Control and Trajectory Tracking

Setup of Code

The URL of all the code and associated document (Including this one) is:

https://github.com/HarisAshraf/Project_6

The repository includes the following Files:

main.cpp

pid_controller.cpp

pid_controller.h

steer.png

throttle.png

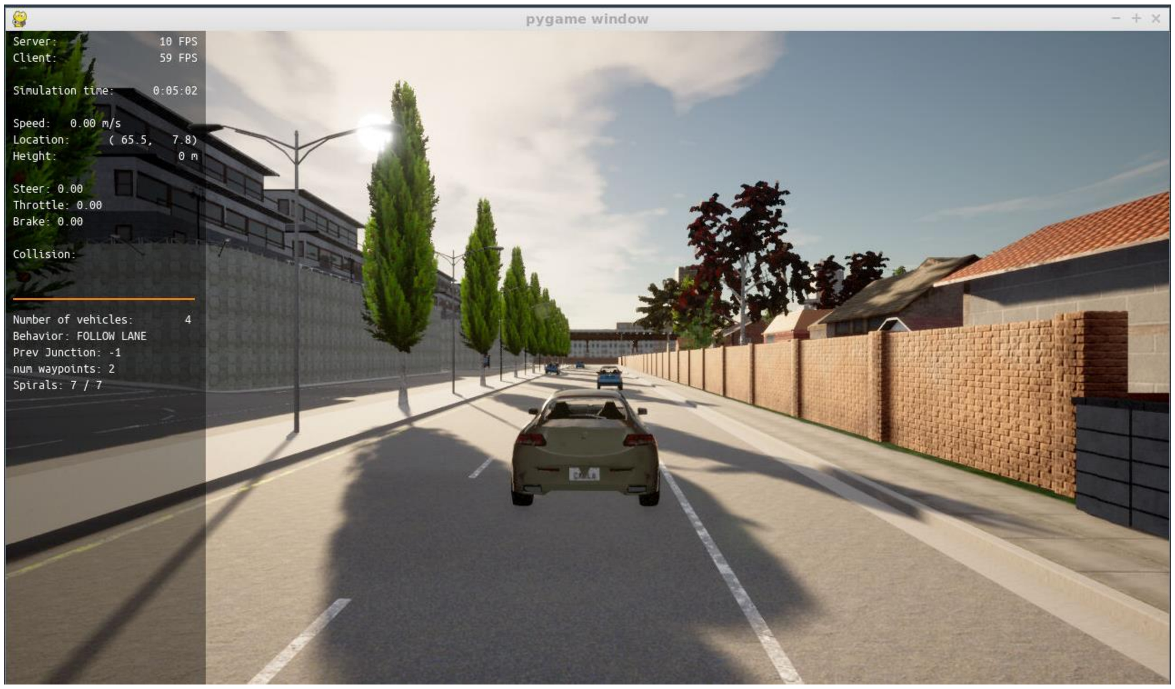
The analysis will follow the following steps (It is assumed that classroom workspace is provided)

Step1: Build the PID controller object

Files pid_controller .cpp and pid_controller.h were modified to include a header and and a pid object. A separate initialization routine was used to set the controller parameters.

Although not part of the instructions, a windup protection on the integral error was also provided.

As the controller objects were not hooked up in the main function, the car did not move if the code was executed. This is shown in *Figure 1*.



```
Server:                 10 FPS
Client:                 59 FPS

Simulation time:       0:05:02

Speed:   0.00 m/s
Location:       ( 65.5,   7.8)
Height:                    0 m

Steer: 0.00
Throttle: 0.00
Brake: 0.00

Collision:

Number of vehicles:          4
Behavior: FOLLOW LANE
Prev Junction: -1
num waypoints: 2
Spirals: 7 / 7
```

Figure 1: Stopped Car

Step2: PID Controller for Throttle

The vector **v_points**, contained the velocity trajectory. The last point on the vector was used as the setpoint to the PID controller.

After quite a few iterations the loop was tuned and its performance is shown in *Figure 2*. Also, the braking output was reduced to half as it was found that the braking was too action was too aggressive. The output was limited to [-1,+1]

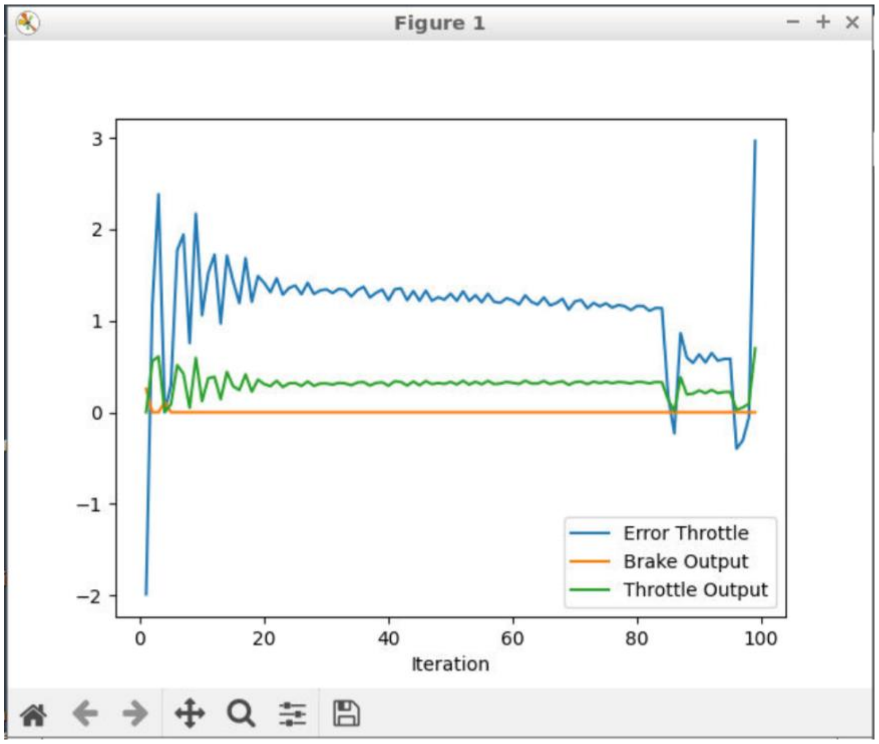


Figure 2: Braking and Throttle Performance

The Gains of the PID Throttle controller are as follows:

K_p = 0.2

K_i = 0.0009

K_d = 0.1

Step 3: PID Controller for Steer

This was the most challenging part of the project as the desired yaw computation was problematic. It was found that some of the points in the trajectory were sometimes behind the car itself. It was chosen to use the last point in the trajectory as the setpoint to the steer controller. However, the best performance achieved was still very oscillatory and mor tuning will be needed to bring it to the acceptable performance. Figure 2 two shows how the Steer controller performed.

The output was limited to [-1.2,1.2]

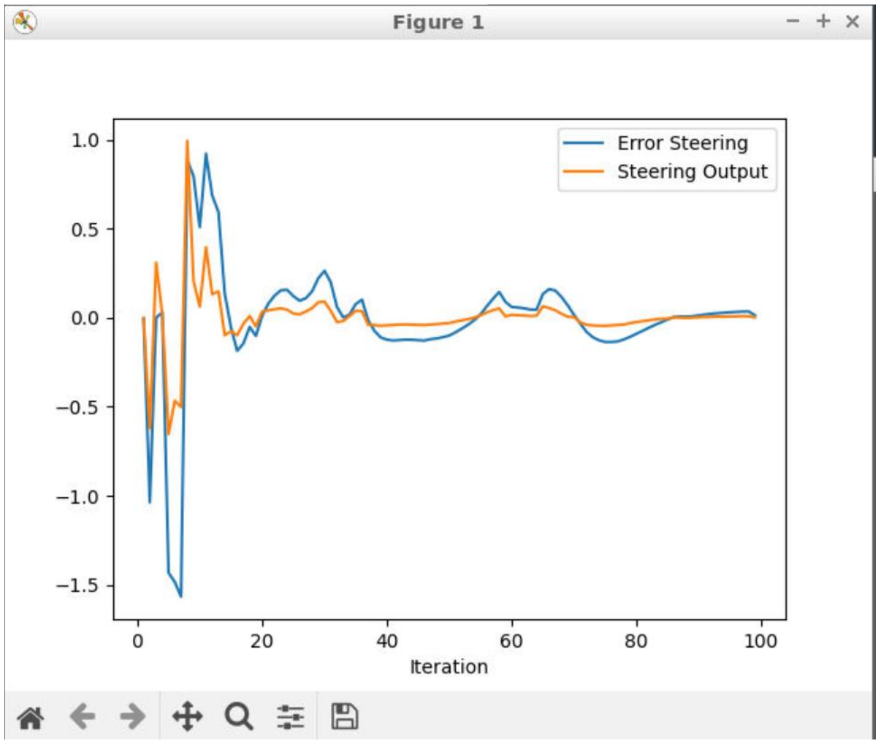


Figure 3: Steer Performance

The Gains of the PID Steer controller are as follows:

K_p = 0.3

K_i = 0.001

K_d = 0.3

Step 4: Conclusion

The throttle controller shows a steady state error. This is due to the physics of the car. Some increased K_i would decrease the error but was found to cause instability in the loop. Too much Integral gin caused te car to always overshoot.

The increase in proportional gain, K_p, increased performance to a point and then made the oscillations grow as time passed. Low gain caused the control to be very slow and the ego car always collided with the first parked car.

The increase of K_d, initially damped the oscillations and then caused instability in the system.

To tune the loop, the algorithm needs to minimize the mean square error as time goes by. Twiddle is one algorithm that does such a function. However, initial values should still be close enough to the final optimized values.

As shown by the plots, al low speeds low gain may be beneficial so a simple tuner algorithm will just change the gain I proportional to 1/velocity.

PID is a model free controller hence it is simpler to implement. The model based controller depends upon how good the model , which is not always possible especially when the conditions are changing.

In order to improve the controller, a simple inverse velocity gain scheduler will help.