Autonomous Vehicles

Project 1 Team 2

Title

1. Lane following: For this project Duckiebot was programmed to follow a miniature street course marked by white and yellow pavement lines
2. Imitation Learning using Duckietown Simulator

Lane Detection Code

```python
import numpy as np
import cv2

from .line_detector_interface import Detections, LineDetectorInterface
import duckietown_utils as dtu


class LineDetector2Dense(dtu.Configurable, LineDetectorInterface):
    def __init__(self, configuration):
        # Images to be processed
        self.bgr = np.empty(0)
        self.hsv = np.empty(0)
        self.edges = np.empty(0)

        param_names = [

            'hsv_white1',
            'hsv_white2',
            'hsv_yellow1',
            'hsv_yellow2',
            'hsv_red1',
            'hsv_red2',
            'hsv_red3',
            'hsv_red4',

            'dilation_kernel_size',
            'canny_thresholds',
            'sobel_threshold',
        ]

        dtu.Configurable.__init__(self, param_names, configuration)
```

```python
    def _colorFilter(self, color):
        # threshold colors in HSV space
        if color == 'white':
            bw = cv2.inRange(self.hsv, self.hsv_white1, self.hsv_white2)
        elif color == 'yellow':
            bw = cv2.inRange(self.hsv, self.hsv_yellow1, self.hsv_yellow2)
        elif color == 'red':
            bw1 = cv2.inRange(self.hsv, self.hsv_red1, self.hsv_red2)
            bw2 = cv2.inRange(self.hsv, self.hsv_red3, self.hsv_red4)
            bw = cv2.bitwise_or(bw1, bw2)
        else:
            raise Exception('Error: Undefined color strings...')

        # binary dilation
        kernel =
cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(self.dilation_kernel_size,
self.dilation_kernel_size))

        # refine edge for certain color
        edge_color = cv2.bitwise_and(cv2.dilate(bw, kernel), self.edges)

        return bw, edge_color

    def _lineFilter(self, bw, edge_color):
        # find gradient of the bw image
        grad_x = -cv2.Sobel(bw/255, cv2.CV_32F, 1, 0, ksize=5)
        grad_y = -cv2.Sobel(bw/255, cv2.CV_32F, 0, 1, ksize=5)
        grad_x *= (edge_color == 255)
        grad_y *= (edge_color == 255)

        # compute gradient and thresholding
        grad = np.sqrt(grad_x**2 + grad_y**2)
        roi = (grad>self.sobel_threshold)

        #print np.unique(grad)
        #print np.sum(roi)

        # turn into a list of points and normals
        roi_y, roi_x = np.nonzero(roi)
        centers = np.vstack((roi_x, roi_y)).transpose()
        normals = np.vstack((grad_x[roi], grad_y[roi])).transpose()
        normals /= np.sqrt(np.sum(normals**2, axis=1, keepdims=True))
```

```
        lines = self._synthesizeLines(centers, normals)

        return lines, normals, centers

    def _findEdge(self, gray):
        edges = cv2.Canny(gray, self.canny_thresholds[0], self.canny_thresholds[1],
apertureSize = 3)
        return edges

    def _checkBounds(self, val, bound):
        val[val<0]=0
        val[val>=bound]=bound-1
        return val

    def _synthesizeLines(self, centers, normals):
        lines = []
        if len(centers)>0:
            x1 = (centers[:,0:1] + normals[:, 1:2] * 6.).astype('int')
            y1 = (centers[:,1:2] - normals[:, 0:1] * 6.).astype('int')
            x2 = (centers[:,0:1] - normals[:, 1:2] * 6.).astype('int')
            y2 = (centers[:,1:2] + normals[:, 0:1] * 6.).astype('int')
            x1 = self._checkBounds(x1, self.bgr.shape[1])
            y1 = self._checkBounds(y1, self.bgr.shape[0])
            x2 = self._checkBounds(x2, self.bgr.shape[1])
            y2 = self._checkBounds(y2, self.bgr.shape[0])
            lines = np.hstack([x1, y1, x2, y2])
        return lines

    def detectLines(self, color):
        bw, edge_color = self._colorFilter(color)
        lines, normals, centers = self._lineFilter(bw, edge_color)
        return Detections(lines=lines, normals=normals, area=bw, centers=centers)

    def setImage(self, bgr):
        self.bgr = np.copy(bgr)
        self.hsv = cv2.cvtColor(bgr, cv2.COLOR_BGR2HSV)
        self.edges = self._findEdge(self.bgr)

    def getImage(self):
        return self.bgr
```

Lane following Code

Important parameters:

v_bar: 0.23

type: float

desc: Nominal linear velocity (m/s).

k_theta: -1.5

type: float

desc: Proportional gain for theta.

k_d: -2

type: float

desc: Proportional gain for d.

d_thres: 0.3490

type: float

desc: Cap for error in d.

theta_thres: 0.523

type: float

desc: Maximum desired theta.

d_offset: 0.00

type: float

desc: A configurable offset from the lane position.

Gain: 0.75

```
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)     # for this topic, no remapping is required, since it is directly
defined in the namespace lane_controller_node by the fsm_node (via it's
default.yaml file)


        self.sub_stop_line =
rospy.Subscriber("~stop_line_reading",StopLineReading, self.cbStopLineReading,
queue_size=1)     # for this topic, no remapping is required, since it is directly
defined in the namespace lane_controller_node by the fsm_node (via it's
default.yaml file)


        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

        # overwrites some of the above set default values (the ones that are
already defined in the corresponding yaml-file (see launch-file of this node))

        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)

        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

        self.velocity_to_m_per_s = 1
        #self.omega_to_rad_per_s = 0.45 * 2 * math.pi

        # 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

        #if pose_source == "obstacle_avoidance":
            # print "obstacle_avoidance pose_msg d_ref: ", input_pose_msg.d_ref
            # print "obstacle_avoidance pose_msg v_ref: ", input_pose_msg.v_ref
            # print "flag_dict[\"obstacle_detected\"]: ",
self.flag_dict["obstacle_detected"]

        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
```

```python
        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_parking_pose.unregister()
        # self.sub_fleet_planning_pose.unregister()
        # self.sub_fleet_planning_lane_following_override_active.unregister()
        # self.sub_implicit_coord_pose.unregister()
        # self.sub_implicit_coord_velocity_limit_active.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:
            self.cross_track_integral = 0
        if abs(self.heading_err) <= 0.051:
            self.heading_integral = 0
        if np.sign(self.cross_track_err) != np.sign(prev_cross_track_err):
            self.cross_track_integral = 0
        if np.sign(self.heading_err) != np.sign(prev_heading_err):
            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
        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()
```

Simulator Code

Training Imitation Learning

```
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]
```

```python
            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)
```

Applying Imitation Learning

```python
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
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()
            env.render()

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