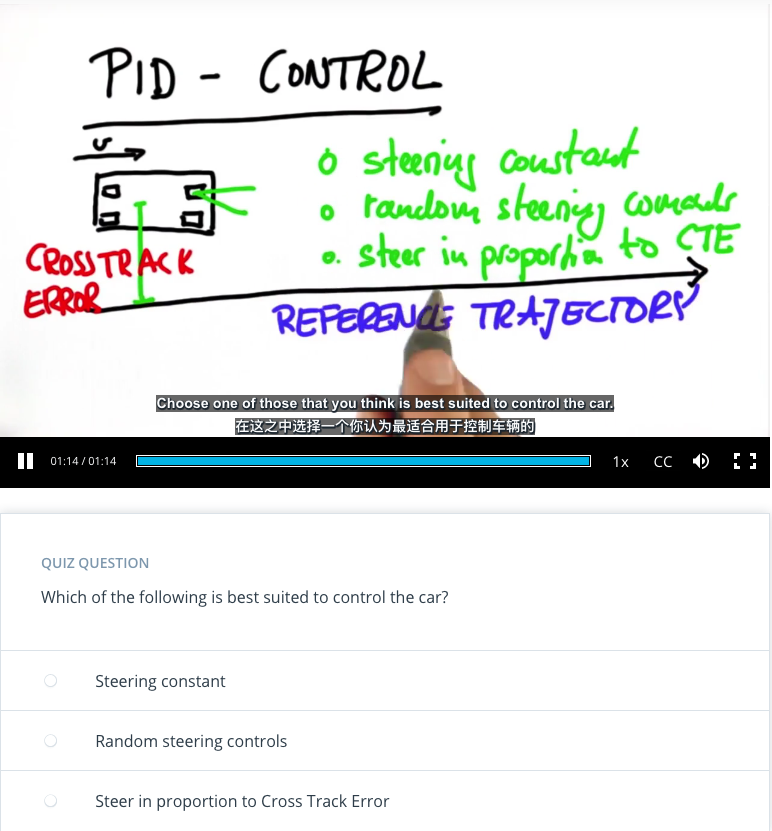
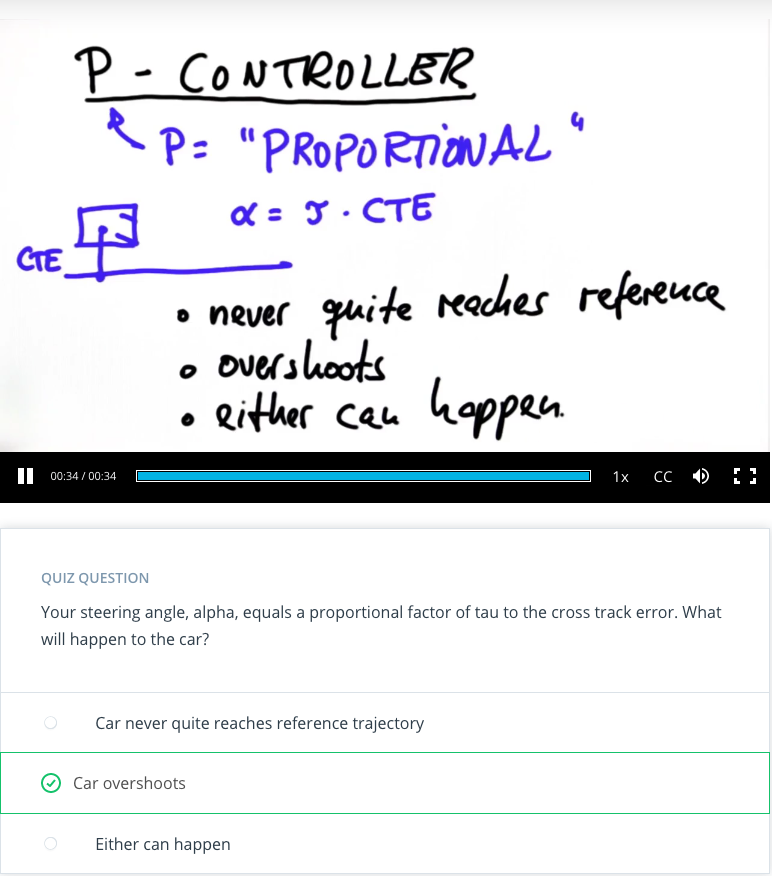
1. Intro
2. PID Control



1. Proportional Control



1. Implement P Controller

*# -----------*

*# User Instructions*

*#*

*# Implement a P controller by running 100 iterations*

*# of robot motion. The desired trajectory for the*

*# robot is the x-axis. The steering angle should be set*

*# by the parameter tau so that:*

*#*

*# steering = -tau \* crosstrack\_error*

*#*

*# You'll only need to modify the `run` function at the bottom.*

*# ------------*

*import random*

*import numpy as np*

*import matplotlib.pyplot as plt*

*# ------------------------------------------------*

*#*

*# this is the Robot class*

*#*

*class Robot(object):*

*def \_\_init\_\_(self, length=20.0):*

*"""*

*Creates robot and initializes location/orientation to 0, 0, 0.*

*"""*

*self.x = 0.0*

*self.y = 0.0*

*self.orientation = 0.0*

*self.length = length*

*self.steering\_noise = 0.0*

*self.distance\_noise = 0.0*

*self.steering\_drift = 0.0*

*def set(self, x, y, orientation):*

*"""*

*Sets a robot coordinate.*

*"""*

*self.x = x*

*self.y = y*

*self.orientation = orientation % (2.0 \* np.pi)*

*def set\_noise(self, steering\_noise, distance\_noise):*

*"""*

*Sets the noise parameters.*

*"""*

*# makes it possible to change the noise parameters*

*# this is often useful in particle filters*

*self.steering\_noise = steering\_noise*

*self.distance\_noise = distance\_noise*

*def set\_steering\_drift(self, drift):*

*"""*

*Sets the systematical steering drift parameter*

*"""*

*self.steering\_drift = drift*

*def move(self, steering, distance, tolerance=0.001, max\_steering\_angle=np.pi / 4.0):*

*"""*

*steering = front wheel steering angle, limited by max\_steering\_angle*

*distance = total distance driven, most be non-negative*

*"""*

*if steering > max\_steering\_angle:*

*steering = max\_steering\_angle*

*if steering < -max\_steering\_angle:*

*steering = -max\_steering\_angle*

*if distance < 0.0:*

*distance = 0.0*

*# apply noise*

*steering2 = random.gauss(steering, self.steering\_noise)*

*distance2 = random.gauss(distance, self.distance\_noise)*

*# apply steering drift*

*steering2 += self.steering\_drift*

*# Execute motion*

*turn = np.tan(steering2) \* distance2 / self.length*

*if abs(turn) < tolerance:*

*# approximate by straight line motion*

*self.x += distance2 \* np.cos(self.orientation)*

*self.y += distance2 \* np.sin(self.orientation)*

*self.orientation = (self.orientation + turn) % (2.0 \* np.pi)*

*else:*

*# approximate bicycle model for motion*

*radius = distance2 / turn*

*cx = self.x - (np.sin(self.orientation) \* radius)*

*cy = self.y + (np.cos(self.orientation) \* radius)*

*self.orientation = (self.orientation + turn) % (2.0 \* np.pi)*

*self.x = cx + (np.sin(self.orientation) \* radius)*

*self.y = cy - (np.cos(self.orientation) \* radius)*

*def \_\_repr\_\_(self):*

*return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y, self.orientation)*

*############## ADD / MODIFY CODE BELOW ####################*

*# ------------------------------------------------------------------------*

*#*

*# run - does a single control run*

*robot = Robot()*

*robot.set(0, 1, 0)*

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

*x\_trajectory = []*

*y\_trajectory = []*

*# TODO: your code here*

*return x\_trajectory, y\_trajectory*

*x\_trajectory, y\_trajectory = run(robot, 0.1)*

*n = len(x\_trajectory)*

*fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))*

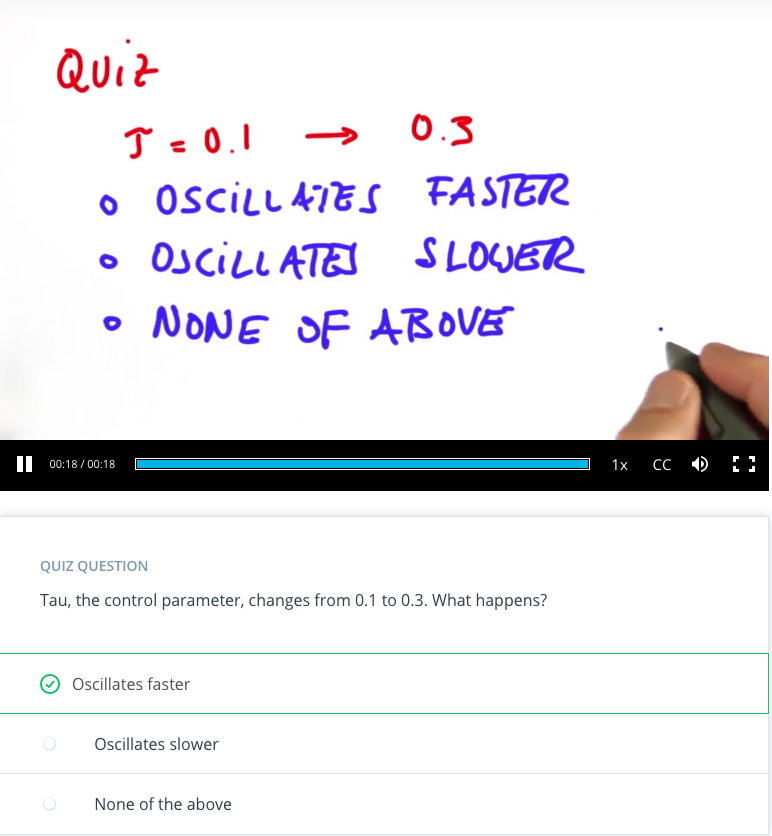
*ax1.plot(x\_trajectory, y\_trajectory, 'g', label='P controller')*

*ax1.plot(x\_trajectory, np.zeros(n), 'r', label='reference')*

1. P Controller Solution



1. Oscillations



1. PD Controller

# -----------

# User Instructions

#

# Implement a PD controller by running 100 iterations

# of robot motion. The steering angle should be set

# by the parameter tau\_p and tau\_d so that:

#

# steering = -tau\_p \* CTE - tau\_d \* diff\_CTE

# where differential crosstrack error (diff\_CTE)

# is given by CTE(t) - CTE(t-1)

#

#

# Only modify code at the bottom! Look for the TODO

# ------------

import random

import numpy as np

import matplotlib.pyplot as plt

# ------------------------------------------------

#

# this is the Robot class

#

class Robot(object):

def \_\_init\_\_(self, length=20.0):

"""

Creates robot and initializes location/orientation to 0, 0, 0.

"""

self.x = 0.0

self.y = 0.0

self.orientation = 0.0

self.length = length

self.steering\_noise = 0.0

self.distance\_noise = 0.0

self.steering\_drift = 0.0

def set(self, x, y, orientation):

"""

Sets a robot coordinate.

"""

self.x = x

self.y = y

self.orientation = orientation % (2.0 \* np.pi)

def set\_noise(self, steering\_noise, distance\_noise):

"""

Sets the noise parameters.

"""

# makes it possible to change the noise parameters

# this is often useful in particle filters

self.steering\_noise = steering\_noise

self.distance\_noise = distance\_noise

def set\_steering\_drift(self, drift):

"""

Sets the systematical steering drift parameter

"""

self.steering\_drift = drift

def move(self, steering, distance, tolerance=0.001, max\_steering\_angle=np.pi / 4.0):

"""

steering = front wheel steering angle, limited by max\_steering\_angle

distance = total distance driven, most be non-negative

"""

if steering > max\_steering\_angle:

steering = max\_steering\_angle

if steering < -max\_steering\_angle:

steering = -max\_steering\_angle

if distance < 0.0:

distance = 0.0

# apply noise

steering2 = random.gauss(steering, self.steering\_noise)

distance2 = random.gauss(distance, self.distance\_noise)

# apply steering drift

steering2 += self.steering\_drift

# Execute motion

turn = np.tan(steering2) \* distance2 / self.length

if abs(turn) < tolerance:

# approximate by straight line motion

self.x += distance2 \* np.cos(self.orientation)

self.y += distance2 \* np.sin(self.orientation)

self.orientation = (self.orientation + turn) % (2.0 \* np.pi)

else:

# approximate bicycle model for motion

radius = distance2 / turn

cx = self.x - (np.sin(self.orientation) \* radius)

cy = self.y + (np.cos(self.orientation) \* radius)

self.orientation = (self.orientation + turn) % (2.0 \* np.pi)

self.x = cx + (np.sin(self.orientation) \* radius)

self.y = cy - (np.cos(self.orientation) \* radius)

def \_\_repr\_\_(self):

return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y, self.orientation)

############## ADD / MODIFY CODE BELOW ####################

# ------------------------------------------------------------------------

#

# run - does a single control run

# previous P controller

def run\_p(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

robot = Robot()

robot.set(0, 1, 0)

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

x\_trajectory = []

y\_trajectory = []

# TODO: your code here

return x\_trajectory, y\_trajectory

x\_trajectory, y\_trajectory = run(robot, 0.2, 3.0)

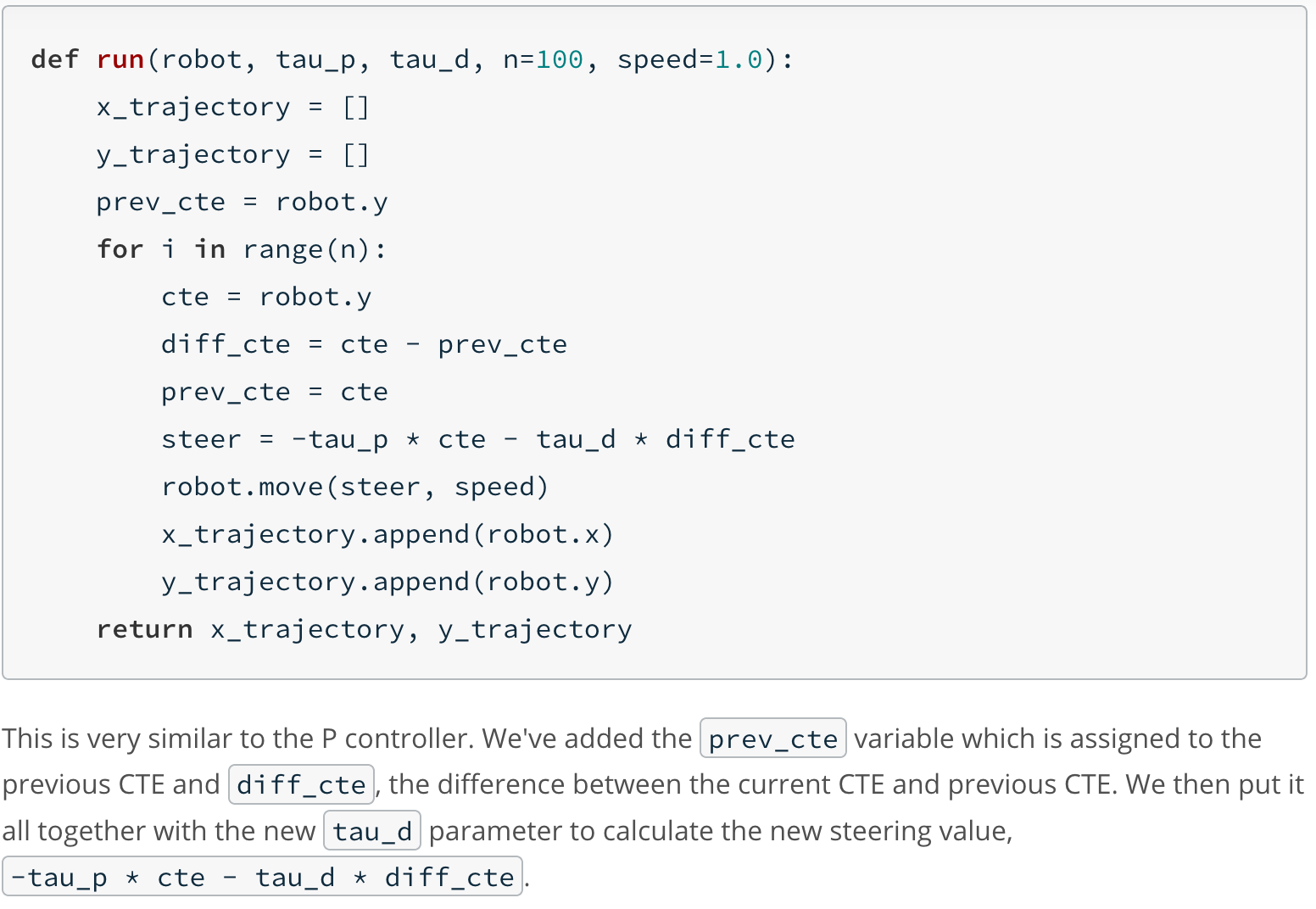
n = len(x\_trajectory)

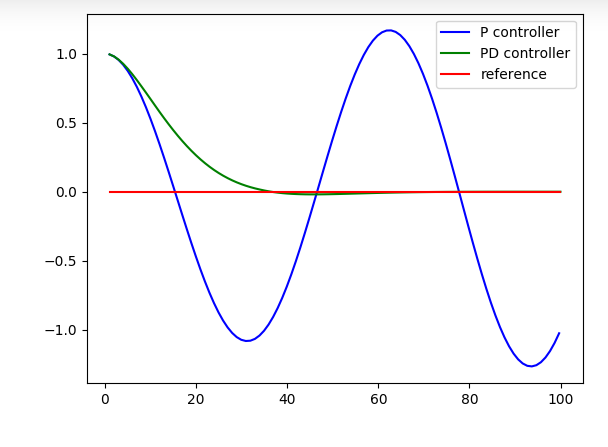
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))

ax1.plot(x\_trajectory, y\_trajectory, 'g', label='PD controller')

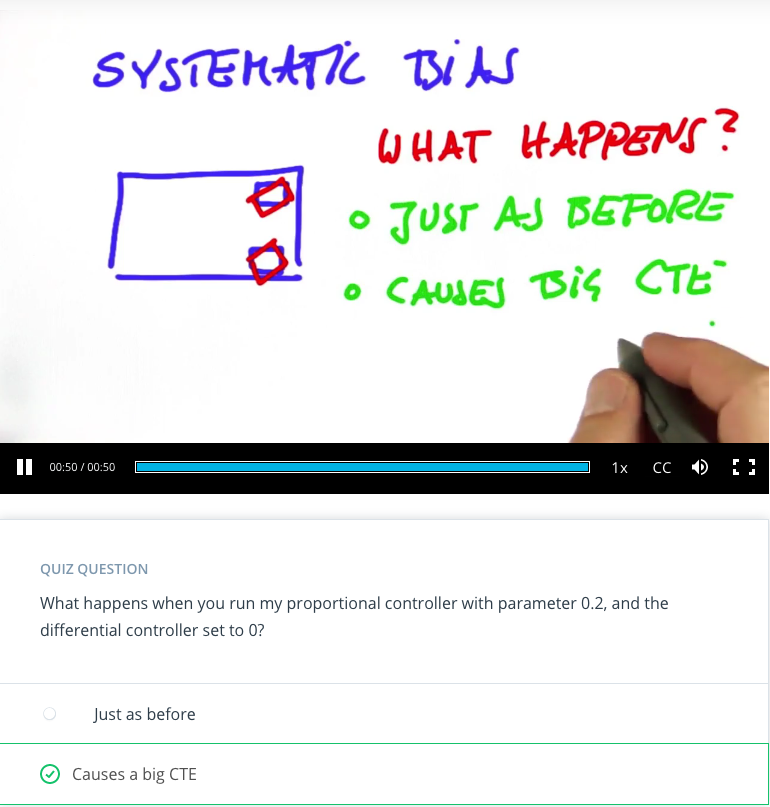
ax1.plot(x\_trajectory, np.zeros(n), 'r', label='reference')

1. PD Controller Solution

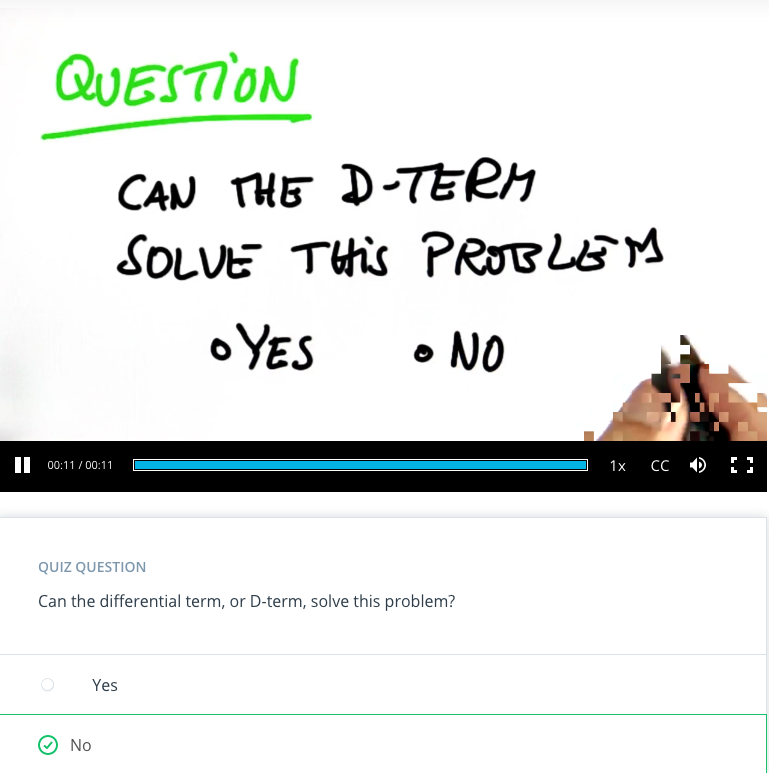




1. Systematic Bias



1. Is PD Enough



1. PID implement

# -----------

# User Instructions

#

# Implement a P controller by running 100 iterations

# of robot motion. The steering angle should be set

# by the parameter tau so that:

#

# steering = -tau\_p \* CTE - tau\_d \* diff\_CTE - tau\_i \* int\_CTE

#

# where the integrated crosstrack error (int\_CTE) is

# the sum of all the previous crosstrack errors.

# This term works to cancel out steering drift.

#

# Only modify code at the bottom! Look for the TODO.

# ------------

import random

import numpy as np

import matplotlib.pyplot as plt

# ------------------------------------------------

#

# this is the Robot class

#

class Robot(object):

def \_\_init\_\_(self, length=20.0):

"""

Creates robot and initializes location/orientation to 0, 0, 0.

"""

self.x = 0.0

self.y = 0.0

self.orientation = 0.0

self.length = length

self.steering\_noise = 0.0

self.distance\_noise = 0.0

self.steering\_drift = 0.0

def set(self, x, y, orientation):

"""

Sets a robot coordinate.

"""

self.x = x

self.y = y

self.orientation = orientation % (2.0 \* np.pi)

def set\_noise(self, steering\_noise, distance\_noise):

"""

Sets the noise parameters.

"""

# makes it possible to change the noise parameters

# this is often useful in particle filters

self.steering\_noise = steering\_noise

self.distance\_noise = distance\_noise

def set\_steering\_drift(self, drift):

"""

Sets the systematical steering drift parameter

"""

self.steering\_drift = drift

def move(self, steering, distance, tolerance=0.001, max\_steering\_angle=np.pi / 4.0):

"""

steering = front wheel steering angle, limited by max\_steering\_angle

distance = total distance driven, most be non-negative

"""

if steering > max\_steering\_angle:

steering = max\_steering\_angle

if steering < -max\_steering\_angle:

steering = -max\_steering\_angle

if distance < 0.0:

distance = 0.0

# apply noise

steering2 = random.gauss(steering, self.steering\_noise)

distance2 = random.gauss(distance, self.distance\_noise)

# apply steering drift

steering2 += self.steering\_drift

# Execute motion

turn = np.tan(steering2) \* distance2 / self.length

if abs(turn) < tolerance:

# approximate by straight line motion

self.x += distance2 \* np.cos(self.orientation)

self.y += distance2 \* np.sin(self.orientation)

self.orientation = (self.orientation + turn) % (2.0 \* np.pi)

else:

# approximate bicycle model for motion

radius = distance2 / turn

cx = self.x - (np.sin(self.orientation) \* radius)

cy = self.y + (np.cos(self.orientation) \* radius)

self.orientation = (self.orientation + turn) % (2.0 \* np.pi)

self.x = cx + (np.sin(self.orientation) \* radius)

self.y = cy - (np.cos(self.orientation) \* radius)

def \_\_repr\_\_(self):

return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y, self.orientation)

############## ADD / MODIFY CODE BELOW ####################

# ------------------------------------------------------------------------

#

# run - does a single control run

robot = Robot()

robot.set(0, 1, 0)

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

x\_trajectory = []

y\_trajectory = []

# TODO: your code here

pre\_cte = robot.y

i\_cte = 0

for i in range(n):

cte = robot.y

diff\_cte = cte - pre\_cte

pre\_cte = cte

i\_cte = i\_cte+cte

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

robot.move(steer,speed)

x\_trajectory.append(robot.x)

y\_trajectory.append(robot.y)

return x\_trajectory, y\_trajectory

x\_trajectory, y\_trajectory = run(robot, 0.2, 3.0, 0.004)

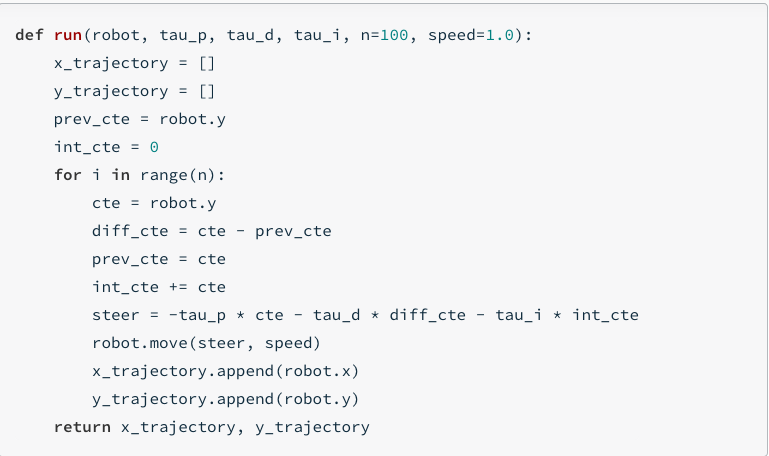
n = len(x\_trajectory)

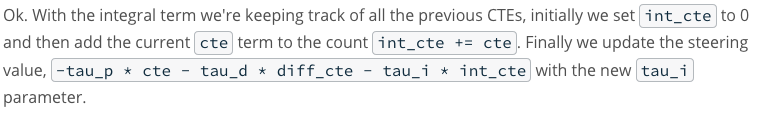
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8,8))

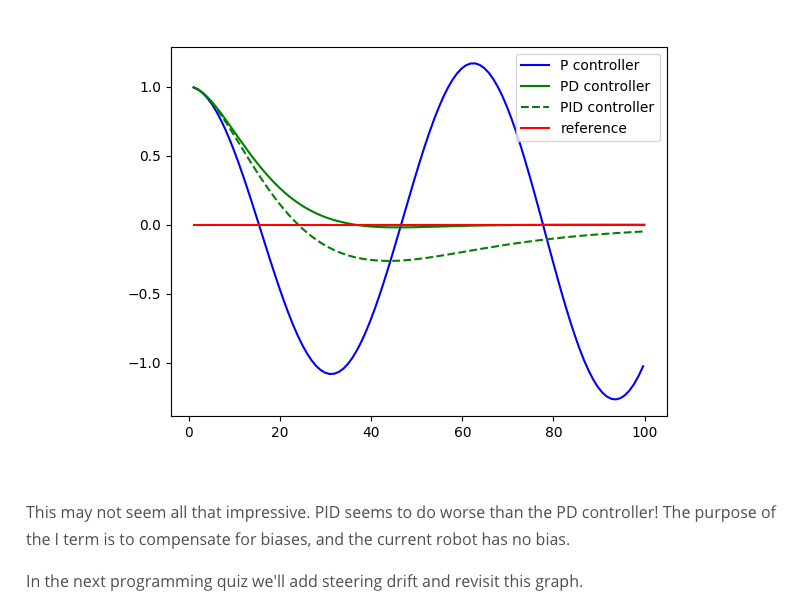
ax1.plot(x\_trajectory, y\_trajectory, 'g', label='PID controller')

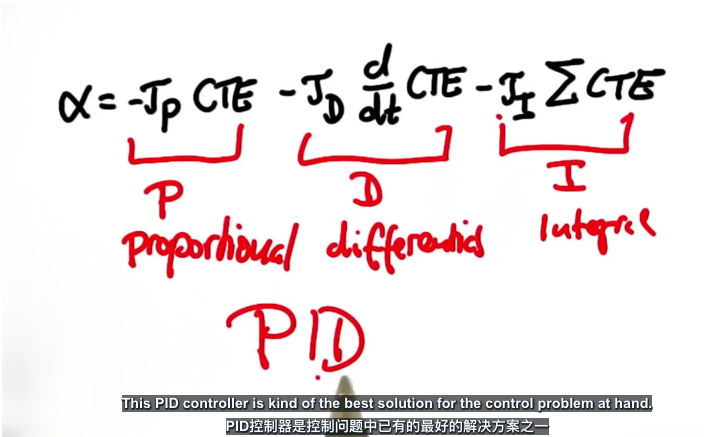
ax1.plot(x\_trajectory, np.zeros(n), 'r', label='reference')

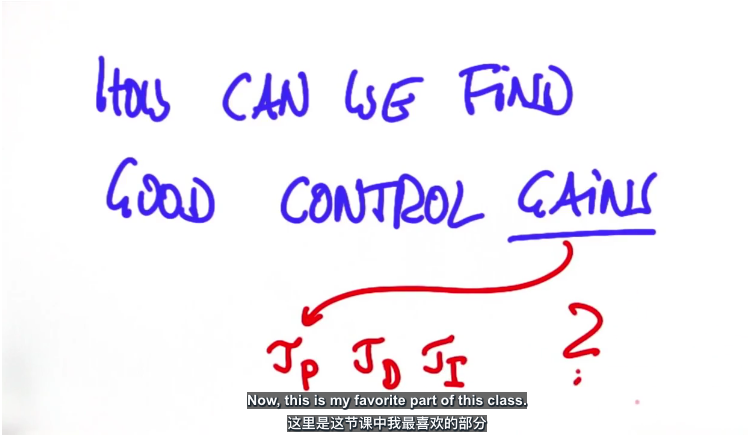
1. PID implement Solution





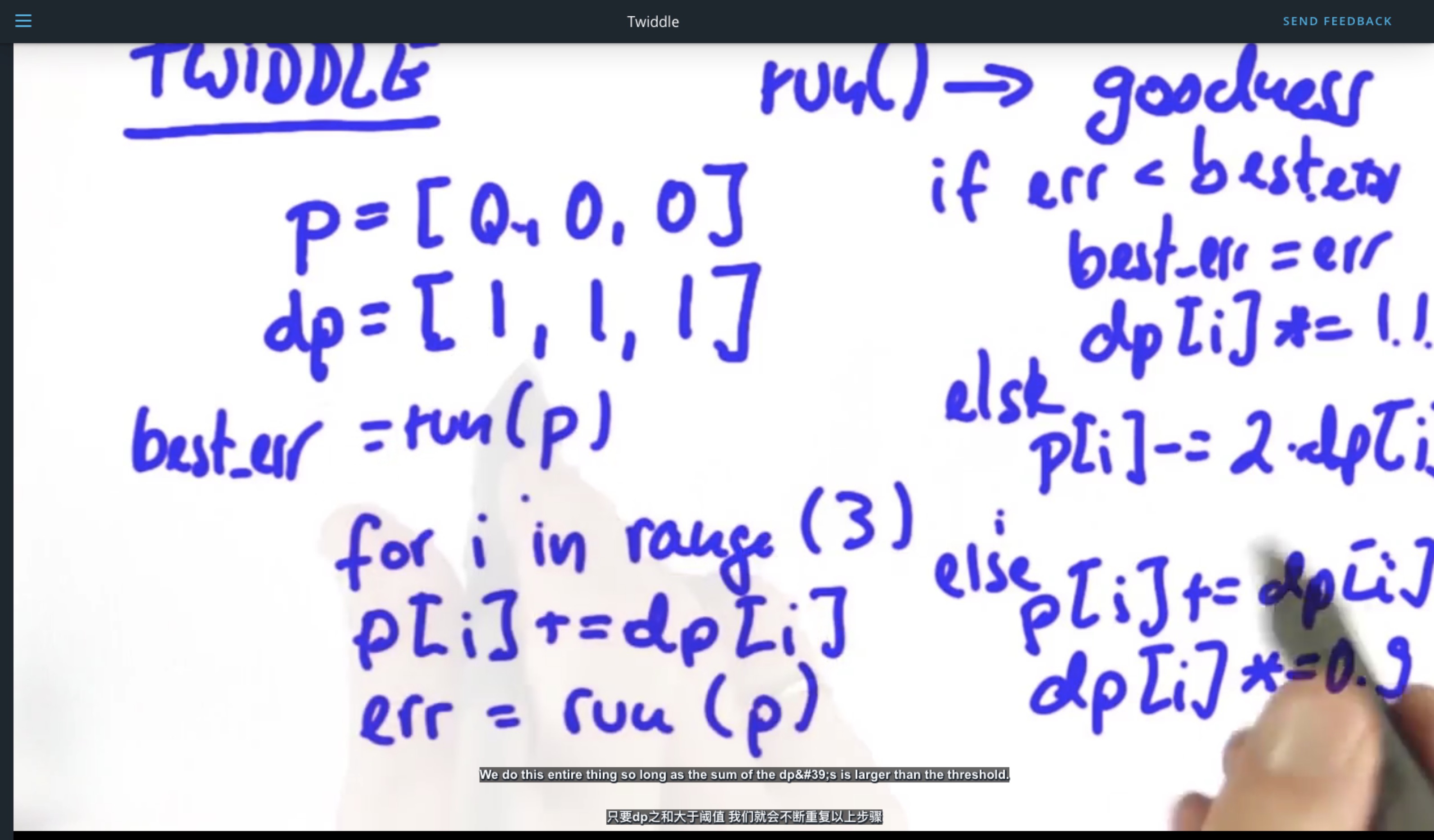








1. Twiddle



1. Parameter Optimization

# ----------------

# User Instructions

#

# Implement twiddle as shown in the previous two videos.

# Your accumulated error should be very small!

#

# You don't have to use the exact values as shown in the video

# play around with different values! This quiz isn't graded just see

# how low of an error you can get.

#

# Try to get your error below 1.0e-10 with as few iterations

# as possible (too many iterations will cause a timeout).

#

# No cheating!

# ------------

import random

import numpy as np

import matplotlib.pyplot as plt

# ------------------------------------------------

#

# this is the Robot class

#

class Robot(object):

def \_\_init\_\_(self, length=20.0):

"""

Creates robot and initializes location/orientation to 0, 0, 0.

"""

self.x = 0.0

self.y = 0.0

self.orientation = 0.0

self.length = length

self.steering\_noise = 0.0

self.distance\_noise = 0.0

self.steering\_drift = 0.0

def set(self, x, y, orientation):

"""

Sets a robot coordinate.

"""

self.x = x

self.y = y

self.orientation = orientation % (2.0 \* np.pi)

def set\_noise(self, steering\_noise, distance\_noise):

"""

Sets the noise parameters.

"""

# makes it possible to change the noise parameters

# this is often useful in particle filters

self.steering\_noise = steering\_noise

self.distance\_noise = distance\_noise

def set\_steering\_drift(self, drift):

"""

Sets the systematical steering drift parameter

"""

self.steering\_drift = drift

def move(self, steering, distance, tolerance=0.001, max\_steering\_angle=np.pi / 4.0):

"""

steering = front wheel steering angle, limited by max\_steering\_angle

distance = total distance driven, most be non-negative

"""

if steering > max\_steering\_angle:

steering = max\_steering\_angle

if steering < -max\_steering\_angle:

steering = -max\_steering\_angle

if distance < 0.0:

distance = 0.0

# apply noise

steering2 = random.gauss(steering, self.steering\_noise)

distance2 = random.gauss(distance, self.distance\_noise)

# apply steering drift

steering2 += self.steering\_drift

# Execute motion

turn = np.tan(steering2) \* distance2 / self.length

if abs(turn) < tolerance:

# approximate by straight line motion

self.x += distance2 \* np.cos(self.orientation)

self.y += distance2 \* np.sin(self.orientation)

self.orientation = (self.orientation + turn) % (2.0 \* np.pi)

else:

# approximate bicycle model for motion

radius = distance2 / turn

cx = self.x - (np.sin(self.orientation) \* radius)

cy = self.y + (np.cos(self.orientation) \* radius)

self.orientation = (self.orientation + turn) % (2.0 \* np.pi)

self.x = cx + (np.sin(self.orientation) \* radius)

self.y = cy - (np.cos(self.orientation) \* radius)

def \_\_repr\_\_(self):

return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y, self.orientation)

############## ADD / MODIFY CODE BELOW ####################

# ------------------------------------------------------------------------

#

# run - does a single control run

def make\_robot():

"""

Resets the robot back to the initial position and drift.

You'll want to call this after you call `run`.

"""

robot = Robot()

robot.set(0, 1, 0)

robot.set\_steering\_drift(10 / 180 \* np.pi)

return robot

# NOTE: We use params instead of tau\_p, tau\_d, tau\_i

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

x\_trajectory = []

y\_trajectory = []

err = 0

prev\_cte = robot.y

int\_cte = 0

for i in range(2 \* n):

cte = robot.y

diff\_cte = cte - prev\_cte

int\_cte += cte

prev\_cte = cte

steer = -params[0] \* cte - params[1] \* diff\_cte - params[2] \* int\_cte

robot.move(steer, speed)

x\_trajectory.append(robot.x)

y\_trajectory.append(robot.y)

if i >= n:

err += cte \*\* 2

return x\_trajectory, y\_trajectory, err / n

# Make this tolerance bigger if you are timing out!

def twiddle(tol=0.2):

# Don't forget to call `make\_robot` before every call of `run`!

p = [0, 0, 0]

dp = [1, 1, 1]

robot = make\_robot()

x\_trajectory, y\_trajectory, best\_err = run(robot, p)

# TODO: twiddle loop here

it = 0

while sum(dp) > tol:

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

print("Iteration {}, Para = {},best error = {}".format(it, p,best\_err))

return p, best\_err

params, err = twiddle()

print("Final twiddle error = {}".format(err))

robot = make\_robot()

x\_trajectory, y\_trajectory, err = run(robot, params)

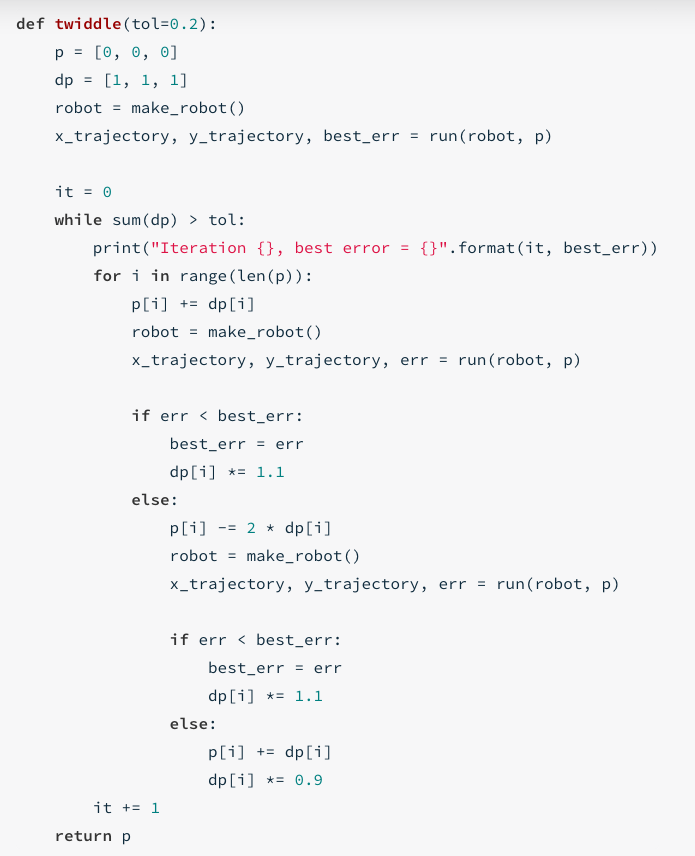
n = len(x\_trajectory)

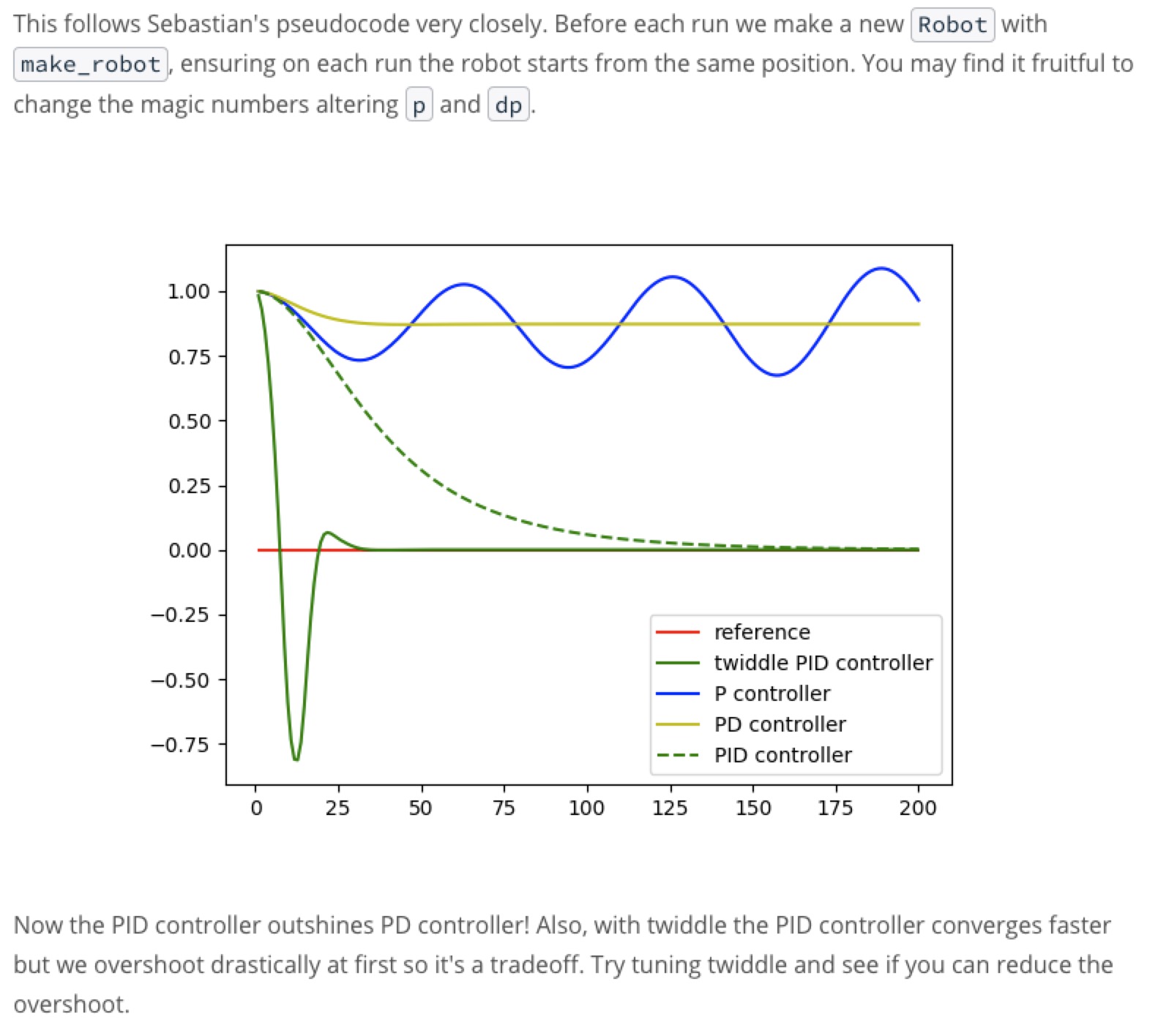
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))

ax1.plot(x\_trajectory, y\_trajectory, 'g', label='Twiddle PID controller')

ax1.plot(x\_trajectory, np.zeros(n), 'r', label='reference')

1. Parameter Optimization Solution





1. Outro

“Great work”…”transfer python to C++”

1. Bonus Round:Control[Optional]

**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.