Autonomy I Final Project Report

N. Keller, J. Poirier

December 9, 2025

Abstract

This project explores controller design for an autonomous vehicle. The goal of this project is to apply all of the previous learnings of this class to solve various problems of autonomous driving. Tasks 1 through 3 serve as set up and validation for code and controller designs to be used for the more open-ended Task 4, which implements autonomous lane changing and fuel optimization.

Contents

1 Task 1

1.1 Deriving $F_{d,max}$

The first step is to derive the maximum force that the engine can provide.

1.1.1 Givens & Assumptions

- The engine's relationship to force is modeled with **equation 1**. In this equation, η_g and η_d are related to the gear and drive ratios, r_w is the wheel radius, and ζ is a constant which describes the efficiency of the drive train.

$$T_e = \frac{1}{\zeta}\frac{r_w}{\eta_g \eta_d} F_d \tag{1}$$

- The maximum engine torque ($T_{e,max}$) is $200N \cdot m$.

1.1.2 Derivation

Equation 1 is simply rewritten as **equation 2**.

$$F_{d,max} = \frac{T_{e,max}\ \eta_g \eta_d}{r_w}\zeta \tag{2}$$

The solved value for $F_{d,max}$ given the vehicle parameter values given in task 1 of the project assignment is 1698.82 Newtons

1.2 Beta Demonstration

Code was then written to plot the road profile, β, as a function of horizontal distance x. This plot was created using an amplitude, Amp, of 3. The initial 500 meters of road grade is set to be flat, and the remaining road grade is calculated via **equation 3**, resulting in the plot shown in **Figure 1**.

$$\beta = Amp * \sin(\frac{2\pi}{1000 * x} + 300) \tag{3}$$

1.3 BSFC Contour and Surface

Next, the contour and surface plots for the brake-specific fuel consumption (BSFC) are created. The BSFC model used for this project is shown by **equation 4**, where N_e is the engine speed defined by **equation 5**.

$$BSFC = (\frac{N_e - 2700}{12000})^2 + (\frac{T_e - 150}{600})^2 + 0.07 \tag{4}$$

$$N_e = \frac{60}{2\pi}\frac{\eta_g \eta_d}{r_w} v \tag{5}$$

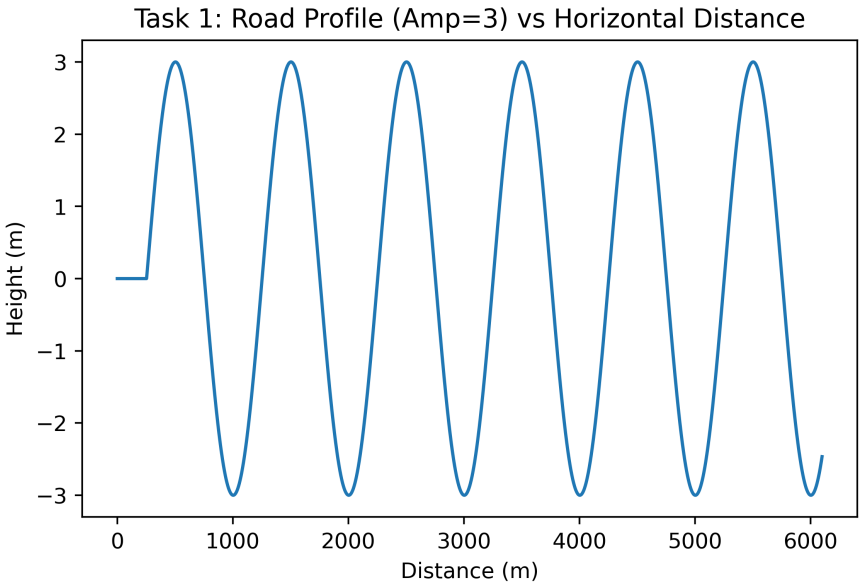

Figure 1: Road Grade β as a function of Horizontal Distance x

Using these equations, the contour and surface plots of the BSFC function can be created. These plots are shown below in **Figures 2 and 3**.

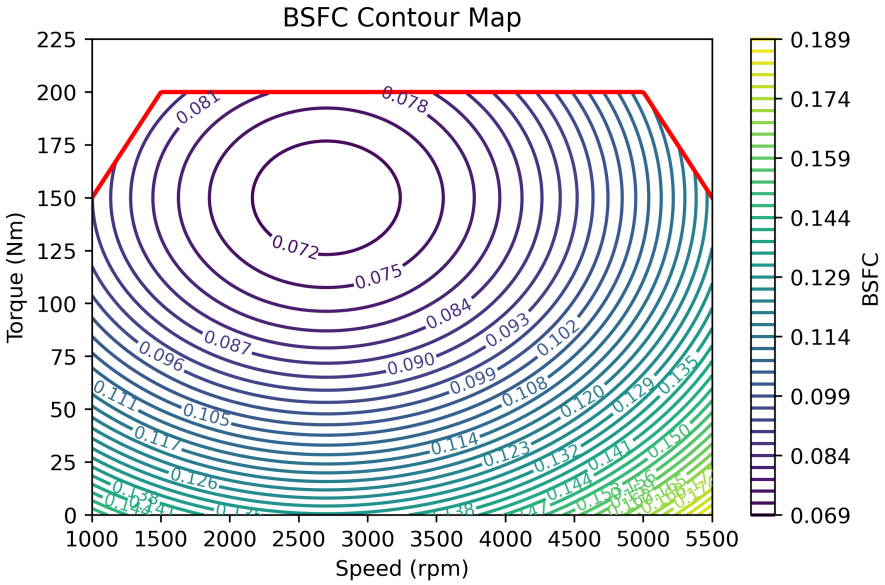

Figure 2: BSFC Contour Plot

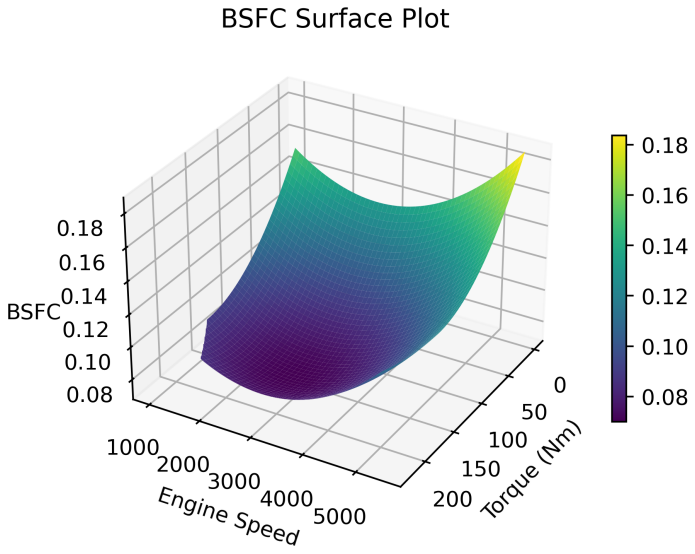

Figure 3: BSFC Surface Plot

1.4 Fuel Conversion Factor

Fuel economy is typically measured in miles per gallon (MPG) in the US, but this project model cares about fuel consumption in milligrams per second. Thus, the conversion factor to get from mg/s to MPG must be calculated.

1.4.1 Givens and Assumptions

- Converting from mg/s directly to MPG is not possible in isolation, as they are not equivalent types of measurements, and so it must be known how many grams one gallon of gasoline is equivalent to. It is given that one gallon of gasoline is 2835 grams.

1.4.2 Derivation

The series of unit conversions needed is shown in **equation 6**.

$$(\frac{meter}{milligram})(\frac{mile}{1609.34meters})(\frac{2835grams}{gallon})(\frac{1000milligrams}{gram}) = (\frac{miles}{gallon}) \tag{6}$$

Evaluating this conversion yields a conversion factor to get from mg/s to MPG of 1761.59.

1.5 Finding $F_D^{ss}(v, \beta)$

1.5.1 Givens & Assumptions

- Mass m of the car is 1300 Kg, and acceleration due to gravity, g, is 9.8 m/s^2.

- $F_D^{ss}(v, \beta)$ defines the relationship between the current speed and incline, with the driving force required to sustain that state (F_d^{ss}).
- The plant dynamics is given in **equation 7**.

$$m\dot{v} = F_d - F_{air} - F_c - F_{roll} \tag{7}$$

- F_c refers to the force applied by gravity on sloped sections of road and is defined in **equation 8**. In this context, β refers to the grade of the road.

$$F_c = mg\sin(\beta) \tag{8}$$

- The nonlinear equation governing air resistance is shown in **equation 9**. Where v is the speed of the car.

$$F_{air} = av^2 + bv \tag{9}$$

- The values of a and b are 0.2 Ns^2/m^2 and 20 Ns/m respectively.
- The steady state speed is 27.78 m/s (V_0^{ss}).
- F_{roll}^{ss} is defined as 100 newtons at high speeds

1.5.2 Derivation

To find F_d^{ss}, it is required to balance all forces such that $\sum_i F_i = 0$. This is given in **equation 10**. A further substitution leads to the equation for $F_d^{ss}(v, \beta)$ in **equation 11**.

$$F_d^{ss} = F_{air}^{ss} + F_c^{ss} + F_{roll}^{ss} \tag{10}$$

$$F_d^{ss}(v, \beta) = av^2 + bv + mg\sin(\beta) + F_{roll}^{ss} \tag{11}$$

Evaluating **equation 11** at V_0^{ss} with β=0 yields the result 809.95 N.

1.6 Finding $J^{ss}(v, \beta)$

1.6.1 Givens & Assumptions

- $J^{ss}(v, \beta)$ is defined as the steady state fuel rate, in mg/s, defined by

$$\text{fuel rate} = \max(\frac{1}{\zeta}\text{BSFC} * F_d * v,\ \bar{F}) \tag{12}$$

where BSFC refers to **equation 4**, and F_d refers to the steady state value found from **equation 11**.

1.6.2 Derivation

The derivation to get the expression for $J^{ss}(v, \beta)$ is a simple substitution of the BSFC equation, yielding the rather nasty looking **equation 13**.

$$J^{ss}(v,\beta) = \max\left(\frac{1}{\zeta} * \left[\left(\frac{\frac{60}{2\pi}\frac{\eta_g\eta_d}{r_w}v - 2700}{12000}\right)^2 + \left(\frac{\frac{1}{\zeta}\frac{r_w}{\eta_g\eta_d}F_d^{ss} - 150}{600}\right)^2 + 0.07\right] * F_d^{ss} * v,\ \bar{F}\right) \quad (13)$$

Using **equation 13**, a surface plot of $J^{ss}(v, \beta)$ is created over the ranges of $v \in [23,\ 28]$ m/s and $\beta \in [-2.5,\ 2.5]$ degrees (which are converted to radians). This plot is shown in **Figure 4**.

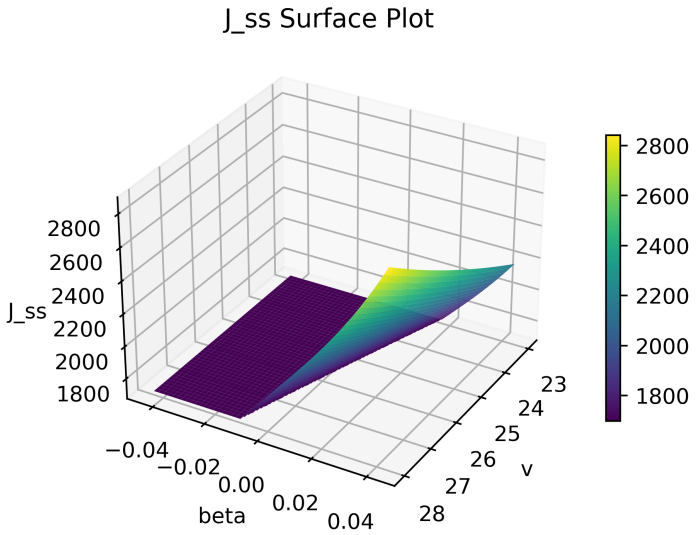

Figure 4: $J^{ss}(v, \beta)$ Surface Plot

1.7 Model Verification

Now that the model thus far has been derived and implemented, it must be verified. First, the model will be verified using a simulated flat road (i.e. Amp=0). Python code was written to simulate the car dynamics as derived previously in this report, along with the Car class created. A simulation loop is initialized to simulate the car for 150 seconds with a step input F_d at t = 50 seconds, using an initial value of $F_d^{ss}(27.78, 0)$ (found to be 809.95 N.m) and a final value of 3000 N.m. Running this simulation yields the plots shown in **Figure 5**.

Next, the same simulation is run again, except this time with an amplitude of Amp=3, which corresponds to a maximum road grade of 5%. Additionally, the step input is removed, and instead a constant input of $F_d^{ss}(27.78, 0)$ is used for the entire simulation. This simulation yields the plots shown in **Figure 6**.

These results may seem out of place at first glance, but make sense when explained. First, the intuitive thought is that since the force applied by the engine during the simulation is

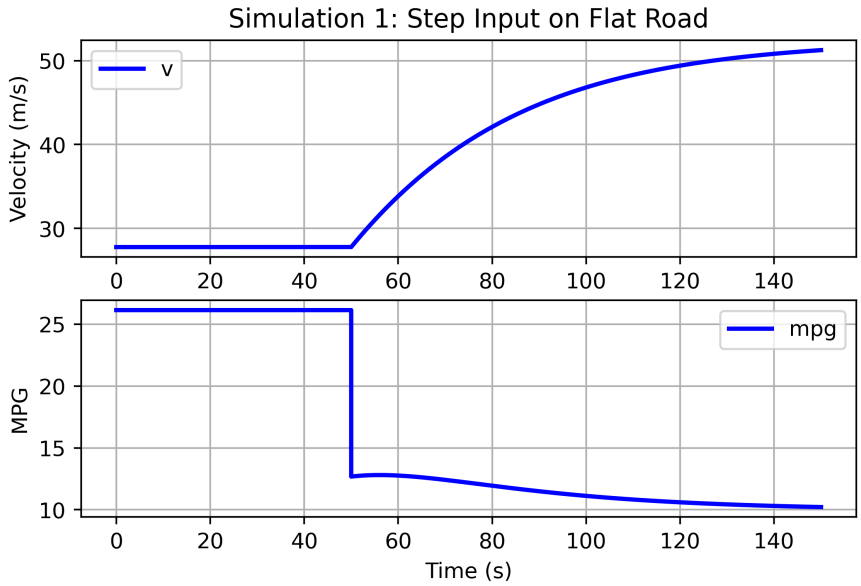

Figure 5: Task 1 Model Validation: Flat Road

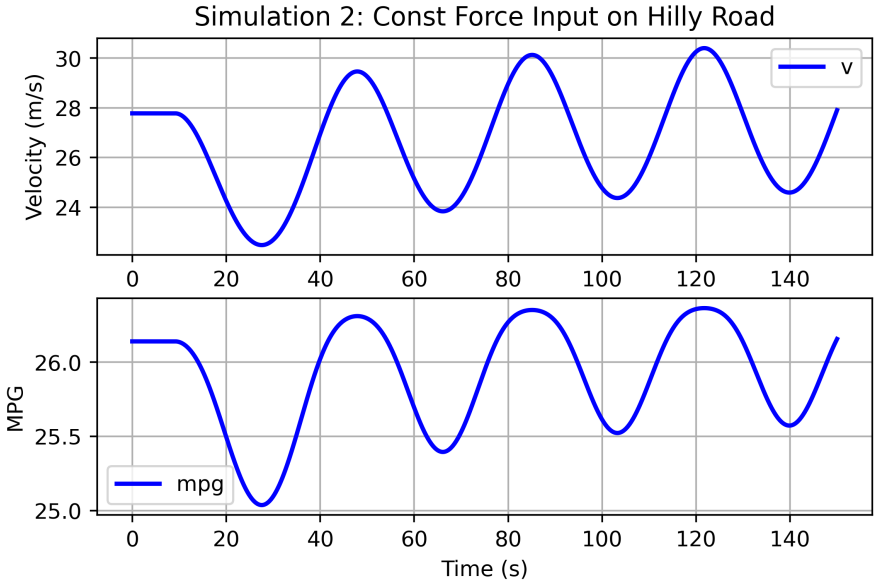

Figure 6: Task 1 Model Validation: Hilly Road

constant, one would imagine that fuel economy would also be constant. However, this incorrectly assumes that fuel economy is directly proportional to engine force. In reality, fuel economy is related to engine power, not force. The power necessary to maintain a constant force is *not constant*, and it changes with speed and road grade. Fuel economy is affected by where on the BSFC map shown in **Figure 3** the current operating point is, so as

that operating point changes, so does the fuel economy. For the hilly road simulation, when the car is climbing uphill, the constant engine force will have differing forces to overcome, with larger uphills being more difficult and steeper downhills being less difficult. Because of the oscillatory forces in the hilly road, the fuel efficiency and velocity outputs also exhibit oscillatory behavior.

2 Task 2

2.1 Finding $\Delta V \longrightarrow \Delta F_d$

This deliverable is to find the linearized system that models the plant.

2.1.1 Givens & Assumptions

- All given and assumptions from **section 1.5.1**
- The desired form of the final equation for F_{air} is in **equation 14**.

$$F_{air}(v) \approx cv = F_{lin}(v) \tag{14}$$

- The rolling friction (F_{roll}), and the force from gravity on a slope (F_c) can be ignored in this model.

2.1.2 Derivation

In order to find the linearized system it is first required to linearize all the component functions. In this case the dynamics are reduced down to **equation 15**. In this equation, F_d is an input, so only F_{air} needs to be linearized.

$$m\Delta\dot{v} = \Delta F_d - F_{air} \tag{15}$$

Linearizing F_{air}

Using a first-order Taylor series as shown in **equation 16** expansion we can find the equation for ΔF_{air}. The simplified form can be found in **equation 17**.

$$\begin{aligned} F_{air}(v) \approx F_{lin}(v) &= F_{air}(v_0^{ss}) + (v - v_0^{ss})F'_{air}(v_0^{ss}) \\ &= (v - v_0)(2av_0 + b) = (v - v_0) * C \end{aligned} \tag{16}$$

$$C = (2av_0 + b) \tag{17}$$

If the velocity used for linearization is V_0^{ss} (the equivariant of 100km/h), the final value for C is 31.112.

Finding the Differential Equation & Transfer Function

The differential equation is found by substituting the linearized F_{air} into the ODE. This yields **equation 18** which simplifies to **equation 19**

$$m\Delta\dot{v} = \Delta F_d - C\Delta V \tag{18}$$

$$\begin{aligned} ms\Delta V &= \Delta F_d - C\Delta V \\ \Delta V(ms + C) &= \Delta F_D \\ \frac{\Delta V}{\Delta F_d} &= \frac{1}{ms + c} \end{aligned} \tag{19}$$

Step Simulation & Model Validation

In order to verify this result, a step response was simulated. The first linearization was performed with a step of 1 Newton. This is shown in **figure 7**. It is clear that with this small of a step the linear approximation is nearly identical to the non-linear model. This process was repeated with a step size of 600 Newtons. This is shown in **figure 8**. The same procedure was repeated at a different equilibrium point, specifically where $V_0 = 150$ km/hr (41.67 m/s). These are shown in **figures 9** and **10**. It is easy to see that selecting a relevant equilibrium point is important for simulating the system.

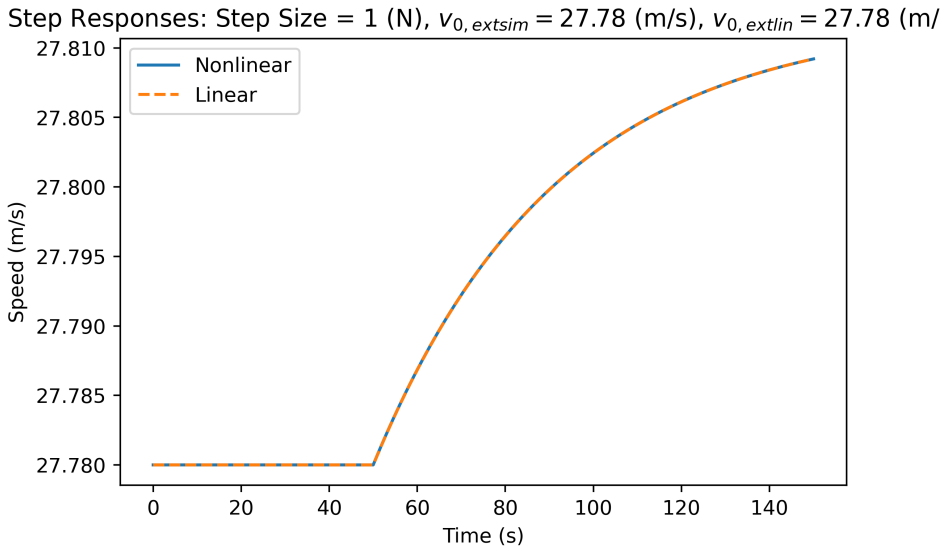

Figure 7: Task 2 linearized system: 1 Newton step

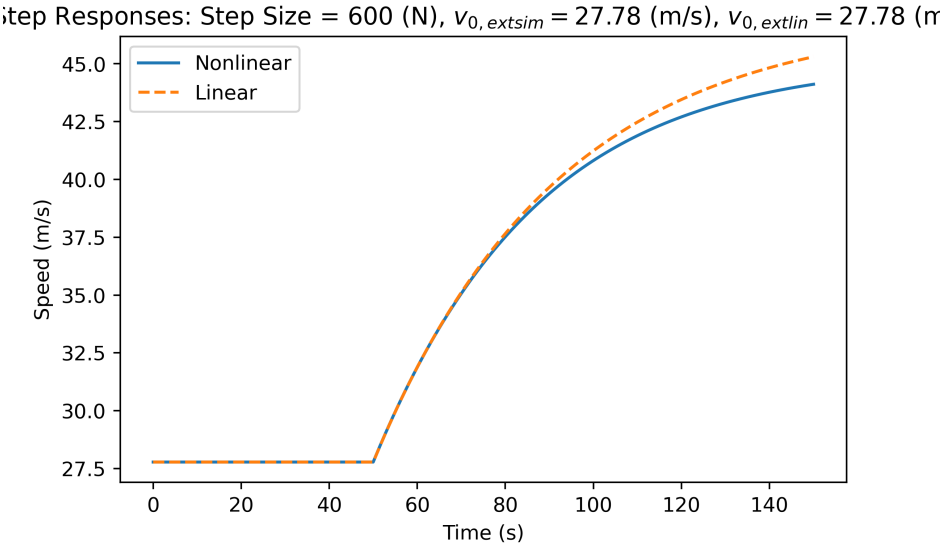

Figure 8: Task 2 linearized system: 600 Newton step

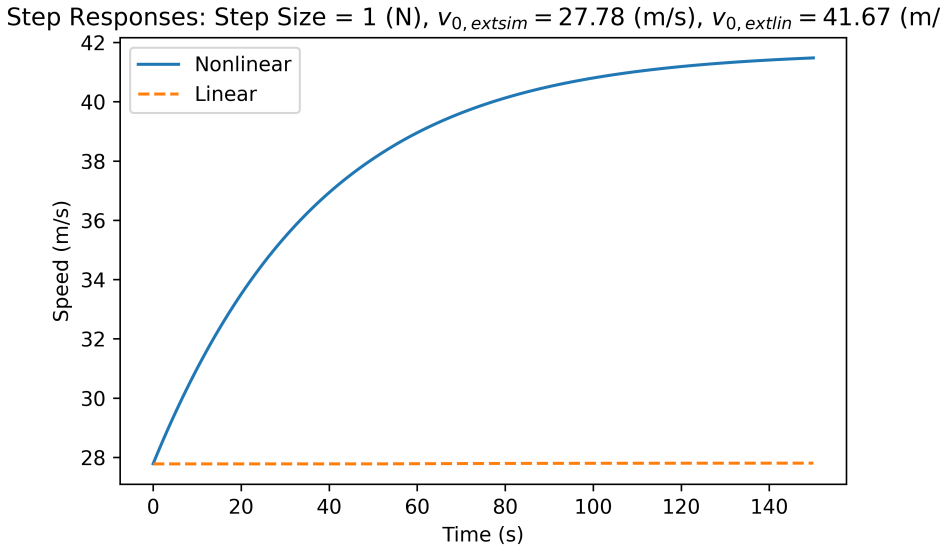

Figure 9: Task 2 linearized system (alternate speed): 1 Newton step

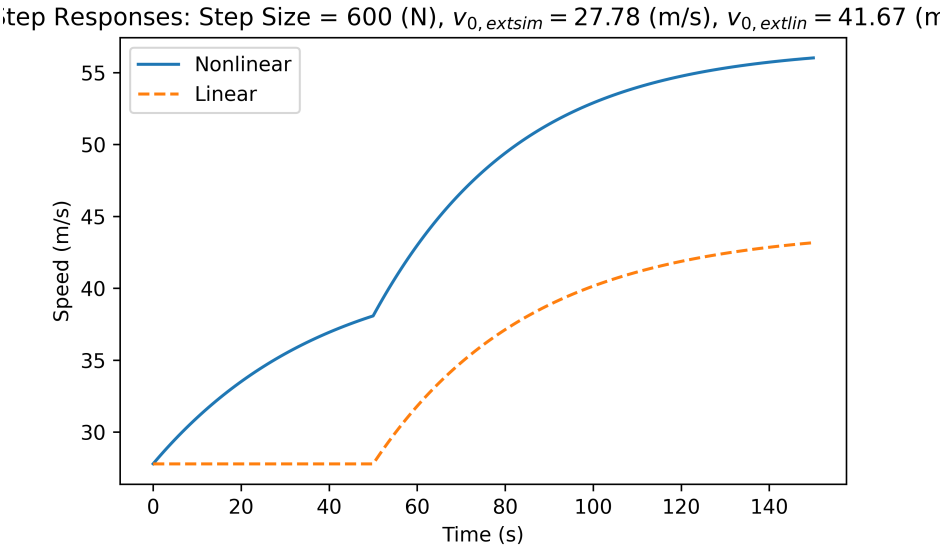

Figure 10: Task 2 linearized system (alternate speed): 600 Newton step

These plots show that, especially at small steps in the input, the linearized model very effectively approximates the nonlinear dynamics around the equilibrium point at which it was linearized. For these systems in particular, changing the equilibrium point changes only c. Even with large step inputs, the difference between the linearized model and the nonlinear model is minimal. Linearizing about a different equilibrium point than the one being simulated at yields a much worse approximation. However, this is expected. When linearizing a system, it can only be done at an equilibrium point. So, as you move into operating regions farther and farther from that equilibrium point, the linear approximation

becomes worse.

2.2 Designing the Velocity controller

The next step is to design the cruise controller to autonomously adjust the vehicle speed.

2.2.1 Givens & Assumptions

- The only measurement available to the controller is the vehicle's longitudinal speed, v.
- The controller design must meet the following specifications:
  - Zero steady state tracking error despite constant disturbances.
  - Rise time for step tracking must be between 1 and 3 seconds (time to go from 10% to 90% of final value).
  - No overshoot for step tracking.
  - Closed-loop response to a unit step disturbance applied at the plant should settle in less than 10s (defined as the time it takes for the output to reach and remain within [-1×10^{-6}, 1×10^{-6}]).

2.2.2 Controller Design

Using the transfer function from **equation 19** at the operating point of 27.78 m/s, a PI controller is designed to track speed setpoints. In the context of **equation 19**, c is the acting forces on the vehicle due to physics. This is calculated using **equation 17**, and is found to have a value of 31.112 N. **Equation 19** is the plant for this system. The general form of a PI controller is shown in **equation 20**, and the general form of the closed-loop transfer function (CLTF) is shown in **equation 21**.

$$C(s) = \frac{K_p s + K_i}{s} \tag{20}$$

$$\text{CLTF} = \frac{C(s)P(s)}{1 + C(s)P(s)} \tag{21}$$

Given the controller from **equation 20** and the plant from **equation 19**, the full CLTF can be written as shown in **equation 22**.

$$\text{CLTF} = \frac{K_p s + K_i}{ms^2 + (c + K_p)s + K_i} \tag{22}$$

This CLTF has a zero at $\frac{K_i}{K_p}$, and thus a precompensator is used to cancel out this zero. The precompensator must also have unity gain, and thus the precompensator is designed as shown in **equation 23**.

$$F(s) = \frac{K_i}{K_p s + K_i} \tag{23}$$

The new transfer function, implementing the precompensator to cancel the zero, becomes that shown in **equation 24**.

$$\text{TF} = F(s)\text{CLTF} = \frac{K_i}{ms^2 + (c + K_p)s + K_i} \tag{24}$$

Now, the design specifications ask for no overshoot, but a quick response is still desireable. Thus, the controller will be designed to be critically damped (i.e $\zeta = 1$). The generic form of a transfer function is shown in **equation 25**.

$$\text{Generic Form} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{25}$$

Using **equation 25**, coefficient mathcing can be used to get solution forms for K_i and K_p, which are shown in **equations 26 and 27**.

$$K_i = m\omega_n^2 \tag{26}$$

$$K_p = 2\zeta\omega_n m - c \tag{27}$$

These equations are then used, along with heuristics, to design the controller. The common heuristic used is that rise time, t_r, is approximately $\frac{3.35}{\omega_n}$. The specifications of the controller require the rise time to be between 1 and 3 seconds, so for design, it is set to be 2 seconds. This yields that $\omega_n = 1.675$ rad/s. With $m = 1300$ kg defined, $\zeta = 1$ for a critically damped system, $c = 31.112$ as found from **equation 17**, and now $\omega_n = 1.675$, the values for K_i and K_p can be evaluated. This yields the final design results of $K_i = 3647.3125$ and $K_p = 4323.888$. The block diagram for this system is shown in **Figure 11**.

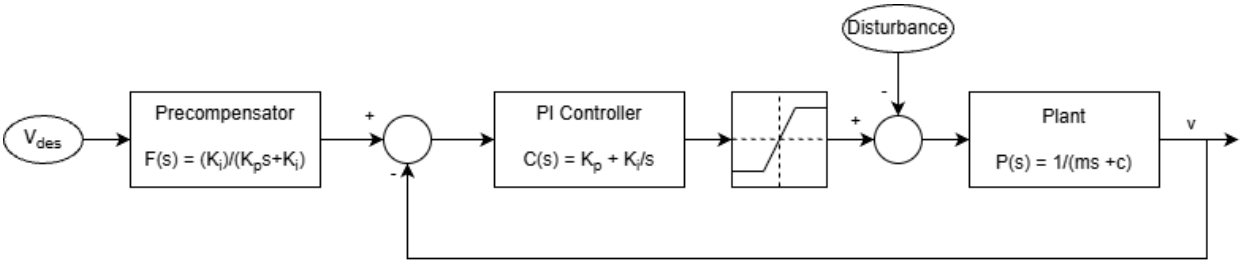

Figure 11: Task 2 System Block Diagram

Controller Validation

With the controller now designed, it can be validated. The script shown in **appendix section 6.1.5** is a script used to run analysis on the designed controller. This script outputs a step response plot, as well as various metrics like overshoot and rise time. The step response is shown in **Figure 12**.

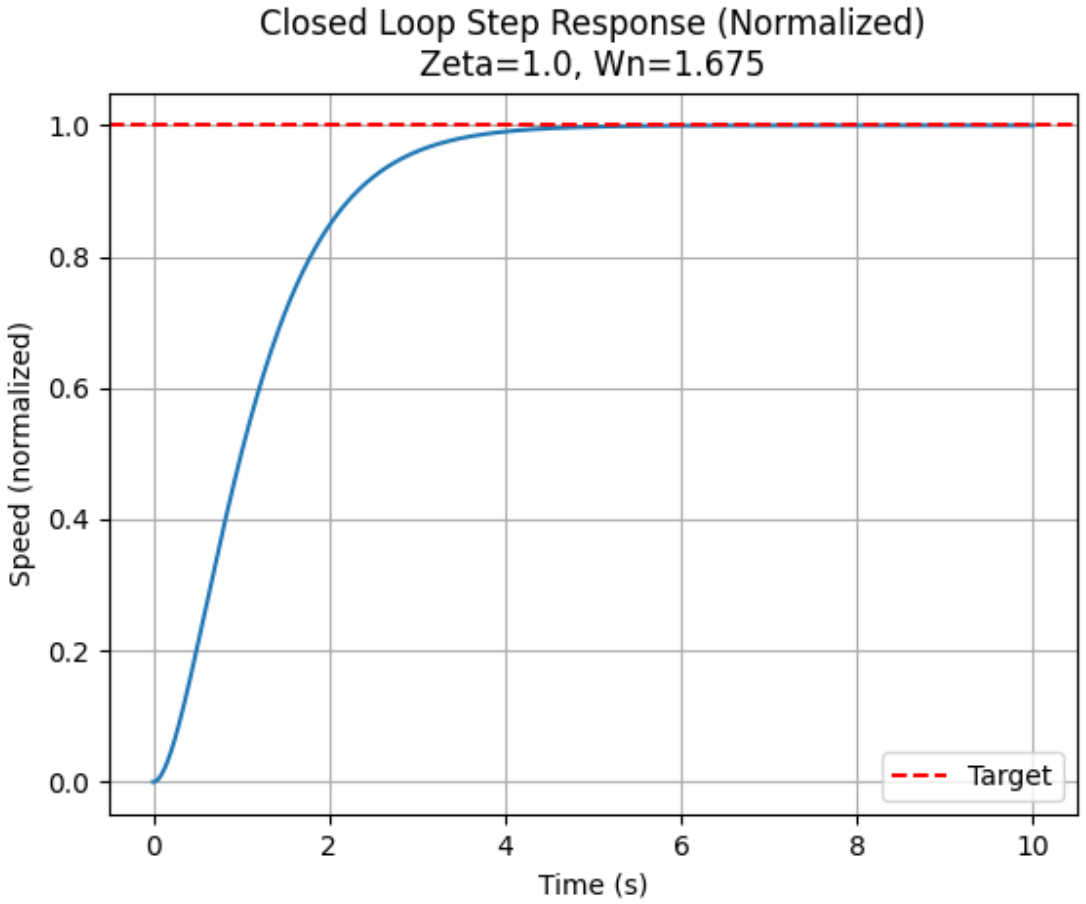

Figure 12: Task 2 Controller Step Response

The step response output of the controller is exactly as desired. It has no overshoot, perfect tracking, a rise time of 2.002 seconds, and a settling time of 3.4835 seconds, all of which are well withing the specifications set for the controller.

The gain and phase margins of this controller can also be observed. These plots are shown in **Figure 13**. These gain and phase margins show that the controller is both stable and robust.

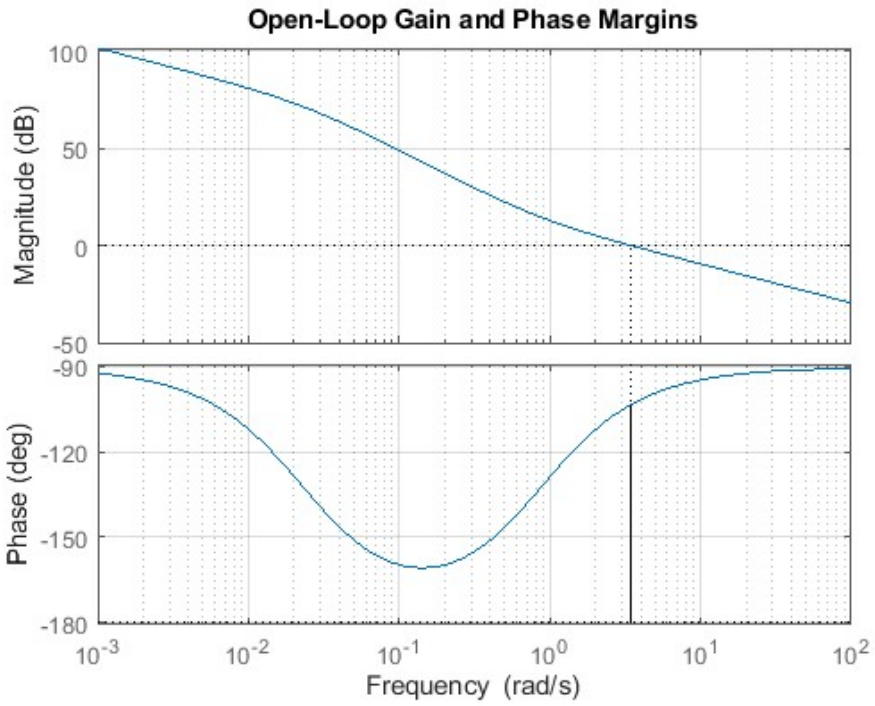

Figure 13: Task 2 Controller Gain and Phase Margins

Controller Implementation

Now that the controller design has been verified, it can be implemented to work with the car class previously created. The controller code is created to implement the designed controller. This code is shown in **appendix section 6.1.8**. For the first implementation test, anti-windup is set to 0, and the step response is simulated with a step at t=50, starting from 100km/h and stepping up to 101km/h. When this simulation is run, the plot in **Figure 14**.

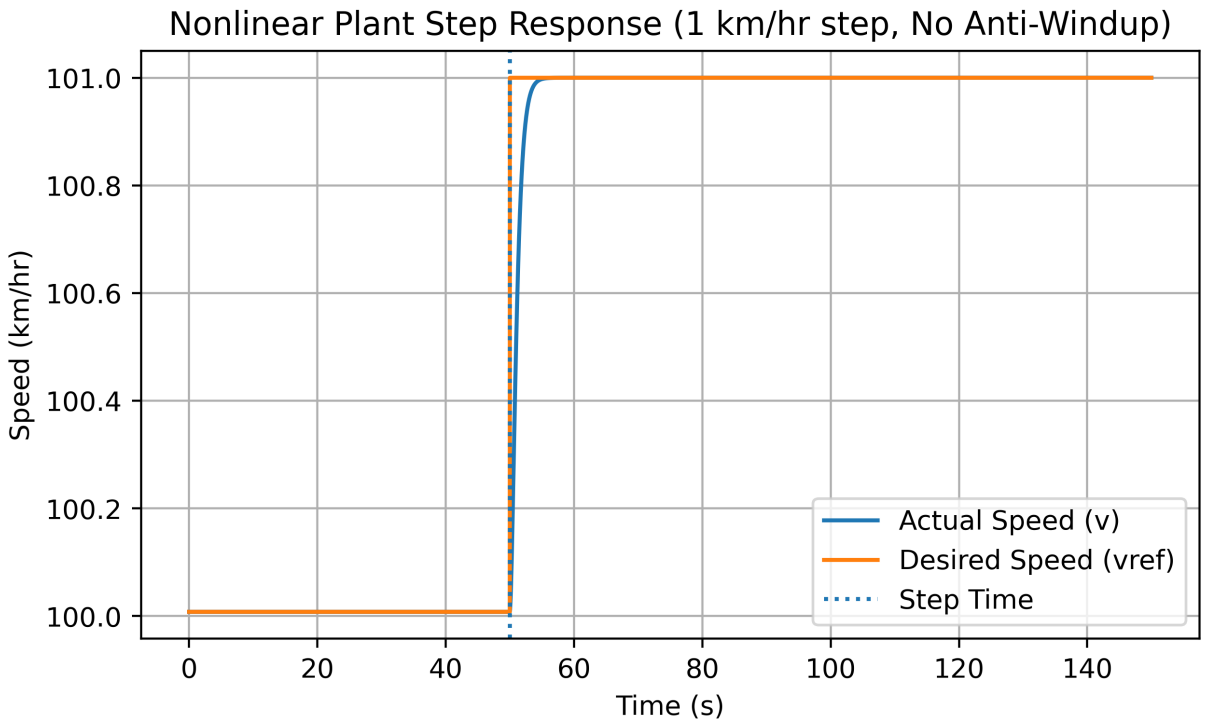

Figure 14: Task 2 Controller Step Response - No Anti-Windup, 1km/h Step

The step response is then run again, this time with a step size of 50km/h (i.e. starting at 100km/h, ending at 150km/h). The plot generated from this is shown in **Figure 15**.

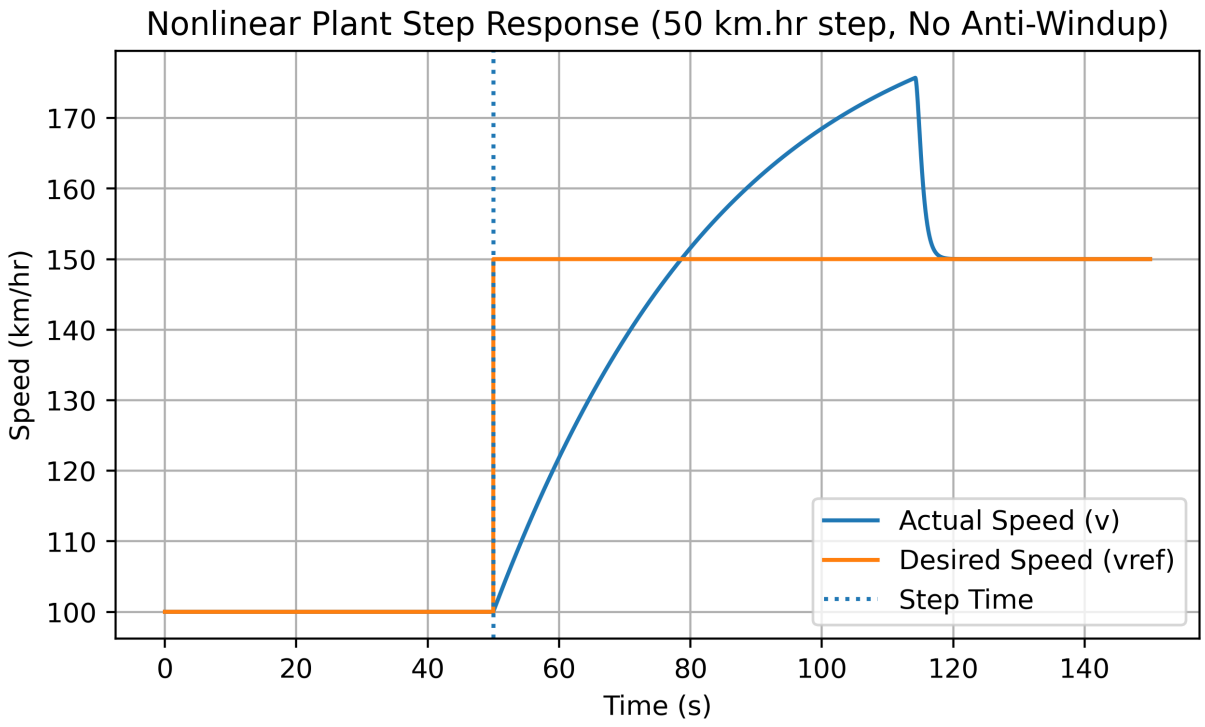

Figure 15: Task 2 Controller Step Response - No Anti-Windup, 50km/h Step

This plot obviously does not meet the design specifications of the controller, since there is a huge overshoot. However, this is expected. Since there is no integral anti-windup implemented in this step response, the integral winds up much past the desired set point, and so there is a massive overshoot. The last simulation keeps the 50km/h step, but includes anti-windup. This is shown in **Figure 16**.

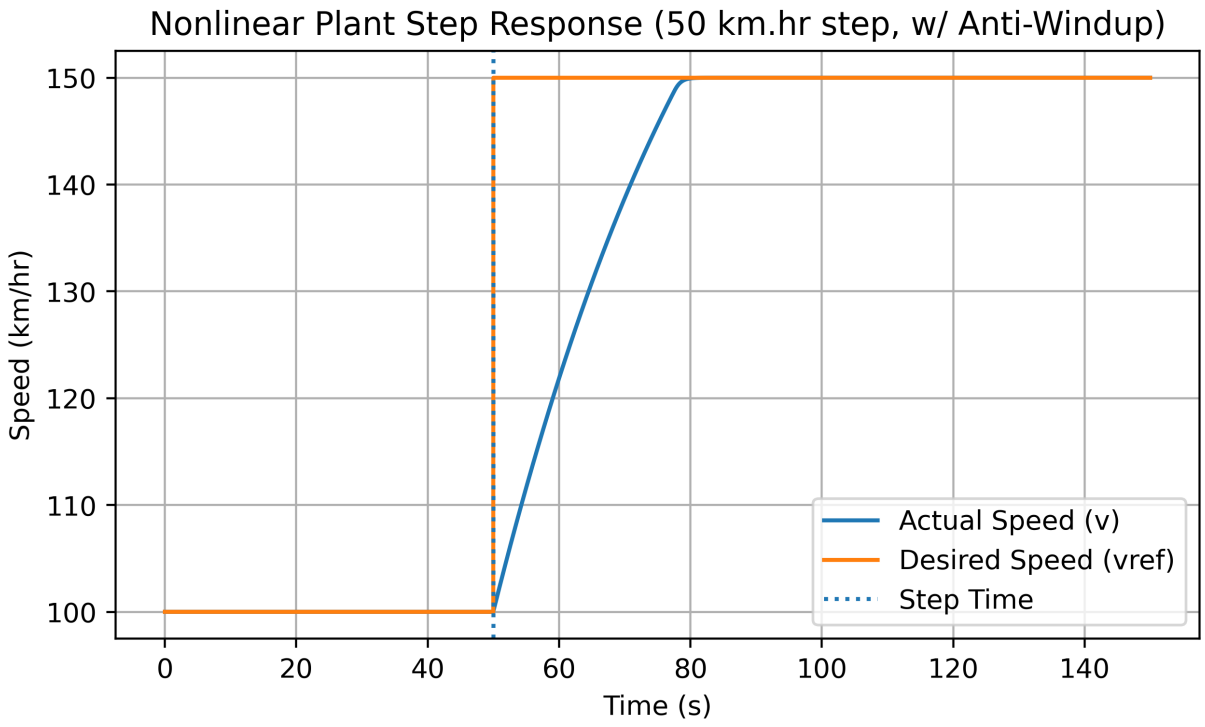

Figure 16: Task 2 Controller Step Response - with Anti-Windup, 50km/h Step

With the anti-windup integrated, the controller again meets the design specifications necessary for the controller. The final point of validation is to look at total fuel usage. For this simulation, the road grade amplitude is set to 3, and the step input is turned off and set to a constant 100km/h input. The simulation is then run with the gains found in the controller design, and again with half the value of those gains. The total fuel usage found for the full-gains simulation is 269,833.49 mg, and the total fuel usage for the half-gains simulation is 269,632.75 mg. The reason that the halved gains produce a more fuel-efficient vehicle is because lowered controller gains produces more aggressive throttle control. When the throttle control is more aggressive, the vehicle is able to operate in the optimal range for more of the simulation, but it also lowers performance based on the metrics provided by the design specifications. If the code from **appendix section 6.1.5** is run with the halved values of the gains, it results in a significant overshoot, as well as a worse settling time. Since the controller is more aggressive, the rise time is faster, but it then violates the other design specifications. The plot produced by running this controller analysis is shown in **Figure 17**.

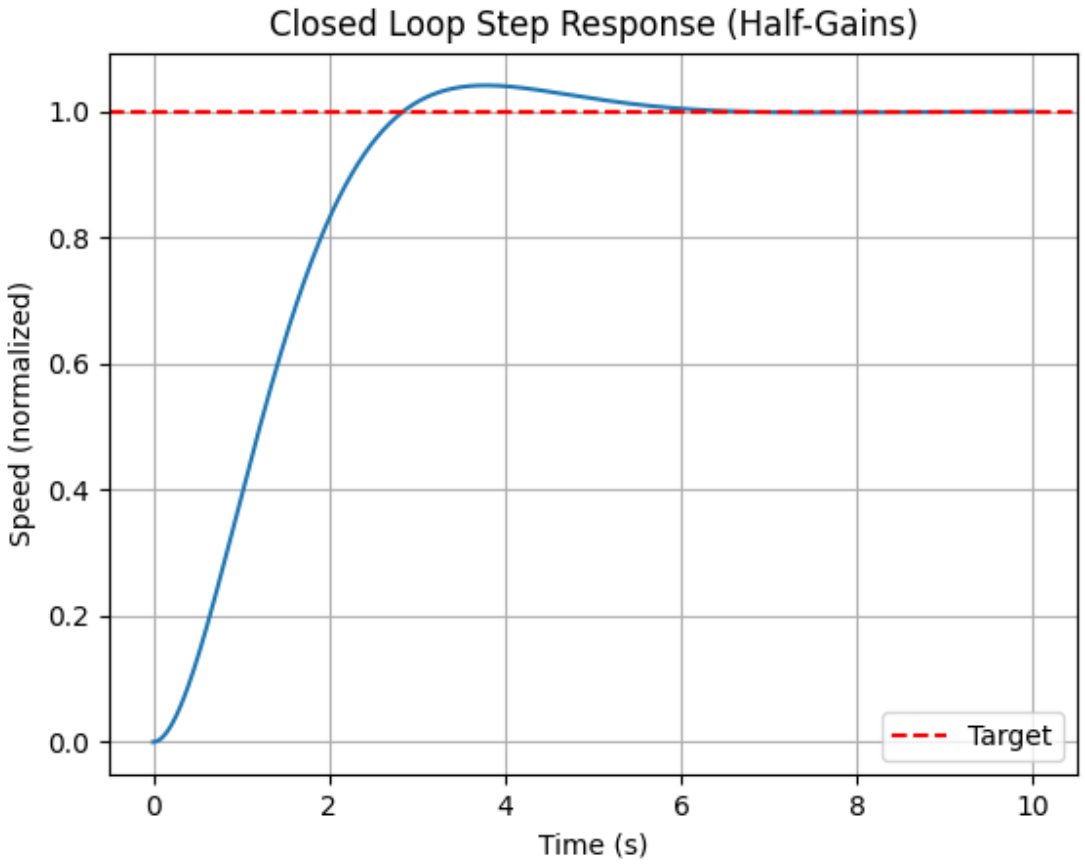

Figure 17: Task 2 Controller Step Response - Half Gains

3 Task 3

3.1 Finding $\Delta V \longrightarrow \Delta F_d$

This deliverable is to find the linearized system that models the plant.

3.1.1 Givens & Assumptions

- The heading and lateral Dynamics are modeled by **equation 28** and **equation 29**, where y is the lateral position, ψ is the heading, L is the length of the wheelbase, and δ is the steering angle.

$$\dot{\psi} = \frac{v}{L} tan(\delta) \tag{28}$$

$$\dot{y} = v \sin(\psi) \tag{29}$$

- δ has a limit value of $\pm\delta_{max} = \pm 0.05$ radians.
- The lateral position controller requirements are:
  1. No overshoot
  2. Settling time less than three seconds
  3. Perfect steady state tracking despite step disturbance
- It is well known that settling time is given by:

$$\sigma = \frac{4.6}{t_s}$$

  where t_s is the settling time, and sigma is the location of the pole closest to the origin.
- All given and assumptions from **section 1.5.1**

3.2 Linearize ODE

To linearize the ODE for this task, the same process is used as in **section 2.1.2**. To begin, an equilibrium point must be chosen. v_0 was chosen to be 27.78 m/s, as this is the contsant value that the car has been tested at throughout this project. For the heading, ψ_0, and the steering angle, δ_0, both were chosen to be 0 degrees. This is because the operating range of the vehicle is symmetric (i.e. the car could turn 15 degrees to the left, or 15 degrees to the right), so 0 degrees is the center point. Additionally, the car will be travelling straight ahead most of the time, and thus its steering angle and heading will most often be 0 degrees.

With the equilibrium point chosen, the ODE can be linearized about that point. The small-angle approximation, which is derived from the taylor series, can be used for this. The small angle approximation states that $\sin(\psi) \approx \psi$ for small angles ψ. Additionally, it states

that $\tan(\psi) \approx \psi$ for small angles ψ Given this, the functions shown in **equations 28 and 29** can be approximated as **equations 30 and ??**.

$$\Delta\dot{\psi} = \frac{v_0}{L}\Delta\delta \tag{30}$$

$$\Delta\dot{y} = v_0\Delta\psi \tag{31}$$

$$\Delta\ddot{y} = v_0\Delta\dot{\psi} = \frac{v_0^2}{L}\Delta\delta \tag{32}$$

Given these linear approximations, the transfer functions $\frac{\Delta\Psi(s)}{\Delta\delta(s)}$ and $\frac{\Delta Y(s)}{\Delta\Psi(s)}$ can be found. These transfer functions are shown in **equations 33, 34, and 35**.

$$\frac{\Delta\Psi(s)}{\Delta\delta(s)} = \frac{v_0}{L}\frac{1}{s} \tag{33}$$

$$\frac{\Delta Y(s)}{\Delta\Psi(s)} = \frac{v_0}{s} \tag{34}$$

$$\frac{\Delta Y(s)}{\Delta\delta(s)} = \frac{v_0^2}{s}\frac{1}{s^2} \tag{35}$$

From these linearized ODE's, known values can be plugged in as needed. These linearized ODE's can be compared to the non-linear dynamics given a step input. **Figures 18, 19, and 20**. From these plots, it is clear that the linearized model provides a strong approximation of the non-linear dynamics. The lateral linearization exhibits much higher errors than the heading linearization, but the error overall is still very small, only reaching an error of -0.016 after a simulation of 2 seconds.

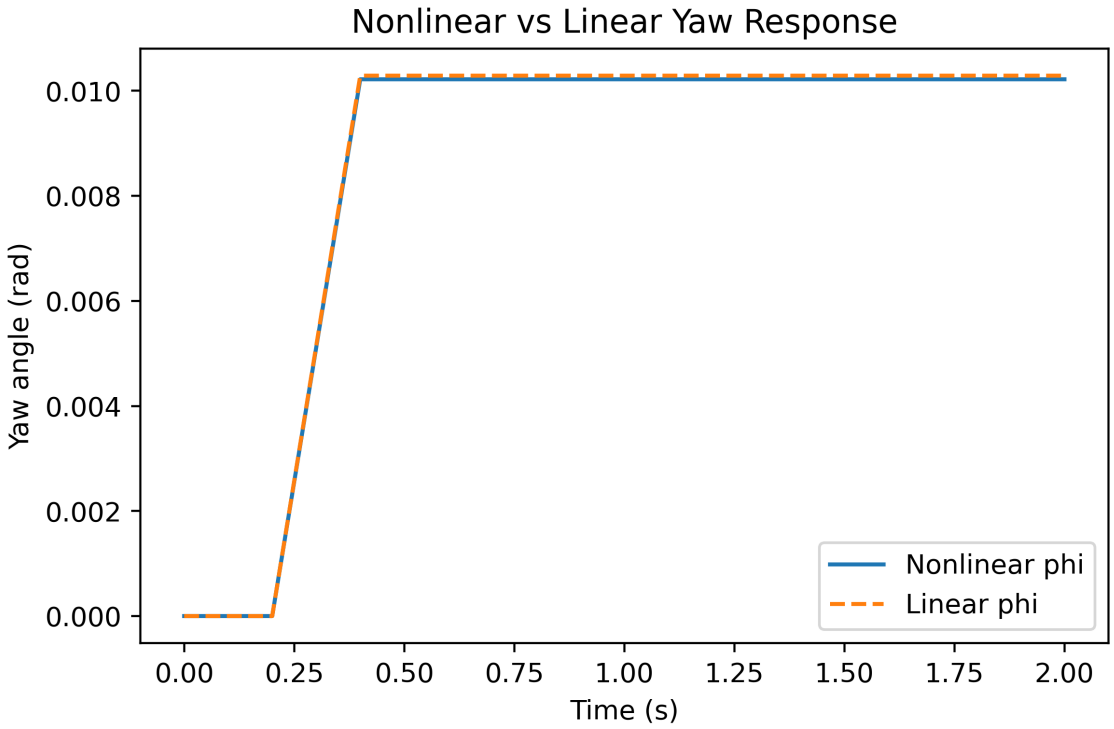

Figure 18: Task 3 - Yaw Linearization vs Non-Linear Dynamics

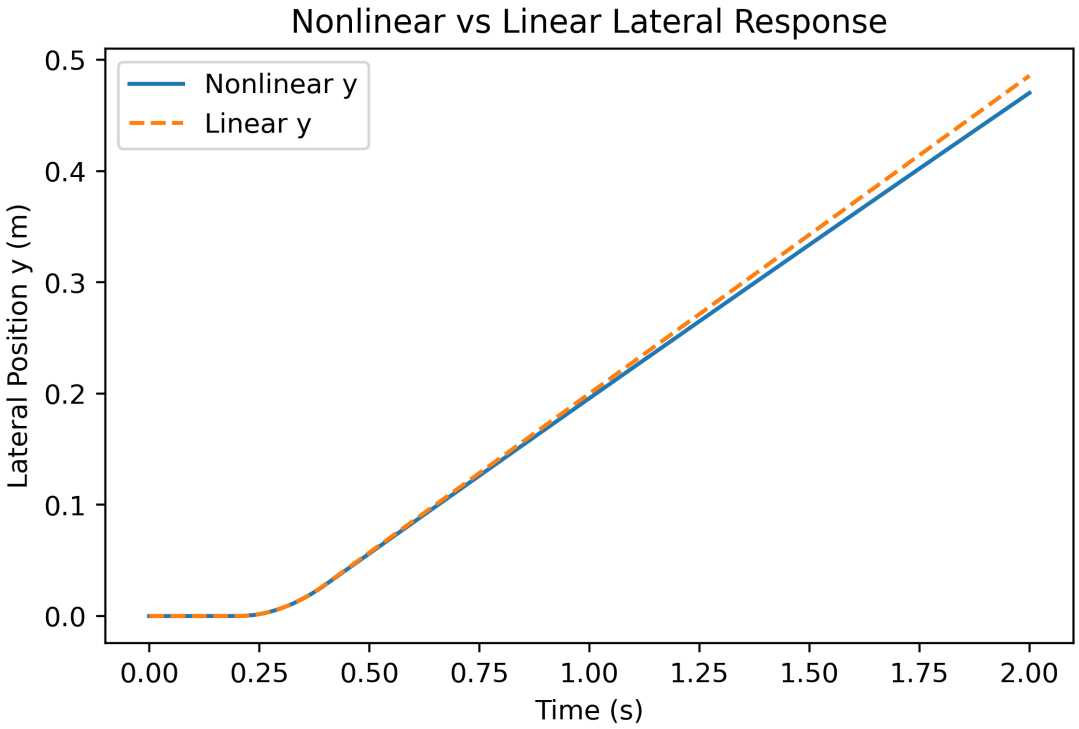

Figure 19: Task 3 - Position Linearization vs Non-Linear Dynamics

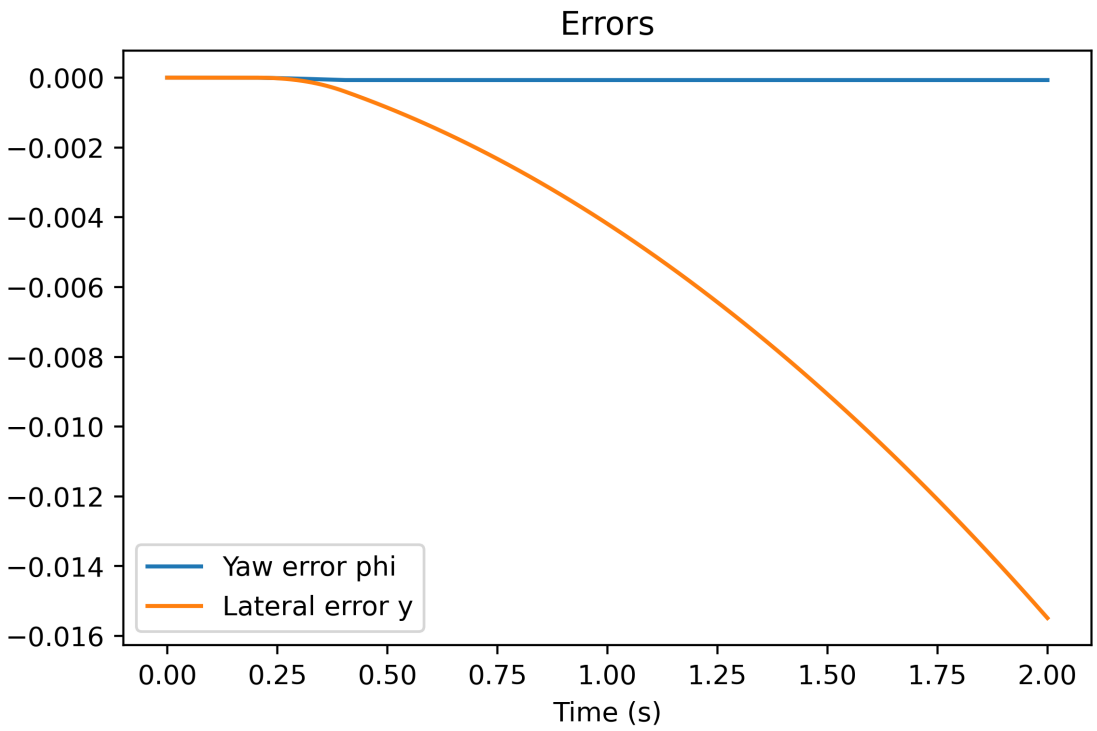

Figure 20: Task 3 - Linearization Errors

3.3 Designing Controller

The first step was to interpret the requirements. The no overshoot requirement is rather straightforward, meaning that a pre-compensator is likely required. The settling time translated to all poles being further than $\frac{4.6}{4} = 1.15 = \sigma$. Finally to fulfill the perfect tracking requirement, an integrator is required in one of the controllers.

The next step was to design the controller architecture. The architecture for the controller is shown in **figure 21**.

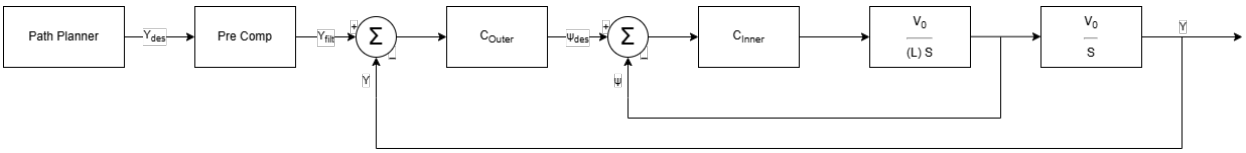

Figure 21: Task 3 controller architecture

The next step was to decide what the controller architecture would be. For simplicity it was decided that the C_{inner} would be a P-type. This was an easy choice because there is already an integrator in the corresponding plant. The result is that the inner loop has perfect steady-state tracking (assuming there are no disturbances). Furthermore, the gain

of C_{inner} directly determines the settling time. A gain of 1 was selected for this controller, simplifying later analysis.

Next, C_{outer} was selected to be a PI-type controller. Without disturbances, a P-type controller would have sufficed. However, because of the steady state tracking despite disturbances requirement, an integrator is required. Shown below is the process used to find the gains. It was assumed that the inner loop would settle with a DC gain of 1.

$$C_{outer} = k_p + \frac{k_i}{s}$$

$$\text{OLTF}_{outer} = P_{outer}C_{outer} = (k_p + \frac{k_i}{s})\frac{v_0}{s} = \frac{v_0 k_p s + k_i v_0}{s^2}$$

$$\text{CLTF}_{outer} = \frac{P_{outer}C_{outer}}{1 + P_{outer}C_{outer}} = \frac{\frac{v_0 k_p s + k_i v_0}{s^2}}{1 + \frac{v_0 k_p s + k_i v_0}{s^2}}$$

$$= \frac{(v_0 k_p)s + (k_i v_0)}{s^2 + (v_0 k_p)s + (k_i v_0)}$$

Decide: poles $= (P_l, -P_l)$ where P_l is the pole location of the resulting CLTF:

$$P_l = \frac{-v_0 k_p \pm \sqrt{(v_0 k_p)^2 - 4(1)(k_i v_0)}}{2}$$

Because the roots are repeated the discriminator must be zero. This yields the following system of equations:

$$2P_l = v_0 k_p$$

$$0 = \sqrt{(v_0 k_p)^2 - 4(1)(k_i v_0)}$$

This simplifies down to the equation for the pole locations:

$$k_p = \frac{2P_l}{v_0}$$

$$k_i = \frac{v_0 k_p^2}{4}$$

Solving this for $P_l = 2$, an arbitrary "fast" location yields:

$$k_p = k_i = 0.144$$

Plugging back into C_{outer} and simplifying following is obtained:

$$\frac{0.144(s+1)}{s}$$

Finally, a compensator can be designed with a pole at -1. Additionally because this controller has perfect steady state tracking despite a disturbance, the DC gain is 1. It is important to note that this compensator does not count toward the $\sigma < 1.15$ requirement.

This is because it cancels out with the controller's zero at the same location, thus nullifying the effect.

After implementing and testing the controller, it was found that it behaved somewhat strangely and oscillated. It was theorized that this was due to the inner loop, which controls δ. This is because of the tight limits on δ which resulted in extensive saturation. To solve this, the gain of the inner loop was halved. This resulted in much smaller inputs, and less oscillation.

The final, full controller is shown in **figure 22**, and the implemented version with saturation is shown in **figure 23**.

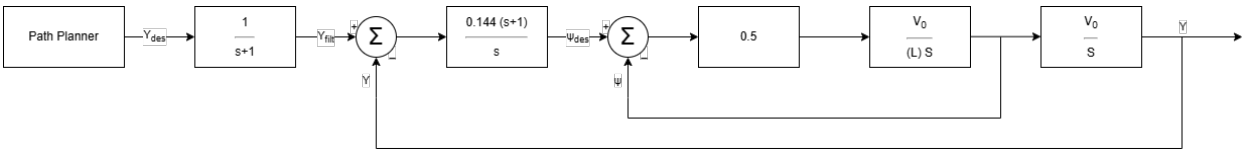

Figure 22: Task 3 controller design

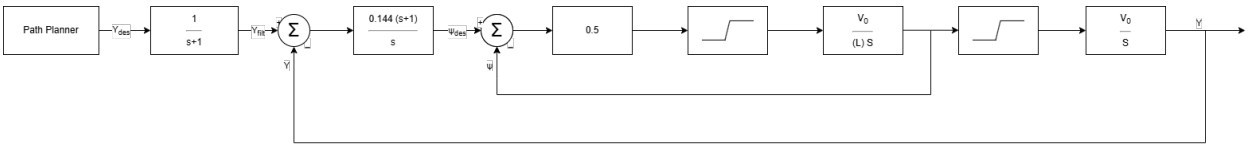

Figure 23: Task 3 controller design with saturation

Controller Validation

Using MATLAB the linear controller was tested. The step response of the full system, and the gain and phase plots are shown in **figures 24 and 25**.

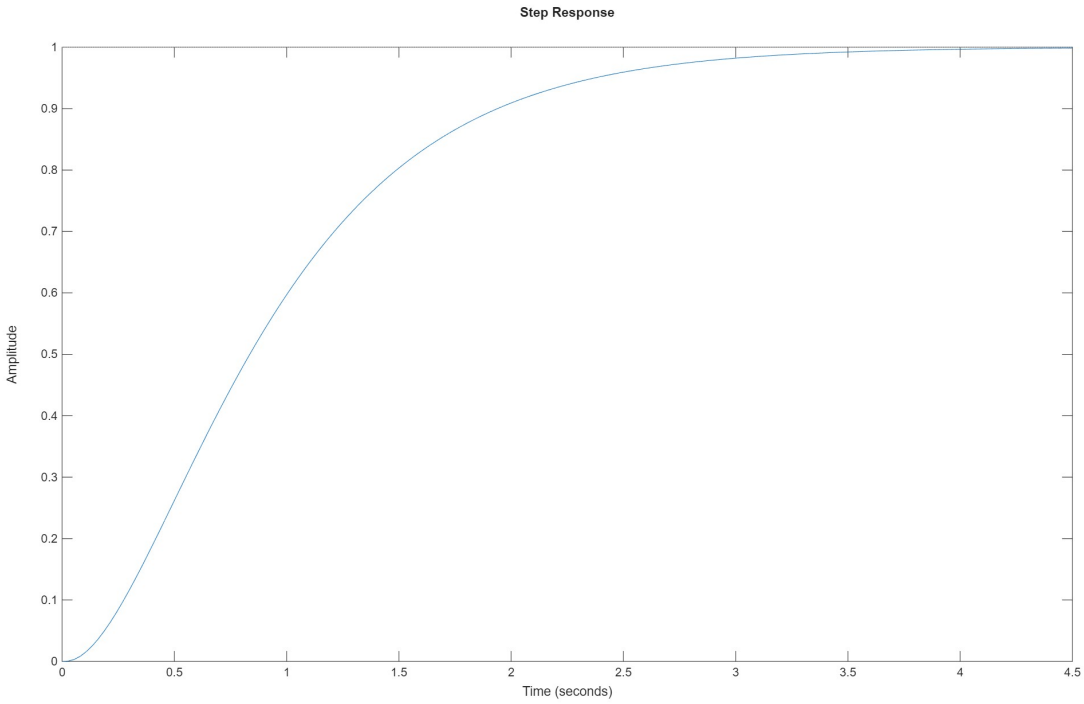

Figure 24: Task 3 controller step response

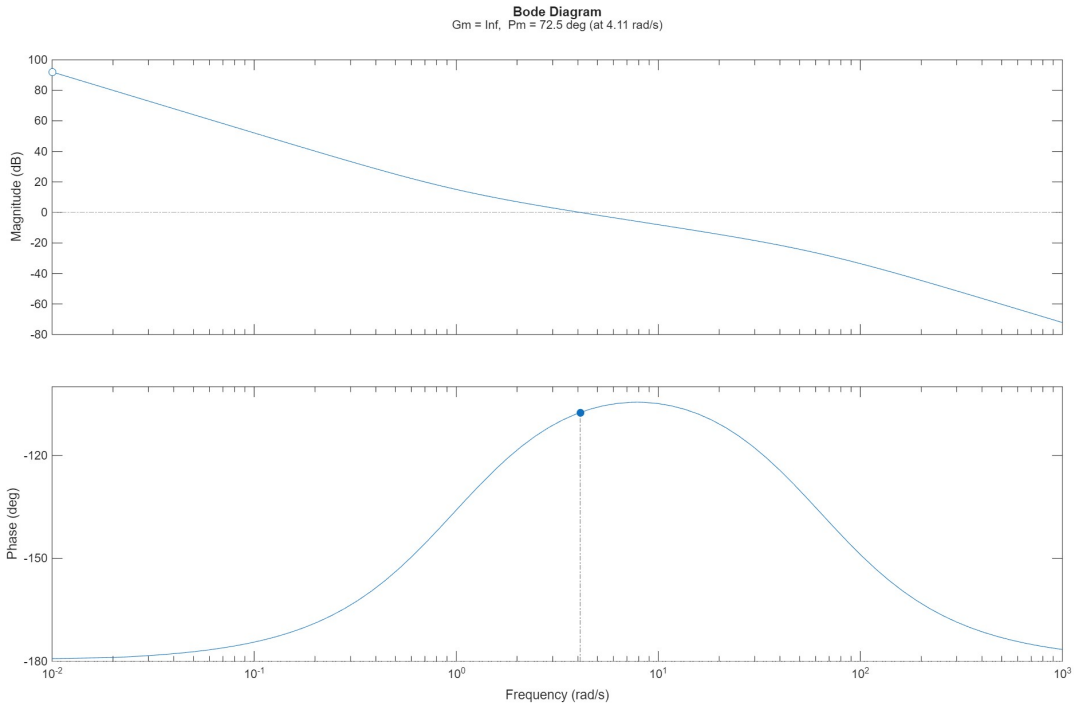

Figure 25: Task 3 controller gain and phase margins

MATLAB provides the step response information as well as gain and phase margins. The controller performs well within the limits. Some key results are:

- No overshoot (MATLAB)
- A rise time of 2.37 seconds (measured from 0% to 95%) as well as a 99% settling time of 3.35 seconds (MATLAB)
- Perfect steady state tracking despite step disturbance due to integrator (by analysis)
- Infinity gain margin
- 72.5° phase margin

In order to provide a more rigorous test, the controller was simulated with the non-linear car dynamics. Two tests were performed. The first, shown in **figure 26**, shows a step response of the non-linear controller. It was conducted with a step size of 0.1 to minimize the non-linear saturation of the controller. This step works well for testing the controller, but is not a realistic senario. Thus, the second test, shown in **figure 27** repeats the test with a step size of 20.

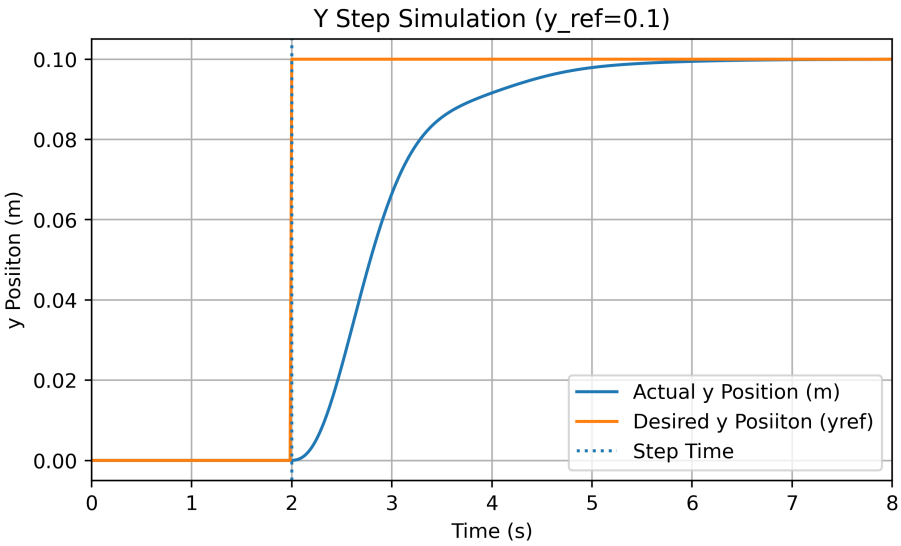

Figure 26: Task 3 small controller step response

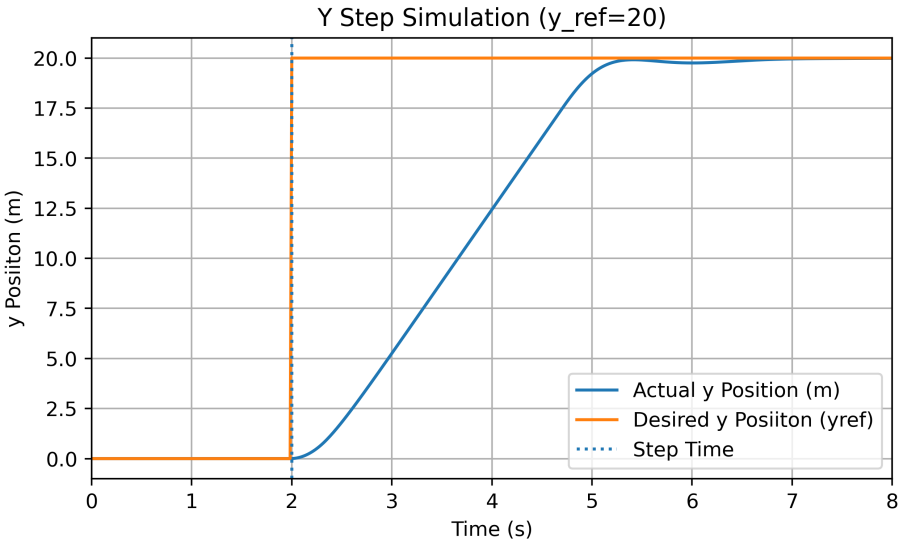

Figure 27: Task 3 large controller step response

It is clear from these simulations that the controller still meets the requirements, even when used in a non-linear system.

3.4 The Complete Block Diagram

The completed block diagram is shown in **figure 28**.

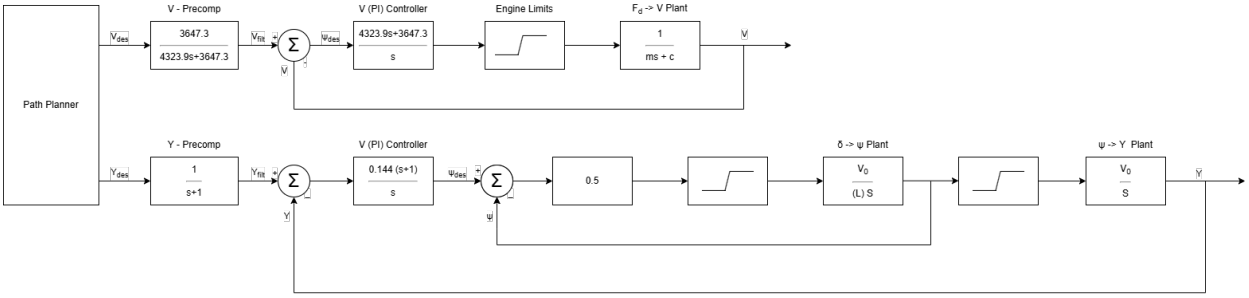

Figure 28: Full block diagram

4 Task 4

4.1 Fuel Minimization and Path Planning Methodology

The final part of this project is to use optimization and path planning to have the car autonomously traverse simulated traffic and optimize fuel. To do this, the task was split up into two parts detailed in this section:

- Fuel Minimization Algorithm
- Path Planning and Safety System

4.2 Fuel Minimization

The first portion of this task is to optimize fuel usage. Before, the speed controller would always try to get v_{driver} to match v_{des}. Now, with fuel optimization being implemented, the controller will instead track v_{des} with slack, allowing for the speed to be adjusted to minimize fuel consumption. The specifications for this fuel optimization are as follows:

1. The self-driving controller will treat speed as a slack for fuel optimization. The desired speed may be chosen within the range $[\text{sat}_{20.83}^{27.78}(v_{driver}-3), \text{sat}_{20.83}^{27.78}(v_{driver}+3)]$ such that the total fuel consumed is minimized over a prediction horizon.
2. If there are no vehicles ahead, the controller computes v_{des} to minimize fuel.
3. If there is a vehicle ahead, the ego vehicle must reduce speed to maintain a safe following distance or initiate a lane change if safe to do so.
4. The ego vehicle should at no point, under no circumstances, be within 7 meters of a vehicle ahead of it.
5. The autonomous driving should avoid unnecessary or aggressive accelerations or lane changes if possible.

To accomplish this task, a model predictive controller (MPC) is used. This code can be found in **appendix section 6.1.8**, specifcally from the "BEGIN MPC" comment to the "END MPC" comment. This MPC uses optimization to override the desired speed under safe conditions to be optimized for minimal fuel usage. The optimization used in MPC also cannot singificantly slow down computation efficiency, as the simulation must remain at 60 FPS or above. Thus, a coarse discretization is used for the MPC, and the linearized models provided earlier in this report are utilized to reduce computational complexity.

The ego vehicle's longitudinal motion was modeled using a first-order approximation, as shown in **equations 36 and 37**, where $T_s = 1$ second was chosen for a coarse MPC time step, $\tau_v = 1$ second approximates the closed-loop response of the speed controller, and v_{des} is contant over the prediction horizon. A prediction horizon of 10 was chosen to balance the computational complexity and model predictive accuracy.

$$v_{k+1} = v_k + \frac{T_s}{\tau_v}(v_{des} - v_k) \tag{36}$$

$$x_{k+1} = x_k + T_s v_{k+1} \tag{37}$$

The prediction of the future positions of other vehicles is also calculated. The model in **equation 38** is a heavily simplified, but computationally fast model to predict the future positions of the other cars. It uses the position and speed of the other vehicles relative to the ego vehicle.

$$x_{other,k} = (x_0 + \text{rel\_x}) + (v_0 + \text{rel\_speed})kT_s \tag{38}$$

Next, to model fuel consumption, the physical resistance model from **equation 11**, the BSFC model from **equation 4**, and the minimum engine fuel of $\underline{F} = 200$ mg/s are used. With these models in place, the fuel cost can be modeled by **equation 39**.

$$\text{fuel rate} = \frac{1}{\zeta} * \text{BSFC} * F_d * v \tag{39}$$

This model results in a bias toward lower speeds to improve fuel efficiency. The optimization uses a search step of 0.1 m/s, which is sufficiently small for the purposes of this application. The fuel cost of each speed candidate is evaluated, and then an optimization is run to return the desired speed to achieve the lowest fuel cost.

The MPC is implemented in a way that always prioritizes safety. The permissible minimum distance between the ego vehicle and another vehicle is 7 meters, but a value of 10 meters was used for additional safety and to account for any gap decrease that happens during a lane change maneuver. If the predicted gap between the ego vehicle and another vehicle is less than the permissible minimum distance of 10 meters, the cost function evaluation simply returns a cost of 10^9, effectively making that speed candidate invalid.

With the horizon distance of 10 chosen, and the one-second time constants, the MPC is effectively looking ahead 10 seconds. This gives more than enough time to make lane and speed change maneuvers, and naturally penalizes aggressive controller behavior. This leads to a lack of unnecessary accelerations or lane changes.

This design successful achieves fuel optimization while maintaining safety for the ego vehicle and other vehicles. When the simulations are run using this MPC controller, it is clear that the fuel optimization is functional. To test this, two simulations were run, both with the same set seed (tested on SEED = 59). The first simulation does not implement the MPC fuel optimization and only includes autonomous lane control, which always has the driver speed track the desired speed. This simulation, when run for 90 seconds, had a total fuel usage of 179 grams. For the second simulation, the MPC is implemented. This simulation, when also run for 90 seconds on the same seed, had a total fuel usage of 146 grams. This is a significant improvement and clearly demonstrates the effectiveness of the MPC. Additionally, both with and without the MPC active, the ego vehicle completed several successful lane change maneuvers with no errors in gap distance calculations.

4.3 Path Planning and Safety System

While the fuel minimization algorithm determines the speed to travel at, the path planning algorithm must take that into account. The logic flow follows three rules. These are general procedures followed by human drivers whenever possible.

1. Exit the passing (left) lane if possible.
2. If the path is blocked and the other lane is clear, move to the other lane.
3. If the path is blocked and the other lane is not clear slow down to avoid hitting the car in front.

To do this, a finite state machine was used. The flow chart of this state machine is shown in **figure 29**. It has 3 main states:

- Drive Straight
- Move Left
- Move Right

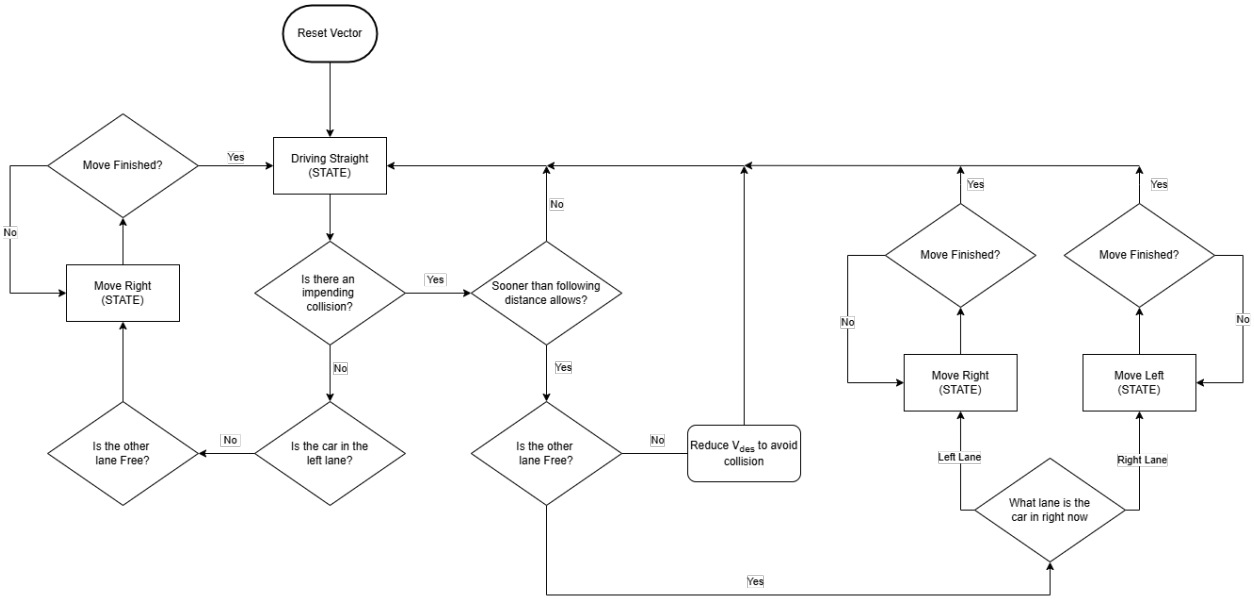

Figure 29: Task 3 large controller step response

This system controls the lateral position, but also impacts the desired speed sent to the controller. In order to not invalidate the fuel use minimizer, the controller makes sure to pass through the MPC optimized speed unless a collision needs to be avoided.

5 Conclusion

Overall, this project has been very insightful, and a good reflection of this course. This project used physics to model the dynamics of the vehicle, and Python code to simulate the dynamics on discrete time intervals. Linearization was used to approximate the nonlinear models to design linear controllers and their performance was validated.

Many challenges were overcome during the course of this project, primarily from the difficulty of implementing controllers and system dynamics into actual software. For example, in task 3, difficulties were encountered. with the nonlinear saturation functions.

If this project were to be completed again, there are very few things we would do differently. That being said, making more time for this project earlier on would have been helpful, as there was somewhat of a time crunch at the end. Additionally, schedule coordination was an issue, leaving the coordination of information for the project relatively poor.

6 Appendix

6.1 Code

6.1.1 Car.py

```
import numpy as np

class Car:
    #consts
    m = 1300.0
    F_roll = 100.0 # N
    a = 0.2 # Ns^2/m2
    b = 20.0 # Ns/m
    g = 9.8 # m/s^2
    L = 2.7 # m
    zeta = 0.95
    eta_g = 0.8
    eta_d = 3.8
    rw = 0.34 # m
    Te_max = 200 # Nm
    #end consts

    F_max_force = (Te_max * eta_g * eta_d) / rw * zeta
    F_bar = 200.0

    AMP = 0.0
    road_x = np.arange(0, 6101, 1)
    road_beta_deg = AMP * np.sin(2*np.pi/1000*road_x + 300)
    road_beta = np.deg2rad(road_beta_deg)


    def __init__(self, Ts, initial_speed = 27.78, mass=1300,
   initial_position=0.0):
        self.Ts = Ts

        # Car states
        self.speed = initial_speed #m/s
        self.x = initial_position #longitudinal position (m)
        self.phi = 0.0 #yaw angle (rad)
        self.total_fuel = 0.0 #total fuel consumed (mg)
        self.total_distance_m = 0.0 #total distance traveled (m)\
        self.fuel_rate = 0

        self.fd_min = -7000.0 #min force (N)
        self.delta_max = 0.05 #steering limit (rad)
        self.vx = 0
        self.vy = 0

        self.phi_dot = 0
        self.y = 0
        # self.x = 0
```

```
 def update(self, Fd, delta, beta=0):
     Ts = self.Ts

     #saturations
     F_sat = np.clip(Fd, self.fd_min, self.F_max_force)
     delta_sat = np.clip(delta, -self.delta_max, self.delta_max)

     #calculate forces
     F_air = self.a * self.speed**2 + self.b * self.speed
     F_gravity = self.m * self.g * np.sin(beta)
     F_t = F_sat - F_air - self.F_roll - F_gravity


     #longitudinal dynamics
     accel = F_t / self.m
     self.speed += accel * Ts
     self.speed = np.maximum(0.0, self.speed) #non-negative speed


     #longitudinal dynamics
     self.phi += Ts * self.speed/self.L * np.tan(delta_sat)


     self.vx = self.speed * np.cos(self.phi) # speed in direction of
road
     self.vy = self.speed * np.sin(self.phi) # Speed left and right


     self.y += Ts * self.vy

     #position and distance updates
     if self.speed > 0:
         distance_travelled = self.speed * Ts
         self.x += distance_travelled
         self.total_distance_m += distance_travelled

     #J_ss calculations
     N_e = (60/(2*np.pi)) * (self.eta_g * self.eta_d / self.rw) * self.
speed
     T_e = (1/self.zeta) * (self.rw / (self.eta_g * self.eta_d)) *
F_sat
     BSFC = ((N_e - 2700) / 12000)**2 + ((T_e - 150) / 600)**2 + 0.07

     self.fuel_rate = np.maximum((1/self.zeta) * BSFC * F_sat * self.
speed, self.F_bar) # was fuel_rate_mg_per_s

     #update total fuel used
     self.total_fuel += self.fuel_rate * Ts

     #return values
     return self.speed, self.x, self.total_fuel
```

6.1.2 controller Task2 Task3.py

```
from PID import PID
from car import Car
import numpy as np
from FuelOptimizerMPC_controller_JP import mpc_select_v_des

import enum

class States(enum.Enum):
    STRAIGHT = enum.auto()
    MOVING_LEFT = enum.auto()
    MOVING_RIGHT = enum.auto()


class Other_Car:
    def __init__(self, rel_x, rel_speed):
        self.rel_x = rel_x
        self.rel_speed = rel_speed


# Task 2 controllers

class Precompensator:
    def __init__(self, Kp, Ki, Ts, initial_setpoint_ref):
        self.KP = Kp
        self.Ki = Ki
        self.Ts = Ts

        denom = Kp + Ki * Ts
        self.alpha = (Ki * Ts) / denom
        self.beta = Kp / denom
        self.Vf_prev = initial_setpoint_ref

    def filter(self, Setpoint_ref_current):
        Vf_current = self.alpha * Setpoint_ref_current + self.beta * self.
   Vf_prev
        self.Vf_prev = Vf_current
        return Vf_current

class V_controller:

    def __init__(self, Kp=4323.888, Ki=3647.3125, Ts=1/60, umax=10000,
   umin=-10000, Kaw=1):
        #Kaw = 1.0/Ts #standard value

        v0 = 27.78
        a = 0.2
        b = 20
        F_roll = 100
        F_drag = F_roll + a * v0**2 + b * v0
        #insantiate controller
        self.PI = PID(Kp=Kp, Ki=Ki, Ts=Ts, umax=umax, umin=umin, Kaw=Kaw,
```

```
   initialState=F_drag)
        self.precomp = Precompensator(Kp=Kp, Ki=Ki, Ts=Ts,
   initial_setpoint_ref=v0)
        self.last_filtered_ref = 0.0

    def update(self, desired_speed, speed):
        V_ref_filtered = self.precomp.filter(desired_speed)

        force = self.PI.update(V_ref_filtered, speed)

        self.last_filtered_ref = V_ref_filtered
        return force


# Task 3 Controllers
#TODO never used??
class Precompensator_t3:
    def __init__(self, Ts = 1/60, alpha = 1.5):
        self.y_f = 0
        self.alpha = alpha
        self.Ts = Ts

    def filter(self, y):

        self.y_f += self.Ts * self.alpha*(y-self.y_f)

        return self.y_f


class Y_controller:
    def __init__(self, Ts=1/60,
                 Kp_inner=0.5, delta_max=0.05, delta_min=-0.05,
                 phi_max=np.deg2rad(15), phi_min=-np.deg2rad(15), P_loc =
   2):

        # Caluclate controller/Precomp Values:
        v0 = 27.78

        Kp_outer = 2*P_loc/v0
        Ki_outer = v0*Kp_outer**2/4

        self.c_inner = PID(Kp=Kp_inner, Ts=Ts, umax=delta_max, umin=
   delta_min) # Detla Controller Psi des -> Delta
        self.c_outer = PID(Kp=Kp_outer, Ki=Ki_outer, Ts=Ts, umax=phi_max,
   umin=phi_min, Kaw=1.5) #TODO change from 0, initialState=0.0) #Y
   controller Y-> psi_des

        self.precomp = Precompensator_t3(alpha=1) #Precompensator(Kp=
   Kp_outer, Ki=Ki_outer, Ts=Ts, initial_setpoint_ref=0) # Precomp




    def update(self, y_des, y, phi):
```

```
        y_filt = self.precomp.filter(y_des)

        phi_des = self.c_outer.update(y_filt, y)

        delta = self.c_inner.update(phi_des, phi)

        return delta


class Controller:
    #lane geometry constants
    LANE_MIDPOINT_OFFSET = 11.25   #center of lane in meters

    def __init__(self, Ts, initial_conditions):
        self.Ts = Ts
        self.Fd_cmd = initial_conditions[0]
        self.speed = initial_conditions[1]

        # Force limits
        self.F_d_min = -7000.0
        c = Car(0, 0)
        self.F_d_max = c.F_max_force

        # Instantiate inner controllers
        self.v_controller = V_controller(
            Ts=Ts,
            umin=self.F_d_min,
            umax=self.F_d_max
        )

        self.y_controller = Y_controller()

        self.last_v_des_opt = 0.0

    def update(self, speed, x, y, phi, desired_speed, des_lane, other_cars
   , grade):
        v_des = desired_speed
        self.last_v_des_opt = v_des
        desired_y = des_lane * self.LANE_MIDPOINT_OFFSET
        delta_cmd = self.y_controller.update(desired_y, y, phi)
        Fd_cmd = self.v_controller.update(v_des, speed)

        return Fd_cmd, delta_cmd
```

6.1.3 Task1.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import os

#Consts
m = 1300
Froll = 100 #N
a = 0.2 #Ns^2/m2
b = 20 #Ns/m
g = 9.8 #m/s^2
fd_min = -7000 #N
zeta = 0.95
eta_g = 0.8
eta_d = 3.8
rw = 0.34 #n
F_bar =  200 #ng/d
Te_max = 200 #Nm
F_max = (Te_max * eta_g * eta_d) / rw * zeta #mg/s from engine
F_min = -7000 #mg/s from brakes
L = 2.7 #m
delta_max = 0.05 #rad
step_size = 300 #steps per second
sim_length = 150 # s

v0 = 27.78
#End Consts

#save figures
SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
FIGS_PATH = os.path.join(SCRIPT_DIR, "figs")

Amp = 3

x = np.arange(0, 6101, 1)                       # 6.1 km total
beta = Amp * np.sin(2*np.pi/1000*x + 300)      # unit is degrees here!
beta[(x < 500) & (beta < 0)] = 0    # initial 500m is flat

#plotting variables
N_e = np.linspace(1000, 5500, 300)
T_e = np.linspace(0, 225, 300)
N, T = np.meshgrid(N_e, T_e)

#piecewise T_max function
def tmax(N):
    return np.where(N < 3250, 0.1 * N + 50, -0.1 * N + 700)

#given functions
T_max = tmax(N)
T_max = np.clip(T_max, 0, 200)
BSFC = ((N - 2700)/12000)**2 + ((T - 150)/600)**2 + 0.07
BSFC_masked = np.ma.masked_where(T > T_max, BSFC)
```

```
fig_ratio = 1.61803398875
fig_height = 4
figsize = (fig_height * fig_ratio, fig_height)

#Road profile plot
plt.figure(figsize = figsize)
plt.plot(x, beta)
plt.title("Task 1: Road Profile (Amp=3) vs Horizontal Distance")
plt.xlabel("Distance (m)")
plt.ylabel("Height (m)")
fig_path = os.path.join(FIGS_PATH, "road_shape.png")
plt.savefig(fig_path, dpi = 400)
plt.grid()
if __name__ == "__main__":
    plt.show()


#2D Contour Plot
plt.figure(figsize=figsize)
cont = plt.contour(N, T, BSFC_masked, levels=40, cmap='viridis')
plt.plot(N_e, T_max[0, :], 'r', linewidth=2, label='Torque Limit')
plt.clabel(cont, inline=True, fontsize=8)
plt.xlabel("Speed (rpm)")
plt.ylabel("Torque (Nm)")
plt.title("BSFC Contour Map")
plt.colorbar(cont, label="BSFC")
fig_path = os.path.join(FIGS_PATH, "BSFC_contour.png")
plt.savefig(fig_path, dpi = 400)
plt.grid()
if __name__ == "__main__":
    plt.show()

#3D Surface Plot
fig = plt.figure(figsize=figsize)
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(T, N, BSFC_masked, cmap='viridis', edgecolor='none'
   )
ax.view_init(elev = 30, azim = 30)
ax.set_xlabel("Torque (Nm)")
ax.set_ylabel("Engine Speed")
ax.set_zlabel("BSFC")
ax.set_title("BSFC Surface Plot")
fig.colorbar(surf, ax=ax, shrink=0.6)
fig_path = os.path.join(FIGS_PATH, "BSFC_3d.png")
plt.savefig(fig_path, dpi = 400)
if __name__ == "__main__":
    plt.show()

#Fdss derivation
def F_d_ss(v, beta):
    return m*g*np.sin(beta) + Froll + a*v**2 + b*v

print(f"F_d_ss = {F_d_ss(v0, 0)} N")
```

```
#Jss derivation
def J_ss(v, beta):
    F_d = F_d_ss(v, beta)
    N = (60/(2*np.pi)) * (eta_g * eta_d / rw) * v
    T = (1/zeta) * (rw / (eta_g * eta_d)) * F_d
    BSFC = ((N - 2700)/12000)**2 + ((T - 150)/600)**2 + 0.07
    return np.maximum((1/zeta)*(BSFC) * F_d * v, F_max)

#Jss surface plot
v = np.linspace(23, 28, 1000)
beta = np.linspace(np.deg2rad(-2.5), np.deg2rad(2.5), 1000)

V, B = np.meshgrid(v, beta)
Jss_vals = J_ss(V, B)

fig = plt.figure(figsize=figsize)
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(V, B, Jss_vals, cmap='viridis', edgecolor='none')
ax.view_init(elev = 30, azim = 30)
ax.set_xlabel("v")
ax.set_ylabel("beta")
ax.set_zlabel("J_ss")
ax.set_title("J_ss Surface Plot")
fig.colorbar(surf, ax=ax, shrink=0.6)
fig_path = os.path.join(FIGS_PATH, "Jss_3d.png")
plt.savefig(fig_path, dpi = 400)
plt.show()


#============Validation of model===============
from car import Car
mpg_conversion = 1761.59 #calculated by hand

def simulate(car_obj, Fd_func, duration=150, grade = None, x = None):
    #time vector
    t_vec = np.arange(0, duration, car_obj.Ts)

    #logging lists
    v_log = []
    mpg_log = []
    t_log = []
    x_pos = 0

    for t in t_vec:
        Fd_input = Fd_func(t)
        grade_at_x = 0
        # Interp for grade at x
        if(x is not None):
            grade_at_x = np.interp(x_pos, x, grade)

        speed, x_pos, fuel = car_obj.update(Fd_input, 0, grade_at_x)
```

```
        if len(mpg_log) > 0: #skip if on first test
            pass

        F_sat_log = np.clip(Fd_input, car_obj.fd_min, car_obj.F_max_force)

        Ne_log = (60/(2*np.pi)) * (car_obj.eta_g * car_obj.eta_d / car_obj
   .rw) * speed
        Te_log = (1/car_obj.zeta) * (car_obj.rw / (car_obj.eta_g * car_obj
   .eta_d)) * F_sat_log
        BSFC_log = ((Ne_log - 2700) / 12000)**2 + ((Te_log - 150) / 600)
   **2 + 0.07
        fuel_rate_log = np.maximum((1/car_obj.zeta) * BSFC_log * F_sat_log
    * speed, car_obj.F_bar)

        if fuel_rate_log > 0 and speed > 0.1:
            inst_mpg = (speed / fuel_rate_log) * mpg_conversion
        else:
            inst_mpg = 0

        v_log.append(speed)
        mpg_log.append(inst_mpg)
        t_log.append(t)
    return t_log, v_log, mpg_log

#====Sim 1, flat road=====
car_1 = Car(Ts=0.1/step_size, initial_speed=27.78)

flat_beta = np.zeros_like(x)
car_1.road_beta = np.deg2rad(flat_beta)

F_eq_val = F_d_ss(27.78, 0.0)

def step_force(t):
    if t < 50:
        return F_eq_val
    else:
        return 3000.0

t1, v1, mpg1 = simulate(car_1, step_force)

#plot
plt.figure(figsize=figsize)
plt.subplot(2, 1, 1)
plt.plot(t1, v1, 'b', linewidth=2, label='v')
plt.ylabel('Velocity (m/s)')
plt.title('Simulation 1: Step Input on Flat Road')
plt.grid()
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(t1, mpg1, 'b', linewidth=2, label='mpg')
plt.ylabel('MPG')
plt.xlabel('Time (s)')
plt.grid()
```

```
plt.legend()

fig_path = os.path.join(FIGS_PATH, "Validation_Sim_1_Flat_Road.png")
plt.savefig(fig_path, dpi = 400)

if __name__ == "__main__":
    plt.show()

#====Sim 2, Amp=3=====
car_2 = Car(Ts=1.0/step_size, initial_speed=27.78)

sim_x = np.arange(0, 6101, 1)
sim_beta_deg = Amp * np.sin(2*np.pi/1000*sim_x + 300)
sim_beta_deg[(sim_x < 500) & (sim_beta_deg < 0)] = 0 #flatten
sim_beta_rad = np.deg2rad(sim_beta_deg)


def const_force(t):
    return F_eq_val

t2, v2, mpg2 = simulate(car_2, const_force, grade=sim_beta_rad, x = sim_x)

#plot
plt.figure(figsize=figsize)
plt.subplot(2, 1, 1)
plt.plot(t2, v2, 'b', linewidth=2, label='v')
plt.ylabel('Velocity (m/s)')
plt.title('Simulation 2: Const Force Input on Hilly Road')
plt.grid()
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(t2, mpg2, 'b', linewidth=2, label='mpg')
plt.ylabel('MPG')
plt.xlabel('Time (s)')
plt.grid()
plt.legend()

fig_path = os.path.join(FIGS_PATH, "Validation_Sim_2_Hilly_Road.png")
plt.savefig(fig_path, dpi = 400)
if __name__ == "__main__":
    plt.show()
```

6.1.4 Task2.py

```
import numpy as np
import matplotlib.pyplot as plt
import os
from car import Car
import control as ct

#from task 1
SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
FIGS_PATH = os.path.join(SCRIPT_DIR, "figs")
fig_ratio = 1.61803398875
fig_height = 4
figsize = (fig_height * fig_ratio, fig_height)


Tp = 1/300
Ts = 1/60
m = 1300 #kg
Froll = 100 #N
a = 0.2 #Ns^2/m2
b = 20 #Ns/m
g = 9.8 #m/s^2
fd_min = -7000 #N
zeta = 0.95
eta_g = 0.8
eta_d = 3.8
rw = 0.34 #m
F_bar =  200 # m   /s
Te_max = 200 #Nm
F_max = (Te_max * eta_g * eta_d) / rw * zeta #mg/s from engine
F_min = -7000 #mg/s from brakes
L = 2.7 #m
delta_max = 0.05 #rad


v0_100 = 27.78 # KM/H
v0_150 = 150 * v0_100/100 #KM/H


def F_d_ss(v, beta):
    return m*g*np.sin(beta) + Froll + a*v**2 + b*v

def plot_lin_analisis(v0_sim = 27.78, v0_lin = None, step_size = 1,
   sim_length = 150, step_time = 50):

    # Use v0_sim as the default linearization point if not specified
    if v0_lin is None:
        v0_lin = v0_sim

    t = np.arange(0, sim_length + Ts, Ts)

    # Define plant - linearized about v0_lin
    c = 2*a*v0_lin + b
```

```
    plant = ct.tf(1,(m,c))

    # Run Linear Sim
    _, step_res_linaprox = ct.step_response(plant, T = t[int(round(
   step_time/Ts)):])
    step_res_linaprox = np.concatenate((np.zeros((int(round(step_time/Ts))
   )), step_res_linaprox))

    # Non lim sim
    eq_input = F_d_ss(v0_lin, 0)
    print(f"EQ input (Linearized at v0_lin={round(v0_lin,2)}): {eq_input}N
   ")


    # Create input sequence
    F_d_in = t * 0 + eq_input
    F_d_in[int(round(step_time/Ts)):] = eq_input + step_size

    vel_nonlin_sim_out = np.zeros(len(t))
    nonlin_speed = v0_sim # NONLINEAR SIMULATION STARTS AT v0_sim

    for i,_ in enumerate(t):
        F_c = 0 # no grade
        F_sat = F_d_in[i]

        #Non lin sim
        F_air_nonlin = a*nonlin_speed**2 + b*nonlin_speed
        # Sum of forces
        F_t_nonlin = F_sat - F_air_nonlin - F_c - Froll
        # Longitudinal dynamics (Euler)
        nonlin_speed += Ts * F_t_nonlin/m
        vel_nonlin_sim_out[i] = nonlin_speed


    plt.figure(figsize=figsize)
    plt.plot(t, vel_nonlin_sim_out, label="Nonlinear")
    plt.plot(t, step_res_linaprox * step_size + v0_sim, "--", label="
   Linear")

    plt.legend()
    plt.title(f"Step Responses: Step Size = {step_size} (N), $v_{{0, \text
   {{sim}}}} = {round(v0_sim,2)}$ (m/s), $v_{{0, \text{{lin}}}} = {round(
   v0_lin,2)}$ (m/s)")
    plt.xlabel("Time (s)")
    plt.ylabel("Speed (m/s)")

    fig_path = os.path.join(FIGS_PATH, f"t2_Linsim_step{step_size}_v0{
   round(v0_lin)}.png")
    plt.savefig(fig_path, dpi = 400)

    if __name__ == "__main__":
        plt.show()

plot_lin_analisis(v0_sim = v0_100)
```

```
plot_lin_analisis(v0_sim = v0_100, step_size=600)
plot_lin_analisis(v0_sim=v0_100, v0_lin=v0_150)
plot_lin_analisis(v0_sim=v0_100, v0_lin=v0_150, step_size=600)

#==========Controller Simulations============
from controller import V_controller as controller

print(f'\n==========Beginning Simulations=========')

SIM_TIME = 150.0
STEP_TIME = 50.0

V_STEP = 101.0 / 3.6 #m/s

def simulate_step_response(Kp, Ki, Kaw, v_start, v_target, plot_title,
   filename, road_amp=0.0):

    # Update car class road profile manually
    Car.AMP = road_amp
    Car.road_beta_deg = Car.AMP * np.sin(2*np.pi/1000*Car.road_x + 300)
    Car.road_beta = np.deg2rad(Car.road_beta_deg)

    #defaults are set appropriately for designed controller of this
   specific task
    cruise_controller = controller(Kp=Kp, Ki=Ki, Ts=Ts, umax=F_max, umin=
   F_min, Kaw=Kaw)

    # Instantiate car object
    car_task2 = Car(Ts=Ts, initial_speed=v_start)

    time_data = np.arange(0, SIM_TIME, Ts)
    speed_data = np.zeros_like(time_data)
    ref_data = np.zeros_like(time_data)
    force_data = np.zeros_like(time_data)

    current_speed = v_start
    x_pos = 0

    for i, t in enumerate(time_data):
        desired_speed = v_start
        if t >= STEP_TIME:
            desired_speed = v_target
        ref_data[i] = desired_speed

        control_force = cruise_controller.update(desired_speed,
   current_speed)
        force_data[i] = control_force

        # Using car class update
        grade_at_x = np.interp(x_pos, Car.road_x, Car.road_beta)
        current_speed, x_pos, total_fuel = car_task2.update(control_force,
    0, grade_at_x)
        speed_data[i] = current_speed
```

```
    #convert back to km/hr
    speed_data_kph = speed_data * 3.6
    ref_data_kph = ref_data * 3.6

    #plotting
    if filename:

        #Speed vs Time
        plt.figure(figsize=figsize)
        plt.plot(time_data, speed_data_kph, label='Actual Speed (v)')
        plt.plot(time_data, ref_data_kph, label='Desired Speed (vref)')
        plt.axvline(STEP_TIME, linestyle=':', label='Step Time')
        plt.title(plot_title)
        plt.xlabel('Time (s)')
        plt.ylabel('Speed (km/hr)')
        plt.legend()
        plt.grid(True)

        plt.tight_layout()
        save_path = os.path.join(FIGS_PATH, filename)
        plt.savefig(save_path, dpi = 400)
        plt.show()

    return total_fuel


#defined gains
Kp = 4323.888
Ki = 3647.3125



if __name__ == "__main__":

    #run with no anti_windup, step of 1 km/hr
    simulate_step_response(Kp=Kp, Ki=Ki, Kaw=0.0, v_start=v0_100, v_target
   =101.0/3.6,
        plot_title='Nonlinear Plant Step Response (1 km/hr step, No Anti-
   Windup)',
        filename="Task2_Controller_Part3.png",
        road_amp=0
    )

    #rerun with new values
    simulate_step_response(Kp=Kp, Ki=Ki, Kaw=0.0, v_start=v0_100, v_target
   =150.0/3.6,
        plot_title='Nonlinear Plant Step Response (50 km.hr step, No Anti-
   Windup)',
        filename="Task2_Controller_Part4.png",
        road_amp=0
    )

    #rerun with anti_windup
    simulate_step_response(Kp=Kp, Ki=Ki, Kaw=1/Ts, v_start=v0_100,
```

```
    v_target=150.0/3.6,
         plot_title='Nonlinear Plant Step Response (50 km.hr step, w/ Anti-
    Windup)',
         filename="Task2_Controller_Part5.png",
         road_amp=0
     )

     #fuel consumption with road grade
     fuel_full = simulate_step_response(Kp=Kp, Ki=Ki, Kaw=1/Ts, v_start=
    v0_100, v_target=v0_100, # Constant speed
         plot_title="", filename=None, #no plot needed
         road_amp=3.0)
     print(f"Total Fuel (Full Gains): {fuel_full:.2f} mg")

     #half Gains
     fuel_half = simulate_step_response(Kp=Kp/2, Ki=Ki/2, Kaw=1/Ts, v_start
    =v0_100, v_target=v0_100,
         plot_title="", filename=None,
         road_amp=3.0)
     print(f"Total Fuel (Half Gains): {fuel_half:.2f} mg")
```

6.1.5 controller_task2_analysis.py

```
import numpy as np
import matplotlib.pyplot as plt
import control as ct

m = 1300.0
a = 0.2
b = 20.0
v0 = 27.78
c = 2 * a * v0 + b
print(f'Linearized damping coefficient = {c:.4f} Ns/m')

zeta = 1.0 #critically damped
omega_n = 1.675 #chosen

#gain calculations (coefficient matching)
Ki = m * (omega_n**2)
Kp = 2 * zeta * omega_n * m - c


print(f'\nDesigned Controller Gains:')
print(f'Kp: {Kp:.4f}')
print(f'Ki: {Ki:.4f}')

s = ct.TransferFunction.s #setup transfer functions
P = 1 / (m * s + c) #plant
C = (Kp * s + Ki) / s #controller (PI)
TF = C * P #transfer function

#bode plot
plt.figure()
ct.bode_plot(TF, dB=True, display_margins=True)
plt.suptitle(f'Bode Plot of Transfer Function')

F = Ki / (Kp * s + Ki) #pre-comp

T_feedback = ct.feedback(TF, 1)
T_final = F * T_feedback

#step Response
t, y = ct.step_response(T_final, T=np.linspace(0, 10, 1000))
y_final = y[-1]
y_max = np.max(y)

#overshoot calculation
overshoot = (y_max - y_final) / y_final * 100 #percentage
if overshoot < 0: overshoot = 0 #if no overshoot

#rise time calculation
t_10_id = np.where(y >= 0.1 * y_final)[0][0] #index where response crosses
    10%
t_90_id = np.where( y >= 0.9 * y_final)[0][0] #index where response
   crosses 90%
```

```
rt = t[t_90_id] - t[t_10_id]

#settling time calculation
upper = 1.02 * y_final #2% above
lower = 0.98 * y_final #2% below
outside = (y > upper) | (y < lower)
if np.any(outside):
    last_violation = np.where(outside)[0][-1]
    settling_time = t[last_violation + 1] if last_violation < len(t)-1
   else t[-1]
else:
    settling_time = 0.0

print(f'\nPerformance Metrics:')
print(f'Final Value: {y_final:.4f} | (Req: 1 for step input)')
print(f'Overshoot: {overshoot:.2f}% | (Req: 0%)')
print(f'Rise Time: {rt:.4f} s | (Req: 1-3 s)')
print(f'Settling Time: {settling_time:.4f} s | (Req: <10 s)')


plt.figure()
plt.plot(t, y)
plt.axhline(1.0, color='r', linestyle='--', label='Target')
plt.title(f'Closed Loop Step Response (Normalized)\nZeta={zeta}, Wn={
   omega_n}')
plt.xlabel('Time (s)')
plt.ylabel('Speed (normalized)')
plt.grid(True)
plt.legend()

plt.show()
```

6.1.6 Task3.py

```
import numpy as np
import matplotlib.pyplot as plt
import os
from car import Car
from controller import Y_controller as controller

v0 = 27.78
L = 2.7
dt = 0.001
T = 2.0
t = np.arange(0, T + dt, dt)

pulse_amp = 0.005
pulse_start = 0.2
pulse_end = 0.4

def steering_delta(time):
    return pulse_amp * (pulse_start <= time < pulse_end)

#nonlinear simulation
car_nl = Car(Ts=dt, initial_speed=v0)

phi_nl = np.zeros_like(t)
y_nl = np.zeros_like(t)
speed_nl = np.zeros_like(t)

for i in range(len(t) - 1):
    delta = steering_delta(t[i])

    #update longitudinal dynamics
    v, _, _ = car_nl.update(Fd=0.0, delta=delta, beta=0)

    speed_nl[i+1] = v

    dphi = (v / L) * np.tan(delta)
    dy = v * np.sin(phi_nl[i])

    phi_nl[i+1] = phi_nl[i] + dphi * dt
    y_nl[i+1] = y_nl[i] + dy * dt

#linearized simulation
phi_lin = np.zeros_like(t)
y_lin = np.zeros_like(t)

for i in range(len(t) - 1):
    delta = steering_delta(t[i])

    dphi = (v0 / L) * delta
    dy = v0 * phi_lin[i]

    phi_lin[i+1] = phi_lin[i] + dphi * dt
    y_lin[i+1] = y_lin[i] + dy * dt
```

```
err_phi = phi_nl - phi_lin
err_y   = y_nl  - y_lin

print('Pulse amplitude:', pulse_amp, 'rad')
print('Max yaw error  =', np.max(np.abs(err_phi)))
print('Max lateral error =', np.max(np.abs(err_y)))

#plotting
fig_ratio = 1.61803398875
fig_height = 4
figsize = (fig_height * fig_ratio, fig_height)

#save figures
SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
FIGS_PATH = os.path.join(SCRIPT_DIR, "figs")

plt.figure(figsize=figsize)
plt.plot(t, phi_nl, label='Nonlinear phi')
plt.plot(t, phi_lin, '--', label='Linear phi')
plt.xlabel('Time (s)')
plt.ylabel('Yaw angle (rad)')
plt.title('Nonlinear vs Linear Yaw Response')
plt.legend()
fig_path = os.path.join(FIGS_PATH, 'Task3_Linearization_Yaw')
plt.savefig(fig_path, dpi = 400)
plt.grid()
plt.show()

plt.figure(figsize=figsize)
plt.plot(t, y_nl, label='Nonlinear y')
plt.plot(t, y_lin, '--', label='Linear y')
plt.xlabel('Time (s)')
plt.ylabel('Lateral Position y (m)')
plt.title('Nonlinear vs Linear Lateral Response')
plt.legend()
fig_path = os.path.join(FIGS_PATH, 'Task3_Linearization_Position')
plt.savefig(fig_path, dpi = 400)
plt.grid()
plt.show()

plt.figure(figsize=figsize)
plt.plot(t, err_phi, label='Yaw error phi')
plt.plot(t, err_y, label='Lateral error y')
plt.xlabel('Time (s)')
plt.title('Errors')
plt.legend()
fig_path = os.path.join(FIGS_PATH, 'Task3_Linearization_Errors')
plt.savefig(fig_path, dpi = 400)
plt.grid()
plt.show()

# ---------------------- SIM --------------------------------
```

```
C_inner_Kp = 2

C_outer_Kp = 0.2160
C_outer_Ki = 0.3240

road_grade = 0
y_ref_list = [0.1, 20]

car_lin = Car(Ts=dt, initial_speed=v0)

#car parameters
Tp = 1/300
Ts = 1/60
m = 1300 #kg
Froll = 100 #N
a = 0.2 #Ns^2/m2
b = 20 #Ns/m
g = 9.8 #m/s^2
fd_min = -7000 #N
zeta = 0.95
eta_g = 0.8
eta_d = 3.8
rw = 0.34 #m
F_bar =  200 # m   /s
Te_max = 200 #Nm
F_max = (Te_max * eta_g * eta_d) / rw * zeta #mg/s from engine
F_min = -7000 #mg/s from brakes
L = 2.7 #m
delta_max = 0.05 #rad


v0_100 = 27.78 # KM/H
v0_150 = 150 * v0_100/100 #KM/H

SIM_TIME = 150.0
STEP_TIME = 2


def simulate_step_response(y_start, y_target, plot_title, filename,
   road_amp=0.0):

    # Update car class road profile manually
    Car.AMP = road_amp
    Car.road_beta_deg = Car.AMP * np.sin(2*np.pi/1000*Car.road_x + 300)
    Car.road_beta = np.deg2rad(Car.road_beta_deg)

    #defaults are set appropriately for designed controller of this
   specific task
    y_controller = controller(Ts=Ts)

    # Instantiate car object
    car_task3 = Car(Ts=Ts, initial_speed=v0)

    time_data = np.arange(0, SIM_TIME, Ts)
```

```
    y_pos_data = np.zeros_like(time_data)
    ref_data = np.zeros_like(time_data)
    delta_list = np.zeros_like(time_data)

    car_task3.y = y_start

    def F_d_ss(v, beta):
        return m*g*np.sin(beta) + Froll + a*v**2 + b*v

    # stay at v0 for sim
    control_force =  F_d_ss(v0, 0)

    for i, t in enumerate(time_data):
        desired_y = y_start
        if t >= STEP_TIME:
            desired_y = y_target
        ref_data[i] = desired_y

        delta = y_controller.update(desired_y, car_task3.y, car_task3.phi)
        delta_list[i] = delta

        # Using car class update
        car_task3.update(control_force, delta, 0)
        y_pos_data[i] = car_task3.y
    #plotting

    if filename:

        #Speed vs Time
        plt.figure(figsize=figsize)
        plt.plot(time_data, y_pos_data, label='Actual y Position (m)')
        plt.plot(time_data, ref_data, label='Desired y Posiiton (yref)')
        plt.axvline(STEP_TIME, linestyle=':', label='Step Time')
        plt.title(plot_title)
        plt.xlabel('Time (s)')
        plt.ylabel('y Posiiton (m)')
        plt.xlim([0,8])
        plt.legend()
        plt.grid(True)

        plt.tight_layout()
        save_path = os.path.join(FIGS_PATH, filename)
        plt.savefig(save_path, dpi = 400)
        plt.show()


for y_ref in y_ref_list:
        simulate_step_response(y_start=0, y_target=y_ref, plot_title=f'Y
   Step Simulation (y_ref={y_ref})', filename=f"T3_ystep_sim_yref_{int(
   y_ref)}.png", road_amp=0.0)
```

6.1.7 controller_task3_analysis.py

```
import numpy as np
import matplotlib.pyplot as plt
import control as ct


v0 = 27.78
L = 2.7

#-------------inner-------------
#gains
Kp_inner = 1

print(f'\nUsing Controller Gains:')
print(f'Kp = {Kp_inner}')

s = ct.TransferFunction.s

#plant
P_inner = v0/L * 1/s

#controller
C_inner = Kp_inner

TF_inner = C_inner * P_inner

#bode plot
plt.figure()
ct.bode_plot(TF_inner, dB=True, display_margins=True)
plt.suptitle(f'Bode Plot of Inner Transfer Function')

#closed-loop TF
T_feedback_inner = ct.feedback(TF_inner, 1)

#step response
t_inner = np.linspace(0, 1, 2000)
t_resp_inner, y_inner = ct.step_response(T_feedback_inner, T=t_inner)

y_final_inner = y_inner[-1]
y_max_inner = np.max(y_inner)

# Overshoot
overshoot_inner = max(0, (y_max_inner - y_final_inner) / y_final_inner *
    100)

# Rise time 10  90  %
t_10 = np.where(y_inner >= 0.1 * y_final_inner)[0][0]
t_90 = np.where(y_inner >= 0.9 * y_final_inner)[0][0]
rise_time_inner = t_resp_inner[t_90] - t_resp_inner[t_10]

# Settling time (2%)
upper_inner = 1.02 * y_final_inner
lower_inner = 0.98 * y_final_inner
```

```
outside_inner = (y_inner > upper_inner) | (y_inner < lower_inner)
if np.any(outside_inner):
    last_idx_inner = np.where(outside_inner)[0][-1]
    settling_time_inner = t_resp_inner[last_idx_inner]
else:
    settling_time_inner = 0

# --- Print Metrics ---
print(f'\nPerformance Metrics:')
print(f'Final Value: {y_final_inner:.4f}')
print(f'Overshoot: {overshoot_inner:.2f}%')
print(f'Rise Time: {rise_time_inner:.4f} s')
print(f'Settling Time: {settling_time_inner:.4f} s')

# --- Plot Step Response ---
plt.figure()
plt.plot(t_resp_inner, y_inner)
plt.axhline(1.0, color='r', linestyle='--', label='Reference')
plt.title('Step Response with Prefiltered Closed Loop')
plt.xlabel('Time (s)')
plt.ylabel('Output')
plt.grid(True)
plt.legend()

plt.show()

#-------------outer-------------
#gains
Ki = 0.4039776818
Kp = 0.2411807055

print(f'\nUsing Controller Gains:')
print(f'Kp = {Kp}')
print(f'Ki = {Ki}')

# plant (outer plant includes closed-loop inner loop)
P_outer = T_feedback_inner * (v0 / s)

#controller
C = (Kp * s + Ki) / s

TF_outer = C * P_outer

#bode plot
plt.figure()
ct.bode_plot(TF_outer, dB=True, display_margins=True)
plt.suptitle(f'Bode Plot of Transfer Function')

#closed-loop TF
T_feedback = ct.feedback(TF_outer, 1)

#precompensator
F = (Ki/Kp) / (s + Ki/Kp)
```

```
# Final transfer function
T_final = F * T_feedback

#step response
t = np.linspace(0, 10, 2000)
t_resp, y = ct.step_response(T_final, T=t)

y_final = y[-1]
y_max = np.max(y)

#overshoot
overshoot = max(0, (y_max - y_final) / y_final * 100)

#rise time 10  90  %
t_10 = np.where(y >= 0.1 * y_final)[0][0]
t_90 = np.where(y >= 0.9 * y_final)[0][0]
rise_time = t_resp[t_90] - t_resp[t_10]

#settling time 2%
upper = 1.02 * y_final
lower = 0.98 * y_final
outside = (y > upper) | (y < lower)
if np.any(outside):
    last_idx = np.where(outside)[0][-1]
    settling_time = t_resp[last_idx]
else:
    settling_time = 0

#metrics
print(f'\nPerformance Metrics:')
print(f'Final Value: {y_final:.4f}')
print(f'Overshoot: {overshoot:.2f}%')
print(f'Rise Time: {rise_time:.4f} s')
print(f'Settling Time: {settling_time:.4f} s')

#step response
plt.figure()
plt.plot(t_resp, y)
plt.axhline(1.0, color='r', linestyle='--', label='Reference')
plt.title('Step Response with Prefiltered Closed Loop')
plt.xlabel('Time (s)')
plt.ylabel('Output')
plt.grid(True)
plt.legend()

plt.show()
```

6.1.8 controller.py

```
from PID import PID
from car import Car
import numpy as np
import math

import enum
#THE BEGINNING OF THIS IS THE SAME AS TASK 2 AND 3 CONTROLLERS
class States(enum.Enum):
    STRAIGHT = enum.auto()
    MOVING_LEFT = enum.auto()
    MOVING_RIGHT = enum.auto()


class Other_Car:
    def __init__(self, rel_x, rel_speed):
        self.rel_x = rel_x
        self.rel_speed = rel_speed


# Task 2 controllers

class Precompensator:
    def __init__(self, Kp, Ki, Ts, initial_setpoint_ref):
        self.KP = Kp
        self.Ki = Ki
        self.Ts = Ts

        denom = Kp + Ki * Ts
        self.alpha = (Ki * Ts) / denom
        self.beta = Kp / denom
        self.Vf_prev = initial_setpoint_ref

    def filter(self, Setpoint_ref_current):
        Vf_current = self.alpha * Setpoint_ref_current + self.beta * self.
   Vf_prev
        self.Vf_prev = Vf_current
        return Vf_current

class V_controller:

    def __init__(self, Kp=4323.888, Ki=3647.3125, Ts=1/60, umax=10000,
   umin=-10000, Kaw=1):
        #Kaw = 1.0/Ts #standard value

        v0 = 27.78
        a = 0.2
        b = 20
        F_roll = 100
        F_drag = F_roll + a * v0**2 + b * v0
        #insantiate controller
        self.PI = PID(Kp=Kp, Ki=Ki, Ts=Ts, umax=umax, umin=umin, Kaw=Kaw,
```

```
   initialState=F_drag)
        self.precomp = Precompensator(Kp=Kp, Ki=Ki, Ts=Ts,
   initial_setpoint_ref=v0)
        self.last_filtered_ref = 0.0

    def update(self, desired_speed, speed):
        V_ref_filtered = self.precomp.filter(desired_speed)

        force = self.PI.update(V_ref_filtered, speed)

        self.last_filtered_ref = V_ref_filtered
        return force


# Task 3 Controllers
#TODO never used??
class Precompensator_t3:
    def __init__(self, Ts = 1/60, alpha = 1.5):
        self.y_f = 0
        self.alpha = alpha
        self.Ts = Ts

    def filter(self, y):

        self.y_f += self.Ts * self.alpha*(y-self.y_f)

        return self.y_f


class Y_controller:
    def __init__(self, Ts=1/60,
                 Kp_inner=0.5, delta_max=0.05, delta_min=-0.05,
                 phi_max=np.deg2rad(15), phi_min=-np.deg2rad(15), P_loc =
   2):

        # Caluclate controller/Precomp Values:
        v0 = 27.78

        Kp_outer = 2*P_loc/v0
        Ki_outer = v0*Kp_outer**2/4

        self.c_inner = PID(Kp=Kp_inner, Ts=Ts, umax=delta_max, umin=
   delta_min) # Detla Controller Psi des -> Delta
        self.c_outer = PID(Kp=Kp_outer, Ki=Ki_outer, Ts=Ts, umax=phi_max,
   umin=phi_min, Kaw=1.5) #TODO change from 0, initialState=0.0) #Y
   controller Y-> psi_des

        self.precomp = Precompensator_t3(alpha=1) #Precompensator(Kp=
   Kp_outer, Ki=Ki_outer, Ts=Ts, initial_setpoint_ref=0) # Precomp




    def update(self, y_des, y, phi):
```

```
        y_filt = self.precomp.filter(y_des)

        phi_des = self.c_outer.update(y_filt, y)

        delta = self.c_inner.update(phi_des, phi)

        return delta


#Task 4: putting it all together

#=============BEGIN MPC=================================
#vehicle parameters
m = 1300.0
Froll = 100.0
a = 0.2
b = 20.0
g = 9.8
zeta = 0.95
eta_g = 0.8
eta_d = 3.8
rw = 0.34
F_bar = 200.0  # mg/s minimum fuel
v_min = 20.83
v_max = 27.78

# MPC parameters
Ts_mpc = 1.0
N = 10 #horizon length
tau_v = 1.0 #closed loop speed
lambda_tracking = 1.0

# BSFC function
def bsfc_from_Ne_Te(Ne, Te):
    return ((Ne - 2700.0)/12000.0)**2 + ((Te - 150.0)/600.0)**2 + 0.07

def F_d_ss(v, beta):
    return m*g*np.sin(beta) + Froll + a*v**2 + b*v

def fuel_rate_ss(v, beta):
    Fd = F_d_ss(v, beta)
    # engine torque and speed
    Te = (1.0/zeta) * (rw / (eta_g * eta_d)) * Fd
    Ne = (60.0/(2.0*math.pi)) * (eta_g*eta_d / rw) * v
    bsfc = bsfc_from_Ne_Te(Ne, Te)
    fr = (1.0/zeta) * bsfc * Fd * v
    return max(fr, F_bar)

def predict_other_positions_global(x0, other_cars, k, Ts, v0):
    #Predict global positions of other cars after k steps.
    preds = []
    for oc in other_cars:
        other_speed = v0 + oc[2]
```

```
        other_x_k = (x0 + oc[0]) + other_speed * (k * Ts)
        preds.append(other_x_k)
    return preds

def mpc_cost_for_vdes(v_des_candidate, x0, v0, grade_obj, other_cars,
   v_driver, Ts=Ts_mpc, N=N, tau=tau_v, safety_gap_min=10.0):
    #simulate coarse dynamics forward for constant v_des_candidate and
   return total cost.
    x = x0
    v = v0
    total_cost = 0.0

    #if any current other car is already closer than safety gap, reject
   candidates that reduce gap
    for oc in other_cars:
        if oc[0] < safety_gap_min:
            pass

    for k in range(N):
        #road grade at x
        beta_deg = grade_obj.grade_at(x)
        beta = math.atan(beta_deg/100.0)

        #fuel burn at current v
        Jss = fuel_rate_ss(v, beta)
        total_cost += Jss * Ts

        #first-order to v_des
        v = v + (Ts/tau) * (v_des_candidate - v)
        x = x + Ts * v  #predicted ego position at next step

        #predict other vehicles' positions at future time
        for oc in other_cars:
            other_speed = v0 + oc[2]
            other_x_k = (x0 + oc[0]) + other_speed * ((k+1)*Ts)
            ego_x_k = x
            gap = other_x_k - ego_x_k

            #if other vehicle is behind, skip
            if gap <= 0:
                continue

            #if gap ever drops below safety threshold, reject
            if gap < safety_gap_min:
                return 1e9 #huge penalty, candidate not safe

    total_cost += lambda_tracking * (v_des_candidate - v_driver)**2
    return total_cost

def mpc_select_v_des(x0, v0, grade_obj, other_cars, v_driver,
   search_resolution=0.1, safety_gap_min=16.0):
    # allowable slack: clamp (v_driver +/- 3) into [v_min, v_max]
    low = max(v_min, v_driver - 3.0)
    high = min(v_max, v_driver + 3.0)
```

```
    candidates = np.arange(low, high + 1e-6, search_resolution)
    best_v = v_driver
    best_cost = 1e12
    for cand in candidates:
        c = mpc_cost_for_vdes(cand, x0, v0, grade_obj, other_cars,
   v_driver, Ts=Ts_mpc, N=N, tau=tau_v, safety_gap_min=safety_gap_min)
        if c < best_cost:
            best_cost = c
            best_v = cand
    return best_v

#================END MPC=================================================

class Controller:

    LANE_MIDPOINT_OFFSET = 11.25
    LANE_THRESH = LANE_MIDPOINT_OFFSET*0.9
    LANE_DIFF_LATERAL_READING = 45 # The reading when a car is in another
   lane
    LAT_READING_CONVERSION_FACTOR = (2*LANE_MIDPOINT_OFFSET)/
   LANE_DIFF_LATERAL_READING # Conversion factor for other car rel y to
   meters

    # Safety params
    FOLLOWING_TIME_S = 6.0   # seconds headway to consider for passing
    MIN_ABS_GAP_M = 12.0     # absolute min gap (m) that triggers
   immediate evasive lane change if possible
    SAFETY_MIN_GAP_PRED = 16.0 # safety gap used for MPC predictions (kept
    in sync with MPC settings)


        # Ts: sample time of the controller;
    # initial_conditions: two element vector representing the equilibrium
    # i.e., [equilibrium of driving force Fd, equilibrium of vehicle speed
   ]
    def __init__(self, Ts, initial_conditions):
        # parameters go here
        self.Ts = Ts
        self.Fd_cmd = initial_conditions[0]
        self.speed = initial_conditions[1]

        self.F_d_min = -7000.0 #min force (N) (Given)
        c = Car(0,0)
        self.F_d_max = c.F_max_force

        # states/controller initialization go here
        self.pos = 0

        self.v_controller = V_controller(Ts = Ts,
                                         umin = self.F_d_min,
                                         umax = self.F_d_max)

        self.y_controller = Y_controller()
```

```
        self.state = States.MOVING_RIGHT
        self.last_v_des_opt = 0.0



    def time_to_impact(self, rel_x, rel_speed):
        # rel_x: positive if other car is ahead
        # rel_speed = other_speed - ego_speed
        # We only care if we are closing (rel_speed < 0), i.e., other is
slower than us
        if rel_speed >= -0.01:  # not closing (or nearly zero)
            return None

        # avoid divide-by-zero and extremely large times due to tiny
rel_speed
        safe_rel_speed = abs(rel_speed)
        if safe_rel_speed < 0.1:
            # treat as "not imminently closing" (will rely on absolute gap
)
            return None

        return abs(rel_x / rel_speed)


    def state_machine(self, y, other_cars, desired_speed, speed):
        FOLLOWING_DISTANCE = self.FOLLOWING_TIME_S # sec
        MIN_VERT_FOR_LANE_CHANGE = 11 # meters

        #init return, desired_speed defaults to input value
        desired_y = None
        anticipated_lane_right = None

        # ------------------------------------ Check if car needs to
finish lane change -------------------------
        # Check if a previously started lane change was completed if not
complete that turn first
        if (((self.state == States.MOVING_RIGHT) and (y > self.LANE_THRESH
))       # Arrived in right lane
            or ((self.state == States.MOVING_LEFT) and (y < -self.
LANE_THRESH))):  # Arrived in left lane

            # Set to keep no longer turning
            self.state = States.STRAIGHT
        elif((self.state == States.MOVING_RIGHT) or (self.state == States.
MOVING_LEFT)):
            if (self.state == States.MOVING_LEFT):
                desired_y = -self.LANE_MIDPOINT_OFFSET
            elif (self.state == States.MOVING_RIGHT):
                desired_y = self.LANE_MIDPOINT_OFFSET
            else:
                print("ERROR finishing turn")

            return desired_y, desired_speed
```

```
        # Determine which lane we are anticipating being in for planning
        if (self.state == States.MOVING_RIGHT) or ((y >= self.LANE_THRESH)
                                                   and (self.state ==
States.STRAIGHT)):
            anticipated_lane_right = True
        elif (self.state == States.MOVING_LEFT) or ((y <= -self.
LANE_THRESH)
                                                    and (self.state ==
States.STRAIGHT)):
            anticipated_lane_right = False
        else:
            # fallback
            anticipated_lane_right = (y >= 0)




        #---------------------------------------- Gather info on other
cars --------------------------------
        # each index: (rel pos, rel speed, time to impact)
        cars_right_lane = []
        cars_left_lane = []

        # Read in all car data
        for npc_car in other_cars:

            rel_x = npc_car[0]

            # If car is behind us by enough of a margin we don't care
            if rel_x < -MIN_VERT_FOR_LANE_CHANGE:
                continue

            rel_y = npc_car[1] * self.LAT_READING_CONVERSION_FACTOR
            rel_speed = npc_car[2]  # this is other.speed - ego.speed
            abs_y = y + rel_y

            # Only consider cars that are in approximate lane center (left
/right)
            if abs_y > self.LANE_THRESH:
                # In right lane
                tti = self.time_to_impact(rel_x, rel_speed)
                cars_right_lane.append((rel_x, rel_speed, tti))

            elif abs_y < -self.LANE_THRESH:
                # In left lane
                tti = self.time_to_impact(rel_x, rel_speed)
                cars_left_lane.append((rel_x, rel_speed, tti))

            else:
                # treat as in same lane - consider for immediate safety
                tti = self.time_to_impact(rel_x, rel_speed)
                # place it in the closer lane list (based on sign of rel_y
)
```

```
                if rel_y >= 0:
                    cars_right_lane.append((rel_x, rel_speed, tti))
                else:
                    cars_left_lane.append((rel_x, rel_speed, tti))

        # print(f"Car Y: {y:<6.2f}| Rel x: {npc_car[0]:<6.2f}| Rel y:
{npc_car[1]* self.LAT_READING_CONVERSION_FACTOR:<6.2f}| rel speed: {
npc_car[2]:<6.2f}| LL car #: {len(cars_left_lane)}| RL car #: {len(
cars_right_lane)}")



    #---------------------------------------- Plan next step
------------------------------

    # ______ From here on out, we are currently in a steady state
---------

    # Check for impending collision:
    closest_impact_time = None

    # Utility to find min tti and min absolute gap in a list
    def analyze_lane_list(lst):
        min_tti = None
        min_gap = None
        for (rx, rs, tti) in lst:
            if min_gap is None or rx < min_gap:
                min_gap = rx
            if tti is not None:
                if min_tti is None or tti < min_tti:
                    min_tti = tti
        return min_gap, min_tti

    right_min_gap, right_min_tti = analyze_lane_list(cars_right_lane)
    left_min_gap, left_min_tti = analyze_lane_list(cars_left_lane)

    # Safety checks use absolute gap first (immediate)
    # When anticipating moving right:
    if anticipated_lane_right:
        # If immediate right lane car is dangerously close, try to
move left (if free)
        if right_min_gap is not None and right_min_gap < self.
MIN_ABS_GAP_M and not cars_left_lane:
            self.state = States.MOVING_LEFT
            # immediate return: start moving now
            desired_y = -self.LANE_MIDPOINT_OFFSET
            return desired_y, desired_speed

        # Otherwise check time-to-impact for cars ahead in our lane
        if right_min_tti is not None and (right_min_tti <
FOLLOWING_DISTANCE):
            # If left lane is clear, pass
            if not cars_left_lane:
                self.state = States.MOVING_LEFT
```

```
                else:
                    # slow down to match leader speed (use rel_speed of
closest)
                    # pick the minimal tti occuring car to compute
rel_speed
                    # find that car
                    min_tti = right_min_tti
                    for (rx, rs, tti) in cars_right_lane:
                        if tti == min_tti:
                            desired_speed = speed + rs
                            break
        else:
            # anticipating left lane
            if left_min_gap is not None and left_min_gap < self.
MIN_ABS_GAP_M and not cars_right_lane:
                self.state = States.MOVING_RIGHT
                desired_y = self.LANE_MIDPOINT_OFFSET
                return desired_y, desired_speed

            if left_min_tti is not None and (left_min_tti <
FOLLOWING_DISTANCE):
                if not cars_right_lane:
                    self.state = States.MOVING_RIGHT
                else:
                    min_tti = left_min_tti
                    for (rx, rs, tti) in cars_left_lane:
                        if tti == min_tti:
                            desired_speed = speed + rs
                            break

        # find the desired y from desired lane movement
        if (self.state == States.MOVING_LEFT):
            desired_y = -self.LANE_MIDPOINT_OFFSET
        elif (self.state == States.MOVING_RIGHT):
            desired_y = self.LANE_MIDPOINT_OFFSET
        elif (self.state == States.STRAIGHT):
            #Keep lane
            if anticipated_lane_right:
                desired_y = self.LANE_MIDPOINT_OFFSET
            else:
                desired_y = -self.LANE_MIDPOINT_OFFSET

        return desired_y, desired_speed



    # speed, y, phi: ego vehicle's speed, lateral position, heading
    # desired_speed: user defined speed setpoint (flexible by plus minus 3
    degrees up to speed limits)
    # des_lane: desired lane specified by the user. -1 is left lane, +1 is
    right lane
    # other_cars: each row of this list corresponds to one of the other
vehicles on the road
    #             the columns are: [relative longitudinal position,
```

```
relative lateral position, relative speed]
 # grade is a road grade object. Use grade.grade_at to find the road
grade at any x
 def update(self, speed, x, y, phi, desired_speed, des_lane, other_cars
, grade):
     #fuel optimization MPC. must not be allowed to override safety
decisions
     v_driver = desired_speed
     #for extra safety, pass MPC a safety_gap value consistent with
controller settings
     v_des_opt = mpc_select_v_des(x, speed, grade, other_cars, v_driver
, safety_gap_min=self.SAFETY_MIN_GAP_PRED)
     #never go faster than what the planner suggested, MPC may suggest
slower speeds for fuel saving,

     v_des_opt = min(v_des_opt, desired_speed)
     self.last_v_des_opt = v_des_opt



     # Path planning & lane-change decision
     # NOTE: state_machine uses current 'speed' and raw other_cars (not
 affected by MPC)
     self.desired_y, desired_speed = self.state_machine(y, other_cars,
v_des_opt, speed)
     #self.desired_y, desired_speed = self.state_machine(y, other_cars,
 27.78, speed) # Kept for easy checking for instructor =) (uncomment
for max speed to demo more lane changes if needed)


     self.delta_cmd= self.y_controller.update(self.desired_y, y, phi)



     self.Fd_cmd = self.v_controller.update(desired_speed, speed)

     #self.Fd_cmd = self.v_controller.update(desired_speed, speed) #
used before fuel optimization)

     return self.Fd_cmd, self.delta_cmd
```