PID Control Project Reflections

Project Objective

Implement a PID controller in C++ to maneuver the vehicle around the lake race track.

Principle:

PID controllers are found in a wide range of applications for industrial process control. PID stands for Proportional-Integral-Derivative. These three controllers are combined in such a way that it produces a control signal.

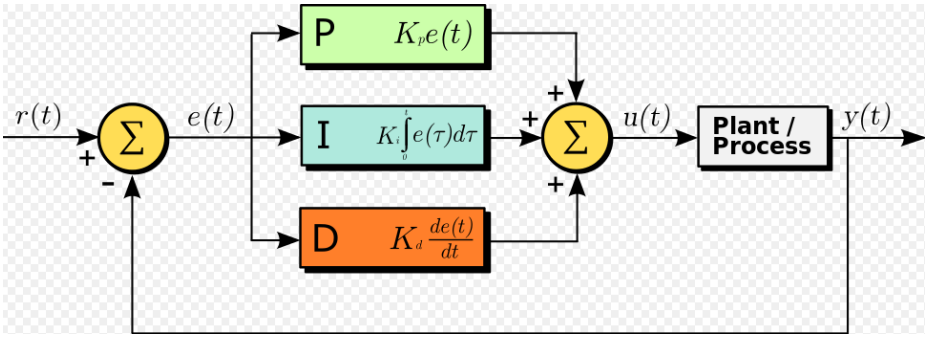

Figure 1 PID Control (*Image Reference: https://en.wikipedia.org/wiki/PID_controller*)

Proportional or P- controller gives output which is proportional to current error e (t). It compares desired or set point with actual value or feedback process value. The resulting error is multiplied with proportional constant to get the output. If the error value is zero, then this controller output is zero. Due to limitation of p-controller where there always exists an offset between the process variable and set point, **I-controller** is needed, which provides necessary action to eliminate the steady state error. It integrates the error over a period of time until error value reaches to zero. It holds the value to final control device at which error becomes zero. I-controller doesn't have the capability to predict the future behavior of error. So it reacts normally once the set point is changed. **D-controller** overcomes this problem by anticipating future behavior of the error. Its output

depends on rate of change of error with respect to time, multiplied by derivative constant. It gives the kick start for the output thereby increasing system response.

Tuning Method:

The tuning method to find the best parameters for the PID controller is illustrated below.

```
def twiddle(tol=0.2):
    p = [0, 0, 0]
    dp = [1, 1, 1]
    robot = make_robot()
    x_trajectory, y_trajectory, best_err = run(robot, p)

    it = 0
    while sum(dp) > tol:
        print("Iteration {}, best error = {}".format(it, best_err))
        for i in range(len(p)):
            p[i] += dp[i]
            robot = make_robot()
            x_trajectory, y_trajectory, err = run(robot, p)

            if err < best_err:
                best_err = err
                dp[i] *= 1.1
            else:
                p[i] -= 2 * dp[i]
                robot = make_robot()
                x_trajectory, y_trajectory, err = run(robot, p)

                if err < best_err:
                    best_err = err
                    dp[i] *= 1.1
                else:
                    p[i] += dp[i]
                    dp[i] *= 0.9
        it += 1
    return p
```

Figure 2 Twiddle Method

In this project, I used two different cost functions to tune parameters. The first cost function is to measure the sum squared cross track errors for a given time steps. The second cost function is to measure the maximum speed and the duration of that high speed. For the first cost function, the training method is to tune parameters to minimize the sum squared cross track errors for a

given time steps, while for the second one, the training method is to tune parameters to reach high speed and stay that high speed as long as possible.

Experiments:

- Cost Function 1

  To train the first cost function, I used the initial parameters [Kp = 0.1, Kd = 3.0, Ki = 0] and the first 550 time steps. The training feedback information **CTE: 0.579, Count: 50 [0.1,3,0] 24.1262** means the cross track error is 0.579 and the sum squared CTE from the beginning is 24.1262 at time step 50 with the PID parameters [Kp = 0.1, Kd = 3.0, Ki = 0]. After each run, the program will displayed the previous sum squared CTE such as 0 and the current sum squared CTE such as 331.841 and the adjustment array DPS such as [1,1,1].

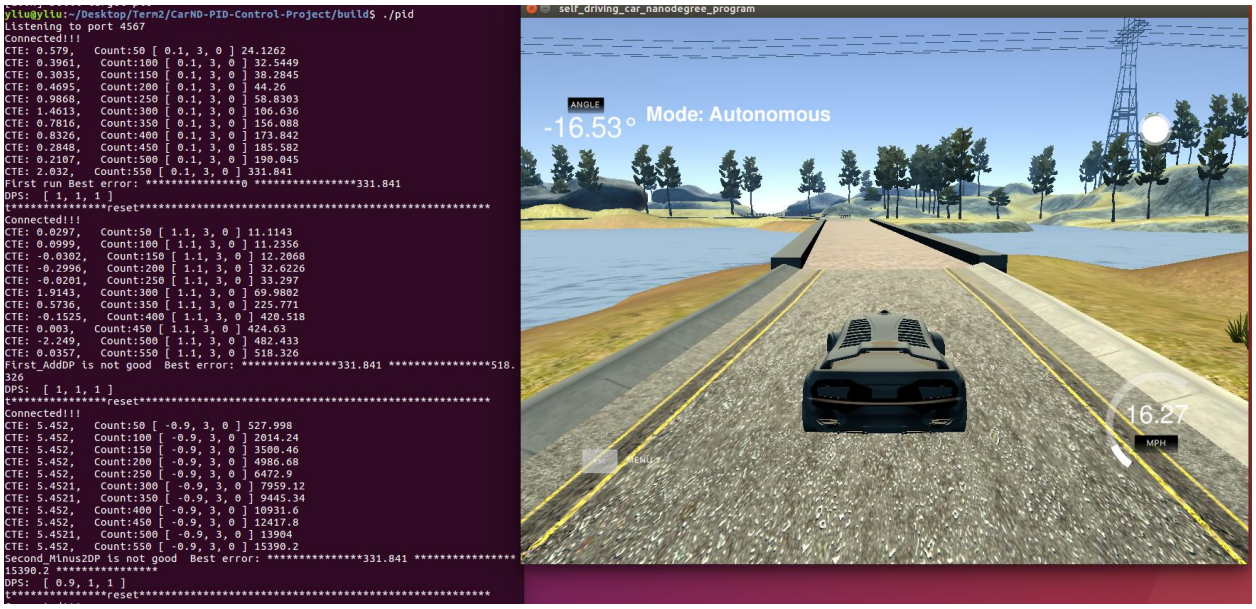

Figure 3 Beginning of the CTE training

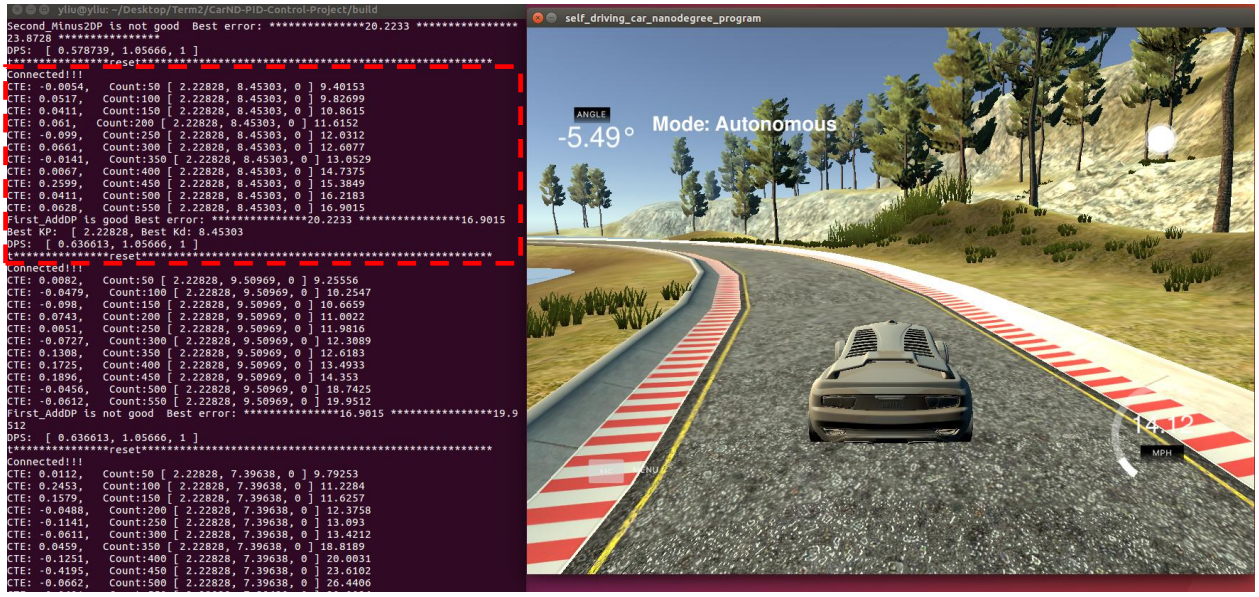

Figure 4 Good PID Parameters from 2 hours of Training

The above figure shows the good PID parameters got from 2 hours of training. They reduced the sum squared CTE to 16.9015 for 550 time steps of driving which is greater than 300 at the beginning of the training.

- Cost Function 2

  To train the second cost function, I used the initial parameters [Kp = 0.1, Kd = 3.0, Ki = 0] and the first 550 time steps.

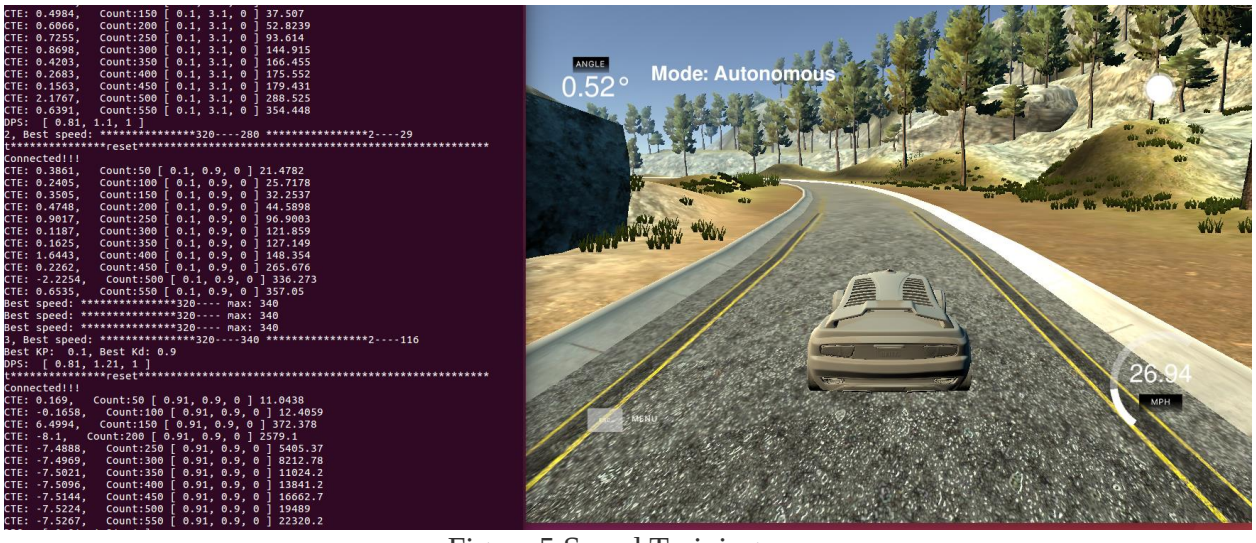

Figure 5 Speed Training

Most training feedback information are the same as the CTE training except that it will display the maximum speed and how many time steps in that speed after each run. The information 320, 280, 2, 29 means the maximum speed is 32 MPH and 2 time steps stay in that speed in previous runs, the maximum speed and the number of time steps stay in that speed for the current run are 28 MPH and 29.

Results:

In order to save time, I use the twiddle function to train only the Kp and Kd parameters and leave the Ki parameters to be zero all the time. I used the original settings for the throttle. After the training process, I got two sets of PID parameters. Parameters [Kp =2.22828, Kd = 8.45303, Ki = 0.0] is able to achieve the smallest sum squared CTE during the training. Parameter [Kp =0.1, Kd = 0.623038, Ki = 0.0] is able to keep vehicle at high speed for most of time steps while still drive safely. In order to get the smallest sum squared CTE, vehicle adjusted its angle very frequently to stay around the center line most of the time. Due to the frequent angle adjustment, the speed of the vehicle is always less than 20 MPH. In order to get high speed, vehicle has to drive very smoothly with less angle adjustment.

I used those two sets of parameters to control the vehicle to run a loop and had the following results.

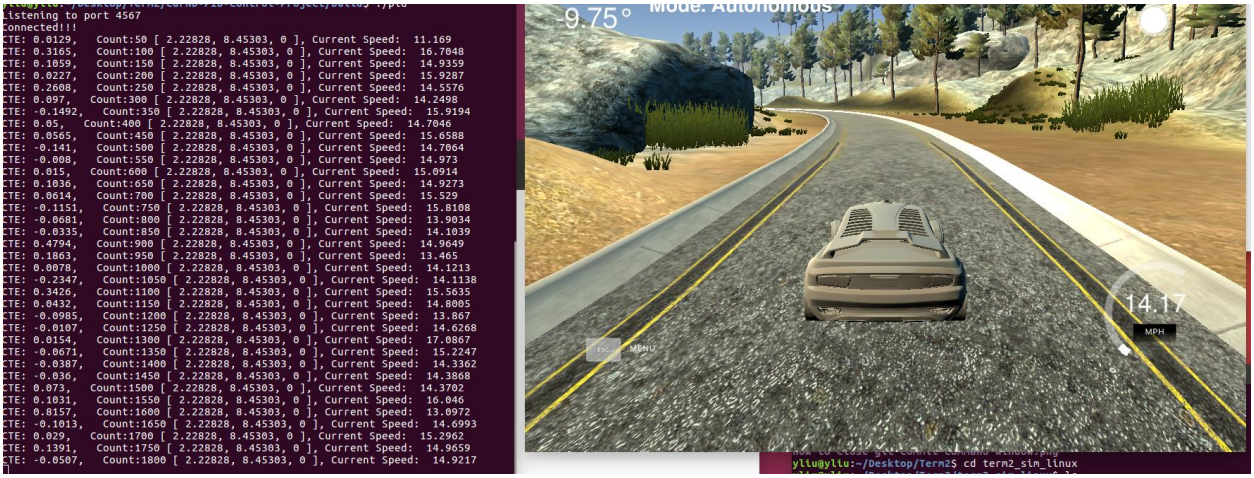

Figure 6 One Loop of Driving With Smallest CTE Parameters

Figure 6 One Loop of Driving With Fastest Speed Parameters

The vehicle finished one loop around 800 time steps by using the fastest speed parameters while the vehicle stayed around the middle line most of time by using the smallest CTE parameters.