In this exercise we are asked to implement PID filter. PID filters work similar to a control feedback loop. Quoting wikipedia: A PID controller continuously calculates an *error value* e (t) , as the difference between a desired setpoint, and a measured [process variable and a measured process variable, which give the controller its name.
In practical terms it automatically applies accurate and responsive correction to a control function. An everyday example is the cruise control “Cruise control”) on a road vehicle; where external influences such as gradients would cause speed changes, and the driver has the ability to alter the desired set speed. The PID algorithm restores the actual speed to the desired speed in the optimum way, without delay or overshoot, by controlling the power output of the vehicle’s engine.
So how do we do it? we have three principal functions in this implementation: PID.cpp which contains the main code for the PID implementation. Twiddle does the optimization, and it contains information about what variable is being optimized, inside we have the iterator which increases the value or reduces the value of the variable in questions.
How are the parameters chosen? Well, we again return to revising the theoretical principles of the parameters. Again, quoting wikipedia:
"The proportional term produces an output value that is proportional to the current error value. The proportional response can be adjusted by multiplying the error by a constant _K_p, called the proportional gain constant.

The proportional term is given by
Pout=Kpe(t)
A high proportional gain results in a large change in the output for a given change in the error. If the proportional gain is too high, the system can become unstable In contrast, a small gain results in a small output response to a large input error, and a less responsive or less sensitive controller. If the proportional gain is too low, the control action may be too small when responding to system disturbances. Tuning theory and industrial practice indicate that the proportional term should contribute the bulk of the output change."
Basically if P gain is too big, we get oversteering, that is the car dangerously turns towards left or right, with a movement like a balancing boat. If P gain is too small, however, we don’t correct the dangerous turns we see with a high P an so, our car can veer towards the left excessively.
To see the effect of the integral term, we have the following:

The contribution from the integral term is proportional to both the magnitude of the error and the duration of the error. The “Integral” in a PID controller is the sum of the instantaneous error over time and gives the accumulated offset that should have been corrected previously. The accumulated error is then multiplied by the integral gain (_K_i) and added to the controller output.
So, in theory this should shape and correct the residual error. In our case, we don’t have an instantaneous error, so the effect of this component is very small, perhaps a little under steering with big gains in I.
Finally the derivative parameter works by smoothing out the movement of the car, as follows:
Derivative action predicts system behavior and thus improves settling time and stability of the system. implementations of PID controllers include an additional low-pass filtering for the derivative term to limit the high-frequency gain and noise.
In our systems, if D is too low, it fails to correct the balancing motion of the car, if D is too big, it exacerbates (doesnt correct) the balancing motion of the car.
How did you choose the value of the parameters P,I,D?

Sorry too say, but the three parameters were chosen based on those suggested in udacity forums. I plugged in the values suggested and I did run the simulation with this parameters. I found that they were effective enough to meet the rubric(the car doesn't move out of the track). I tried other values, but i found that some made the movement more stable but at some points sent the car towards the lake. I would have tried more possibilities but i am currently short of time.

Video output:

It does meet by a small margin the rubric of the exercise. Althought the movement is quite jerky, and sometime it moves towards the borders of the track, it regains its momentum winding towards the center(in the rubric it says it can cross the lines).

CODE ANALYSIS.

We will proceed to analyze in detail each of the three main files that we will use: PID.cpp, TWIDDLE.cpp, and main.cpp.

PID.cpp

contains the implementation of the PID filter.

In INIT we initialize the PID constructor, we use the following paramters:

p,i,d(error), previous constant value, and the sum of square errors.

Gains returns the vector gains which is composed of the PID constants we showed above.

UpdateError-updates the error by setting the p_error to the constant, the integral error adds the error from the preceding step and the derivative substracts current constant from previous constant.We also calculate the sum_squared_error which is the error squared by two, and increment the iteration.

TotalError returns the total error for the whole PID filter as follows (kp(0),ki(0),kd(0))*(p_error)+ (kp(1),ki(1),kd(1))*(i_error)+(kp(2),ki(2),kd(2))*(d_error).

PID_accumulatedSquaredError returns the square accumulated error, dividing the squared accumulated error by the current iteration.

Finally PID_Update Gain updates the current coefficients(in the gains matrix) by a delta.

Twiddle.cpp.

First we have the constructor. The constructor receives the delta we must update each value in the gain matrix. We set the initial gain deltas matrix, as vector, and resize gain_deltas vector to 3.

In twiddle reset we set gain_deltas values to those of the initial delta value. We also set the current gain_index, set the best error to -1, and set current case to 0.

In MoveToNextGainIndex, we jump to next current gain, if however this value exceeds the size of gain_deltas we set it to 0.

In Twiddle iterate we set best-error to accumulated squared error is this error is below zero(it's initialization).

And then we switch on current case,we update gains, with pid_UpdateGains with gain deltas(this only is done if we havent done it before, with current_case 0), we then set current_case to 0.

The following two cases do the following: if the error we have found is less than the best error(then we are understeering) and so we must update gain deltas by multiplying by 1.1. We set current_case to 0 and jump to the next gain index.

If however the error is more, we reduce the gain deltas by 2.0 points and jump to case 2. If best error is less than the current error, we multiply by 1.1(we make deltas higher) if not we make deltas lower by multiplying by 0.9.

We end by jumping to next gain index, and setting current case to 0.

Finally iterate returns only one if the gain_delta vector is more than 0.001, if less then we dont iterate and it returns 0.

Twiddle GainDeltas returns GainDeltas,
MAIN.CPP.
We will cover the modifications i have implemented so that it works with twiddle. Main modifications we have implemented are as follows:
PID pid_steering;
These values were suggested from the forums:
pid_steering.Init(0.0837465, 2.0899e-06, 1.44656);
Twiddle pid_steering_twiddle(pid_steering, 0.0109888, 1.0943e-07, 0.110536);

PID pid_throttling;
pid_throttling.Init(0.2, 0.0, 3.0);
Additionally i have tried different values by setting the following to true and running twiddle:
bool twiddle_in_progress = true;
I have settled on those values, which give a solution that fits with the requirements.
The steering error is updated with pid_steering as follows:
pid_steering_UpdateError(cte).
The speed_error is updated with
pid_throttling.UpdateError(speed_error);
double throttle_value = pid_throttling.TotalError() * (1.0 / (1.0 + fabs(angle)));
The key inner workings of the algorithm are here:
```
 if (twiddle_in_progress) {
if (pid_steering.Iteration() > 2000) {
twiddle_in_progress = pid_steering_twiddle.Iterate();
auto gains = pid_steering.Gains();
auto gains_deltas = pid_steering_twiddle.GainDeltas();
}
```
What is happening here? we are iterating with twiddle 2000 times, if at any moment sum of gains is below 0.001, the optimizing process has finished. And so, we jump to pid_steering_twiddle.Reset();
NOTE:
Due to time shortages, i managed to fill in most of the PID code myself. However, the twiddle function was pretty tough so i had a lot of outside help(github repositories), help from tutor,etc. However by extending myself on the explanation, as shown in the readme, i show that i do understand most of the code i've submitted, even if it comes from udacity forums, udacity tutors, github repositories.

> Written with StackEdit.