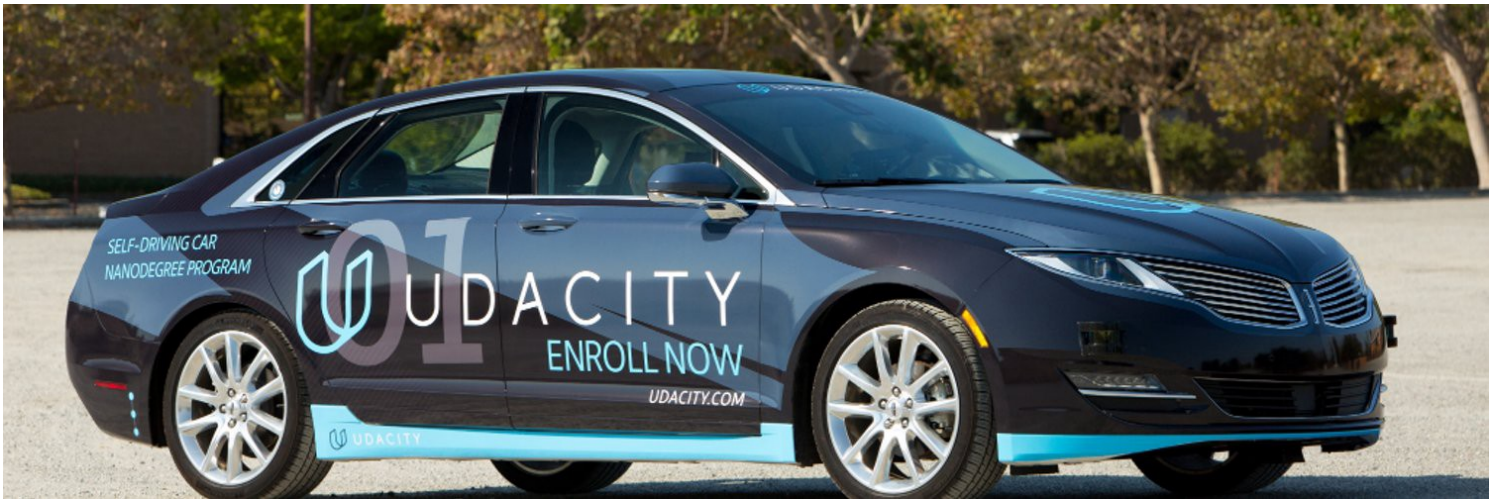
PID Controller-Udacity's Self-driving Car Nanodegree

Dhanoop Karunakaran Follow
Mar 24, 2018 · 7 min read



This article discuss about the PID controller implementation in C++ as part of the Udacity's nanodegree program.

Introduction

PID stands for Proportional-Integral-Derivative. These three components are combined in such a way that it produces a control signal. Before getting into the implementation, let's discuss the PID controller components. I am going to discuss based on how to use PID for steering angles.

Cross Track Error

A cross track error is a distance of the vehicle from trajectory. In theory it's best suited to control the car by steering in proportion to Cross Track Error(CTE).

P component

It sets the steering angle in proportion to CTE with a proportional factor tau.

```
-tau * cte
```

In other words, the P, or "proportional", component had the most directly observable effect on the car's behaviour. It causes the car to steer proportional (and opposite) to the car's distance from the lane center(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.

D component

It's the differential component of the controller which helps to take temporal derivative of error. This means when the car turned enough to reduce the error, it will help not to overshoot through the x axis.

In other words, the D, or "differential", component counteracts the P component's tendency to ring and overshoot the center line. A properly tuned D parameter will cause the car to approach the center line smoothly without ringing.

```
diff_cte = cte - prev_cte
prev_cte = cte
- tau_d * diff_cte
```

I component

It's the integral or sum of error to deal with systematic biases.

In other words, the I, or "integral", component counteracts a bias in the CTE which prevents the P-D controller from reaching the center line. This bias can take several forms, such as a steering drift , but I believe that in this particular implementation the I component particularly serves to reduce the CTE around curves.

```
int_cte += cte
tau_i * int_cte
```

And combination of these we can get PID controller to control the steering value.

```
cte = robot.y
diff_cte = cte - prev_cte
prev_cte = cte
int_cte += cte
steer = -tau_p * cte - tau_d * diff_cte - tau_i * int_cte
```

Parameter optimisation can be done manually or using Twiddle algorithm.

Pseudo code for implementing the Twiddle algorithm is as follows:

```
function(tol=0.2) {
    p = [0, 0, 0]
    dp = [1, 1, 1]
    best_error = move_robot()
    loop untill sum(dp) > tol
        loop until the length of p using i
            p[i] += dp[i]
            error = move_robot()

            if err < best_err
                best_err = err
                dp[i] *= 1.1
            else
                p[i] -= 2 * dp[i]
                error = move_robot()

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

PID controller implementation

Udacity provides a simulator to do the project and the aim is to write the PID controller to calculate the steering value to control the car using Cross Track Error(CTE) provided by simulator.

This is very basic controller in control theory and using proportional, integral, and derivative components using CTE to find out the control signal.

Pseudo code to find out the steering value based PID controller algorithm as follows:

```
cte = robot.y
diff_cte = cte - prev_cte
prev_cte = cte
int_cte += cte
steer = -tau_p * cte - tau_d * diff_cte - tau_i * int_cte
```

PID project code to find out steering value based on the CTE:

```
void PID::Init(double Kp, double Ki, double Kd) {
    this->Kp = Kp;
    this->Ki = Ki;
    this->Kd = Kd;
    this->p_error = 0.0;
    this->i_error = 0.0;
    this->d_error = 0.0;
}
```

```
void PID::UpdateError(double cte) {
    d_error = cte - p_error;
    p_error = cte;
    i_error += cte;
}
```

```
double PID::TotalError() {
    return (-Kp * p_error) - (Ki * i_error) - (Kd *
d_error);
}
```

```
void PID::Twiddle() {


}
```

> **cte** *is the* **cross track error** *fed into the controller from a sensor.*

> **p_error***, **i_error**, and **d_error** are* proportional*,* integral *and* derivative *components' errors respectively.*

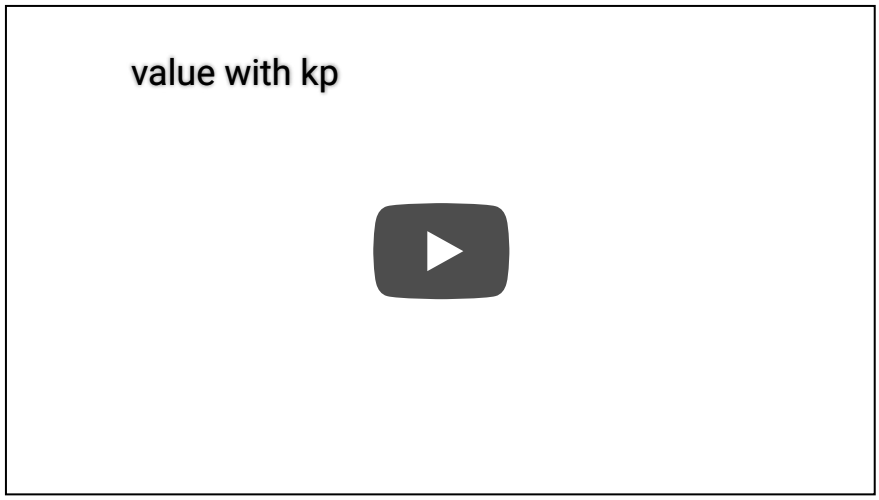
> **Kp***, **Ki,** and **Kd** are those **P, I, and D** parameters to be optimized.*

Finding initial value for Kp, Ki, Kd

The intial value for Kp, Ki, Kd selected using trail and error method. It is a simple method of PID controller tuning. In this method, first we have to set Ki and Kd values to zero and increase proportional term (Kp) until system reaches to oscillating behaviour. Then Kd was tuned to reduced oscillation and then Ki to reduce steady-state error.

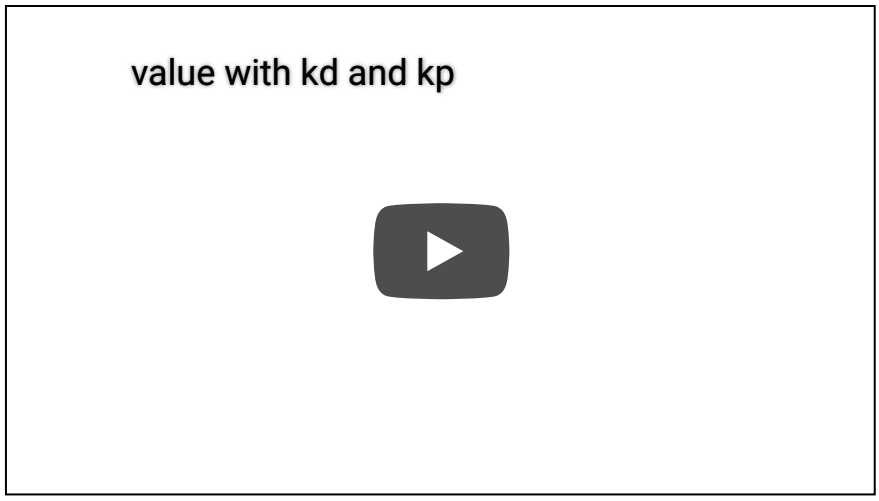
Car started to oscillate when KP value set to 0.05 whereas Ki and Kd set to zero.

Here is the video with oscillating behaviour:



Then found the Kd value that stops the oscillating behaviour which is set to 1.5 along with 0.05 for Kp and zero for Ki.

Here is the video after value for Kp and Kd added:



Finally value Ki set as 0.0001 to reduce the steady-state error.

Initial value for Kp, Ki, and Kd set as below:

```
{0.05, 0.0001, 1.5}
```

Twiddle

Once we set the initial value, Twiddle algorithm is used to optimise the parameters.

I modified the main.cpp to implement Twiddle algorithm. When twiddle variable set to true, simulator runs the car with confidents till the maximum steps set initially and go through the twiddle algorithm. After competition of each iteration, simulator reset to initial stage and car runs starts from the beginning to maximum steps. This process continuous until tol value below the allowed value.

Optimised value we got as below:

```
0.06, 0.00031, 1.29
```

Once I found the optimised value, set the twiddle variable to false to run the car through the simulator.

```
bool twiddle = false;
```

Actual twiddle algorithm in python as follows:

```
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
```

And I had to modify it for this project and code as follows:

```
if (twiddle == true){
    total_cte = total_cte + pow(cte,2);
    if(n==0){

      pid.Init(p[0],p[1],p[2]);
    }
    //Steering value
```

```
    pid.UpdateError(cte);
    steer_value = pid.TotalError();


    // DEBUG
    //std::cout << "CTE: " << cte << " Steering Value: " <<
steer_value << " Throttle Value: " << throttle_value << "
Count: " << n << std::endl;
    n = n+1;
    if (n > max_n){
```

```
      //double sump = p[0]+p[1]+p[2];
      //std::cout << "sump: " << sump << " ";
      if(first == true) {
        std::cout << "Intermediate p[0] p[1] p[2]: " << p[0]
<< " " << p[1] << " " << p[2] << " ";
        p[p_iterator] += dp[p_iterator];
        //pid.Init(p[0], p[1], p[2]);
        first = false;
      }else{
        error = total_cte/max_n;
```

```
        if(error < best_error && second == true) {
            best_error = error;
            best_p[0] = p[0];
            best_p[1] = p[1];
            best_p[2] = p[2];
            dp[p_iterator] *= 1.1;
            sub_move += 1;
            std::cout << "iteration: " << total_iterator <<
" ";
            std::cout << "p_iterator: " << p_iterator << "
";
            std::cout << "p[0] p[1] p[2]: " << p[0] << " "
<< p[1] << " " << p[2] << " ";
            std::cout << "error: " << error << " ";
            std::cout << "best_error: " << best_error << "
";
            std::cout << "Best p[0] p[1] p[2]: " <<
best_p[0] << " " << best_p[1] << " " << best_p[2] << " ";
        }else{
          //std::cout << "else: ";
          if(second == true) {
            std::cout << "Intermediate p[0] p[1] p[2]: " <<
p[0] << " " << p[1] << " " << p[2] << " ";
            p[p_iterator] -= 2 * dp[p_iterator];
            //pid.Init(p[0], p[1], p[2]);
            second = false;
          }else {
            std::cout << "iteration: " << total_iterator <<
" ";
            std::cout << "p_iterator: " << p_iterator << "
";
            std::cout << "p[0] p[1] p[2]: " << p[0] << " "
<< p[1] << " " << p[2] << " ";
            if(error < best_error) {
                best_error = error;
                best_p[0] = p[0];
```

```
                best_p[1] = p[1];
                best_p[2] = p[2];
                dp[p_iterator] *= 1.1;
                sub_move += 1;
            }else {
                p[p_iterator] += dp[p_iterator];
                dp[p_iterator] *= 0.9;
                sub_move += 1;
            }
            std::cout << "error: " << error << " ";
            std::cout << "best_error: " << best_error << "
";
            std::cout << "Best p[0] p[1] p[2]: " <<
best_p[0] << " " << best_p[1] << " " << best_p[2] << " ";
          }
        }
```

```
      }
```

```
      if(sub_move > 0) {
        p_iterator = p_iterator+1;
        first = true;
        second = true;
        sub_move = 0;
      }
      if(p_iterator == 3) {
        p_iterator = 0;
      }
      total_cte = 0.0;
      n = 0;
      total_iterator = total_iterator+1;


      double sumdp = dp[0]+dp[1]+dp[2];
      if(sumdp < tol) {
        //pid.Init(p[0], p[1], p[2]);
        std::cout << "Best p[0] p[1] p[2]: " << best_p[0] <<
best_p[1] << best_p[2] << " ";
        //ws.close();
        //std::cout << "Disconnected" << std::endl;
      } else {
        std::string reset_msg = "42[\"reset\",{}]";
        ws.send(reset_msg.data(), reset_msg.length(),
uWS::OpCode::TEXT);
      }
```

```
    } else {
      msgJson["steering_angle"] = steer_value;
      msgJson["throttle"] = throttle_value;
      auto msg = "42[\"steer\"," + msgJson.dump() + "]";
      ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
    }
```

```
  }
```

If you would like to see full code in action, visit my github repo.