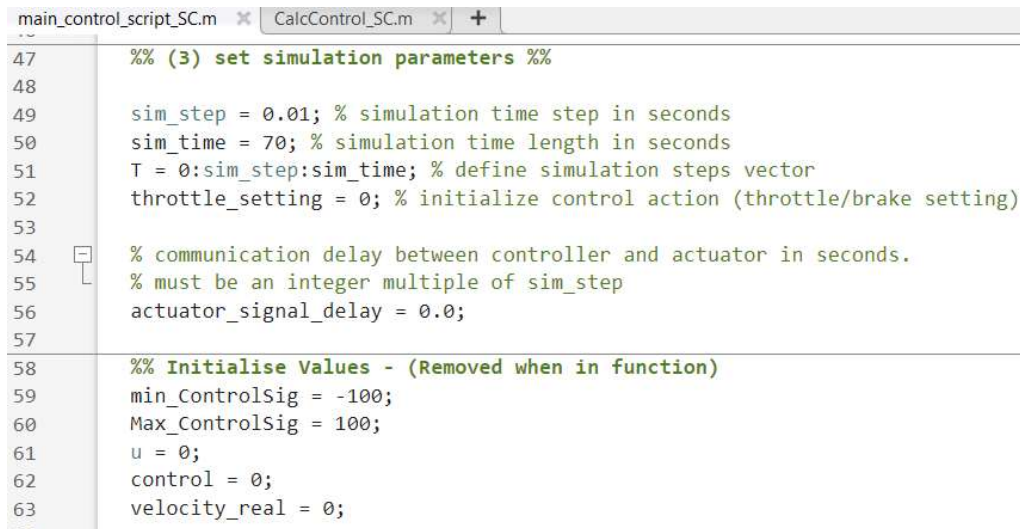
The image shows a MATLAB script editor with two tabs: 'main_control_script_SC.m' and 'CalcControl_SC.m'. The script in the active tab is titled '%% (2) set implementation parameters %%'. It defines the sampling time 'Ts' as 0.05. It then sets parameters for three signals: velocity sensor, velocity demand, and throttle/brake actuator. Each signal is configured with a range in mph and a bit resolution of 16 bits. The velocity sensor range is [0, 150] mph. The velocity demand range is [30, 110] mph. The throttle/brake actuator range is [-100, 100] mph, where [-100, 0) is percentage of braking and (0, 100] is percentage of throttle. The bit resolution for all three signals is 16 bits, and the resolution 'R' is calculated as 2 to the power of 16.

```
25 %% (2) set implementation parameters %%
26
27 % sampling time must be an integer multiple of sim_step
28 Ts = 0.05;
29
30 % number of bits used to encode velocity sensor signal. velocity sensor
31 % range in mph [0,150]
32 min_velocityS = 0;
33 Max_velocityS = 150;
34 n_bits_sensor_signal = 16;
35
36 % number of bits used to encode velocity demand signal. demand
37 % range in mph [30,110]
38 min_demand = 30; Max_demand = 110;
39 n_bits_demand_signal = 16;
40
41 % number of bits used to encode throttle/brake actuator signal. actuator
42 % range in mph [-100,100] where [-100,0) is percentage of braking and
43 % (0,100] is percentage of throttle
44 n_bits_actuator_signal = 16;
45 R = 2.^n_bits_actuator_signal;
```

Figure 15 - MATLAB main control script implementation parameters MATLAB (2024)

Figure 15 above outlines the key implementation parameters for the main script. Here, the sampling time for the simulation is defined, which in this case is set at 0.05 seconds. This interval determines how frequently the controller computes adjustments needed to achieve the desired output. An optimal sampling time is required as this determines the system accuracy and responsiveness. This should be determined before tuning the controller as changing the sampling time would result in the requirement of retuning. Considerations should be made to the cost of the system also as shorter sampling times would require higher computational load, meaning a higher cost or risk overloading an under specification controller.

Additionally, this portion of the code specifies the bit resolutions used to encode various signals within the system simulation. Adjusting these bit settings directly impacts the simulation results. As seen in this script, all signals are currently encoded at 16 bits. Specifically, it sets the bit resolutions for the velocity sensor signal, velocity demand signal, and throttle/brake actuator signal. For the velocity sensor, the range is configured between 0 and 150 mph. The velocity demand signal is set with a minimum of 30 mph and a maximum of 110 mph.



```

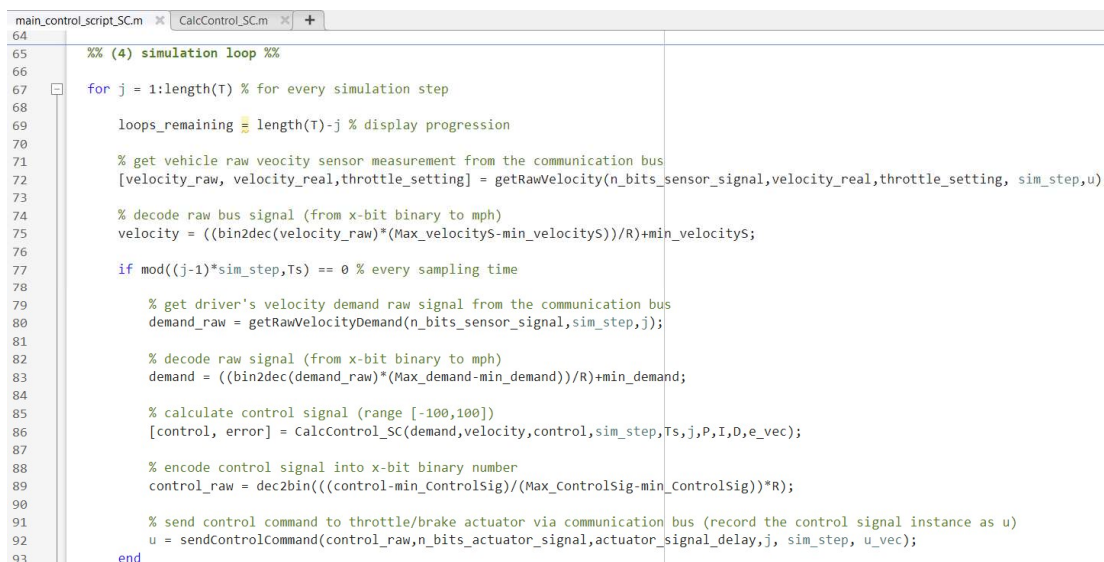
main_control_script_SC.m x CalcControl_SC.m x +
47 %% (3) set simulation parameters %%
48
49 sim_step = 0.01; % simulation time step in seconds
50 sim_time = 70; % simulation time length in seconds
51 T = 0:sim_step:sim_time; % define simulation steps vector
52 throttle_setting = 0; % initialize control action (throttle/brake setting)
53
54 % communication delay between controller and actuator in seconds.
55 % must be an integer multiple of sim_step
56 actuator_signal_delay = 0.0;
57
58 %% Initialise Values - (Removed when in function)
59 min_ControlSig = -100;
60 Max_ControlSig = 100;
61 u = 0;
62 control = 0;
63 velocity_real = 0;

```

Figure 16 - MATLAB main control script set simulation parameters MATLAB (2024)

Figure 16 presents the section of code that sets the core parameters for the simulation. These include the time step of 0.01 seconds, the simulation duration 70 seconds, the throttle setting of 0%, and the communication delay between the controller and actuator of 0.

The 'sim_step' parameter controls how often the simulation updates and recalculates the control response. The 'sim_time' parameter defines the total run time for the simulation, while the 'throttle_setting' determines the initial throttle output at the start of the simulation.



```

main_control_script_SC.m x CalcControl_SC.m x +
64
65 %% (4) simulation loop %%
66
67 for j = 1:length(T) % for every simulation step
68     loops_remaining = length(T)-j % display progression
69
70     % get vehicle raw velocity sensor measurement from the communication bus
71     [velocity_raw, velocity_real, throttle_setting] = getRawVelocity(n_bits_sensor_signal, velocity_real, throttle_setting, sim_step, u);
72
73     % decode raw bus signal (from x-bit binary to mph)
74     velocity = ((bin2dec(velocity_raw)*(Max_velocityS-min_velocityS))/R)+min_velocityS;
75
76     if mod((j-1)*sim_step, Ts) == 0 % every sampling time
77
78         % get driver's velocity demand raw signal from the communication bus
79         demand_raw = getRawVelocityDemand(n_bits_sensor_signal, sim_step, j);
80
81         % decode raw signal (from x-bit binary to mph)
82         demand = ((bin2dec(demand_raw)*(Max_demand-min_demand))/R)+min_demand;
83
84         % calculate control signal (range [-100,100])
85         [control, error] = CalcControl_SC(demand, velocity, control, sim_step, Ts, j, P, I, D, e_vec);
86
87         % encode control signal into x-bit binary number
88         control_raw = dec2bin(((control-min_ControlSig)/(Max_ControlSig-min_ControlSig))*R);
89
90         % send control command to throttle/brake actuator via communication bus (record the control signal instance as u)
91         u = sendControlCommand(control_raw, n_bits_actuator_signal, actuator_signal_delay, j, sim_step, u_vec);
92     end
93

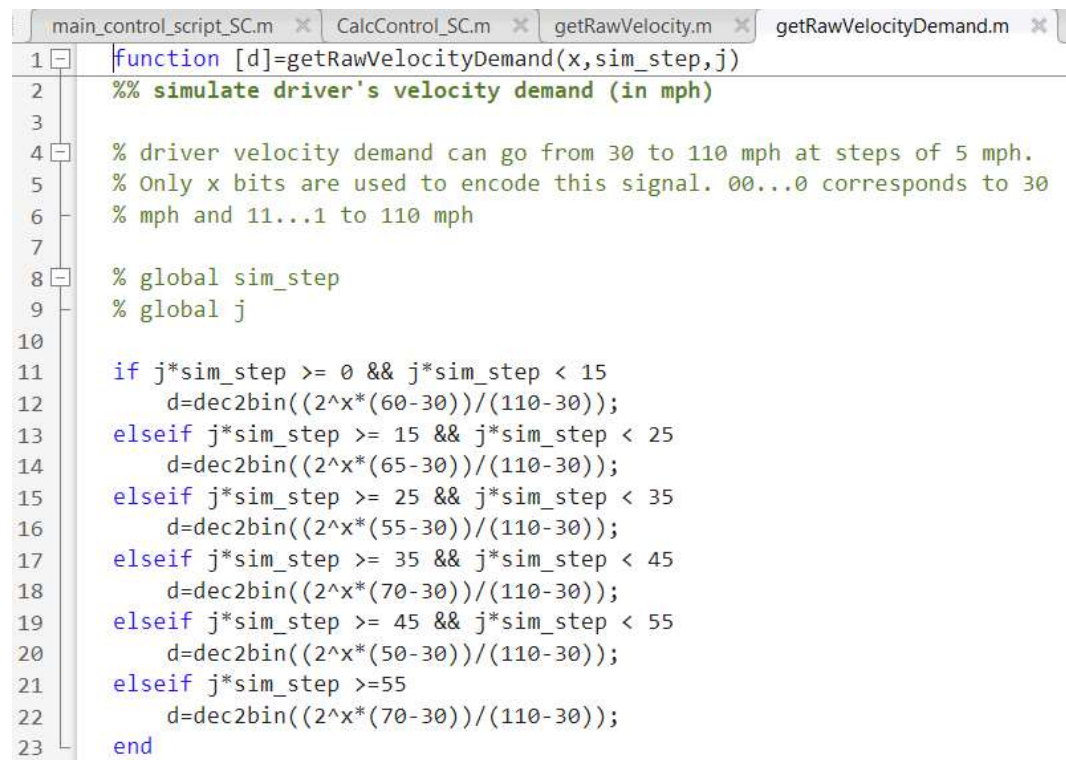
```

Figure 17 - MATLAB main control script simulation loop MATLAB (2024)

Figure 17 illustrates the loop of the control system simulation in the main script. This section cycles through time steps, calculates control actions based on driver demand

and sensor readings, and updates the system's state whilst storing values for graph plotting. The 'getRawVelocity' function retrieves a raw velocity sensor measurement, converting it from a binary value to mph. This measurement is used to determine the error between current and desired velocities. Based on this error, a control command is sent to adjust the throttle or brake to achieve the target speed.

The GetRawVelocityDemand function, figure 18, simulates driver speed input, which changes during the simulation to mimic adjustments to the system. Different speed ranges are handled using 'elseif' statements, and the results are converted from decimal to binary via the 'dec2bin' command.



```

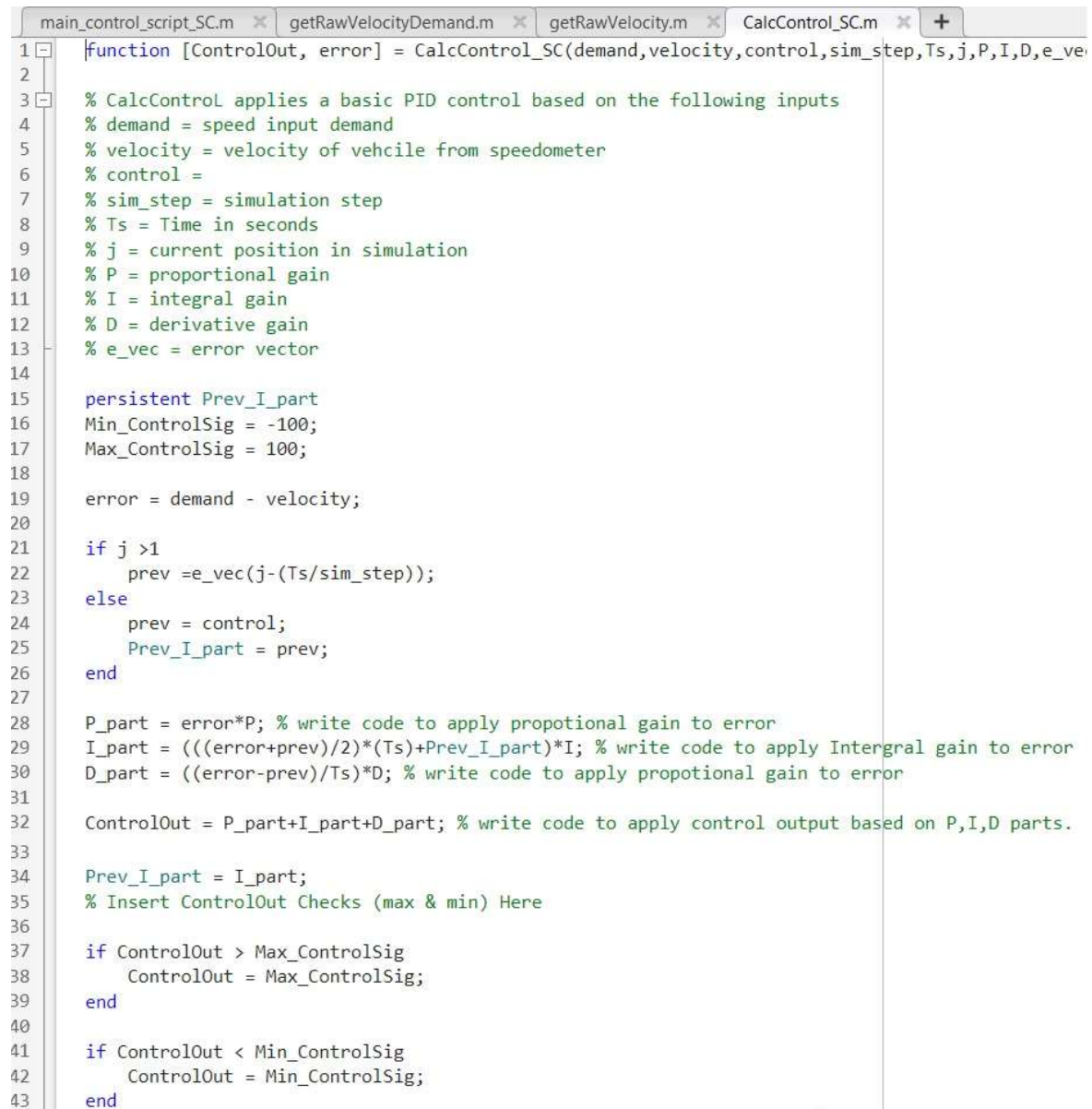
1 function [d]=getRawVelocityDemand(x,sim_step,j)
2     %% simulate driver's velocity demand (in mph)
3
4     % driver velocity demand can go from 30 to 110 mph at steps of 5 mph.
5     % Only x bits are used to encode this signal. 00...0 corresponds to 30
6     % mph and 11...1 to 110 mph
7
8     % global sim_step
9     % global j
10
11     if j*sim_step >= 0 && j*sim_step < 15
12         d=dec2bin((2^x*(60-30))/(110-30));
13     elseif j*sim_step >= 15 && j*sim_step < 25
14         d=dec2bin((2^x*(65-30))/(110-30));
15     elseif j*sim_step >= 25 && j*sim_step < 35
16         d=dec2bin((2^x*(55-30))/(110-30));
17     elseif j*sim_step >= 35 && j*sim_step < 45
18         d=dec2bin((2^x*(70-30))/(110-30));
19     elseif j*sim_step >= 45 && j*sim_step < 55
20         d=dec2bin((2^x*(50-30))/(110-30));
21     elseif j*sim_step >=55
22         d=dec2bin((2^x*(70-30))/(110-30));
23     end

```

Figure 18 – get raw velocity demand script MATLAB (2024)

The CalcController function, shown in Figure 19, page 15, implements PID control to generate a control signal based on the error between the desired and actual velocity. The error is calculated using error is demand minus velocity. Within this part of the

code an IfElse statement is used to ensure that the simulation stays within the minimum and maximum control signals.



```

1 function [ControlOut, error] = CalcControl_SC(demand,velocity,control,sim_step,Ts,j,P,I,D,e_vec)
2
3 % CalcControl applies a basic PID control based on the following inputs
4 % demand = speed input demand
5 % velocity = velocity of vehicle from speedometer
6 % control =
7 % sim_step = simulation step
8 % Ts = Time in seconds
9 % j = current position in simulation
10 % P = proportional gain
11 % I = integral gain
12 % D = derivative gain
13 % e_vec = error vector
14
15 persistent Prev_I_part
16 Min_ControlSig = -100;
17 Max_ControlSig = 100;
18
19 error = demand - velocity;
20
21 if j > 1
22     prev = e_vec(j-(Ts/sim_step));
23 else
24     prev = control;
25     Prev_I_part = prev;
26 end
27
28 P_part = error*P; % write code to apply propotional gain to error
29 I_part = (((error+prev)/2)*(Ts)+Prev_I_part)*I; % write code to apply Intergral gain to error
30 D_part = ((error-prev)/Ts)*D; % write code to apply propotional gain to error
31
32 ControlOut = P_part+I_part+D_part; % write code to apply control output based on P,I,D parts.
33
34 Prev_I_part = I_part;
35 % Insert ControlOut Checks (max & min) Here
36
37 if ControlOut > Max_ControlSig
38     ControlOut = Max_ControlSig;
39 end
40
41 if ControlOut < Min_ControlSig
42     ControlOut = Min_ControlSig;
43 end

```

Figure 19 -- calculation control script MATLAB (2024)

4.2 PID tuning methods

4.2.1 Trial and error

The trial-and-error method for tuning PID parameters follows a set sequence adjusting the proportional then the integral, and finally derivative gain. Typically, it is suggested to start with an existing set of parameters, making small adjustments to optimise performance. For new loops, begin with a cautious initial guess.

Advantages of this method are that it can be quick and easy to obtain reasonable results and little process knowledge is required. Disadvantages are that it is time consuming, and it does not guarantee reaching a stable and robust solution (incatools, 2024).

4.2.2 method-based tuning

Method based tuning involves adjusting the K_p , K_i and K_d parameters to achieve desired control system performance. Common methods include:

1. Ziegler-Nichols Method: A heuristic approach that uses step responses or sustained oscillations to set parameters. This method is simple but often leads to aggressive tuning, causing overshoot and oscillations (Graf, 2013).
2. Cohen-Coon Method: Another rule-based method designed for first-order systems with time delay. It provides better results than Ziegler-Nichols for processes with significant delay (incatools, 2024).

4.2.3 model-based tuning

Model-based PID tuning optimises the K_p , K_i , and K_d parameters by considering control objectives such as disturbance rejection and setpoint tracking, system models, and engineering specs. Unlike trial and error or rule-based methods, which rely on trial and error, model-based tuning offers a structured approach that aligns process behaviour with control needs.

Though initially more time-consuming, once properly tuned, model-based PID settings provide long-lasting benefits with minimal need for adjustment unless the process changes (incatools, 2024).

5.0 Tuning

To tune this PID simulation, it was decided that the trial-and-error method would be most beneficial as it would aid deeper understanding of the tuning of a PID device, and as suggested by incatools (2024) it can be quick and easy but achieve good results.

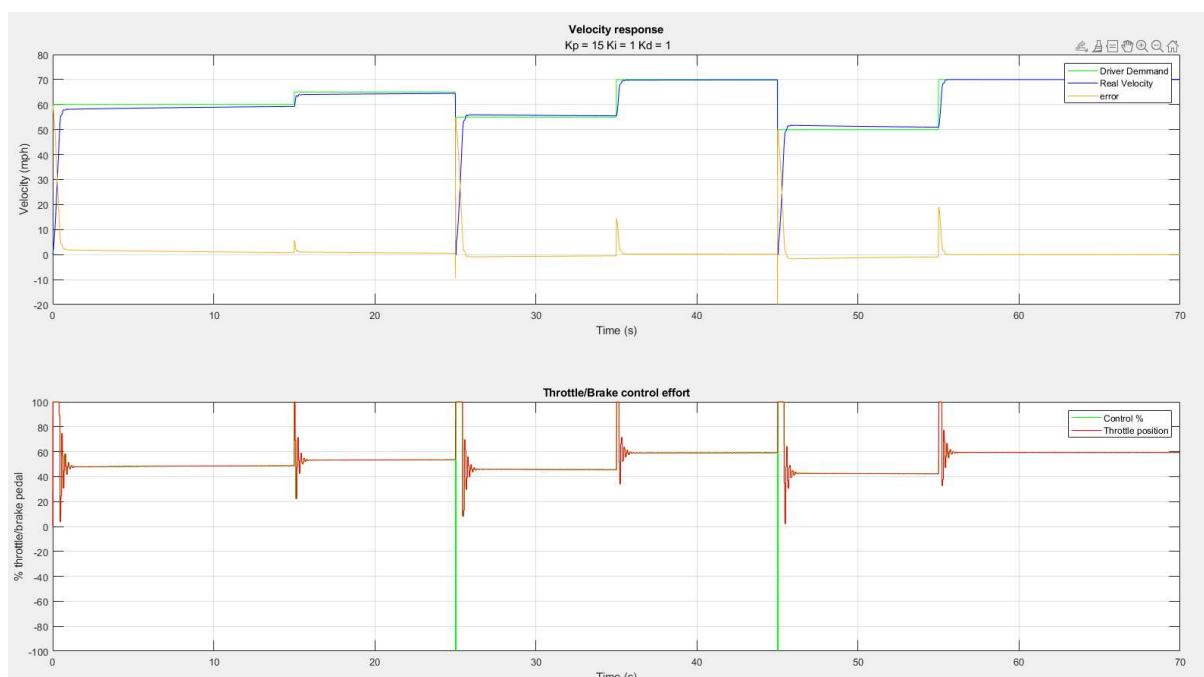


Figure 20 – Initial MATLAB graph created with settings created using trial and error approach (2024)

After using this method, the initial settings were as below: -

K_p – 15

K_i – 1

K_d – 1

These setting yielded the results seen in figure 20 which show no oscillation in output speed, but some oscillation in throttle/brake control output. Very little

overshoot is observed, and steady state error is absorbed by the integral gain setting. Throttle control as shown in the bottom graph though, show a high demand that is oscillating a lot. This oscillation would put extra stress on the throttle and braking system. There is also a discrepancy in the output graph which shows the output speed momentarily drop to 0 miles per hour (mph) when selecting a lower speed. To counteract this sample time was adjusted, as this system does not need to be so responsive. After reducing the sample time, the gains required adjusting again (see figure 21).

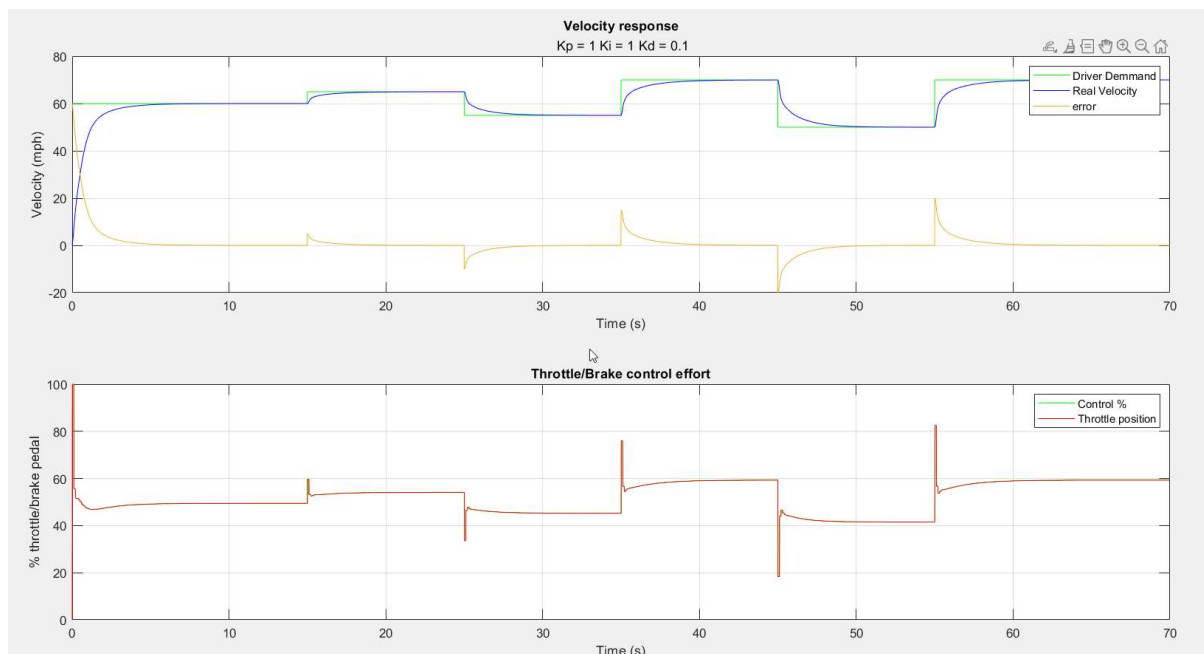


Figure 21 - Final MATLAB graph created with settings created using trial and error approach (2024)

This system with a revised sample time of 0.1 seconds appears to be much more stable. The discrepancy of momentary 0 mph speed has also been resolved. The throttle and brake control effort are now much more stable, with less oscillation, meaning the system is not working as hard to stabilise. These settings would be considered sufficient for a cruise control system in a motor vehicle.

Final settings: -

Sample time - 0.1 seconds

Kp - 1

Ki - 1

Kd - 0.1

6.0 Conclusion

The responsive control system simulated and illustrated in figure 21, page 18 may not accurately reflect a real-life scenario for cruise control of a vehicle. The simulation itself does not consider the performance of the vehicle it is controlling. For example, from 0 mph a demand is set for 60mph. A speed of 52mph is reached within just 1.4 seconds. Not only would this be uncomfortable for a driver, but the fastest production car manufactured today is capable of a 0-60 time of 1.66 seconds (Jones, 2024).

Sample time for the system has also been discovered to be very important as to how a control system functions. The reduction in sample time from 0.05s to 0.1s yielded sufficient results for this control system to be considered sufficient for a motor vehicle cruise control system.

7.0 Bibliography

- Ahmed, F., 2016. *Fluid Power Control Systems*. s.l.:Lulu.com.
- Bolton, W., 2015. *Programmable Logic Controllers*. 6th ed. s.l.:Newnes.
- Dorf, R. C. & Bishop, R. H., 2010. *Modern Control Systems*. 12th ed. s.l.:Pearson.
- Graf, J., 2013. *PID Control: Ziegler-Nichols Tuning*. s.l.:s.n.
- Gupta, V. & Kumar, S., 2018. Application of Fibre Optic Sensors in Structural Health Monitoring. *Journal of Engineering Technology*, 36(4), pp. 211-225.
- Hägglund, T. & Åström, K. J., 2006. *Advanced PID Control*. s.l.:ISA-The Instrumentation, Systems, and Automation Society.
- incatools, 2024. *Explore the 3 PID tuning methods*. [Online]
Available at: <https://www.incatools.com/pid-tuning/pid-tuning-methods/#:~:text=In%20broad%20terms%2C%20there%20are,has%20its%20pros%20and%20cons.>
[Accessed 19 August 2024].
- Johnson, C. D., 2013. *Process Control Instrumentation Technology*. 8th ed. s.l.:Pearson.
- Johnson, R., 2019. Proximity Sensors in Automated Systems. *Journal of Automation Technology*, 34(2), pp. 23-134.
- Jones, E., 2024. *The World's Top 25 Fastest 0-60 MPH Production Cars*. [Online]
Available at: <https://news.dupontregistry.com/features/fastest-0-60-mph-production-cars/>
[Accessed 22 August 2024].
- Mathworks, 2024. *Mathworks*. [Online]
Available at: <https://uk.mathworks.com/matlabcentral/fileexchange/37058-clr-clear-all-close-all-clc>
[Accessed 15 August 2024].
- Mitsubishi Electric, 2024. *Factory Automation*. [Online]
Available at: <https://www.mitsubishielectric.com/fa/products/cnt/plceng/items/index.html>
[Accessed 8 August 2024].
- OMRON, 2024. *OMRON Industrial Automation*. [Online]
Available at: <https://industrial.omron.co.uk/en/products/cx-programmer>
[Accessed 8 August 2024].
- Patel, A., 2018. Pneumatic Actuators in Material Handling Systems. *International Journal of Industrial Engineering*, 45(1), pp. 56-67.
- Rockwell Automation, 2024. *PLC Programmable Controllers*. [Online]
Available at: <https://www.rockwellautomation.com/en-us/products/hardware/allen->

<bradley/programmable-controllers.html>

[Accessed 08 08 2024].

Seborg, D. E., Edgar, T. F., Mellichamp, D. A. & Doyle III, F. J., 2016. *Process Dynamics and Control*. 4th ed. s.l.:s.n.

Siemens , 2024. *From PROFIBUS to PROFINET – Time to change over*. [Online] Available at: <https://www.siemens.com/us/en/products/automation/industrial-communication/profinet/pb-2-pn.html>

[Accessed 9 August 2024].

Walker, J. & Hill, M., 2021. Pressure Sensor Applications in Hydraulic Systems. *Journal of Mechanical Engineering*, 58(3), pp. 301-310.

8.0 Appendix A

```
%% (1) Initialize %%

clc
clear all
close all % house keeping

% define global variables
global sim_step u j u_vec

% initialize vector where to store results for plots
d_vec = []; u_vec = []; y_vec = []; e_vec = [] ; T_vec = [];

P = 15; % proportional gain
I = 1; % integral gain
D = 1; % derivative gain

%% (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]
min_velocityS = 0;
Max_velocityS = 150;
n_bits_sensor_signal = 16;

% number of bits used to encode velocity demand signal. demand
% range in mph [30,110]
min_demand = 30; Max_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; % define simulation steps vector
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;

%% Initialise Values - (Removed when in function)
min_ControlSig = -100;
Max_ControlSig = 100;
u = 0;
control = 0;
velocity_real = 0;

%% (4) simulation loop %%

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

    loops_remaining = length(T)-j % display progression

    % 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)*(Max_velocityS-
min_velocityS))/R)+min_velocityS; % INCORRECT DECODING! INSERT YOUR DECODING
FUNCTION HERE

    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)*(Max_demand-min_demand))/R)+min_demand;%
INCORRECT DECODING! INSERT YOUR DECODING FUNCTION HERE

        % calculate control signal (range [-100,100])

        [control, error] =
CalcControl_SC(demand,velocity,control,sim_step,Ts,j,P,I,D,e_vec); % INCORRECT
CONTROL FUNCTION! INSERT CORRECT FUNCTION HERE

        % encode control signal into x-bit binary number

        control_raw = dec2bin(((control-min_ControlSig)/(Max_ControlSig-
min_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 = [e_vec error]; % Error Value Vector
    d_vec = [d_vec demand]; % Demand Value Vector
    u_vec = [u_vec control]; % Control Percentage Value Vector
    y_vec = [y_vec velocity_real]; % Real Velocuty Value Vector
    T_vec = [T_vec throttle_setting]; % Throttle Setting Value Vector

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')

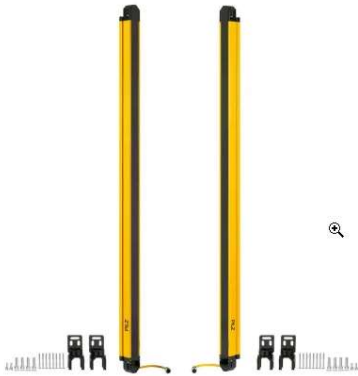
% plot throttle/brake actuator effort
subplot(2,1,2)
stairs(T,u_vec,'g-');
hold on;
plot(T,T_vec,'r-'); grid;
title('Throttle/Brake control effort');
ylabel('% throttle/brake pedal'); xlabel('Time (s)');
legend ('Control %','Throttle position')

```

7.1 Appendix B

Pilz PSEN opII4H Series Light Curtain Receiver, Transmitter, 30mm Resolution

RS Stock No.: 191-9845 | Mfr. Part No.: 632065 | Brand: [Pilz](#)



Price Each

£1,692.93

(exc. VAT)

£2,031.52

(inc. VAT)

1

+

Units

Add to basket

14 In stock for delivery within 1 working days

FREE delivery for orders over £50.00

Check stock levels

Add to a parts list

Not available for guaranteed pre 9am/10am delivery

View all Light Curtains

Units	Per unit
1 +	£1,692.93

Figure 3