Describe the effect each of the P, I, D components had in your implementation.

The P, or "proportional", impacts the car behavior the most. It causes the car to steer proportional to CTE - if the car is far to the right it steers hard to the left, if it's slightly to the left it steers slightly to the right.

The D, or "differential", component counteracts the P component and prevents overshooting the center line.

A properly tuned D parameter will cause the car to approach the center line smoothly without ringing.

The I, or "integral", component counteracts a bias in the CTE. This bias can take several forms, such as a steering drift and particularly serves to reduce the CTE around curves.

These are the impact I felt the components impacts the system.

Describe how the final hyper parameters were chosen.

I used Twiddle logic taught in the classroom to come up with the values of the hyper parameters. The Hyper parameter in arrived are P-0.13,I-0.0001,D-2.0

The code block I have implemented is attached to the files. Also the code is as below

**********TWIDDLE IMPLEMENTED in MAIN.CPP********

```
int count=0;
//Param vector
std::vector<double> p{1,0,0};
//Param change factor
std::vector<double> dp{1,1,1};

int flag=0;
int flag_c=0;
int flag_p=0;
float pre_error=1.0;
int stop=0;

int twiddle(PID &pid)
{
  if((dp[0]+dp[1]+dp[2])<0.2)
  {
          return 1;
  }

  if(pid.t_error<=pre_error and flag_c==0)
  {
          dp[flag]*=1.1;
    if(flag==0)
          {
```

```
          flag=1;
        }
        else if(flag==1)
        {
          flag=2;
        }
        else if(flag==2)
        {
          flag=0;
        }
        pre_error=pid.t_error;
        p[flag]+=dp[flag];
}
else if(flag_p==0)
{
        flag_p=1;
        flag_c=1;
        p[flag]-=2*dp[flag];
        pid.Init(p[0],p[1],p[2]);
  return 0;
}

if(pid.t_error<=pre_error and flag_p==1)
{
        pre_error=pid.t_error;
        dp[flag]*=1.1;
        flag_c=0;
        flag_p=0;
        if(flag==0)
        {
          flag=1;
        }
        else if(flag==1)
        {
          flag=2;
        }
        else if(flag==2)
        {
          flag=0;
        }
        p[flag]+=dp[flag];
}

else if(flag_p==1)
{
        p[flag]+=dp[flag];
        dp[flag]*=0.9;
        flag_c=0;
```

```
          flag_p=0;
          if(flag==0)
          {
            flag=1;
          }
          else if(flag==1)
          {
            flag=2;
          }
          else if(flag==2)
          {
            flag=0;
          }
          p[flag]+=dp[flag];
  }

  pid.Init(p[0],p[1],p[2]);
  return 0;
}



 int values(double a, double b,double c){


  uWS::Hub h;
  PID pid;
  pid.Init(a,b,c);


  h.onMessage([&pid](uWS::WebSocket<uWS::SERVER> ws, char *data, size_t
length, uWS::OpCode opCode) {
    // "42" at the start of the message means there's a websocket message event.
    // The 4 signifies a websocket message
    // The 2 signifies a websocket event


        count+=1;
    if (length && length > 2 && data[0] == '4' && data[1] == '2')
    {
      auto s = hasData(std::string(data).substr(0, length));
      if (s != "") {
        auto j = json::parse(s);
        std::string event = j[0].get<std::string>();
        if (event == "telemetry") {
          // j[1] is the data JSON object
          double cte = std::stod(j[1]["cte"].get<std::string>());
          double speed = std::stod(j[1]["speed"].get<std::string>());
          double angle = std::stod(j[1]["steering_angle"].get<std::string>());
          double steer_value;
          /*
          * TODO: Calcuate steering value here, remember the steering value is
          * [-1, 1].
        * NOTE: Feel free to play around with the throttle and speed. Maybe use
        * another PID controller to control the speed!
        */
        pid.UpdateError(cte);
                steer_value=pid.TotalError();
                //std::cout<<count<<std::endl;


                if(count==500)
                {
                        pid.t_error=std::abs(pid.i_error)/count;
                        count=0;
                        std::cout<<pid.Kp<<' '<<pid.Ki<<' '<<pid.Kd<<'
'<<pid.t_error<<std::endl;
                        stop=twiddle(pid);
                        if(stop==1)
                        {
                          ws.close();
                        }
                        std::cout<<pid.Kp<<' '<<pid.Ki<<' '<<pid.Kd<<'
'<<pre_error<<std::endl;
                        std::string msg("42[\"reset\",{}]");
                        ws.send(msg.data(),msg.length(), uWS::OpCode::TEXT);


                }
```

```
          // DEBUG
          //std::cout << "CTE: " << cte << " Steering Value: " << steer_value <<
std::endl;

          json msgJson;
          msgJson["steering_angle"] = steer_value;
          msgJson["throttle"] = 0.1;
          auto msg = "42[\"steer\"," + msgJson.dump() + "]";
          //std::cout << msg << std::endl;
          ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
        }
      } else {
        // Manual driving
        std::string msg = "42[\"manual\",{}]";
        ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
      }
    }
  });

  h.onHttpRequest([](uWS::HttpResponse *res, uWS::HttpRequest req, char *data,
size_t, size_t) {
    const std::string s = "<h1>Hello world!</h1>";
    if (req.getUrl().valueLength == 1)
    {
```

```
    res->end(s.data(), s.length());
   }
   else
   {
    // i guess this should be done more gracefully?
    res->end(nullptr, 0);
   }
  });

  h.onConnection([&h](uWS::WebSocket<uWS::SERVER> ws, uWS::HttpRequest
req) {
   std::cout << "Connected!!!" << std::endl;
  });

  h.onDisconnection([&h](uWS::WebSocket<uWS::SERVER> ws, int code, char
*message, size_t length) {
   ws.close();
   std::cout << "Disconnected" << std::endl;
  });

  int port = 4567;
  if (h.listen(port))
  {
   std::cout << "Listening to port " << port << std::endl;
  }
  else
  {
    std::cerr << "Failed to listen to port" << std::endl;
    return -1;
  }
  h.run();
  }



int main()
{
  values(1,0,0);
}
```

********END OF TWIDDLE********