**Intro to Autonomous Vehicles Final Design Project**

Daniel Taylor

Florida Polytechnic University

24 November 2024

EEL4332: Intro to Autonomous Vehicles

**Part 1 – Designing & Implementing Pure Pursuit Controller**

To implement a longitudinal and lateral controller in MATLAB, I first compared at the Stanley controller and Pure Pursuit controllers from the MATLAB Simulink libraries. I installed packages for each. I used the MATLAB Add-On Explorer to download the Navigation Toolbox, Automated Driving Toolbox, Curve Fitting Toolbox, Simulink Coder, Statistics and Machine Learning Toolbox, Vehicle Dynamics Blockset, DSP System Toolbox, Signal Processing Toolbox, and the ROS Toolbox. This allowed me to view the functions and Simulink blocks for the Pure Pursuit and Stanley controllers.

A screenshot of a computer

Description automatically generatedThe Stanley controllers had two blocks, one for lateral control, and the other for longitudinal control, while the Pure Pursuit controller just had one block, as seen below. For this reason, I went with pure pursuit.

Figure 1 - Stanley and Pure Pursuit Blocks in Simulink

As can be seen from the photo, the Pure Pursuit controller has two input ports: Pose and Waypoints. Pose takes in a three-element state vector of xy-positions and vehicle heading, (theta), while Waypoints accepts an N-by-2 matrix as xy-coordinates. These are generated later in Part 3, so they were left initially unconnected. The controller also has two output ports: Linear Velocity and Angular Velocity. These return linear and angular velocities as scalar values, respectively, according to MATLAB. These outputs were then connected to the Kinematic and Longitudinal vehicle models made previously. The controller also has a lookahead distance, which can be tuned to determine how closely the test car follows the waypoint path in Part 3. To improve the path

**Part 1 – Designing & Implementing Pure Pursuit Controller (cont.)**

tracking, the distance should decrease, however this may lead to instability issues. I specified this to be an exact input port, modeled by the equation,

where “ld” is the lookahead distance, R is the radius of the wheels on the vehicle, and alpha, , is the “lookahead angle”, the angle between the vehicle’s current heading and the line connecting the vehicle’s position to the next waypoint. Since we are using the Kinematic model, the radius R can also be found by the following equation:

This means we can find the steering angle, , by substitution. This results in the equation:

For implementation, as can be seen in the figure below, I utilized the equations above to find the lookahead distance, set that as the parameter for the Pure Pursuit controller, and then connected the inputs and outputs to the longitudinal and kinematic model. After the waypoints are generated in Part 3, they will be connected back to the Pure Pursuit controller to obtain an output.

A diagram of a computer scheme

Description automatically generated

Figure 2 - Implementation of Pure Pursuit Controller

**Part 2 - Designing/Implementing the PID Controller**

A Proportional-Integral-Derivative (PID) controller is an incredibly useful technology in an autonomous vehicle to regulate the longitudinal motion under different circumstances. It takes in the actual speed at t = 0, calculates the speed with the PID algorithm and uses that value in a feedback loop to calculate the error. The Proportional (P) component will respond to current error by generating an output proportional to the error’s magnitude [1]. This is most commonly seen when first driving a car from rest, like a typical 0-60 test. The Integral (I) component will use the area under the error curve for long-term deviations. This is used commonly when the vehicle is cruising, as the speed is relatively constant. Lastly, the Derivative (D) component is used when sharp changes in error occur, like when a car hits a speedbump or brakes suddenly for half a second.

When designing the PID controller, I was initially looking at just the “PID Controller” Simulink block, but I decided I wanted to model my own to further my understanding on how PID Controllers work. For my model, I implemented the following transfer function:

A screenshot of a computer

Description automatically generated Then, to first test my PID controller, I implemented a simple step function for the input, and used a sum block, the proportional gain Kp, and then the PID transfer function from above. With the “|+-” as the list of signs, and the input going into the plus sign, and the output going into the minus sign, this meant that the error could be found and fed back into the PID controller to correct it and get closer to the actual speed. I used a 2x1 multiplexer with the inputs as the input and output, which then connected to a Scope to test the output. The result is seen in the figure below.

Figure 3 - Early Form of PID Controller with Output

**Part 2 - Designing/Implementing the PID Controller (cont.)**

A screenshot of a computer

Description automatically generatedWith a proportional gain of 4 in this rudimentary PID controller, the controller corrects itself to the actual speed within around 6 seconds. However, as one can notice, this value is about 90% of the actual value. This is to be expected, as we have only implemented the “P” in “PID”, meaning the integral and derivative parts of the PID controller are left. The speed is constant at the end, so this instance of the error getting “stuck” can be fixed by the Integral component’s gain. To expand upon the controller, I used another sum block, the integrator block, the derivative block, and two gain blocks for Integral and Derivative gains, KI and KD. The second sum block has the list of signs “+++”, so that the Proportional, Integral, and Derivative components can be added together before being run through the transfer function. Below is a screenshot of the Simulink model along with the output. It is important to remember that for this demonstration of the PID controller, the derivative with the step function will throw an error with Simulink. However, since the actual implementation will be with the controller connected to the models we made, we can ignore this for now.

Figure 4 - Updated PID Controller

**Part 2 - Designing/Implementing the PID Controller (cont.)**

A diagram of a circuit

Description automatically generatedNow that the PID Controller is designed, it is time to implement it to our models. This is accomplished by connecting the velocity output from the pure pursuit controller to the PID controller, as seen below.

**Part 3 – Reference Path Through Driving Scenario Driver**

Using the Driving Scenario Designer, I initially added a road at the origin (0,0) of the graph. I then expanded the road, gave it a curve which offset it from the origin, and increased the number of lanes from one to three. Three lanes were chosen, as the scenario would be Dr. Adla realizing she missed Exit 41 on Interstate-4 (I4) on the way from Tampa, Florida to Florida Polytechnic University. This meant that her Tesla would have to switch across 3 lanes (the required 3 lane changes as part of the design requirement) from the left lane to the right lane. I also had the vehicle perform one lane change before the realization for further testing of my model. One can imagine this as before she realizes the turn was missed, her Tesla was weaved out of the left lane to the middle lane and back to pass a slower driver. As seen in Figure 5, a rudimentary diagram of I4, exit 41, and the toll road Polk Parkway was created.

A graph of a road

Description automatically generated

Figure 5 - Diagram of Road Scenario

**Part 3 – Reference Path Through Driving Scenario Driver (cont.)**

After creating the test environment, the next step is to insert an actor. This actor is a vehicle and can be either a car or a truck. Both classes are models shaped like a box, with the truck model having an increased height and width. The 3D display type can also be changed, which for cars was between a Sedan, Muscle Car, SUV, Small Pickup Truck, Hatchback, Box Truck, or Cuboid. For trucks, it was the same selection, however MATLAB would automatically select the “Box

Truck” optionA white cube on a road

Description automatically generated Since this scenario was based off Dr. Adla’s own white Tesla, I went with the Car class with the Sedan 3D Display Type. After selecting, MATLAB will show a 3D model of the car on the road you create as an Ego-Centric View. Below is a screenshot of what that looked like.

Figure 6 - Ego Centric View of Car Provided by MATLAB

After choosing the correct actor, the next step was to add the waypoints the test car would follow to simulate my scenario. The waypoints that were added were to simulate the actor making lane changes whilst driving along the road, and then make a sharp lane change to successfully make the exit. It then cruises along the following road. This completed my scenario for Dr. Adla’s Tesla to weave throughout I4, almost miss Exit 41, and then cruise along Polk Parkway on the way to campus. Of course, it is important to note that the Tesla’s self-driving capabilities are what is maneuvering the car throughout this entire scenario. Dr. Adla would most definitely not drive recklessly under any circumstances. According to MATLAB, a total of 40 waypoints were used that each generated its own (x,y) coordinate. The figure below shows the waypoints in white, matching the color of the actor car.

**Part 3 – Reference Path Through Driving Scenario Driver (cont.)**

A graph of a curve

Description automatically generated

Figure 7 - Road Diagram with the Waypoints Set

Now that the waypoints have been created and tweaked around, the simulation can be run. Below is a video of the scenario in action.

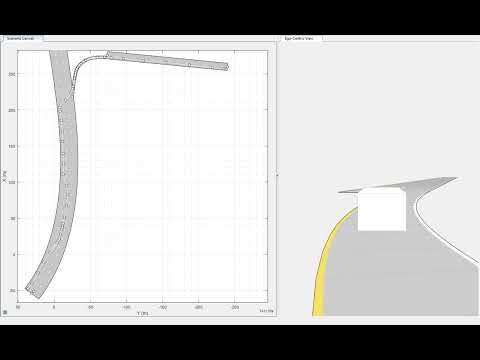
[](https://www.youtube.com/embed/vrWCvQH8QPk?feature=oembed)

Figure 8 - Video of the Actor Vehicle Following the Set Waypoints

**Part 4 - Testing Controller in the Scenario**

To test the controller, the matrix outputs given from the Driving Scenario Designer can be used for the Pose and Waypoints inputs of the Pure Pursuit controller. As explained in parts 1 and 2, this is connected to the PID controller, and to the vehicle model. This will create a feedback loop and calculate the position, velocity, acceleration, yaw angle, and steering angle for each time step, feeding these data values back into the system for the car to perform the scenario.

The Pose Matrix and Waypoints Matrix come from the Simulink import sources, which use their respective .MAT files created by the Driving Scenario Design scenario. This would be accomplished further by using Multiplexer blocks to get the output as one line for it to work with the Pure Pursuit Controller.

A diagram of a signal

Description automatically generated

Figure 9 - Connecting Output of Driving Scenario Designer to Model

**Part 4 - Testing Controllers in the Scenario (cont.)**

Unfortunately, my Simulink was having errors outputting. I tried for hours to fix the errors, going through multiple guides on how to import arrays with a Pure Pursuit controller in Simulink from to getting the data directly from the Driving Scenario Designer, but I just could not figure it out. I was not sure if my model, Pure Pursuit controller, PID controller, or simulation was at fault. I heard that Dr. Adla said for this assignment that an explanation of why our model did not work was alright for submission. For the sake of the assignment, I believe it did not work due to the fact I had trouble importing the matrixes from Driving Scenario Designer. However, I was determined to see an output and was not ready to give up. I restarted my model and went with the Stanley controller approach instead. This was the better option when compared to Pure Pursuit for my scenario, which I did not initially think about.

A diagram of a computer

Description automatically generatedUtilizing a Stanley lateral controller and Stanley subsystem, I connected this to a redone version of the model, as seen below.

Figure 10 - Stanley Controller Connected to Model

Then, I reutilized the PID controller I constructed before and connected that to the model.

A close-up of a diagram

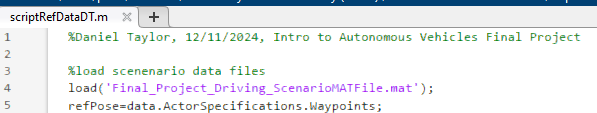
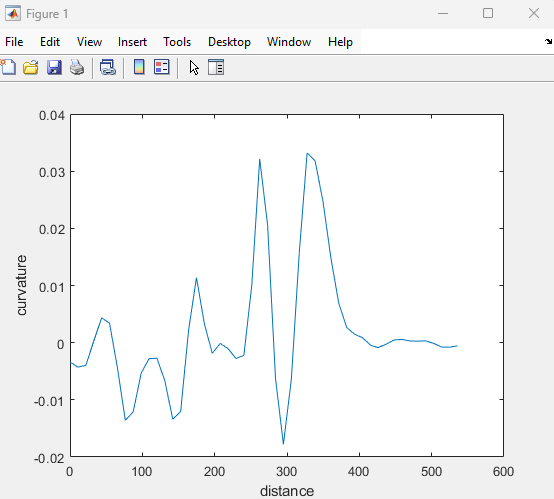
Description automatically generated

Figure 11 – PID Controller Connected to the Model

**Part 4 - Testing Controllers in the Scenario (cont.)**

This meant that my design was finally complete. I had the PID controller for longitudinal correction, and the Stanley controller for lateral correction. Using the same driving scenario as before, the .mat file gave the following output graph based on the .m file below.

Figure 12 - MATLAB Code and Corresponding Output Graph



The meaning of the graph made sense, as the sharp dip halfway between should be from the sudden lane changes, and then the almost-constant looking graph towards the end must be from the car cruising on the road after the exit.

**Part 4 - Testing Controllers in the Scenario (cont.)**

A graph with a red line

Description automatically generatedAfterwards, it was time to implement the scenario with the newly created model. The output worked this time, giving the following graphs:

Figure 13 - Graph of XY Coordinates

A graph with red and blue lines

Description automatically generated

Figure 14 - Uncorrected (Orange) vs. Corrected (Blue)

**Part 4 - Testing Controllers in the Scenario (cont.)**

**A graph on a black background

Description automatically generated**

Figure 15 - PID Controller Output Graph

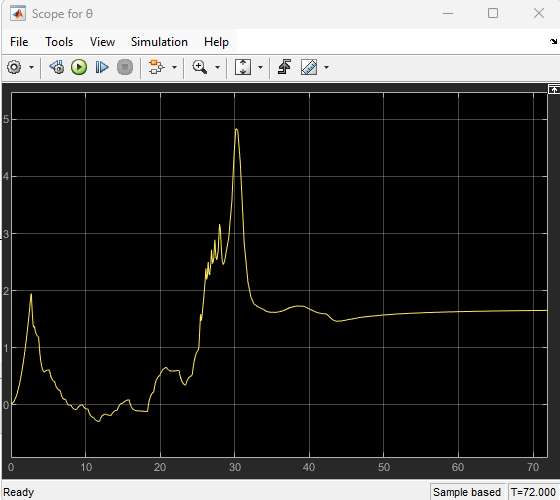


Figure 16 - Orientation Angle Output Graph

**Part 4 - Testing Controllers in the Scenario (cont.)**

For each graph, time was stepped from 0 to 72. Analyzing the graphs, these graphs make sense when considering the scenario at play. In the graph of the XY coordinates, the sharp turn from the driving scenario is clearly visible. At the start of the graph, it moves to the right, while the car in the scenario moves forward on the scenario grid. After the turn, the car goes perpendicular to its initial velocity, moving to the right instead of forward. This is reflected in the XY graph, as the end of the graph is perpendicular to the beginning of the graph.

The Uncorrected vs. Corrected graph makes it easy to understand how the Stanley and PID controllers make corrections using the pose, velocity, position, and direction of the vehicle. These corrections allow the output graph to be smoother.

The PID controller graph shows how the error decreases over time and the speed calculated by the controller catches up to the actual speed using the Proportional, Integral, and Derivative components.

The Orientation Angle output graph shows exactly how the car moves in the scenario. At the beginning, it is a little shaky with a spike from the first lane change. Then, a large spike is seen from what can assumed to be the quick two lane changes as the Tesla in the scenario almost misses the exit. After the exit is made, the vehicle can cruise, so the graph shows the orientation angle as constant.

**Conclusion**

In the end, an output was achieved for this project. It was interesting implementing the Stanley, Pure Pursuit, and PID controllers after learning about them from lecture. It was also fun making a driving scenario in the Driving Scenario Designer program. Although there were hardships to overcome, such as restarting and going with the Stanley controller approach instead of the Pure Pursuit controller approach, and the annoyance of getting the MATLAB files to work with the Simulink files. The outputs all generated made sense, and fit the scenario that was created, correctly fulfilling the 3 lane changes requirement.

**References**

“The PID Controller & Theory Explained.” *NI*, www.ni.com/en/shop/labview/pid-theory-explained.html#:~:text=As%20the%20name%20suggests%2C%20PID,varied%20to%20get%20optimal%20response. Accessed 9 Dec. 2024.