To run the project three C++ files have been updated. The files are main.cpp, pid_controller.cpp and pid_controller.h

pid_controller.cpp
In this class four main functions have been developed. The first function was

```
void PID::Init(double Kpi, double Kii, double Kdi, double output_lim_maxi,
double output_lim_mini)
```

In this function, we define the main variables for PID controller which are proportional, integral and derivative initial gains along with the minimum and maximum values for control action.

```
  kp = Kpi;
  ki = Kii;
  kd = Kdi;
  minLim = output_lim_mini;
  maxLim = output_lim_maxi;
  p_error = 0.0;
  i_error = 0.0;
  d_error = 0.0;
```

The next function is:

```
void PID::UpdateError(double cte)
```

In this function, we define the control actions calculated by proportional, integral and derivative components and add them together. The input to the function is error.
We have descritised the control signals by dt.

```
if(dt>0){d_error = (cte – p_error)/dt;} //sanity check to avoid division
by zero
  else{d_error = 0.0;}

  p_error = cte;
  i_error += cte*dt;
```

The third function is:

```
double PID::TotalError()
```

In this function, we calculate the total control action. We also check if the control action is within the minimum and maximum values. If not, either minimum or maximum value will be assigned to the control action.

```
double control; //control output AKA control signal
control = (kp*p_error + kd*d_error + ki*i_error);

if (control < minLim) {
  control = minLim;
}
else if (control > maxLim){
  control = maxLim;
}
return control;
```

}

The last function is

```
double PID::UpdateDeltaTime(double new_delta_time) {
    /**
    * TODO: Update the delta time with new value
    */
      time_step = new_delta_time;

       return time_step;
```

In this function, we update the time step at each iteration.
Pid_controller.h

In this file we define the PID class and its constructors:

```
class PID {
public:

  /**
   * TODO: Create the PID class
   **/
    /*
    * Errors
    */
     double p_error, i_error, d_error;
    /*
    * Coefficients
    */
     double kp, ki, kd;
    /*
    * Output limits
    */
    double minLim, maxLim;
    /*
    * Delta time
    */
     double dt;
    /*
    * Constructor
    */
    PID();
    /*
    * Destructor.
    */
    virtual ~PID();
    /*
    * Initialize PID.
    */
    void Init(double Kp, double Ki, double Kd, double output_lim_max,
double output_lim_min);
    /*
    * Update the PID error variables given cross track error.
    */
    void UpdateError(double cte, bool debugMode = false);
```

```
    /*
    * Calculate the total PID error.
    */
    double TotalError();

    /*
    * Update the delta time.
    */
    double UpdateDeltaTime(double new_delta_time);
};
```

Main.cpp:

The first modification in this file was initialisation of PID controller for steering and throttle commands. To initialise the PID controllers, the initial values for Kp, Ki and Kd as well as the maximum and minimum of control actions were given as the input of the functions.

```
// initialize pid steer
  /**
  * TODO (Step 1): create pid (pid_steer) for steer command and initialize values
  **/
  PID pid_steer = PID();
  pid_steer.Init(0.5, 0.005, 0.5, 1.2, -1.2);


  // initialize pid throttle
  /**
  * TODO (Step 1): create pid (pid_throttle) for throttle command and initialize values
  **/
  PID pid_throttle = PID();
  pid_throttle.Init(0.5, 0.005, 0.5, 1.0, -1.0);
```

The next modification was the implementation of PID controller to adjust steering command. Here, first we update the change in time step and then calculate the steering error. Here, I have used the error in heading angle instead of steering angle as both heading angle and steering angle are indirectly related. Then, I used pid_steer.UpdateError function to calculate the control action. Using pid_steer.TotalError when saturate control actions between the minimum and maximum values of action for safety reasons.

```
  ////////////////////////////////////
          // Steering control
          ////////////////////////////////////
          pid_steer.UpdateDeltaTime(new_delta_time);


          double error_steer;

          double steer_output;

          double yaw_new = angle_between_points(x_points[x_points.size()-
2], y_points[y_points.size()-2], x_points[x_points.size()-1],
y_points[y_points.size()-1]);
```

```
            error_steer = yaw_new - yaw;


            pid_steer.UpdateError(error_steer);
            steer_output = pid_steer.TotalError();
```

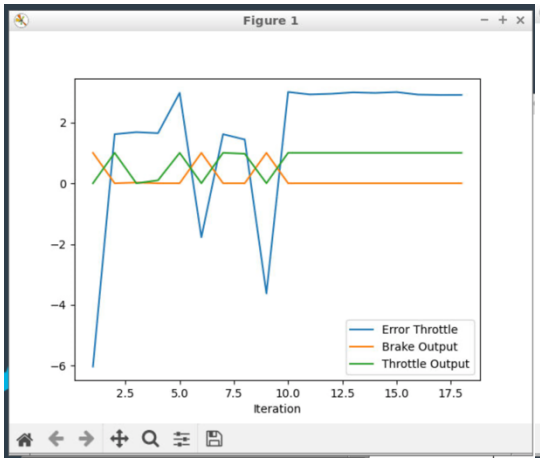
I followed the same thermology to calculate control action to control the throttle.

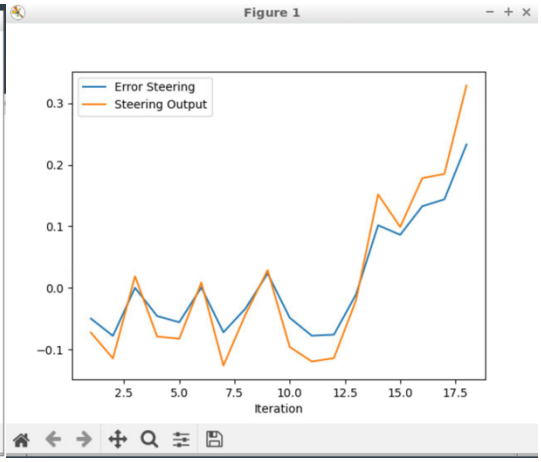
```
// Throttle control
            ////////////////////////////////////

            /**
            * TODO (step 2): uncomment these lines
            **/
            // Update the delta time with the previous command
            pid_throttle.UpdateDeltaTime(new_delta_time);

            // Compute error of speed
            double error_throttle;
            /**
            * TODO (step 2): compute the throttle error (error_throttle) from the position and the
desired speed
            **/
            // modify the following line for step 2
            error_throttle = v_points.back() - velocity;


            double throttle_output;
            double brake_output;

            /**
            * TODO (step 2): uncomment these lines
            **/
            // Compute control to apply
            pid_throttle.UpdateError(error_throttle);
            double throttle = pid_throttle.TotalError();
```

Results:

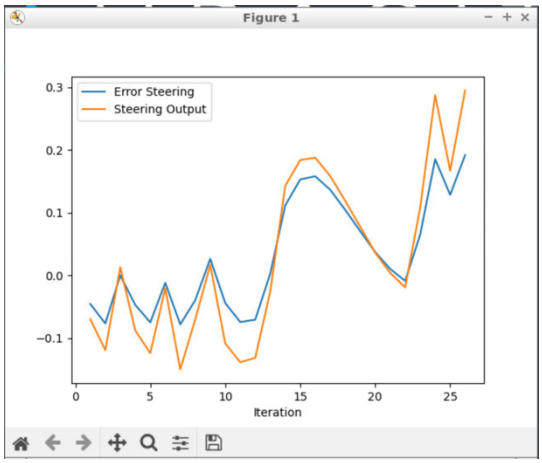
The figures below show the results for different sets of gains. As the figures show, the steering controller is not able to direct error to zero and throttle control has a steady state error. The performance can get better with further tuning of PID gains. However, it is not an easy task and needs lot of trial and errors. To better understand the effects of the gains on control performance, we assumed the gains for pid_throttle.Init(0.5, 0.1, 0.3, 1.0, -1.0) are fixed and we just changed the gains for pid_steer. Case 1 represents the bench mark gains. Case 2 and case 3 represents the effect of increasing and decreasing the gain of integral controller. Case 4 and case 5 represents the effect of increasing gain of differential part. It seems increasing the gain of differential part improves the overall controller performance and change in integral gain effects on tracking performance.

Case 1:
pid_steer.Init(1.3, 0.02, 0.3, 1.2, -1.2)

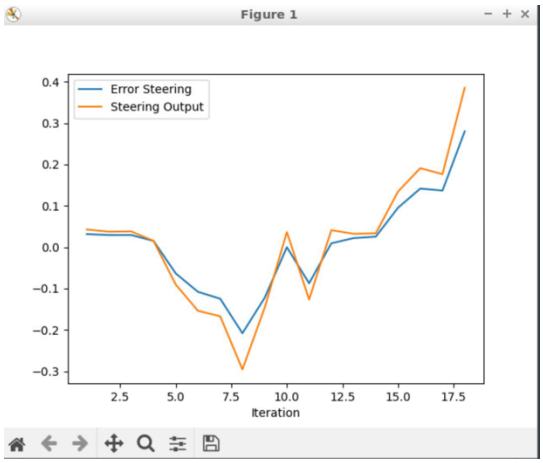




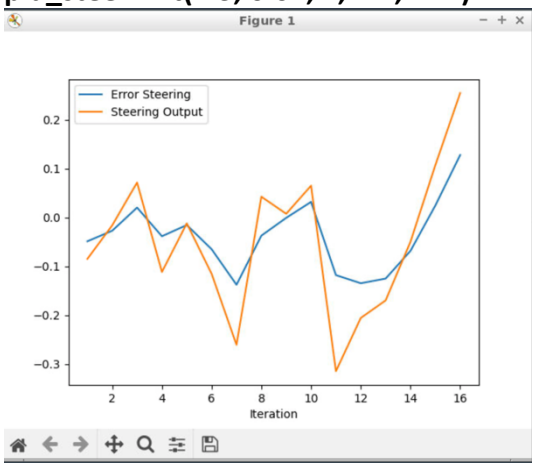
Case 2:
pid_steer.Init(1.3, 0.05, 0.3, 1.2, -1.2);



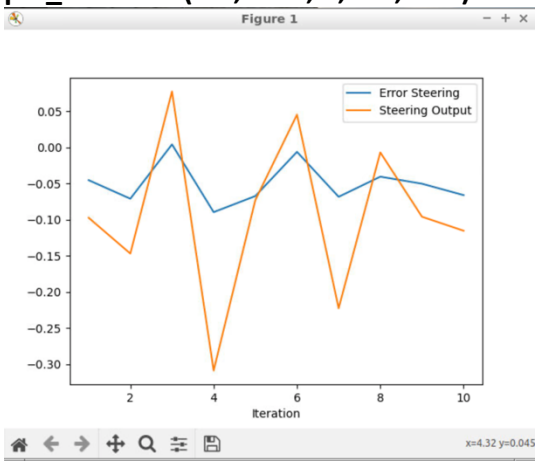
Case 3:
pid_steer.Init(1.3, 0, 0.3, 1.2, -1.2);



Case 4:
pid_steer.Init(1.3, 0.02, 1, 1.2, -1.2)



Case 4:
pid_steer.Init(1.3, 0.02, 2, 1.2, -1.2)



Answer the following questions:

1- Add the plots to your report and explain them (describe what you see)

The file just plots the steering output. For some reasons the reference trajectory is zero and the error is high. I tried to find the source of the problem but I couldn't find it. I don't know why the file does not plot the other errors.

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

Proportional controller directs the error towards zero but the large values may unstablise the system. Differential controller improve the stability of the system and integral controller helps to remove the steady state error as explained in the lecture video.

3- How would you design a way to automatically tune the PID parameters?

The most interesting approach is to use reinforcement learning technique to update automatically the gains of PID controllers real-time according to driving conditions. Also, it is

possible to use classical control techniques such as adaptive controllers to automatically tune the PID gains.

4- 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?

The model free control systems such as PID controller make the controller robust against model uncertainties and parameter variations but tunning the gains is a challenge. Another example of model free controllers is reinforcement learning controllers such as value-based techniques and actor-critic techniques.

5- (Optional) What would you do to improve the PID controller?

Certainly, automatic real-time tuning of PID gains is a great improvements. But other techniques such is integral wind-up can be used to improve the error accumulation due to the integral part of PID controller.