PID Control

Fabrizio Frigeni, 2017.11.26

Specifications

The goal of this project is to design a PID controller to drive a car around a track.

The observed variables are current speed, steering angle, and cross-track error, i.e. the distance between the car and the center of the track.

The controlled variables are steering value and throttle value.

Note: although there is no required minimum speed, we tried to optimize the lap time at cost of driving smoothness. The result is more a fast but jerky race car behavior rather than a slow smooth city car drive. Nevertheless, the drive is safe, in the sense that at no time the vehicle hits the sides of the track or violates its limits.

Implementation

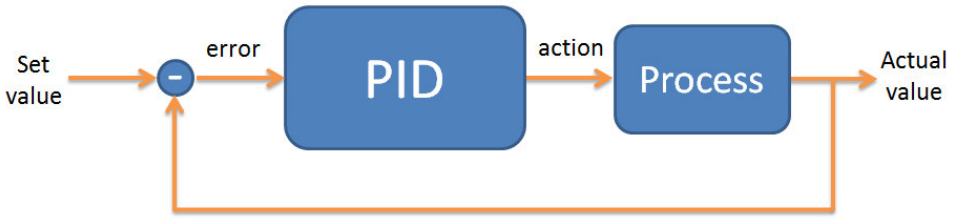
The controller is implemented in C++. All the project files are saved in the *src* folder.

The important files are:

- PID.cpp: this is where the core PID class is implemented.
- main.cpp: this is where the observed variables are read, the controller is initialized and called cyclically, and the output variables are sent to the simulator.

PID class

PID stands for Proportional, Integral and Derivative controller. Its basic functionality is to keep an observed value as close as possible to a given set value:



In our case we use two PID controllers, one to control the steering of the car, the other to control its speed.

By denoting the input error with *e* and the output action with *u*, the formula for the PID controller is:

$$u = K_p e + K_i \int e + K_d \dot{e}$$

The *proportional gain* K_p generates an action that increases with larger error values.

The *integral gain* K_i generates an action that increases when the error adds up over time.

The *derivative gain* K_d generates an action that increases if the error magnitude changes over time.

Main program

The main program is where the observed variables are read from the simulator and provided to the controllers. The output actions of the controllers are then sent back to the simulator.

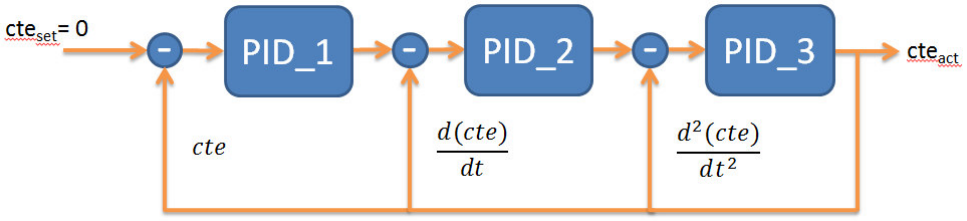
We implemented the following strategy to improve control behavior:

- a three-layer cascaded PID controller for steering
- a single PID controller for throttle
- anti-windup for integral action
- gain scheduling to recover quickly from extreme errors
- dead zone to avoid slowing down too often
- output action clamping

Cascaded controller for steering

A single PID controller is often hard to tune. Great improvements in stability and performance can be obtained by cascading several controllers in series. This is a common strategy when controlling motion applications, e.g. servo motors.

We implemented the following cascaded controller:



The outer loop is responsible for controlling the magnitude of the error: it should prevent the error from getting too large. It responds slower than all other loops and it should be tuned with a low proportional gain to avoid instability.

The middle loop is responsible for controlling the variation of the error over time. At steady state it should prevent the error from changing too quickly.

Finally, the inner loop is responsible for controlling the acceleration of the error over time: at steady state it should prevent the error's speed from changing too quickly. The inner loop has the fastest response and the highest gains.

We achieved good results with the following values:

	P	I	D
PID_1 (outer loop)	0.15	0.005	0.1
PID_2 (middle loop)	0.25	0.005	0.1
PID_3 (inner loop)	3	0.005	0.1

Integral anti-windup

The integration error of a PID controller can quickly reach very large values and saturate the output action with the risk of bringing the system into instability.

A common procedure to avoid a large integral action is to clamp it between given values. This strategy is called anti-windup:

```
//integral anti-windup
if (i_error > 0.5) i_error = 0.5;
if (i_error < -0.5) i_error = -0.5;
```

Gain scheduling

While running the simulator we noticed that sometimes the car was getting dangerously close to the sides of the track and the controller's action was not strong enough to quickly pull the car back towards the middle of the lane.

Instead of increasing the gains altogether, which would increase oscillations in the normal range of actions, we decided to treat the edges of the track as a special zone, with individualized controller gains.

This strategy is called gain scheduling:

```
//add gain scheduling for pid1
if (fabs(pid1.p_error) > 0.9) {
	pid1.gs = 1.5;
} else {
	pid1.gs = 1.0;
}
```

Note that gain scheduling only affects the proportional action of the controller:

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

Dead zone

The car speed controller takes the current steering angle as input and sets the throttle value. The idea is that we can accelerate at maximum throttle when we are driving on a straight line and slow down when we are turning.

However, to improve speed and performance, we decided to create a dead zone for the controller: for steering values lower than 25% of the maximum angle we assume that the car is driving on a straight line and we output maximum throttle anyway.

```
//input with dead zone
double pid_sp_in = -fabs(angle)/25.0;
if (fabs(pid_sp_in) < 0.25) pid_sp_in = 0;
pid_sp.UpdateError(pid_sp_in);
throttle_value = 1 - pid_sp.TotalError();
```

Action clamping

The output actions of both steering and throttle controllers are further manually adjusted before being sent to the simulator.

The steering action is clamped between -1 and 1:

```
//clamp steer_value between -1,1
if (steer_value > 1) steer_value = 1;
if (steer_value < -1) steer_value = -1;
```

The throttle value is first clamped between -0.1 and 1:

```
//clamp throttle_value between -0.1,1
if (throttle_value < -0.1) throttle_value = -0.1;
if (throttle_value > 1) throttle_value = 1;
```

Then we also decided to avoid braking at low speed (<40mph) to avoid wasting precious racing time:

```
//do not brake at low speed
if (speed < 40 && throttle_value < 0) throttle_value = 0;
```

Comments

We tuned the PID controllers to optimize lap time performance with the consequence that the car's movements are quite jerky, similar to a racing behavior.

Alternatively, one could reduce speed and tune the controllers softer, in order to achieve a smoother driving experience.

A major improvement in control quality could be achieved with the addition of a feed-forward action, which would depend on other observed variables not available for this project, for example the curvature of the road ahead of the car.