**def** **run**(robot, tau, n=100, speed=1.0):

x\_trajectory = []

y\_trajectory = []

**for** i **in** range(n):

cte = robot.y

steer = -tau \* cte

robot.move(steer, speed)

x\_trajectory.append(robot.x)

y\_trajectory.append(robot.y)

**return** x\_trajectory, y\_trajectory

The cross track error, cte is the current y position of the robot (our reference is a horizontal line) along the x-axis. To get the steering value we multiply the tau parameter with the cte. We then call the move method which causes the robot to move based on the steer and speed values. Add the x and y coordinates to the respective lists and then return them at the end.

**def** **run**(robot, tau\_p, tau\_d, n=100, speed=1.0):

x\_trajectory = []

y\_trajectory = []

prev\_cte = robot.y

**for** i **in** range(n):

cte = robot.y

diff\_cte = cte - prev\_cte

prev\_cte = cte

steer = -tau\_p \* cte - tau\_d \* diff\_cte

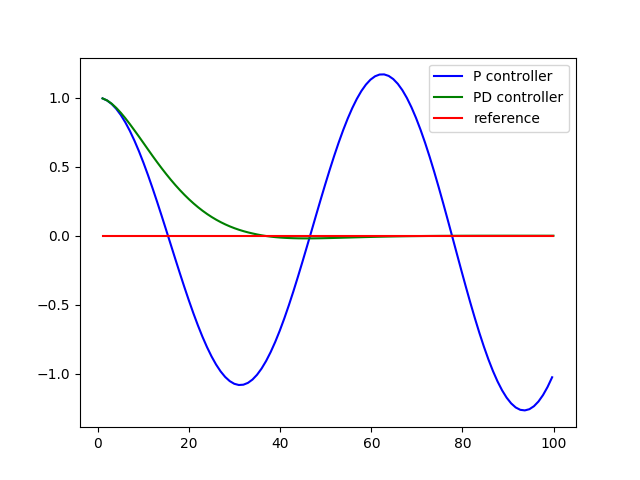
robot.move(steer, speed)

x\_trajectory.append(robot.x)

y\_trajectory.append(robot.y)

**return** x\_trajectory, y\_trajectory

This is very similar to the P controller. We've added the prev\_cte variable which is assigned to the previous CTE and diff\_cte, the difference between the current CTE and previous CTE. We then put it all together with the new tau\_d parameter to calculate the new steering value, -tau\_p \* cte - tau\_d \* diff\_cte.



As we can see from the above image the PD controller performs much better!

**def** **run**(robot, tau\_p, tau\_d, tau\_i, n=100, speed=1.0):

x\_trajectory = []

y\_trajectory = []

prev\_cte = robot.y

int\_cte = 0

**for** i **in** range(n):

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

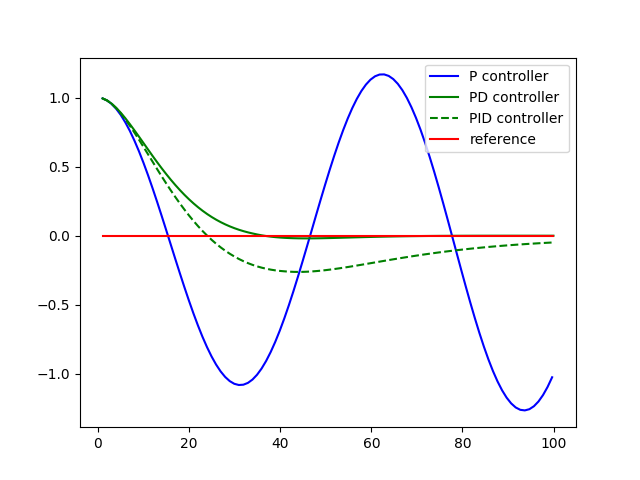
robot.move(steer, speed)

x\_trajectory.append(robot.x)

y\_trajectory.append(robot.y)

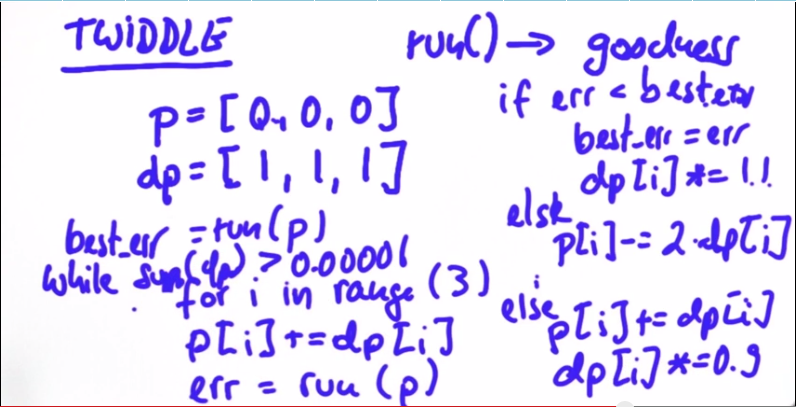
**return** x\_trajectory, y\_trajectory

Ok. With the integral term we're keeping track of all the previous CTEs, initially we set int\_cte to 0 and then add the current cte term to the count int\_cte += cte. Finally we update the steering value, -tau\_p \* cte - tau\_d \* diff\_cte - tau\_i \* int\_cte with the new tau\_i parameter.



This may not seem all that impressive. PID seems to do worse than the PD controller! The purpose of the I term is to compensate for biases, and the current robot has no bias.

In the next programming quiz we'll add steering drift and revisit this graph.

Twiddle notes: 

**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**:

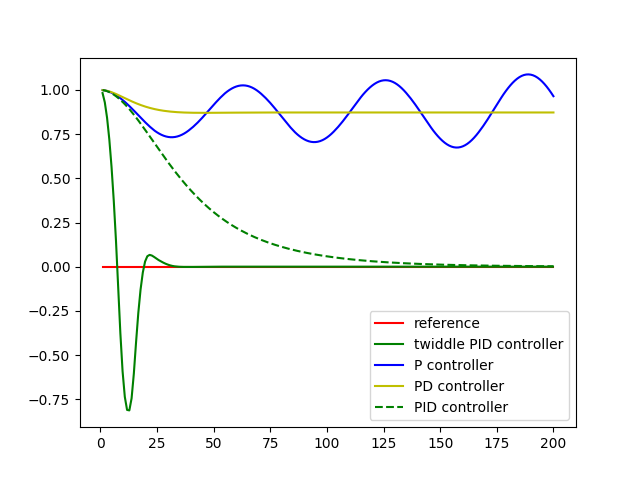
p[i] += dp[i]

dp[i] \*= 0.9

it += 1

**return** p

This follows Sebastian's pseudocode very closely. Before each run we make a new Robot with make\_robot, ensuring on each run the robot starts from the same position. You may find it fruitful to change the magic numbers altering p and dp.



Now the PID controller outshines PD controller! Also, with twiddle the PID controller converges faster but we overshoot drastically at first so it's a tradeoff. Try tuning twiddle and see if you can reduce the overshoot.

**Additional Resources on Control**

Nice work reaching the end of the control content! While you still have the project left to do here, we're also providing some additional resources and recent research on the topic that you can come back to if you have time later on.

Reading research papers is a great way to get exposure to the latest and greatest in the field, as well as expand your learning. However, just like the project ahead, it's often best to *learn by doing* - if you find a paper that really excites you, try to implement it (or even something better) yourself!

**Optional Reading**

All of these are completely optional reading - you could spend hours reading through the entirety of these! We suggest moving onto the project first so you have what you’ve learned fresh on your mind, before coming back to check these out.

We've categorized these papers to hopefully help you narrow down which ones might be of interest, as well as including their *Abstract* section, which summarizes the paper.

**Model Predictive Control (MPC)**

[Vision-Based High Speed Driving with a Deep Dynamic Observer](https://arxiv.org/abs/1812.02071) by P. Drews, et. al.

***Abstract:****In this paper we present a framework for combining deep learning-based road detection, particle filters, and Model Predictive Control (MPC) to drive aggressively using only a monocular camera, IMU, and wheel speed sensors. This framework uses deep convolutional neural networks combined with LSTMs to learn a local cost map representation of the track in front of the vehicle. A particle filter uses this dynamic observation model to localize in a schematic map, and MPC is used to drive aggressively using this particle filter based state estimate. We show extensive real world testing results, and demonstrate reliable operation of the vehicle at the friction limits on a complex dirt track. We reach speeds above 27 mph (12 m/s) on a dirt track with a 105 foot (32m) long straight using our 1:5 scale test vehicle. [...]*

**Reinforcement Learning-based**

[Reinforcement Learning and Deep Learning based Lateral Control for Autonomous Driving](https://arxiv.org/abs/1810.12778) by D. Li, et. al.

***Abstract:****This paper investigates the vision-based autonomous driving with deep learning and reinforcement learning methods. Different from the end-to-end learning method, our method breaks the vision-based lateral control system down into a perception module and a control module. The perception module which is based on a multi-task learning neural network first takes a driver-view image as its input and predicts the track features. The control module which is based on reinforcement learning then makes a control decision based on these features. In order to improve the data efficiency, we propose visual TORCS (VTORCS), a deep reinforcement learning environment which is based on the open racing car simulator (TORCS). By means of the provided functions, one can train an agent with the input of an image or various physical sensor measurement, or evaluate the perception algorithm on this simulator. The trained reinforcement learning controller outperforms the linear quadratic regulator (LQR) controller and model predictive control (MPC) controller on different tracks. The experiments demonstrate that the perception module shows promising performance and the controller is capable of controlling the vehicle drive well along the track center with visual input.*

**Behavioral Cloning**

The below paper shows one of the techniques Waymo has researched using imitation learning (aka behavioral cloning) to drive a car.

[ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst](https://arxiv.org/abs/1812.03079) by M. Bansal, A. Krizhevsky and A. Ogale

***Abstract:****Our goal is to train a policy for autonomous driving via imitation learning that is robust enough to drive a real vehicle. We find that standard behavior cloning is insufficient for handling complex driving scenarios, even when we leverage a perception system for preprocessing the input and a controller for executing the output on the car: 30 million examples are still not enough. We propose exposing the learner to synthesized data in the form of perturbations to the expert's driving, which creates interesting situations such as collisions and/or going off the road. Rather than purely imitating all data, we augment the imitation loss with additional losses that penalize undesirable events and encourage progress -- the perturbations then provide an important signal for these losses and lead to robustness of the learned model. We show that the ChauffeurNet model can handle complex situations in simulation, and present ablation experiments that emphasize the importance of each of our proposed changes and show that the model is responding to the appropriate causal factors. Finally, we demonstrate the model driving a car in the real world.*