CSC 5280
Final Report
CyberPhysical Systems

Austin Burns

Project Overview

This project uses the open source driving simulator Carla [1]. This simulator allows users to control various agents in a server-based simulator environment. The graphic rendering of this simulator is done using the Unreal Engine 4, while the creation and control of agents in the environment is handled using the Carla python API [2]. The python API was the primary point of interaction for this project.

On reviewing the introductory Carla paper from Dosovitskiy [3], we saw the implementation of a modular control pipeline for self driving a vehicle in the Carla environment. This control pipeline involved steering control, throttle/brake control, as well as perception modules using reinforcement learning. In this project, we first implemented a static throttle controller – maintaining the speed of our car on a straight line with no intervention other than the physics of the simulation. We then incorporated object detection sensors for collision avoidance, which required a more refined tuning of our throttle PID, which we will see later in this paper. Finally we incorporate a basic graph based path-planning algorithm which is set up for the arbitrary extension to more complex path planning techniques, which we explore later in this paper. This allowed us to extend our drive time, as well as our test time for PID tuning.

Carla Environment

We set up our testing environment by running the Carla Simulator binary and creating several actors within using the python API. A main 'hero' car is created, as well as cameras and various sensors – the positions of which were tuned for ideal experimentation. Using the default asynchronous mode, we use a while loop in our python program to update the position of our first person camera relative to the car, allowing us to view in real time the activity of the car. Asynchronous mode allows for the simulation time steps to tick uncoupled from the activity of the python API. We found this to be ideal for our experiments, as well as most aligned with real world applications.

For object detection testing, we have modified the example program 'generate traffic.py' to create testing traffic for perturbing our PID controller. We created a shell script to run the simulator, and then parameterized generate traffic by 'n' giving various amounts of cars for testing. We also parameterized the spawn point of the main car, which can be selected from an index of all spawn points given by the python API. Our modified traffic generation avoids spawning test cars in the location we spawned our primary car.

Waypoints are provided through the API, the handling of which we address in 'Path Planning'.

Data collection objects are instantiated in our main program to collect telemetry and control data for testing feedback.

Control

In Control System Design [6], Goodwin identifies PID controllers as a simple and robust method for plant variable control, which in our case is throttle percentage, the range of which is $[0, 1]$, no throttle to full throttle. A PID attempts to control a plant variable u – in our case throttle – using a dynamic or static set point with respect to which it computes the error and attempts to make adjustments resulting in a return of the plant variable to the set point. The proportional term P provides a linear relationship to the error term, along with a coefficient K_p. This coefficient is a tuning parameter for our controller. The I term computes an accumulated error, along with coefficient K_i for tuning. And finally the D term computes the change of the control error with respect to time, multiplied by K_d for tuning.

PID control has been successfully implemented in the autonomous driving setting [5], as well as in a traditional driving system [7]. We attempt to build a version of this which can plug in to any control pipeline, from fully autonomous to our simple example of braking control.

In our initial testing, we implement a straightforward version of this controller based on the basic structure presented by Goodwin [6]. Here is the pseudo code for our initial implementation.

```
def get_control(self, measurement):
        error = setPoint - measurement  # calculate error for proportion
        curr_time = time # uptdate time
        dt = curr_time - prev_time

        prop_term = self.Kp * error
        derivative_term = (error - self.last_error) / dt * self.Kd
        int_term += error * self.Ki * dt

        u_out = prop_term + self.int_term + self.derivative_term

        last_error = error  # reset error
        prev_time = time
return u_out
```

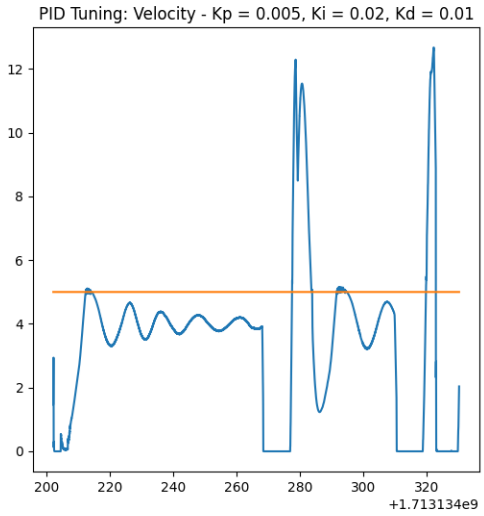
The input of this code is the current speed measurement and the output of this code is given directly to the throttle control. We found this to be sufficient for straight line speed maintenance with minimal parameter tuning.

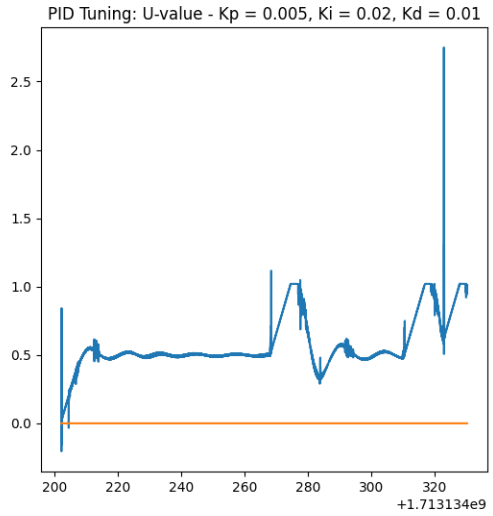
After implementing some basic path following in our planning code, we implemented a linear turning control. This is done by computing the heading error of our car with respect to the next waypoint and adjusting the steering linearly. We found this to be adequate for the turning required in our Carla maps.

Upon turning, we noticed a tendency for the car to take turns too wide, or to lose control during turning. To remedy this, we adjusted our static-set-point car speed to lower the speed set point during turning. Lowering the set point allowed our car to take turns smoothly, and our PID control was able to adjust back and forth between set points.

Finally we introduce other vehicles, as well as another component to our car control: object detection. The Carla python API provides sensors for detecting objects in front of the vehicle at a certain range. We wish to control the vehicle through waypoints like before, only this time through the variable behavior of the surrounding traffic.

Initially we set the collision detection to 5 meters in front of the vehicle, and created a linear brake controller to stop the vehicle upon a detected object. We applied sever tuning techniques including manually applying the Ziegler Nichols method [4]. We found that, while the vehicle

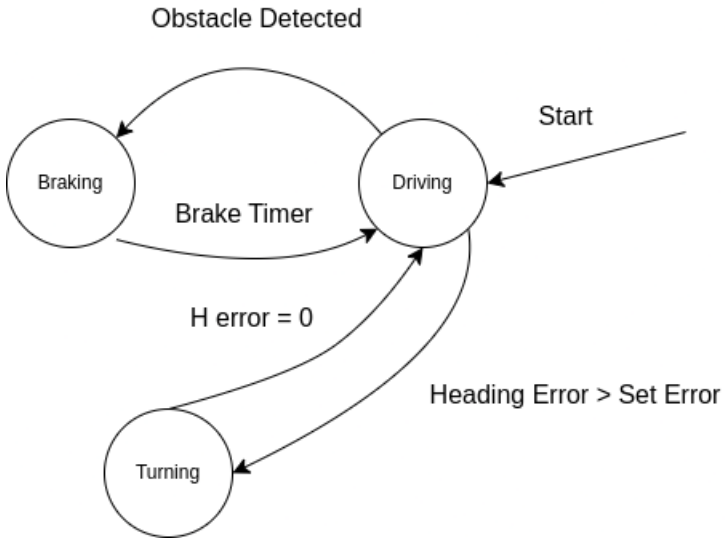




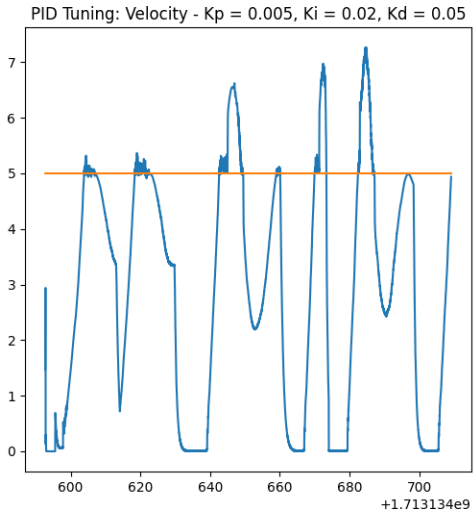
is in the 'braking' state, our integral term was still accumulating error. This resulted in very high overcompensation speeds after braking, which can be seen in the following figures. The yellow line indicates the driving set point on the velocity graph.

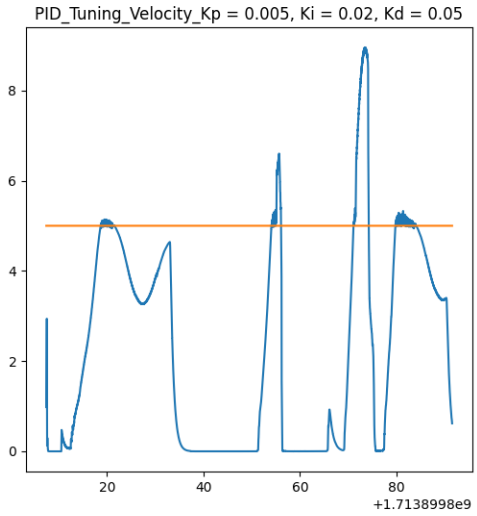
We can see a period of braking with 0 speed, followed by a drastic overshoot in velocity. On the right we can see how the U-value also sharply increases at these points. Even after clamping the U value to within the actuating range, we still see plateaus where our U-value is maximized at 1. To solve this, we used a state variable for the car under braking, resulting in a 3 state system of turning, braking, and continuing. When the car is in the in the braking state, the accumulation of error is prevented and the integral term is reset.

To achieve a continuation of throttle control, we place a timer on our obstacle variable. If the obstacle timer runs for a second and no obstacle is detected in that time, we switch to the driving state. An example of this state transition is seen here. Transitions between braking and turning are omitted since they are identical to those between turning and driving.

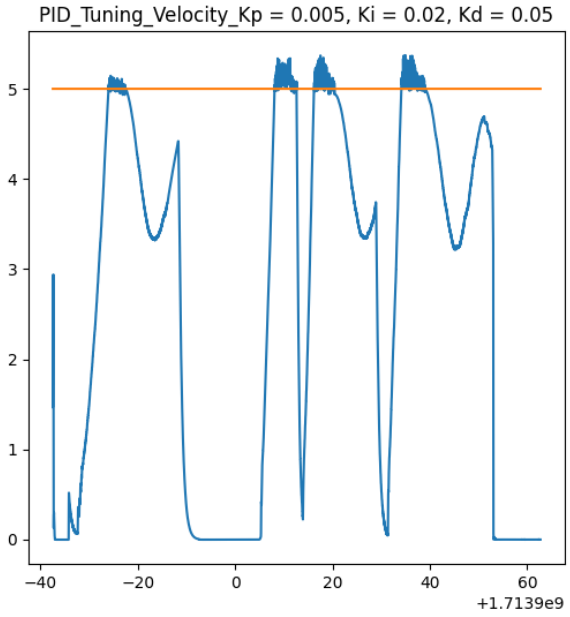


After implementing this, our controller results look much better for braking events. We show the results from two runs with identical parameters. We see some but much less overshoot, as well as relatively quick stabilization around our turning set point of 4.





The first graph is the result of running our simulation for several minutes, showing us that our setup can operate independently for a large amount of time. To address the still extant overshoot, we identified integral windup as the cause, and implemented integral clamping [6]. This is simply the practice of preventing the integral term from increasing once our actuator is saturated. We think that the actuator becomes saturated when trying to regain the set point after braking. Here are the final results.



We see very little overshoot and small oscillations around the turning set point of 4.

Planning

Several methods have been used for route planning in autonomous driving, as surveyed by Ming [8]. A graph search involves representing a map using a topological graph, with edges

representing contiguous driving points in the map. The search takes a point in the graph A and attempts to find a route from this point to another specified point B [8]. In our project we use a graph based planning algorithm, using A and B as the same point – though preventing the selection of the null path. Our algorithm starts relative to the spawn position selected by the user, and then connects adjacent waypoints via directed edges, ultimately exploring the map and creating a graph with loops.

Our planning algorithm then traverses the graph, using any technique we like. We have set up this class to implement any graph traversal algorithm the user desires. Though due to the simplicity of our maps, we opted to find a loop in the graph and then iterate over the waypoints in that path, repeating our path once we find ourselves back at the start.

We found that this setup allowed us to tune our algorithm very well, enabling us to iterate almost 100 times on our system while tuning our parameters.

Results/Conclusion

In this project we were able to successfully implement PID control over the throttle of our vehicle, as well as to implement basic path planning and object collision avoidance. We found a deepened understanding of some elements of a modular control pipeline as we set out to do. We were able to provide a smooth and crash free ride without intervention for more than 10 minutes in an active simulated environment, as well as to provide a control system which is stable under perturbations and whose plant variable is maintained within a certain envelope – avoiding dangerous inputs and uneccesary collisions.

In the future we would like to see the ability of our system to be incorporated into larger pipelines successfully.

References

[1] URL: https://github.com/carla-simulator/carla/tree/master/PythonAPI.

[2] URL: https://carla.readthedocs.io/en/latest/python_api/.

[3] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[4] George Ellis. "Chapter 6 - Four Types of Controllers". In: *Control System Design Guide (Fourth Edition)*. Ed. by George Ellis. Fourth Edition. Boston: Butterworth-Heinemann, 2012, pp. 97–119. ISBN: 978-0-12-385920-4. DOI: https://doi.org/10.1016/B978-0-12-385920-4.00006-0. URL: https://www.sciencedirect.com/science/article/pii/B9780123859204000060.

[5] Mumin Tolga Emirler et al. "Robust PID Steering Control in Parameter Space for Highly Automated Driving". In: *International Journal of Vehicular Technology* 2014 (Feb. 2014), pp. 1–8. DOI: 10.1155/2014/259465.

[6] G.C. Goodwin, S.F. Graebe, and M.E. Salgado. *Control System Design*. Prentice Hall, 2001. ISBN: 9780139586538. URL: https://books.google.com/books?id=7dNSAAAAMAAJ.

[7] Shugang Jiang. 2009. URL: https://www.aandd.jp/support/dsp_papers/2009-01-0370.pdf.

[8] Yu Ming et al. "A Survey of Path Planning Algorithms for Autonomous Vehicles". In: *SAE International Journal of Commercial Vehicles* 14 (Jan. 2021). DOI: 10.4271/02-14-01-0007.