2.1. Lane Detection:

```
def homography_ipmnorm2g(top_view_region):#Top-view Perspective
    src=np.float32([[0,0],[1,0],[0,1],[1,1]])
    H_ipmnorm2g=cv2.getPerspectiveTransform(src,np.float32(top_view_region))
    return H_ipmnorm2g

bev=np.zeros((96,96,3))
H,W=96,96
top_view_region= np.array([[50,-25],[50,25],[0,25],[0,-25]])

cam_xyz=[-5.5,0,2.5]
cam_yaw=0
cam_pitch=8
cam_roll=0

width=1920
height=1080
fov=64

focal=width/(2.0*np.tan(fov*np.pi/360.0))
K=np.identity(3) #Projection Matrix
K[0,0]=K[1,1]=focal
K[0,2]=width/2.0
K[1,2]=height/2.0
#Translation and Rotation Matrix
H_g2cam=np.array(carla.Transform(carla.Location(*cam_xyz),carla.Rotation(yaw=cam_yaw
                        ,pitch=cam_pitch,roll=cam_roll)).get_matrix())
H_g2cam=np.concatenate([H_g2cam[:3,0:2],np.expand_dims(H_g2cam[:3,3],1)],1)
#Multiplication of K and [R t]
trans_mat=np.array([[0,1,0],[0,0,-1],[1,0,0]])
temp_mat=np.matmul(trans_mat, H_g2cam)
H_g2in=np.matmul(K,temp_mat)
#Multiplication of K [R t] with top view region perspective
H_ipmnorm2g=homography_ipmnorm2g(top_view_region)
H_ipmnorm2in=np.matmul(H_g2in,H_ipmnorm2g)
#Top-view projection
S_in_inv=np.array([[1/np.float64(width),0,0],[0,1/np.float64(height),0],[0,0,1]])
M_ipm2im_norm=np.matmul(S_in_inv,H_ipmnorm2in)
```

Figure 1

```
cropped_image = img[:68, :, :]
# Convert to grayscale
gray_state_image = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2GRAY)
```

Figure 2

```
dx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
dy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

# Compute the convolutions of the image with the Sobel kernels
Ix = convolve2d(gray_image, dx, mode='same')
Iy = convolve2d(gray_image, dy, mode='same')

# Compute the magnitude of the gradients
gradient_sum = np.sqrt(Ix**2 + Iy**2)

# Set gradients below the threshold to zero
gradient_sum[gradient_sum < self.gradient_threshold] = 0
```

Figure 3

```
argmaxima=[]
for i in range(gradient_sum.shape[0]):
    row = gradient_sum[i]
    peaks, _ = find_peaks(row, distance=3)
    argmaxima.append(np.array([[peaks,i]]))
```

Figure 4

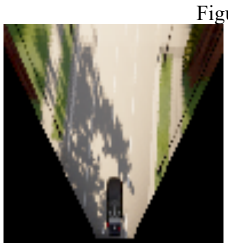


Figure 5

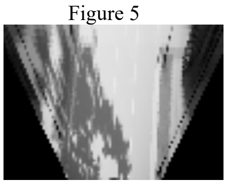


Figure 6

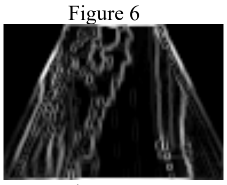


Figure 7

2.1. a. Homography Transform: The front view image is converted to Bird's eye view using the formula $\begin{matrix}x\\y\\1\end{matrix} = [R_x \;\; R_y \;\; t]^{-1}K^{-1}\begin{matrix}u\\v\\1\end{matrix}$ where $[u \; v \; 1]^T$ is the image frame coordinates and $[x \; y \; 1]^T$ is the BEV plane coordinates. The code used to transform this is given in Fig. 1. where the camera parameters and top-view region dimensions are fixed initially followed by computation of the K, [R t] matrix where K= $\begin{matrix}fx & 0 & u0\\0 & fy & v0\\0 & 0 & 1\end{matrix}$ and [R t]= $\begin{matrix}r11 & r12 & r13 & t1\\r21 & r22 & r23 & t2\\r31 & r32 & r33 & t3\end{matrix}$

2.1. b. Edge Detection: Fig. 2. shows the code snippet used in cut_gray() to crop the image in 96*68 to remove the upper part of the car and convert it to grayscale. The BEV was implemented inside cut_gray(). Fig. 3. shows the code used to compute the gradients of the image. Typically, in edge_detection() the kernels of Sobel filter were defined and convolution was performed with the image matrix, followed by computing the overall gradients as sum of squares of gradient in x and y direction. The threshold of 14 or 10 performed well for CARLA, whereas a gradient of around 25 could detect proper lanes in CarRacing(). Fig. 4. shows the code used in find_maxima_gradient_rowwise() where scipy.signal.find_peaks() was used to detect y coordinates of each gradient peak and its y, x coordinate was appended to form a 2D

```
def get_outputs(image, model, threshold):
    with torch.no_grad():
        # forward pass of the image through the modle
        outputs = model(image)

    # get all the scores
    scores = list(outputs[0]['scores'].detach().cpu().numpy())
    # index of those scores which are above a certain threshold
    thresholded_preds_inidices = [scores.index(i)
            for i in scores if i > threshold]
    thresholded_preds_count = len(thresholded_preds_inidices)
    # get the masks
    masks = (outputs[0]['masks']>0.5).squeeze().detach().cpu().numpy()
    # discard masks for objects which are below threshold
    masks = masks[:thresholded_preds_count]
    # get the bounding boxes, in (x1, y1), (x2, y2) format
    boxes = [[(int(i[0]), int(i[1])), (int(i[2]), int(i[3]))]
            for i in outputs[0]['boxes'].detach().cpu()]
    #The centroid of the object is marked with the points
    points= [((int(i[0])+int(i[2]))/2,(int(i[1])+int(i[3]))/2)
            for i in outputs[0]['boxes'].detach().cpu()]
    # discard bounding boxes below threshold value
    boxes = boxes[:thresholded_preds_count]
    points=points[:thresholded_preds_count]
    # get the classes labels
    labels = [coco_names[i] for i in outputs[0]['labels']]
    return masks, boxes, labels,points
```

```
def draw_segmentation_map(image, masks, boxes, labels,points):
    alpha = 1
    beta = 0.6 # transparency for the segmentation map
    gamma = 0 # scalar added to each sum
    for i in range(len(masks)):
        red_map = np.zeros_like(masks[i]).astype(np.uint8)
        green_map = np.zeros_like(masks[i]).astype(np.uint8)
        blue_map = np.zeros_like(masks[i]).astype(np.uint8)
        # apply a randon color mask to each object
        color = COLORS[random.randrange(0, len(COLORS))]
        red_map[masks[i] == 1],
        green_map[masks[i] == 1],
        blue_map[masks[i] == 1]  = color
        # combine all the masks into a single image
        segmentation_map = np.stack([red_map, green_map, blue_map], axis=2)
        #convert the original PIL image into NumPy format
        image = np.array(image)
        #Centroid of the objects is marked red
        for point in points:
            print(point[0])
            print(point[1])
            image[int(point[1])-10:int(point[1])+10
            ,int(point[0])-10:int(point[0])+10]=[0,0,255]
        # apply mask on the image
        cv2.addWeighted(image, alpha, segmentation_map, beta, gamma, image)
        # draw the bounding boxes around the objects
        cv2.rectangle(image, boxes[i][0], boxes[i][1], color=color,
                      thickness=2)
        # put the label text above the objects
        cv2.putText(image , labels[i], (boxes[i][0][0], boxes[i][0][1]-10),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, color,
                    thickness=2, lineType=cv2.LINE_AA)

    return image
```

```
# initialize the model
model = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True, progress=True,
                                                            num_classes=91)
# set the computation device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# load the modle on to the computation device and set to eval mode
model.to(device).eval()
```

```
#image_path = args['input']
image = Image.open('objectdetection/00385439.png').convert('RGB')
# keep a copy of the original image for OpenCV functions and applying ma
orig_image = image.copy()
# transform the image
image = transform(image)
# add a batch dimension
image = image.unsqueeze(0).to(device)
masks, boxes, labels,points = get_outputs(image, model, 0.8)
result = draw_segmentation_map(orig_image, masks, boxes, labels,points)
# visualize the image
#cv2.imshow('Segmented image', result)
#cv2.waitKey(0)
# set the save path
save_path = "./outputs/00385439.png"
cv2.imwrite(save_path, result)
```

Figure 8

2.1. c. Obstacle Detection: Mask RCNN instance segmentation model pretrained with Resnet 50 was used for Obstacle Detection in the captured image. The scores were calculated for each object detected and those passing a detection accuracy of 80% are selected. Bounding boxes are drawn around them and the object centroids were marked with a 20*20 red pixelated box. The images were projected to BEV and the objects other than the ego vehicle itself could be identified with the red stretched line from the centroid mark. Fig. 8. Shows the code for the same and Fig. 10. Shows the detection of the obstacles in the image, Fig. 11. its projection in BEV and Fig. 12. lanes and waypoints detected on the same.

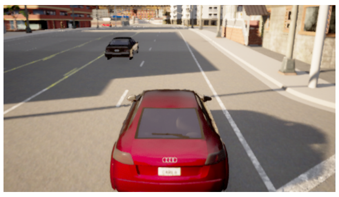


Figure 9

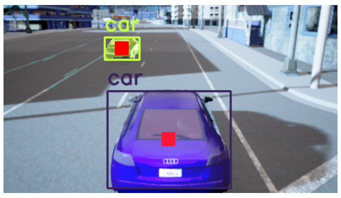


Figure 10

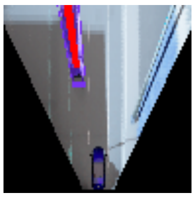


Figure 11

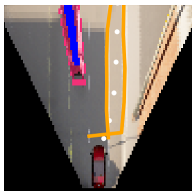
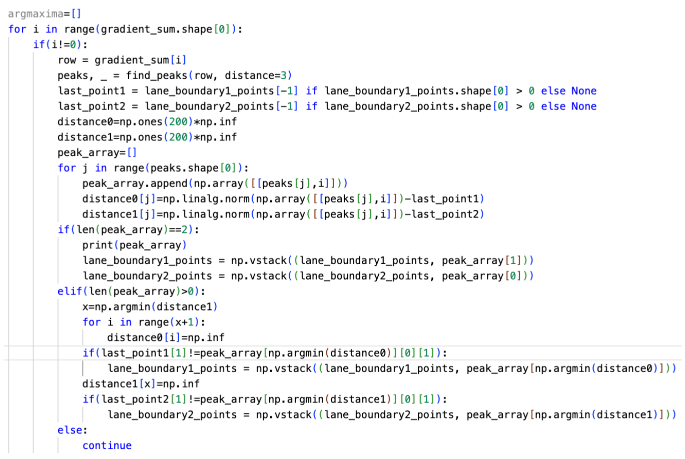
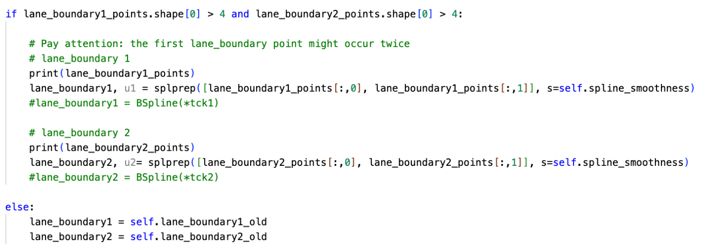


Figure 12



```
argmaxima=[]
for i in range(gradient_sum.shape[0]):
    if(i!=0):
        row = gradient_sum[i]
        peaks, _ = find_peaks(row, distance=3)
        last_point1 = lane_boundary1_points[-1] if lane_boundary1_points.shape[0] > 0 else None
        last_point2 = lane_boundary2_points[-1] if lane_boundary2_points.shape[0] > 0 else None
        distance0=np.ones(200)*np.inf
        distance1=np.ones(200)*np.inf
        peak_array=[]
        for j in range(peaks.shape[0]):
            peak_array.append(np.array([[peaks[j],i]]))
            distance0[j]=np.linalg.norm(np.array([[peaks[j],i]])-last_point1)
            distance1[j]=np.linalg.norm(np.array([[peaks[j],i]])-last_point2)
        if(len(peak_array)==2):
            print(peak_array)
            lane_boundary1_points = np.vstack((lane_boundary1_points, peak_array[1]))
            lane_boundary2_points = np.vstack((lane_boundary2_points, peak_array[0]))
        elif(len(peak_array)>0):
            x=np.argmin(distance1)
            for i in range(x+1):
                distance0[i]=np.inf
            if(last_point1[1]!=peak_array[np.argmin(distance0)][0][1]):
                lane_boundary1_points = np.vstack((lane_boundary1_points, peak_array[np.argmin(distance0)]))
            distance1[x]=np.inf
            if(last_point2[1]!=peak_array[np.argmin(distance1)][0][1]):
                lane_boundary2_points = np.vstack((lane_boundary2_points, peak_array[np.argmin(distance1)]))
        else:
            continue
```

Figure 13



```
if lane_boundary1_points.shape[0] > 4 and lane_boundary2_points.shape[0] > 4:

    # Pay attention: the first lane_boundary point might occur twice
    # lane_boundary 1
    print(lane_boundary1_points)
    lane_boundary1, u1 = splprep([lane_boundary1_points[:,0], lane_boundary1_points[:,1]], s=self.spline_smoothness)
    #lane_boundary1 = BSpline(*tck1)

    # lane_boundary 2
    print(lane_boundary2_points)
    lane_boundary2, u2= splprep([lane_boundary2_points[:,0], lane_boundary2_points[:,1]], s=self.spline_smoothness)
    #lane_boundary2 = BSpline(*tck2)

else:
    lane_boundary1 = self.lane_boundary1_old
    lane_boundary2 = self.lane_boundary2_old
```

Figure 14

2.1. d. Assign Edges to Lane Boundaries: The initial lane position was detected in find_first_lane_point() and for each of the following lane points row wise next maxima was calculated and the maxima closest to the previous lane points were marked. The first boundary point was removed by condition i!=0 to avoid it occurring twice. For the situation where both previous lane points are closer to the same maximum gradient, once closer lane point to last_point_1 is found that point is removed by making minimum distance infinity and rest of the points are searched for the closest point to last_point_2.

2.1. e. Spline Fitting: Fig. 17. shows that lane boundaries were fit in the parametric splines and hence a lane corresponding to it was plotted on the BEV image.

2.1. f. Testing: The implementation was tried and tested in both CARLA and Car Racing environments.

Good parameters: A spline smoothness of 10 and gradient threshold of 14/10 worked well for CARLA.

Limitations of the model:

1. The lane detection often fails when there is shadow in the scene since gradients are not properly calculated in that case as in Fig. 17.
2. When boxes are created around an obstacle with the obstacle detection model it is often taken as a high gradient by the lane detection model and splines are drawn over them which is not really a lane. Gradient threshold changing was tried to avoid finding these wrong peaks but no significant impact was observed.

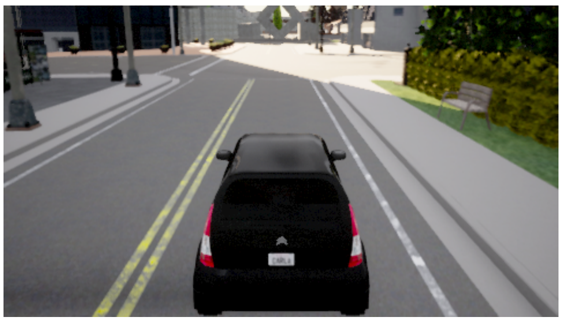


Figure 15

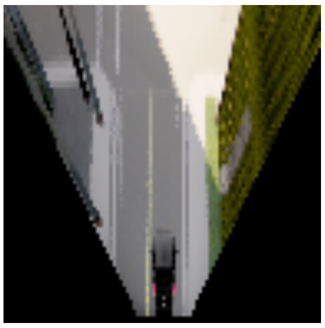


Figure 16

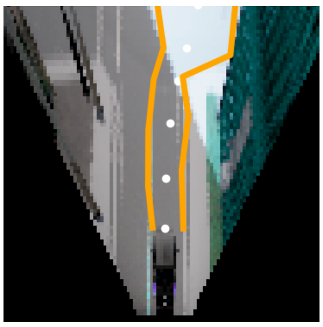
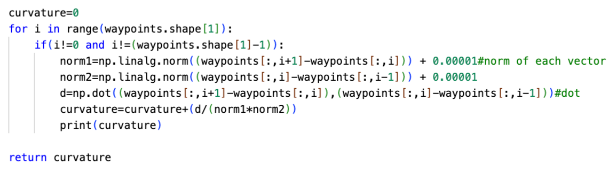


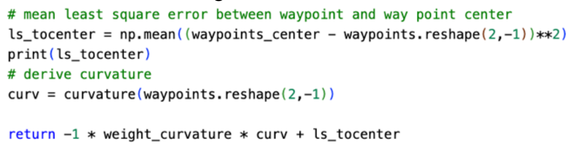
Figure 17

2.2. Waypoint Prediction:



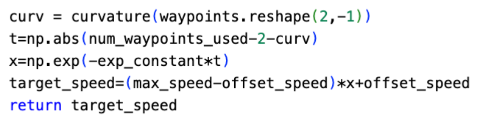
```
curvature=0
for i in range(waypoints.shape[1]):
    if(i!=0 and i!=(waypoints.shape[1]-1)):
        norm1=np.linalg.norm((waypoints[:,i+1]-waypoints[:,i])) + 0.00001#norm of each vector
        norm2=np.linalg.norm((waypoints[:,i]-waypoints[:,i-1])) + 0.00001
        d=np.dot((waypoints[:,i+1]-waypoints[:,i]),(waypoints[:,i]-waypoints[:,i-1]))#dot
        curvature=curvature+(d/(norm1*norm2))
        print(curvature)

return curvature
```

Figure 18



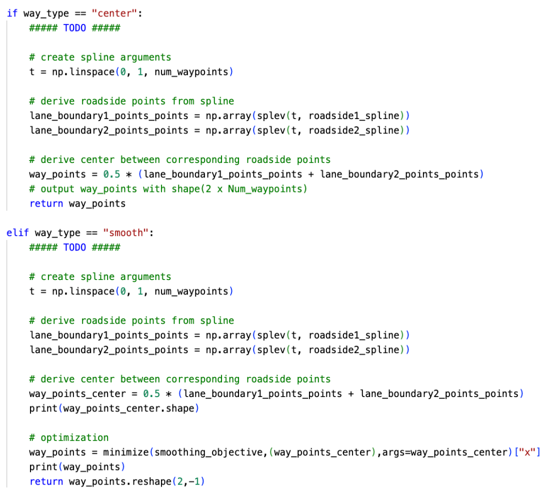
```
# mean least square error between waypoint and way point center
ls_tocenter = np.mean((waypoints_center - waypoints.reshape(2,-1))**2)
print(ls_tocenter)
# derive curvature
curv = curvature(waypoints.reshape(2,-1))

return -1 * weight_curvature * curv + ls_tocenter
```

Figure 19



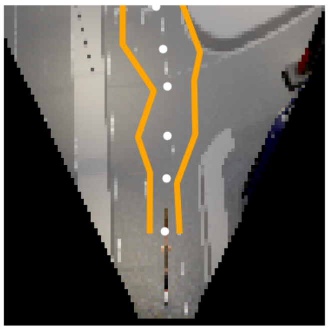
```
curv = curvature(waypoints.reshape(2,-1))
t=np.abs(num_waypoints_used-2-curv)
x=np.exp(-exp_constant*t)
target_speed=(max_speed-offset_speed)*x+offset_speed
return target_speed
```

Figure 20



```
if way_type == "center":
    ##### TODO #####

    # create spline arguments
    t = np.linspace(0, 1, num_waypoints)

    # derive roadside points from spline
    lane_boundary1_points_points = np.array(splev(t, roadside1_spline))
    lane_boundary2_points_points = np.array(splev(t, roadside2_spline))

    # derive center between corresponding roadside points
    way_points = 0.5 * (lane_boundary1_points_points + lane_boundary2_points_points)
    # output way_points with shape(2 x Num_waypoints)
    return way_points

elif way_type == "smooth":
    ##### TODO #####

    # create spline arguments
    t = np.linspace(0, 1, num_waypoints)

    # derive roadside points from spline
    lane_boundary1_points_points = np.array(splev(t, roadside1_spline))
    lane_boundary2_points_points = np.array(splev(t, roadside2_spline))

    # derive center between corresponding roadside points
    way_points_center = 0.5 * (lane_boundary1_points_points + lane_boundary2_points_points)
    print(way_points_center.shape)

    # optimization
    way_points = minimize(smoothing_objective,(way_points_center),args=way_points_center)["x"]
    print(way_points)
    return way_points.reshape(2,-1)
```

Figure 21



2.2. a. Road center: Road center is implemented in Fig. 21. Under way_type=center as the average of the two lanes calculated on both sides. **Situations where center waypoint will not give good results:** In case of curved roads, if one side of the lane is comparatively more curved the waypoint will try to shift towards that side and will not stick to the center path.
A plot for a cyclist's waypoints is added in Fig. 22.

Figure 22

2.2. b. Path Smoothing: The curvature was implemented as dot product of consecutive waypoints divided by their norm as shown in Fig. 18. The ls_tocenter() term ensures that the path stays close to center and a weighted curvature is subtracted from the same to smoothen the curve. The smoothing objective is implemented in Fig. 19 and the smooth waypoint is calculated in Fig. 21.
Purpose of the second term: This term is the curvature term that tries to reduce the angle between two lines in a trajectory by reducing the dot product between the segment vectors. The trajectory becomes smoother as the angle reduces.

2.2. c. Target Speed Prediction: The target_speed_prediction() is implemented as shown in Fig. 20 where a percentage of v_{max}-v_{min} , depending on the curvature, velocity gain factor and number of waypoints is added to the minimum velocity so that the car takes highest velocity on straight roads and goes slower on the curved edges.

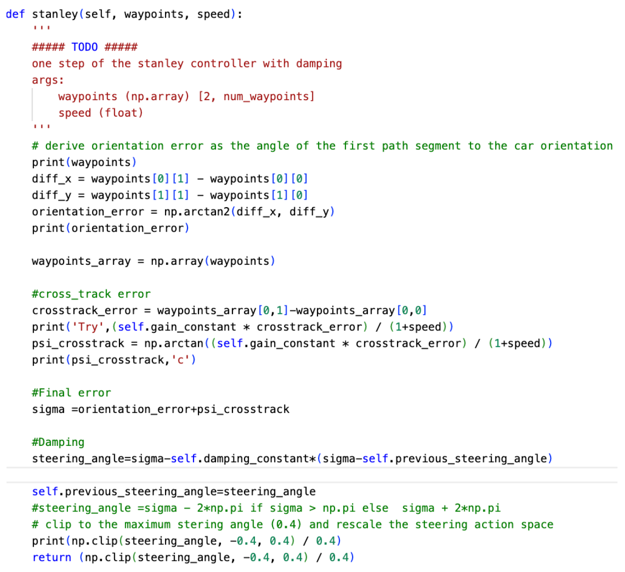
2.3. Lateral Control:

2.3. a. Stanley Controller Theory: In stanley controller, the first term stands for orientation error. This is the angle between the orientation of the car (which is considered to be 0 in bird's eye view) and the orientation of the waypoints that can be calculated by calculating the slope of waypoints.

The second term stands for the steering angle adjustment required depending on the cross-track error. Cross-track error is basically the amount the midpoint of the wheels is shifted on x axis from the path trajectory. The k factor multiplied with (distance/velocity) to calculate the steering angle is the gain parameter.

2.3. b. Stanley Controller: The stanley controller was implemented by calculating the orientation error as the angle between car and waypoint and cross-track error as the difference of x coordinate of the second waypoint and the initial waypoint. The orientation error and psi_crosstrack was used to calculate the steering angle as per the given equation. A default speed of 1 is added to avoid very low values of speed at the start where the value goes to infinity and angles goes to pi/2.

Hyperparameter Tuning: A suitable gain parameter and damping parameter for CarRacing() was 5 and 0.4 however for CARLA tuning these hyperparameters was very challenging. However, a gain parameter of 0.01 and damping constant of 0.5 could keep the car stable mostly.
Car behavior: The trend of the car was to go slightly zigzag initially probably due to the high throttle for straight roads. But later it could stabilize. Fig. 22 shows the implementation of the Stanley controller.

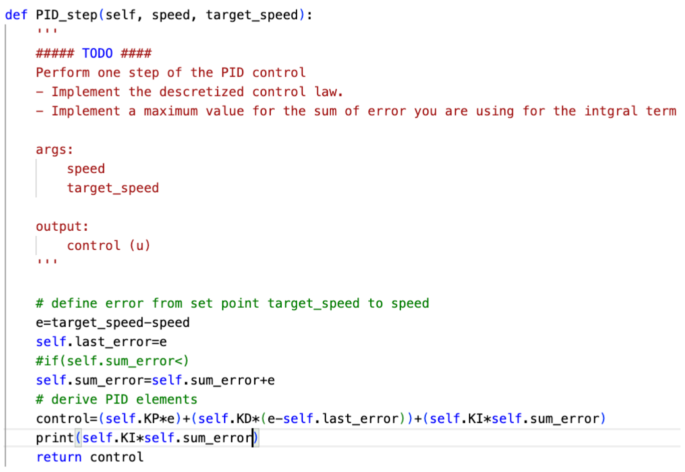


```
def stanley(self, waypoints, speed):
    '''
    ##### TODO #####
    one step of the stanley controller with damping
    args:
        waypoints (np.array) [2, num_waypoints]
        speed (float)
    '''
    # derive orientation error as the angle of the first path segment to the car orientation
    print(waypoints)
    diff_x = waypoints[0][1] - waypoints[0][0]
    diff_y = waypoints[1][1] - waypoints[1][0]
    orientation_error = np.arctan2(diff_x, diff_y)
    print(orientation_error)

    waypoints_array = np.array(waypoints)

    #cross_track error
    crosstrack_error = waypoints_array[0,1]-waypoints_array[0,0]
    print('Try',(self.gain_constant * crosstrack_error) / (1+speed))
    psi_crosstrack = np.arctan((self.gain_constant * crosstrack_error) / (1+speed))
    print(psi_crosstrack,'c')

    #Final error
    sigma =orientation_error+psi_crosstrack

    #Damping
    steering_angle=sigma-self.damping_constant*(sigma-self.previous_steering_angle)

    self.previous_steering_angle=steering_angle
    #steering_angle =sigma - 2*np.pi if sigma > np.pi else  sigma + 2*np.pi
    # clip to the maximum stering angle (0.4) and rescale the steering action space
    print(np.clip(steering_angle, -0.4, 0.4) / 0.4)
    return (np.clip(steering_angle, -0.4, 0.4) / 0.4)
```

Figure 22

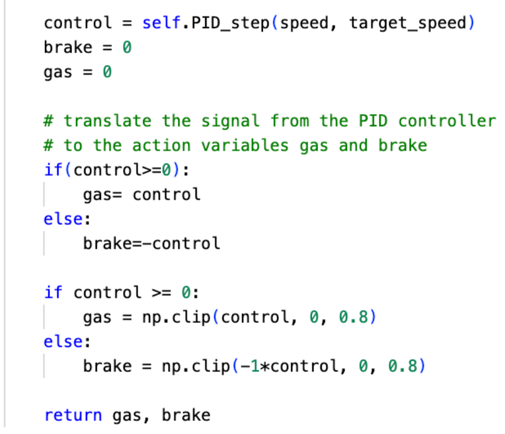
2.3. c. Damping: The damping is also implemented in Fig. 22.
Impact of Damping on Behavior of car: The damping decides what component of previous step steering angle will be added to this step for finding a suitable steering angle relative to the last stage. In the final steering angle for a state (1-damping) times the current steering angle and damping times the last steering angle is used. The car was relatively more stable when damped.
A suitable damping parameter is 0.5.

2.4. Longitudinal Control:



```
def PID_step(self, speed, target_speed):
    '''
    ##### TODO ####
    Perform one step of the PID control
    - Implement the descretized control law.
    - Implement a maximum value for the sum of error you are using for the intgral term

    args:
        speed
        target_speed

    output:
        control (u)
    '''

    # define error from set point target_speed to speed
    e=target_speed-speed
    self.last_error=e
    #if(self.sum_error<)
    self.sum_error=self.sum_error+e
    # derive PID elements
    control=(self.KP*e)+(self.KD*(e-self.last_error))+(self.KI*self.sum_error)
    print(self.KI*self.sum_error)
    return control
```

Figure 23



```
control = self.PID_step(speed, target_speed)
brake = 0
gas = 0

# translate the signal from the PID controller
# to the action variables gas and brake
if(control>=0):
    gas= control
else:
    brake=-control

if control >= 0:
    gas = np.clip(control, 0, 0.8)
else:
    brake = np.clip(-1*control, 0, 0.8)

return gas, brake
```

Figure 24

2.4. a. PID Controller: The PID controller is implemented according to the equations given by calculating the control as shown in Fig. 23 and the gas and brake values as shown in Fig. 24.

2.4.b. Parameter Searching: In CARLA, it was not possible to generate plots and hence different values of all parameters were tried and tested for driving the car. **The best parameters for CARLA are K_P=0.008, K_I=0.0 and K_D=0.0**. Any other parameters for K_I and K_D resulted in the car losing control while turning.

However, parameter tuning was done in CarRacing() by observing the oscillations of the graph generated while driving the car.

K_P parameter:

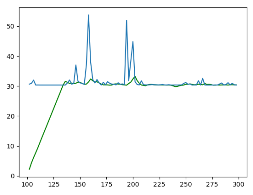


Figure 25

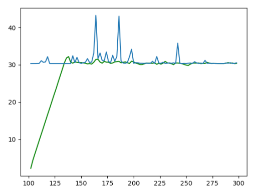


Figure 26

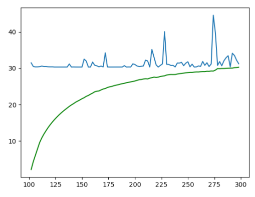


Figure 27

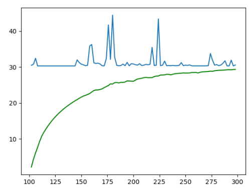


Figure 28

K_D Parameter:

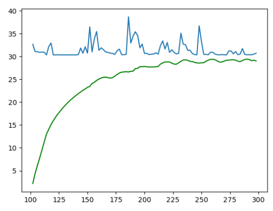


Figure 29

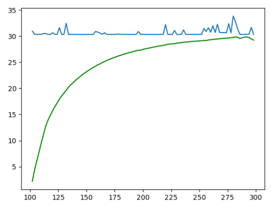


Figure 30

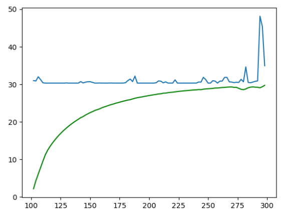


Figure 31

K_I Parameters:

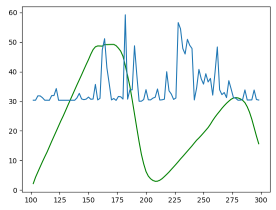


Figure 32

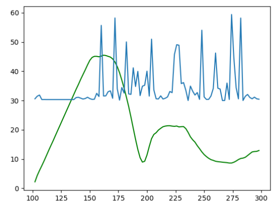


Figure 33

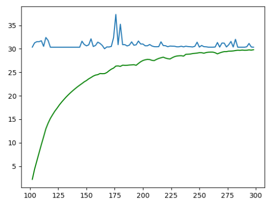
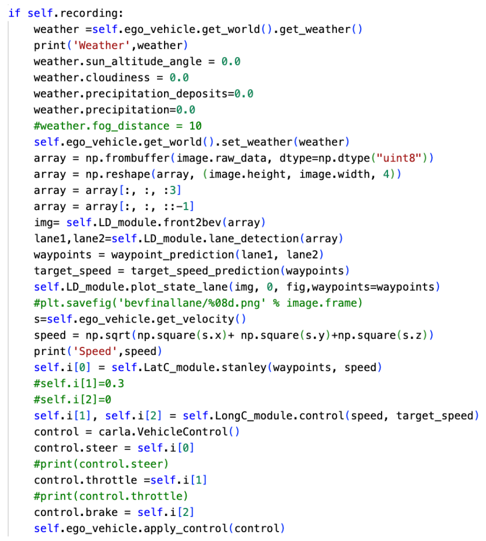


Figure 34

Fig. 25, 26, 27, 28 shows plots for K_P parameter tuning corresponding to K_P=0.1, 0.02, 0.01, 0.009. Fig. 29, 30, 31 shows plots for K_D parameter tuning corresponding to K_D=0.1, 0.05, 0.01 with K_P=0.009 and Fig. 32, 33, 34 shows plots for K_I parameter tuning corresponding to K_I=0.1, 0.01, 0.000001 with K_P=0.009 and K_D=0.05. The final parameters chosen for CarRacing() are K_P=0.009,K_D=0.05 and K_I=0.000001.

Manual Control Script:
BEV Transformation, Lane Detection, Waypoint Prediction, Lateral and Longitudinal Control etc. was implemented to Manual_control and the controls predicted were applied on the car. The code snippet in Fig. 35.


```
if self.recording:
    weather =self.ego_vehicle.get_world().get_weather()
    print('Weather',weather)
    weather.sun_altitude_angle = 0.0
    weather.cloudiness = 0.0
    weather.precipitation_deposits=0.0
    weather.precipitation=0.0
    #weather.fog_distance = 10
    self.ego_vehicle.get_world().set_weather(weather)
    array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
    array = np.reshape(array, (image.height, image.width, 4))
    array = array[:, :, :3]
    array = array[:, :, ::-1]
    img= self.LD_module.front2bev(array)
    lane1,lane2=self.LD_module.lane_detection(array)
    waypoints = waypoint_prediction(lane1, lane2)
    target_speed = target_speed_prediction(waypoints)
    self.LD_module.plot_state_lane(img, 0, fig,waypoints=waypoints)
    #plt.savefig('bevfinallane/%08d.png' % image.frame)
    s=self.ego_vehicle.get_velocity()
    speed = np.sqrt(np.square(s.x)+ np.square(s.y)+np.square(s.z))
    print('Speed',speed)
    self.i[0] = self.LatC_module.stanley(waypoints, speed)
    #self.i[1]=0.3
    #self.i[2]=0
    self.i[1], self.i[2] = self.LongC_module.control(speed, target_speed)
    control = carla.VehicleControl()
    control.steer = self.i[0]
    #print(control.steer)
    control.throttle =self.i[1]
    #print(control.throttle)
    control.brake = self.i[2]
    self.ego_vehicle.apply_control(control)
```
Figure 35

Conditions and Assumptions:

1. The code is written inside the recording, hence it is necessary to press **R** for the code to execute.
2. The weather is set to sunset and no rains because shadows and puddles could result in distortions in lane prediction. Multiple attempts were made to change thresholds to balance this but all attempts in Dynamic weather showed distortions.
3. The obstacle detection part was not included in the manual control since the masks around the obstacles were mistakenly detected as peaks/lanes and the lateral control function gave wrong outputs in that case. It was separately implemented along to lane detection in Jupyter notebook.

Appendix

The zip file contains

1. Lane Detection:
    a. Lane_Detection.py
    b. BEV image (Fig. 5)
    c. Grayscale image (Fig. 6)
    d. Gradient Image (Fig. 7)
    e. Car Image in front view (Fig. 15)
    f. Car Image in BEV (Fig. 16)
    g. Car Image with Lane and Waypoints (Fig. 18)
2. Waypoint Prediction:
    a. Waypoint_Prediction.py
    b. Waypoint Detected Image (Fig. 22)
3. Lateral Control
    a. Lateral_control.py
4. Longitudinal Control
    a. Longitudinal_control.py
    b. K_P parameter tuning graphs (Fig. 25, 26, 27, 28)
    c. K_D parameter tuning graphs (Fig. 29, 30, 31)
    d. K_I parameter tuning graphs (Fig. 32, 33, 34)
5. Obstacle Detection:
    a. ObstacleDetection.ipynb
    b. Original image for obstacle detection (Fig. 9)
    c. Obstacle detected image in front view (Fig. 10)
    d. Obstacle detected image in BEV (Fig. 11)
    e. Lane and waypoint detected image (Fig. 12)
6. Manual_ControlPathPlanning.py(Final test script)