AUTONOMOUS VEHICLES

PROJECT 2 TEAM 2

TITLE

1. DYNAMIC VEHICLES: This task is an extension of Lane Following that includes additional rules of the road and other moving vehicles and static obstacles.
2. Imitation learning using Duckietown simulator.

IMPORTANT PARAMETERS

VARIABLE	VALUE	DESCRIPTION
v_bar	0.23	Nominal linear velocity (m/s).
k_theta	-1.5	Proportional gain for theta.
k_d	-2	Proportional gain for d.
d_thres	0.3490	Cap for error in d.
theta_thres	0.523	Maximum desired theta.
d_offset	0	An offset from the lane position.
Gain	0.75	Sets the gain of the bot
Time_left_turn	2	Left turn time at junction
Time_straight_turn	2	Go straight time at junction
Time_right_turn	2	Right Turn time at junction

DYNAMIC VEHICLES CODE

1. Lane Controller Node

```
import math
import time
import numpy as np
import rospy
from duckietown_msgs.msg import Twist2DStamped, LanePose, WheelsCmdStamped,
BoolStamped, FSMState, StopLineReading
import time
import numpy as np

class lane_controller(object):

    def __init__(self):
        self.node_name = rospy.get_name()
        self.lane_reading = None
        self.last_ms = None
```

```
        self.pub_counter = 0


        self.velocity_to_m_per_s = 0.67
        self.omega_to_rad_per_s = 0.45 * 2 * math.pi
# Setup
parameters
         self.velocity_to_m_per_s = 1.53
         self.omega_to_rad_per_s = 4.75
         self.setGains()

         # Publication
         self.pub_car_cmd = rospy.Publisher("~car_cmd", Twist2DStamped,
queue_size=1)
         self.pub_actuator_limits_received =
rospy.Publisher("~actuator_limits_received", BoolStamped, queue_size=1)
         self.pub_radius_limit = rospy.Publisher("~radius_limit", BoolStamped,
queue_size=1)



         # Subscriptions
         self.sub_lane_reading = rospy.Subscriber("~lane_pose", LanePose,
self.PoseHandling, "lane_filter", queue_size=1)

         self.sub_obstacle_avoidance_pose =
rospy.Subscriber("~obstacle_avoidance_pose", LanePose, self.PoseHandling,
"obstacle_avoidance",queue_size=1)
         self.sub_obstacle_detected = rospy.Subscriber("~obstacle_detected",
BoolStamped, self.setFlag, "obstacle_detected", queue_size=1)

         self.sub_intersection_navigation_pose =
rospy.Subscriber("~intersection_navigation_pose", LanePose,
self.PoseHandling, "intersection_navigation",queue_size=1)
 self.sub_wheels_cmd_executed = rospy.Subscriber("~wheels_cmd_executed",
WheelsCmdStamped, self.updateWheelsCmdExecuted, queue_size=1)
         self.sub_actuator_limits = rospy.Subscriber("~actuator_limits",
Twist2DStamped, self.updateActuatorLimits, queue_size=1)

         # FSM
         self.sub_switch = rospy.Subscriber("~switch",BoolStamped,
self.cbSwitch,  queue_size=1)
```

```
        self.sub_stop_line =
rospy.Subscriber("~stop_line_reading",StopLineReading,
self.cbStopLineReading,  queue_size=1)


        self.sub_fsm_mode = rospy.Subscriber("~fsm_mode", FSMState,
self.cbMode, queue_size=1)

        self.msg_radius_limit = BoolStamped()
        self.msg_radius_limit.data = self.use_radius_limit
        self.pub_radius_limit.publish(self.msg_radius_limit)

        # safe shutdown
        rospy.on_shutdown(self.custom_shutdown)

        # timer
        self.gains_timer = rospy.Timer(rospy.Duration.from_sec(0.1),
self.getGains_event)
        rospy.loginfo("[%s] Initialized " % (rospy.get_name()))

        self.stop_line_distance = 999
        self.stop_line_detected = False

    def cbStopLineReading(self, msg):
        self.stop_line_distance = np.sqrt(msg.stop_line_point.x**2 +
msg.stop_line_point.y**2 + msg.stop_line_point.z**2)
        self.stop_line_detected = msg.stop_line_detected

    def setupParameter(self,param_name,default_value):
        value = rospy.get_param(param_name,default_value)
        rospy.set_param(param_name,value)   # Write to parameter server for
transparancy
        rospy.loginfo("[%s] %s = %s " %(self.node_name,param_name,value))
        return value

    def setGains(self):
        self.v_bar_gain_ref = 0.5
        v_bar_fallback = 0.25  # nominal speed, 0.25m/s
        self.v_max = 1
        k_theta_fallback = -2.0
        k_d_fallback = - (k_theta_fallback ** 2) / (4.0 *
self.v_bar_gain_ref)
        theta_thres_fallback = math.pi / 6.0
```

```
        d_thres_fallback = math.fabs(k_theta_fallback / k_d_fallback) *
theta_thres_fallback
        d_offset_fallback = 0.0

        k_theta_fallback = k_theta_fallback
        k_d_fallback = k_d_fallback

        k_Id_fallback = 2.5
        k_Iphi_fallback = 1.25

        self.fsm_state = None
        self.cross_track_err = 0
        self.heading_err = 0
        self.cross_track_integral = 0
        self.heading_integral = 0
        self.cross_track_integral_top_cutoff = 0.3
        self.cross_track_integral_bottom_cutoff = -0.3
        self.heading_integral_top_cutoff = 1.2
        self.heading_integral_bottom_cutoff = -1.2
        #-1.2
        self.time_start_curve = 0

        use_feedforward_part_fallback = False
        self.wheels_cmd_executed = WheelsCmdStamped()

        self.actuator_limits = Twist2DStamped()
        self.actuator_limits.v = 999.0  # to make sure the limit is not hit
before the message is received
        self.actuator_limits.omega = 999.0  # to make sure the limit is not
hit before the message is received
        self.omega_max = 999.0  # considering radius limitation and actuator
limits   # to make sure the limit is not hit before the message is received

        self.use_radius_limit_fallback = True

        self.flag_dict = {"obstacle_detected": False,
                          "parking_stop": False,
                          "fleet_planning_lane_following_override_active":
False,
                          "implicit_coord_velocity_limit_active": False}

        self.pose_msg = LanePose()
        self.pose_initialized = False
```

```
        self.pose_msg_dict = dict()
        self.v_ref_possible = dict()
        self.main_pose_source = None

        self.active = True

        self.sleepMaintenance = False
        self.v_bar = self.setupParameter("~v_bar",v_bar_fallback)   # Linear
velocity
        self.k_d = self.setupParameter("~k_d",k_d_fallback)     # P gain for
d
        self.k_theta = self.setupParameter("~k_theta",k_theta_fallback)     #
P gain for theta
        self.d_thres = self.setupParameter("~d_thres",d_thres_fallback)     #
Cap for error in d
        self.theta_thres =
self.setupParameter("~theta_thres",theta_thres_fallback)     # Maximum desire
theta
        self.d_offset = self.setupParameter("~d_offset",d_offset_fallback)  #
a configurable offset from the lane position

        self.k_Id = self.setupParameter("~k_Id", k_Id_fallback)     # gain
for integrator of d
        self.k_Iphi = self.setupParameter("~k_Iphi",k_Iphi_fallback)    #
gain for integrator of phi (phi = theta)
        #TODO: Feedforward was not working, go away with this error source!
(Julien)
        self.use_feedforward_part =
self.setupParameter("~use_feedforward_part",use_feedforward_part_fallback)
        self.omega_ff = self.setupParameter("~omega_ff",0)
        self.omega_max = self.setupParameter("~omega_max", 999)
        self.omega_min = self.setupParameter("~omega_min", -999)
        self.use_radius_limit = self.setupParameter("~use_radius_limit",
self.use_radius_limit_fallback)
        self.min_radius = self.setupParameter("~min_rad", 0.0)

        self.d_ref = self.setupParameter("~d_ref", 0)
        self.phi_ref = self.setupParameter("~phi_ref",0)
        self.object_detected = self.setupParameter("~object_detected", 0)
        self.v_ref_possible["default"] = self.v_max


    def getGains_event(self, event):
```

```
        v_bar = rospy.get_param("~v_bar")
        k_d = rospy.get_param("~k_d")
        k_theta = rospy.get_param("~k_theta")
        d_thres = rospy.get_param("~d_thres")
        theta_thres = rospy.get_param("~theta_thres")
        d_offset = rospy.get_param("~d_offset")
        d_ref = rospy.get_param("~d_ref")
        phi_ref = rospy.get_param("~phi_ref")
        use_radius_limit = rospy.get_param("~use_radius_limit")
        object_detected = rospy.get_param("~object_detected")
        self.omega_ff = rospy.get_param("~omega_ff")
        self.omega_max = rospy.get_param("~omega_max")
        self.omega_min = rospy.get_param("~omega_min")
        #FeedForward
        #TODO: Feedforward was not working, go away with this error source!
(Julien)

        self.velocity_to_m_per_s = 1#0.67     # TODO: change after new
kinematic calibration! (should be obsolete, if new kinematic calibration
works properly)
        #self.omega_to_rad_per_s = 0.45 * 2 * math.pi    # TODO: change after
new kinematic calibration! (should be obsolete, if new kinematic calibration
works properly)

        # FeedForward
        self.curvature_outer = 1 / (0.39)
        self.curvature_inner = 1 / 0.175

        use_feedforward_part = rospy.get_param("~use_feedforward_part")


        k_Id = rospy.get_param("~k_Id")
        k_Iphi = rospy.get_param("~k_Iphi")
        if self.k_Id != k_Id:
            rospy.loginfo("ADJUSTED I GAIN")
            self.cross_track_integral = 0
            self.k_Id = k_Id


        params_old =
(self.v_bar,self.k_d,self.k_theta,self.d_thres,self.theta_thres,
self.d_offset, self.k_Id, self.k_Iphi, self.use_feedforward_part,
self.use_radius_limit)
```

```
        params_new = (v_bar,k_d,k_theta,d_thres,theta_thres, d_offset, k_Id,
k_Iphi, use_feedforward_part, use_radius_limit)

        if params_old != params_new:
            rospy.loginfo("[%s] Gains changed." %(self.node_name))

            self.v_bar = v_bar
            self.k_d = k_d
            self.k_theta = k_theta
            self.d_thres = d_thres
            self.d_ref = d_ref
            self.phi_ref = phi_ref
            self.theta_thres = theta_thres
            self.d_offset = d_offset
            self.k_Id = k_Id
            self.k_Iphi = k_Iphi

            self.use_feedforward_part = use_feedforward_part


            if use_radius_limit != self.use_radius_limit:
                self.use_radius_limit = use_radius_limit
                self.msg_radius_limit.data = self.use_radius_limit
                self.pub_radius_limit.publish(self.msg_radius_limit)

    # FSM

    def cbSwitch(self,fsm_switch_msg):
        self.active = fsm_switch_msg.data   # True or False

        rospy.loginfo("active: " + str(self.active))
    # FSM

    def unsleepMaintenance(self, event):
        self.sleepMaintenance = False

    def cbMode(self,fsm_state_msg):

        # if self.fsm_state != fsm_state_msg.state and fsm_state_msg.state ==
"IN_CHARGING_AREA":
        #     self.sleepMaintenance = True
        #     self.sendStop()
```

```
        #     rospy.Timer(rospy.Duration.from_sec(2.0),
self.unsleepMaintenance)

        self.fsm_state = fsm_state_msg.state    # String of current FSM state
        print "fsm_state changed in lane_controller_node to: " ,
self.fsm_state

    def setFlag(self, msg_flag, flag_name):
        self.flag_dict[flag_name] = msg_flag.data
        if flag_name == "obstacle_detected":
             print "flag obstacle_detected changed"
             print "flag_dict[\"obstacle_detected\"]: ",
self.flag_dict["obstacle_detected"]


    def PoseHandling(self, input_pose_msg, pose_source):
        if not self.active:
            return

        if self.sleepMaintenance:
            return
        self.prev_pose_msg = self.pose_msg
        self.pose_msg_dict[pose_source] = input_pose_msg
        # self.fsm_state = "INTERSECTION_CONTROL" #TODO pass this message
automatically
        if self.pose_initialized:
            v_ref_possible_default = self.v_ref_possible["default"]
            v_ref_possible_main_pose = self.v_ref_possible["main_pose"]
            self.v_ref_possible.clear()
            self.v_ref_possible["default"] = v_ref_possible_default
            self.v_ref_possible["main_pose"] = v_ref_possible_main_pose

        if self.fsm_state == "INTERSECTION_CONTROL":
            if pose_source == "intersection_navigation": # for CL
intersection from AMOD use 'intersection_navigation'
                self.pose_msg = input_pose_msg
                self.pose_msg.curvature_ref = input_pose_msg.curvature
                self.v_ref_possible["main_pose"] = self.v_bar
                self.main_pose_source = pose_source
                self.pose_initialized = True
        elif self.fsm_state == "PARKING":
            if pose_source == "parking":
                #rospy.loginfo("pose source: parking!?")
```

```
                self.pose_msg = input_pose_msg
                self.v_ref_possible["main_pose"] = input_pose_msg.v_ref
                self.main_pose_source = pose_source
                self.pose_initialized = True
        else:
            if pose_source == "lane_filter":
                #rospy.loginfo("pose source: lane_filter")
                self.pose_msg = input_pose_msg
                self.pose_msg.curvature_ref = input_pose_msg.curvature


                self.v_ref_possible["main_pose"] = self.v_bar

                # Adapt speed to stop line!
                if self.stop_line_detected:
                    # 60cm -> v_bar, 15cm -> v_bar/2
                    d1, d2 = 0.8, 0.25
                    a = self.v_bar/(2*(d1-d2))
                    b = self.v_bar - a*d1
                    v_new = a*self.stop_line_distance + b
                    v_new = np.max([self.v_bar/2.0, np.min([self.v_bar,
v_new])])
                    self.v_ref_possible["main_pose"] = v_new
                self.main_pose_source = pose_source
                self.pose_initialized = True

        if self.flag_dict["fleet_planning_lane_following_override_active"] ==
True:
            if "fleet_planning" in self.pose_msg_dict:
                self.pose_msg.d_ref =
self.pose_msg_dict["fleet_planning"].d_ref
                self.v_ref_possible["fleet_planning"] =
self.pose_msg_dict["fleet_planning"].v_ref
        if self.flag_dict["obstacle_detected"] == True:
            if "obstacle_avoidance" in self.pose_msg_dict:
                self.pose_msg.d_ref =
self.pose_msg_dict["obstacle_avoidance"].d_ref
                self.v_ref_possible["obstacle_avoidance"] =
self.pose_msg_dict["obstacle_avoidance"].v_ref
                #print 'v_ref obst_avoid=' ,
self.v_ref_possible["obstacle_avoidance"] #For debugging
        if self.flag_dict["implicit_coord_velocity_limit_active"] == True:
            if "implicit_coord" in self.pose_msg_dict:
```

```
                self.v_ref_possible["implicit_coord"] =
self.pose_msg_dict["implicit_coord"].v_ref

        self.pose_msg.v_ref = min(self.v_ref_possible.itervalues())
        #print 'v_ref global=', self.pose_msg.v_ref #For debugging

        if self.pose_msg != self.prev_pose_msg and self.pose_initialized:
            self.cbPose(self.pose_msg)

    def updateWheelsCmdExecuted(self, msg_wheels_cmd):
        self.wheels_cmd_executed = msg_wheels_cmd

    def updateActuatorLimits(self, msg_actuator_limits):
        self.actuator_limits = msg_actuator_limits
        rospy.logdebug("actuator limits updated to: ")
        rospy.logdebug("actuator_limits.v: " + str(self.actuator_limits.v))
        rospy.logdebug("actuator_limits.omega: " +
str(self.actuator_limits.omega))
        msg_actuator_limits_received = BoolStamped()
        msg_actuator_limits_received.data = True

self.pub_actuator_limits_received.publish(msg_actuator_limits_received)

    def sendStop(self):
        # Send stop command
        car_control_msg = Twist2DStamped()
        car_control_msg.v = 0.0
        car_control_msg.omega = 0.0
        self.publishCmd(car_control_msg)

    def custom_shutdown(self):
        rospy.loginfo("[%s] Shutting down..." % self.node_name)

        # Stop listening
        self.sub_lane_reading.unregister()
        self.sub_obstacle_avoidance_pose.unregister()
        self.sub_obstacle_detected.unregister()
        self.sub_intersection_navigation_pose.unregister()
        self.sub_wheels_cmd_executed.unregister()
        self.sub_actuator_limits.unregister()
        self.sub_switch.unregister()
        self.sub_fsm_mode.unregister()
```

```
        # Send stop command
        car_control_msg = Twist2DStamped()
        car_control_msg.v = 0.0
        car_control_msg.omega = 0.0
        self.publishCmd(car_control_msg)

        rospy.sleep(0.5)    #To make sure that it gets published.
        rospy.loginfo("[%s] Shutdown" %self.node_name)

    def publishCmd(self, car_cmd_msg):
        self.pub_car_cmd.publish(car_cmd_msg)

    def cbPose(self, pose_msg):
        self.lane_reading = pose_msg

        # Calculating the delay image processing took
        timestamp_now = rospy.Time.now()
        image_delay_stamp = timestamp_now - self.lane_reading.header.stamp

        # delay from taking the image until now in seconds
        image_delay = image_delay_stamp.secs + image_delay_stamp.nsecs / 1e9

        prev_cross_track_err = self.cross_track_err
        prev_heading_err = self.heading_err

        self.cross_track_err = pose_msg.d - self.d_offset
        self.heading_err = pose_msg.phi

        car_control_msg = Twist2DStamped()
        car_control_msg.header = pose_msg.header

        car_control_msg.v = pose_msg.v_ref

        if car_control_msg.v > self.actuator_limits.v:
            car_control_msg.v = self.actuator_limits.v

        if math.fabs(self.cross_track_err) > self.d_thres:
            rospy.logerr("inside threshold ")
            self.cross_track_err = self.cross_track_err /
math.fabs(self.cross_track_err) * self.d_thres

        currentMillis = int(round(time.time() * 1000))
```

```
        if self.last_ms is not None:
            dt = (currentMillis - self.last_ms) / 1000.0
            self.cross_track_integral += self.cross_track_err * dt
            self.heading_integral += self.heading_err * dt

        if self.cross_track_integral > self.cross_track_integral_top_cutoff:
            self.cross_track_integral = self.cross_track_integral_top_cutoff
        if self.cross_track_integral <
self.cross_track_integral_bottom_cutoff:
            self.cross_track_integral =
self.cross_track_integral_bottom_cutoff

        if self.heading_integral > self.heading_integral_top_cutoff:
            self.heading_integral = self.heading_integral_top_cutoff
        if self.heading_integral < self.heading_integral_bottom_cutoff:
            self.heading_integral = self.heading_integral_bottom_cutoff

        if abs(self.cross_track_err) <= 0.011:  # TODO: replace '<= 0.011' by
'< delta_d' (but delta_d might need to be sent by the lane_filter_node.py or
even lane_filter.py)
            self.cross_track_integral = 0
        if abs(self.heading_err) <= 0.051:  # TODO: replace '<= 0.051' by '<
delta_phi' (but delta_phi might need to be sent by the lane_filter_node.py or
even lane_filter.py)
            self.heading_integral = 0
        if np.sign(self.cross_track_err) != np.sign(prev_cross_track_err):  #
sign of error changed => error passed zero
            self.cross_track_integral = 0
        if np.sign(self.heading_err) != np.sign(prev_heading_err):  # sign of
error changed => error passed zero
            self.heading_integral = 0
        if self.wheels_cmd_executed.vel_right == 0 and
self.wheels_cmd_executed.vel_left == 0:  # if actual velocity sent to the
motors is zero
            self.cross_track_integral = 0
            self.heading_integral = 0

        omega_feedforward = car_control_msg.v * pose_msg.curvature_ref
        if self.main_pose_source == "lane_filter" and not
self.use_feedforward_part:
            omega_feedforward = 0
```

```
        # Scale the parameters linear such that their real value is at
0.22m/s TODO do this nice that  * (0.22/self.v_bar)
        omega = self.k_d * (0.22/self.v_bar) * self.cross_track_err +
self.k_theta * (0.22/self.v_bar) * self.heading_err
        omega += (omega_feedforward)




        # check if nominal omega satisfies min radius, otherwise constrain it
to minimal radius
        if math.fabs(omega) > car_control_msg.v / self.min_radius:
            if self.last_ms is not None:
                self.cross_track_integral -= self.cross_track_err * dt
                self.heading_integral -= self.heading_err * dt
            omega = math.copysign(car_control_msg.v / self.min_radius, omega)

        if not self.fsm_state == "SAFE_JOYSTICK_CONTROL":
            # apply integral correction (these should not affect radius,
hence checked afterwards)
            omega -= self.k_Id * (0.22/self.v_bar) *
self.cross_track_integral
            omega -= self.k_Iphi * (0.22/self.v_bar) * self.heading_integral

        if car_control_msg.v == 0:
            omega = 0
        else:
        # check if velocity is large enough such that car can actually
execute desired omega
            if car_control_msg.v - 0.5 * math.fabs(omega) * 0.1 < 0.065:
                car_control_msg.v = 0.065 + 0.5 * math.fabs(omega) * 0.1




        # apply magic conversion factors
        car_control_msg.v = car_control_msg.v * self.velocity_to_m_per_s
        car_control_msg.omega = omega * self.omega_to_rad_per_s

        omega = car_control_msg.omega
        if omega > self.omega_max: omega = self.omega_max
        if omega < self.omega_min: omega = self.omega_min
        omega += self.omega_ff
```

```
            car_control_msg.omega = omega
            self.publishCmd(car_control_msg)
            self.last_ms = currentMillis



    if __name__ == "__main__":

        rospy.init_node("lane_controller_node", anonymous=False)  # adapted to
    sonjas default file

        lane_control_node = lane_controller()
    rospy.spin()
```

2. Line detector

```
import cv2
import numpy as np
import rospy
from sensor_msgs.msg import Image, CompressedImage
from std_msgs.msg import Float32, Bool
from cv_bridge import CvBridge, CvBridgeError
#from duckietown_msgs.msg import ObstacleImageDetection,
ObstacleImageDetectionList, ObstacleType, Rect, BoolStamped
import sys
import threading
from count_turns import TurnCounter


class Matcher:
    STOP1 = [np.array(x, np.uint8) for x in [[0,140,100], [15, 255,255]] ]
    STOP2 = [np.array(x, np.uint8) for x in [[165,140,100], [180, 255, 255]] ]
    LINE = [np.array(x, np.uint8) for x in [[25,100,150], [35, 255, 255]] ]

    def get_filtered_contours(self,img, contour_type):
        hsv_img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

        if contour_type == "STOP1":
            frame_threshed = cv2.inRange(hsv_img, self.STOP1[0], self.STOP1[1])
            ret,thresh = cv2.threshold(frame_threshed,22,255,0)
        elif contour_type == "STOP2":
            frame_threshed = cv2.inRange(hsv_img, self.STOP2[0], self.STOP2[1])
            ret,thresh = cv2.threshold(frame_threshed,22,255,0)
```

```
        elif contour_type == "LINE":
            frame_threshed = cv2.inRange(hsv_img, self.LINE[0], self.LINE[1])
            ret,thresh = cv2.threshold(frame_threshed,35,255,0)
        else:
            return  []

        filtered_contours = []

        contours, hierarchy = cv2.findContours(\
                thresh,cv2.RETR_CCOMP,cv2.CHAIN_APPROX_SIMPLE)
        contour_area = [ (cv2.contourArea(c), (c) ) for c in contours]
        contour_area = sorted(contour_area,reverse=True, key=lambda x: x[0])

        height,width = img.shape[:2]
        for (area,(cnt)) in contour_area:
        # plot box around contour
            x,y,w,h = cv2.boundingRect(cnt)
            box = (x,y,w,h)
            d =  0.5*(x-width/2)**2 + (y-height)**2
            if not(h>15 and w >15 and d  < 120000):
                    continue
            mask = np.zeros(thresh.shape,np.uint8)
            cv2.drawContours(mask,[cnt],0,255,-1)
            mean_val = cv2.mean(img,mask = mask)
            aspect_ratio = float(w)/h
            filtered_contours.append( (area, (cnt, box, d, aspect_ratio, mean_val,
area)) )
        return filtered_contours


    def contour_match(self, img):
        '''
        Returns 1. Image with bounding boxes added
                2. an ObstacleImageDetectionList
        '''



        height,width = img.shape[:2]
        cv2.rectangle(img, (0, 0) , (width,height/3), (0,0,0),thickness=-5)
        cv2.rectangle(img, (0, 0) , (width/5,height), (0,0,0),thickness=-5)
        cv2.rectangle(img, (4*width/5, 0) , (width,height), (0,0,0),thickness=-5)

        # get filtered contours
```

```
        stop1 = self.get_filtered_contours(img, "STOP1")
        stop2 = self.get_filtered_contours(img, "STOP2")
        line = self.get_filtered_contours(img, "LINE")

        all_contours = stop1 + stop2
        all_contours= sorted(all_contours,reverse=True, key=lambda x: x[0])

        i = 0
        center = -1
        if len(all_contours) > 0:
            area, (cnt, box, ds, aspect_ratio, mean_color, area)  = all_contours[0]

            # plot box around contour
            x,y,w,h = box
            font = cv2.FONT_HERSHEY_SIMPLEX
            cv2.putText(img,"stop line", (x,y), font, 0.5,mean_color,4)
            cv2.rectangle(img,(x,y),(x+w,y+h), mean_color,2)

            #center =  (x + w ) /float(width)

            test = x+w

        if len(all_contours)> 0 and len(line)> 0:
            for area, (cnt, box, ds, aspect_ratio, mean_color, area) in  line:

                # plot box around contour
                x,y,w,h = box
                font = cv2.FONT_HERSHEY_SIMPLEX
                val = (test - x)/float(w)
                if abs(val) > 0.75: continue

                #cv2.putText(img,"%s" % val, (x,y), font, 1.0, (255)*3 ,4)
                cv2.putText(img,"servo line", (x,y), font, 0.5,mean_color,4)
                cv2.rectangle(img,(x,y),(x+w,y+h), mean_color,2)

                center =  (x + w ) /float(width)
                break

        return img, center

class StaticObjectDetectorNode:
    def __init__(self):
        self.name = 'static_object_detector_node'
```

```
        self.tm = Matcher()
        self.active = False
        self.thread_lock = threading.Lock()
        self.turn_counter = TurnCounter()

        self.pub_ibvs = rospy.Publisher("~ibvs", Float32, queue_size=1)
        self.sub_image = rospy.Subscriber("~image_compressed", CompressedImage,
self.cbImage, queue_size=1)
        self.pub_image = rospy.Publisher("~servo_image", Image, queue_size=1)
        self.pub_turns = rospy.Publisher("~turned", Bool, queue_size=1)
        self.bridge = CvBridge()

        rospy.loginfo("[%s] Initialized." %(self.name))

    def cbSwitch(self,switch_msg):
        self.active = switch_msg.data

    def cbImage(self,image_msg):
        thread = threading.Thread(target=self.processImage,args=(image_msg,))
        thread.setDaemon(True)
        thread.start()

    def processImage(self, image_msg):
        if not self.thread_lock.acquire(False):
            return

        np_arr = np.fromstring(image_msg.data, np.uint8)
        #image_cv=self.bridge.imgmsg_to_cv2(image_msg,"bgr8")
        image_cv = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)

        img, center = self.tm.contour_match(image_cv)
        crossing, turns = self.turn_counter.cbmsg(center)
        if crossing:
            # only trigger if it's been awhile
            rospy.loginfo("Crossing.  %d turn" % turns)
            self.pub_turns.publish(Bool(data=True))
        self.pub_ibvs.publish(Float32(data=center))

        height,width = img.shape[:2]
        """
        try:
            self.pub_image.publish(self.bridge.cv2_to_imgmsg(img, "bgr8"))
```

```
        except CvBridgeError as e:
            print(e)
        """
        self.thread_lock.release()

if __name__=="__main__":
    rospy.init_node('arii')
    node = StaticObjectDetectorNode()
    rospy.spin()
```

3. Closed Loop Intersection control

```
Import rospy
from duckietown_msgs.msg import FSMState, BoolStamped, WheelsCmdStamped
from std_msgs.msg import String, Int16, Float32 #Imports msg
import copy

class OpenLoopIntersectionNode(object):
    def __init__(self):
        # Save the name of the node
        self.node_name = rospy.get_name()
        self.mode = None
        self.turn_type = -1
        self.in_lane = False
        self.ibvs_data = -1

        ibvs_topic = "/arii/ibvs"
        self.sub_ibvs = rospy.Subscriber(ibvs_topic, Float32, self.cbIbvs,
queue_size=1)
        self.pub_cmd = rospy.Publisher("~wheels_cmd",WheelsCmdStamped,queue_size=1)
        self.pub_done =
rospy.Publisher("~intersection_done",BoolStamped,queue_size=1)

        self.rate = rospy.Rate(30)

        # Subscribers
        self.sub_in_lane = rospy.Subscriber("~in_lane", BoolStamped, self.cbInLane,
queue_size=1)
        self.sub_turn_type = rospy.Subscriber("~turn_type", Int16, self.cbTurnType,
queue_size=1)
        self.sub_mode = rospy.Subscriber("~mode", FSMState, self.cbFSMState,
queue_size=1)
```

```
    def cbTurnType(self,msg):
        self.turn_type = msg.data

    def cbIbvs (self,data):
        self.ibvs_data = data.data

    def cbInLane(self,msg):
        self.in_lane = msg.data

    def cbFSMState(self,msg):
        if (not self.mode == "INTERSECTION_CONTROL") and msg.state ==
"INTERSECTION_CONTROL":
            self.mode = msg.state
            rospy.loginfo("[%s] %s triggered. turn_type: %s"
%(self.node_name,self.mode,self.turn_type))
            self.servo(self.turn_type)

        self.mode = msg.state


    def servo(self, turn_type):
        #move forward

        wheels_cmd_msg = WheelsCmdStamped()
        end_time = rospy.Time.now() + rospy.Duration(0.5)
        while rospy.Time.now() < end_time:
            wheels_cmd_msg.header.stamp = rospy.Time.now()
            wheels_cmd_msg.vel_left = .4 # go straight
            wheels_cmd_msg.vel_right = .4
            self.pub_cmd.publish(wheels_cmd_msg)

        wheels_cmd_msg = WheelsCmdStamped()
        wheels_cmd_msg.header.stamp = rospy.Time.now()
        while not rospy.is_shutdown():
            #self.mode == "INTERSECTION_CONTROL": # If not in the mode anymore,
return
            angle_direction = (0.5 - self.ibvs_data)
            wheels_cmd_msg = WheelsCmdStamped()
            wheels_cmd_msg.header.stamp = rospy.Time.now()
            wheels_cmd_msg.vel_left = 0
            wheels_cmd_msg.vel_right = 0
            gain = 0.5
            done = False
```

```
            if abs(angle_direction) < 0.1 or self.ibvs_data == -1:
                if self.ibvs_data == -1:
                    rospy.loginfo("nothing detected!")
                    wheels_cmd_msg.vel_left = .2 # go straight
                    wheels_cmd_msg.vel_right = -.2
                else:
                    rospy.loginfo("already centered!")

                    done= True
                    wheels_cmd_msg.vel_left = .4 # go straight
                    wheels_cmd_msg.vel_right = .4
            else:
                if angle_direction > 0:
                    wheels_cmd_msg.vel_left = gain*abs(angle_direction)
                    rospy.loginfo("turning left %f " % angle_direction)
                else:
                    wheels_cmd_msg.vel_right = gain*abs(angle_direction)
                    rospy.loginfo("turning right %f " % angle_direction)
            self.pub_cmd.publish(wheels_cmd_msg)
            if self.in_lane:# and done==True:
                self.pub_done.publish(msg)

            self.rate.sleep()


    def publishDoneMsg(self):
        if self.mode == "INTERSECTION_CONTROL":
            msg = BoolStamped()
            msg.header.stamp = rospy.Time.now()
            msg.data = True
            self.pub_done.publish(msg)
            rospy.loginfo("[%s] interesction_done!" %(self.node_name))

    def on_shutdown(self):
        rospy.loginfo("[%s] Shutting down." %(self.node_name))

if __name__ == '__main__':
    # Initialize the node with rospy
    rospy.init_node('open_loop_intersection_node', anonymous=False)

    # Create the NodeName object
    node = OpenLoopIntersectionNode()
```

```
    # Setup proper shutdown behavior
    rospy.on_shutdown(node.on_shutdown)
    # Keep it spinning to keep the node alive
    rospy.spin()
```

4. Open Loop Intersection Control

```
import rospy
from duckietown_msgs.msg import FSMState, BoolStamped, Twist2DStamped, LanePose,
StopLineReading
from std_srvs.srv import EmptyRequest, EmptyResponse, Empty
from std_msgs.msg import String, Int16 #Imports msg
import copy

class OpenLoopIntersectionNode(object):


    def updateParams(self,event):
        self.maneuvers[0] = self.getManeuver("turn_left")
        self.maneuvers[1] = self.getManeuver("turn_forward")
        self.maneuvers[2] = self.getManeuver("turn_right")



    def __init__(self):
        # Save the name of the node
        self.node_name = rospy.get_name()
        self.mode = None
        self.turn_type = -1
        self.in_lane = False
        self.lane_pose = LanePose()
        self.stop_line_reading = StopLineReading()





        self.trajectory_reparam = rospy.get_param("~trajectory_reparam",1)
        self.pub_cmd = rospy.Publisher("~car_cmd",Twist2DStamped,queue_size=1)
        self.pub_done =
rospy.Publisher("~intersection_done",BoolStamped,queue_size=1)

        # Construct maneuvers
```

```
        self.maneuvers = dict()

        self.maneuvers[0] = self.getManeuver("turn_left")
        self.maneuvers[1] = self.getManeuver("turn_forward")
        self.maneuvers[2] = self.getManeuver("turn_right")
        # self.maneuvers[-1] = self.getManeuver("turn_stop")

        self.srv_turn_left = rospy.Service("~turn_left", Empty, self.cbSrvLeft)
        self.srv_turn_right = rospy.Service("~turn_right", Empty, self.cbSrvRight)
        self.srv_turn_forward = rospy.Service("~turn_forward", Empty,
self.cbSrvForward)

        self.rate = rospy.Rate(30)

        # Subscribers
        self.sub_in_lane = rospy.Subscriber("~in_lane", BoolStamped, self.cbInLane,
queue_size=1)
        self.sub_turn_type = rospy.Subscriber("~turn_type", Int16, self.cbTurnType,
queue_size=1)
        self.sub_mode = rospy.Subscriber("~mode", FSMState, self.cbFSMState,
queue_size=1)
        self.sub_lane_pose = rospy.Subscriber("~lane_pose", LanePose,
self.cbLanePose, queue_size=1)
        self.sub_stop_line = rospy.Subscriber("~stop_line_reading",
StopLineReading, self.cbStopLine, queue_size=1)

        self.params_update = rospy.Timer(rospy.Duration.from_sec(1.0),
self.updateParams)


    def cbSrvLeft(self,req):
        self.trigger(0)
        return EmptyResponse()

    def cbSrvForward(self,req):
        self.trigger(1)
        return EmptyResponse()

    def cbSrvRight(self,req):
        self.trigger(2)
        return EmptyResponse()
```

```
    def getManeuver(self,param_name):
        param_list = rospy.get_param("~%s"%(param_name))
        # rospy.loginfo("PARAM_LIST:%s" %param_list)
        maneuver = list()
        for param in param_list:
            maneuver.append((param[0],Twist2DStamped(v=param[1],omega=param[2])))
        # rospy.loginfo("MANEUVER:%s" %maneuver)
        return maneuver

    def cbTurnType(self,msg):
        if self.mode == "INTERSECTION_CONTROL":
            self.turn_type = msg.data #Only listen if in INTERSECTION_CONTROL mode
            self.trigger(self.turn_type)

    def cbLanePose(self,msg):
        self.lane_pose = msg

    def cbStopLine(self,msg):
        self.stop_line_reading = msg

        # TODO remove in lane it is now handled by the logic_gate_node
    def cbInLane(self,msg):
        self.in_lane = msg.data

    def cbFSMState(self,msg):
        if (not self.mode == "INTERSECTION_CONTROL") and msg.state ==
"INTERSECTION_CONTROL":
            # Switch into INTERSECTION_CONTROL mode
            rospy.loginfo("[%s] %s triggered." %(self.node_name,self.mode))
            start = rospy.Time.now()
            current = rospy.Time.now()
            while current.secs - start.secs < 0.5:
                current = rospy.Time.now()
                self.trigger(-1)
        self.mode = msg.state
        self.turn_type = -1 #Reset turn_type at mode change

    def publishDoneMsg(self):
        msg = BoolStamped()
        msg.header.stamp = rospy.Time.now()
        msg.data = True
        self.pub_done.publish(msg)
        rospy.loginfo("[%s] interesction_done!" %(self.node_name))
```

```python
    def update_trajectory(self,turn_type):
        rospy.loginfo("updating trajectory: distance from stop_line=%s, lane_pose_phi = %s", self.stop_line_reading.stop_line_point.x,  self.lane_pose.phi)
        first_leg = (self.maneuvers[turn_type]).pop(0)
        exec_time = first_leg[0];
        car_cmd   = first_leg[1];
        new_exec_time = exec_time + self.stop_line_reading.stop_line_point.x/car_cmd.v
        rospy.loginfo("old exec_time = %s, new_exec_time = %s" ,exec_time, new_exec_time)
        ###### warning this next line is because of wrong inverse kinematics - remove the 10s after it's fixed
        new_car_cmd = Twist2DStamped(v=car_cmd.v,omega=10*(car_cmd.omega/10 - self.lane_pose.phi/new_exec_time))
        new_first_leg = [new_exec_time,new_car_cmd]
        print "old car command"
        print car_cmd
        print "new_car_command"
        print new_car_cmd
        self.maneuvers[turn_type].insert(0,new_first_leg)

    def trigger(self,turn_type):
        if turn_type == -1: #Wait. Publish stop command. Does not publish done.
            cmd = Twist2DStamped(v=0.0,omega=0.0)
            cmd.header.stamp = rospy.Time.now()
            self.pub_cmd.publish(cmd)
            return

        if (self.trajectory_reparam):
            self.update_trajectory(turn_type)

        published_already = False
        for index, pair in enumerate(self.maneuvers[turn_type]):
            cmd = copy.deepcopy(pair[1])
            start_time = rospy.Time.now()
            end_time = start_time + rospy.Duration.from_sec(pair[0])
            while rospy.Time.now() < end_time:
                if not self.mode == "INTERSECTION_CONTROL": # If not in the mode anymore, return
                    return
                cmd.header.stamp = rospy.Time.now()
                self.pub_cmd.publish(cmd)
```

```
                if index > 1:
                    # See if need to publish interesction_done
                    if self.in_lane and not (published_already):
                        published_already = True
                        self.publishDoneMsg()
                        return
                self.rate.sleep()
            # Done with the sequence
            if not published_already:
                self.publishDoneMsg()

    def on_shutdown(self):
        rospy.loginfo("[%s] Shutting down." %(self.node_name))

if __name__ == '__main__':
    # Initialize the node with rospy
    rospy.init_node('open_loop_intersection_node', anonymous=False)

    # Create the NodeName object
    node = OpenLoopIntersectionNode()

    # Setup proper shutdown behavior
    rospy.on_shutdown(node.on_shutdown)
    # Keep it spinning to keep the node alive
    rospy.spin()
```

5. Intersection Node

```
import rospy
import numpy as np
from duckietown_msgs.msg import TurnIDandType, FSMState, BoolStamped, LanePose,
Pose2DStamped, Twist2DStamped, TurnIDandType
from std_msgs.msg import Float32, Int16, Bool, String
from geometry_msgs.msg import Point, PoseStamped, Pose, PointStamped
from nav_msgs.msg import Path
import time
import math
import json

class UnicornIntersectionNode(object):
    def __init__(self):
        self.node_name = "Unicorn Intersection Node"

        ## setup Parameters
```

```
        self.setupParams()

        ## Internal variables
        self.state = "JOYSTICK_CONTROL"
        self.active = False
        self.turn_type = -1
        self.tag_id = -1
        self.forward_pose = False


        ## Subscribers
        #self.sub_turn_type = rospy.Subscriber("~turn_type", Int16,
self.cbTurnType)
        self.sub_turn_type = rospy.Subscriber("~turn_id_and_type", TurnIDandType,
self.cbTurnType)
        self.sub_fsm = rospy.Subscriber("~fsm_state", FSMState, self.cbFSMState)
        self.sub_int_go = rospy.Subscriber("~intersection_go", BoolStamped,
self.cbIntersectionGo)
        self.sub_lane_pose = rospy.Subscriber("~lane_pose_in", LanePose,
self.cbLanePose)
        self.sub_switch = rospy.Subscriber("~switch",BoolStamped, self.cbSwitch,
queue_size=1)

        ## Publisher
        self.pub_int_done = rospy.Publisher("~intersection_done", BoolStamped,
queue_size=1)
        self.pub_LF_params = rospy.Publisher("~lane_filter_params", String,
queue_size=1)
        self.pub_lane_pose = rospy.Publisher("~lane_pose_out", LanePose,
queue_size=1)
        self.pub_int_done_detailed = rospy.Publisher("~intersection_done_detailed",
TurnIDandType, queue_size=1)

        ## update Parameters timer
        self.params_update = rospy.Timer(rospy.Duration.from_sec(1.0),
self.updateParams)


    def cbLanePose(self, msg):
        if self.forward_pose: self.pub_lane_pose.publish(msg)

    def changeLFParams(self, params, reset_time):
        data = {"params": params, "time": reset_time}
```

```
        msg = String()
        msg.data = json.dumps(data)
        self.pub_LF_params.publish(msg)

    def cbIntersectionGo(self, msg):

        if not self.active:
            return

        if not msg.data: return

        while self.turn_type == -1:
            if not self.active:
                return
            rospy.loginfo("Requested to start intersection, but we do not see an
april tag yet.")
            rospy.sleep(2)

        tag_id = self.tag_id
        turn_type = self.turn_type

        sleeptimes = [self.time_left_turn, self.time_straight_turn,
self.time_right_turn]
        LFparams = [self.LFparams_left, self.LFparams_straight,
self.LFparams_right]
        omega_ffs = [self.ff_left, self.ff_straight, self.ff_right]
        omega_maxs = [self.omega_max_left, self.omega_max_straight,
self.omega_max_right]
        omega_mins = [self.omega_min_left, self.omega_min_straight,
self.omega_min_right]

        self.changeLFParams(LFparams[turn_type], sleeptimes[turn_type]+1.0)
        rospy.set_param("~lane_controller/omega_ff", omega_ffs[turn_type])
        rospy.set_param("~lane_controller/omega_max", omega_maxs[turn_type])
        rospy.set_param("~lane_controller/omega_min", omega_mins[turn_type])
        # Waiting for LF to adapt to new params
        rospy.sleep(1)

        rospy.loginfo("Starting intersection control - driving to " +
str(turn_type))
        self.forward_pose = True

        rospy.sleep(sleeptimes[turn_type])
```

```python
        self.forward_pose = False
        rospy.set_param("~lane_controller/omega_ff", 0)
        rospy.set_param("~lane_controller/omega_max", 999)
        rospy.set_param("~lane_controller/omega_min", -999)

        # Publish intersection done
        msg_done = BoolStamped()
        msg_done.data = True
        self.pub_int_done.publish(msg_done)

        # Publish intersection done detailed
        msg_done_detailed = TurnIDandType()
        msg_done_detailed.tag_id = tag_id
        msg_done_detailed.turn_type = turn_type
        self.pub_int_done_detailed.publish(msg_done_detailed)



    def cbFSMState(self, msg):
        if self.state != msg.state and msg.state == "INTERSECTION_COORDINATION":
            self.turn_type = -1

        self.state = msg.state

    def cbSwitch(self, switch_msg):
        self.active = switch_msg.data


    def cbTurnType(self, msg):
        self.tag_id = msg.tag_id
        if self.turn_type == -1: self.turn_type = msg.turn_type
        if self.debug_dir != -1: self.turn_type = self.debug_dir

    def setupParams(self):
        self.time_left_turn = self.setupParam("~time_left_turn", 2)
        self.time_straight_turn = self.setupParam("~time_straight_turn", 2)
        self.time_right_turn = self.setupParam("~time_right_turn", 2)
        self.ff_left = self.setupParam("~ff_left", 1.5)
        self.ff_straight = self.setupParam("~ff_straight", 0)
        self.ff_right = self.setupParam("~ff_right", -1)
        self.LFparams_left = self.setupParam("~LFparams_left", 0)
        self.LFparams_straight = self.setupParam("~LFparams_straight", 0)
```

```
        self.LFparams_right = self.setupParam("~LFparams_right", 0)
        self.omega_max_left = self.setupParam("~omega_max_left", 999)
        self.omega_max_straight = self.setupParam("~omega_max_straight", 999)
        self.omega_max_right = self.setupParam("~omega_max_right", 999)
        self.omega_min_left = self.setupParam("~omega_min_left", -999)
        self.omega_min_straight = self.setupParam("~omega_min_straight", -999)
        self.omega_min_right = self.setupParam("~omega_min_right", -999)

        self.debug_dir = self.setupParam("~debug_dir", -1)

    def updateParams(self,event):
        self.time_left_turn = rospy.get_param("~time_left_turn")
        self.time_straight_turn = rospy.get_param("~time_straight_turn")
        self.time_right_turn = rospy.get_param("~time_right_turn")
        self.ff_left = rospy.get_param("~ff_left")
        self.ff_straight = rospy.get_param("~ff_straight")
        self.ff_right = rospy.get_param("~ff_right")
        self.LFparams_left = rospy.get_param("~LFparams_left")
        self.LFparams_straight = rospy.get_param("~LFparams_straight")
        self.LFparams_right = rospy.get_param("~LFparams_right")
        self.omega_max_left = rospy.get_param("~omega_max_left")
        self.omega_max_straight = rospy.get_param("~omega_max_straight")
        self.omega_max_right = rospy.get_param("~omega_max_right")
        self.omega_min_left = rospy.get_param("~omega_min_left")
        self.omega_min_straight = rospy.get_param("~omega_min_straight")
        self.omega_min_right = rospy.get_param("~omega_min_right")

        self.debug_dir = rospy.get_param("~debug_dir")


    def setupParam(self,param_name,default_value):
        value = rospy.get_param(param_name,default_value)
        rospy.set_param(param_name,value) #Write to parameter server for
transparancy
        rospy.loginfo("[%s] %s = %s " %(self.node_name,param_name,value))
        return value

    def onShutdown(self):
        rospy.loginfo("[UnicornIntersectionNode] Shutdown.")

if __name__ == '__main__':
    rospy.init_node('unicorn_intersection_node',anonymous=False)
    unicorn_intersection_node = UnicornIntersectionNode()
```

```
rospy.on_shutdown(unicorn_intersection_node.onShutdown)
rospy.spin()
```

DUCKIETOWN SIMULATOR

1. Train Imitation Learning Model

```
import time
import random
import argparse
import math
import json
from functools import reduce
import operator

import numpy as np
import torch
import torch.optim as optim

from utils.env import launch_env
from utils.wrappers import NormalizeWrapper, ImgWrapper, \
    DtRewardWrapper, ActionWrapper, ResizeWrapper
from utils.teacher import PurePursuitExpert

from imitation.pytorch.model import Model


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def _train(args):
    env = launch_env()
    env = ResizeWrapper(env)
    env = NormalizeWrapper(env)
    env = ImgWrapper(env)
    env = ActionWrapper(env)
    env = DtRewardWrapper(env)
    print("Initialized Wrappers")

    observation_shape = (None, ) + env.observation_space.shape
    action_shape = (None, ) + env.action_space.shape

    # Create an imperfect demonstrator
    expert = PurePursuitExpert(env=env)

    observations = []
    actions = []
```

```
    # let's collect our samples
    for episode in range(0, args.episodes):
        print("Starting episode", episode)
        for steps in range(0, args.steps):
            # use our 'expert' to predict the next action.
            action = expert.predict(None)
            observation, reward, done, info = env.step(action)
            observations.append(observation)
            actions.append(action)
        env.reset()
    env.close()

    actions = np.array(actions)
    observations = np.array(observations)

    model = Model(action_dim=2, max_action=1.)
    model.train().to(device)

    # weight_decay is L2 regularization, helps avoid overfitting
    optimizer = optim.SGD(
        model.parameters(),
        lr=0.0004,
        weight_decay=1e-3
    )

    avg_loss = 0
    for epoch in range(args.epochs):
        optimizer.zero_grad()

        batch_indices = np.random.randint(0, observations.shape[0],
(args.batch_size))
        obs_batch =
torch.from_numpy(observations[batch_indices]).float().to(device)
        act_batch = torch.from_numpy(actions[batch_indices]).float().to(device)

        model_actions = model(obs_batch)

        loss = (model_actions - act_batch).norm(2).mean()
        loss.backward()
        optimizer.step()

        loss = loss.data[0]
        avg_loss = avg_loss * 0.995 + loss * 0.005
```

```
        print('epoch %d, loss=%.3f' % (epoch, avg_loss))

        # Periodically save the trained model
        if epoch % 200 == 0:
            torch.save(model.state_dict(), 'imitation/pytorch/models/imitate.pt')


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--seed", default=1234, type=int, help="Sets Gym, TF, and
Numpy seeds")
    parser.add_argument("--episodes", default=3, type=int, help="Number of epsiodes
for experts")
    parser.add_argument("--steps", default=50, type=int, help="Number of steps per
episode")
    parser.add_argument("--batch-size", default=32, type=int, help="Training batch
size")
    parser.add_argument("--epochs", default=1, type=int, help="Number of training
epochs")
    parser.add_argument("--model-directory", default="models/", type=str,
help="Where to save models")

    args = parser.parse_args()

    _train(args)
```

2. Apply Imitation Learning

```
import time
import sys
import argparse
import math

import torch

import numpy as np
import gym

from utils.env import launch_env
from utils.wrappers import NormalizeWrapper, ImgWrapper, \
    DtRewardWrapper, ActionWrapper, ResizeWrapper
```

```python
from utils.teacher import PurePursuitExpert

from imitation.pytorch.model import Model

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def _enjoy():
    model = Model(action_dim=2, max_action=1.)

    try:
        state_dict = torch.load('trained_models/imitate.pt',
map_location=device)
        model.load_state_dict(state_dict)
    except:
        print('failed to load model')
        exit()

    model.eval().to(device)

    env = launch_env()
    env = ResizeWrapper(env)
    env = NormalizeWrapper(env)
    env = ImgWrapper(env)
    env = ActionWrapper(env)
    env = DtRewardWrapper(env)

    obs = env.reset()

    while True:
        obs = torch.from_numpy(obs).float().to(device).unsqueeze(0)

        action = model(obs)
        action = action.squeeze().data.cpu().numpy()

        obs, reward, done, info = env.step(action)
        env.render()

        if done:
            if reward < 0:
                print('*** FAILED ***')
                time.sleep(0.7)

            obs = env.reset()
```

```python
                env.render()

if __name__ == '__main__':
    _enjoy()
```