Control and Trajectory Tracking for Autonomous Vehicles

# Introduction

The goal in this project is to design a PID controller to perform vehicle trajectory tracking. The trajectory will be given as an array of locations. Carla simulator will be used for testing the controller.

# Project Steps

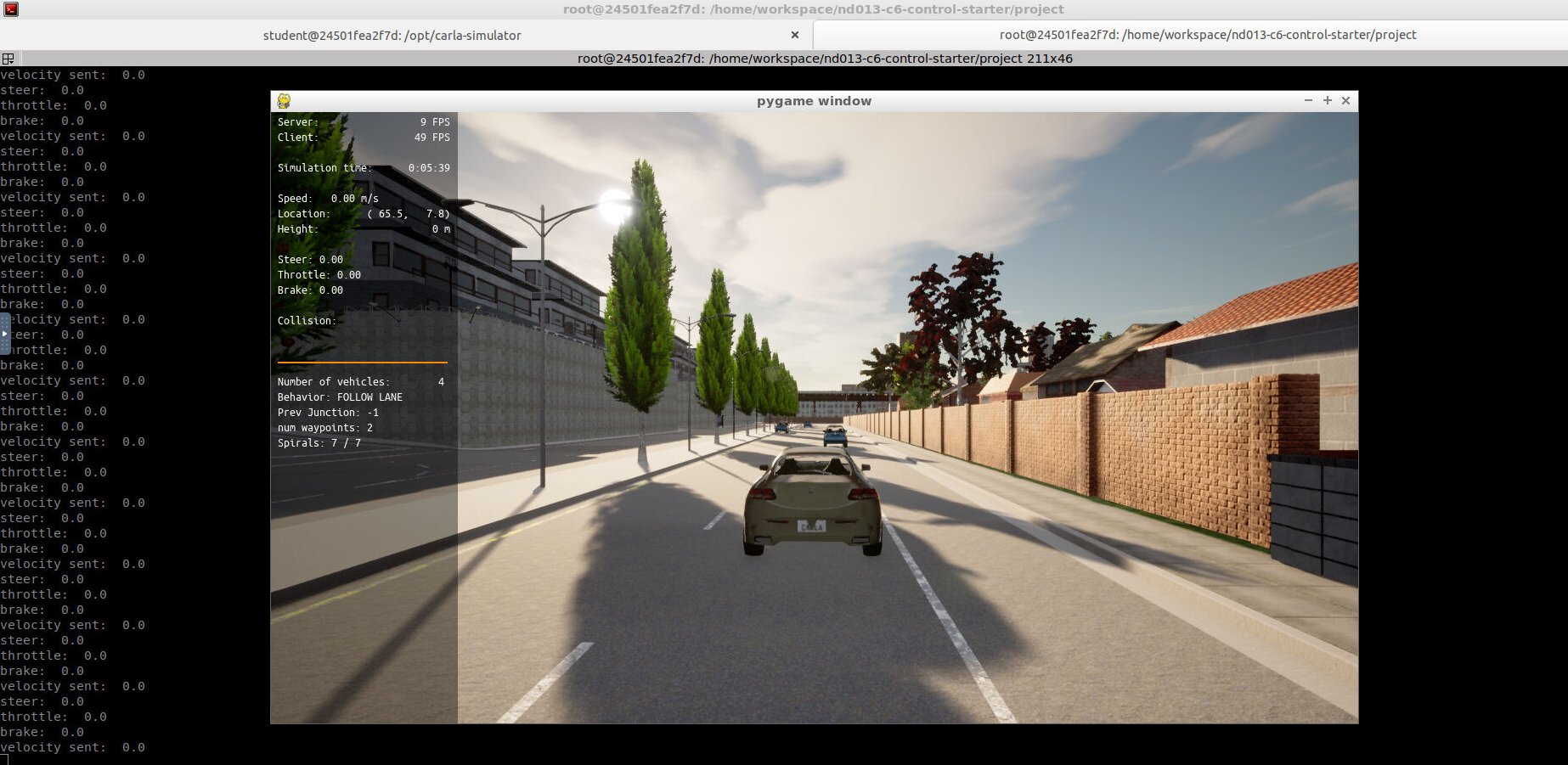
## Step 1: Build PID controller object

In this step, the PID controller was built using the knowledge from the lesson before. Coefficients for proportional, integral and differential elements were defined in the pid\_conroller.h file. Along with that their respective error component were defined. Output limits and delta time were also defined here.

After that we move to pid\_controller.cpp to complete the code for the rest of the controller. We retrieve the values of the coefficients and control limits from the function initialization. We then update the errors. Save previous error and then update the proportional, integral and differential error components.

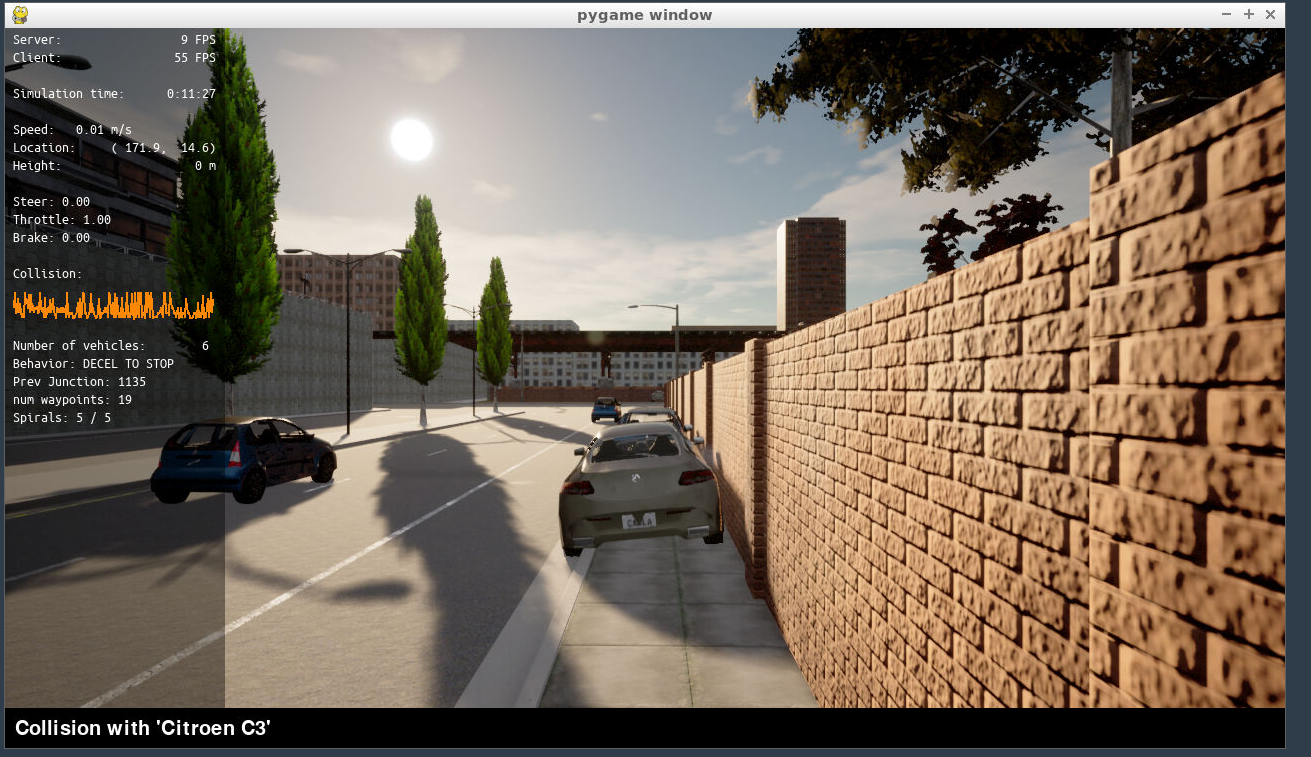
Once that is done the total error is calculated keeping in mind the delta time should be greater than zero. The total error is essentially the control input from the PID controller. The control input is then checked against the output limits and set to output limit if needed.

Next task to do here is to try and run the simulator. It is observed that the simulator window opens but the vehicle does not move. This is because the main function is not updated to send information to the controller. Output from Carla after controller was implemented is below.



## Step 2: PID controller for throttle

The main.cpp script was updated to perform the PID control for throttle and brake inputs. The error model used for this was a simple difference between the current velocity of the vehicle and desired vehicle of the vehicle from the planner point closest to the vehicles current position. A tuning cycle was attempted with just throttle controller and it became clear quickly that a simultaneous tuning of steering and throttle required in order to prevent the vehicle from colliding with objects in the environment. Below is a screen capture from a run where only throttle control was performed and the vehicle kept colliding with the objects in the environment.



## Step 3: PID Controller for Steer

The main.cpp script was updated to perform the PID control for steering. There were two techniques used to model the error here:

1. Yaw error
2. Yaw error combined with position error

When the first tuning cycle was started with just Yaw error after a few attempts it became clear that just yaw was not enough as the vehicle keep getting large values of steering input to keep correcting the yaw error. In order to stabilize the error and the controller another component was added. That was position error. The position error help reduce unnecessary steering inputs that would make the vehicle yaw back and forth when it was not needed.

## Step 4: Tuning and Evaluation of PID efficiency

Most of the initial tunes led to the vehicle colliding with the object in the environment. Another thing to keep in mind before we dive into the details of tuning is that two error models were used for steering controller. That is always mentioned alongside the trial numbers. Below you can see the first tuning trial that led to no collision and parameters for the controller.

pid\_steer.Init(Kp, Kd, Ki, output max, output min); pid\_throttle.Init(Kp, Kd, Ki, output max, output min);

### Trial 1: First result with no collision with just Yaw error

|  |  |
| --- | --- |
|  |  |
| pid\_steer.Init(0.29, 0.001, 0.7, 1.2, -1.2); | pid\_throttle.Init(0.2, 0.001, 0.01, 1.0, -1.0); |

From the steering plot above it is observed that the vehicle is in constant state of correction which can be attributed to the vehicle reaching the location it wants but the yaw value has an error with leads to controller having to correct often and with large values to keep the vehicle on track and stopping it from colliding.

The throttle input for the most part is smooth, but it does show some jerky behavior towards the end which was induced by an object on the road. So, improvement to the steering controller tuning should help with this scenario.

Next domain to explore steering tuning would be to reduce the proportional coefficient to reduce the overreaction to error that leads to solid line crossings and vehicle hitting the curbs. For throttle tunning, Integral component could be increased to reduce the static error as seen in the plot.

### Trial 2: Improvement on Trial 1 with just Yaw error

|  |  |
| --- | --- |
|  |  |
| pid\_steer.Init(0.2, 0.001, 0.8, 1.2, -1.2); | pid\_throttle.Init(0.2, 0.001, 0.02, 1.0, -1.0); |

In this tuning trial the vehicle does well on the steering controller for the most part but still has a flaw where it is not able to correct itself after passing the last object vehicle and that leads to large values of oscillatory behavior which could be attributed to the fact that the error is only driven by yaw. So, the next step is to improve the error model by adding a position component to yaw.

The throttle controller does an adequate job for the most part by reducing the static error towards the end and coming to a stop as smoothly as possible with a PID controller. This tuning will be carried to the next trial.

### Trial 3: Improvement on Trial 2 using Yaw and Position error

|  |  |
| --- | --- |
|  |  |
| pid\_steer.Init(0.2, 0.0011, 0.8, 1.2, -1.2); | pid\_throttle.Init(0.2, 0.001, 0.02, 1.0, -1.0); |

A significant improvement in the tuning can be seen from the steering controller when position component is added to the yaw error. This led to the vehicle not overreacting every time is needs to pass an object and a correction is required. This is evident in how Trial 3 ends the run as opposed to Trial 2. Throttle tune was kept the same as it showed satisfactory behavior for a PID controller. The improvement to steering controller did help throttle controller as well. This can be seen in how the stop behavior is controlled more smoothly in Trial 3 as opposed to Trial 2.

## Questions to be Answered in Report

1. Add plots to your report and explain them

For this please refer to section 2.4. Plots from multiple trails have been shown and their behavior explained.

1. What is the effect of the PID according to the plots, how each part of the PID affects the control command?

The plots help us understand how the coefficients on proportional, integral, and differential component impact the controller. If the proportional component is too high, then that lead to high levels to overreaction to error and overshoot as notice from the Trial 1 to Trail 2 in section 2.4 for steering controller tuning. It can’t be kept too low either as the controller would then have low response to a high error provided.

If coefficient for the integral term is too large, then the controller overshoots the target as the error is built up over time. If its set too low, then the controller would have steady state error. This behavior can be seen from the throttle controller tunning in all Trials with small improvements from 1 to 2 and 3.

Coefficient for the differential term could be used to reduce oscillatory behavior. If it’s too large, then its take longer for the controller to settle to the target point. If its too small, then then controller has high overshoots and oscillations. This behavior can be seen when going from Trial 2 to Trail 3 for steering controller. A small increase in Kd values was needed to reduce the oscillations alongside change to the error model.

1. How would you design a way to automatically tune the PID parameters?

An interesting way to approach this problem would be looking at it like an optimization issue. So, we could build a cost function around the error and use Kp, Ki, and Kd as tuning weights. Thus, the output from that will be the optimal weights for a given scenario. As the optimizer reduces the cost function, it reduces the errors and gives us tuned Kp, Ki, and Kd.

1. PID controller is a model free controller, i.e. it does not use a model of the car. Could you explain the pros and cons of this type of controller?

Pros:

1. Model free controllers don’t need to know the system model that its controlling. This can be very helpful when working with complex systems to model like Traction control and Anti-lock braking systems.
2. Model free controllers are ideal for real time implementation.
3. Model free controllers are widely used in the industry as they are easy to understand and implement.

Cons:

1. Model free controller don’t take into account the model and as such always not knowing what could be the potential impact of the control input it sends.
2. Model free controller are difficult to tune as they are independent of the system that it is trying to control.
3. Tuning for a model free control like PID is usually fixed to certain scenarios and needs to be modified if the scenario changes significantly.
4. Stability of the PID controller is highly dependent on its tuning.
5. What would you do to improve PID controller?

An interesting thing to explore with PID controllers would be more complex error functions. Error function with multiple components with each having their own weight. This might help the final tune to be more robust to noise and different initial states of the vehicle.