**SOURCE CODE**

**CCTV\_Video.py**

# USAGE

import numpy as np

import argparse

import imutils

import time

import cv2

import os

import subprocess

import os, shutil

folder = 'output'

for filename in os.listdir(folder):

file\_path = os.path.join(folder, filename)

try:

if os.path.isfile(file\_path) or os.path.islink(file\_path):

os.unlink(file\_path)

elif os.path.isdir(file\_path):

shutil.rmtree(file\_path)

except Exception as e:

print('Failed to delete %s. Reason: %s' % (file\_path, e))

# construct the argument parse and parse the arguments

ap = argparse.ArgumentParser()

ap.add\_argument("-i", "--input", required=True,

help="path to input video")

ap.add\_argument("-o", "--output", required=False,

help="path to output video")

ap.add\_argument("-y", "--yolo", required=True,

help="base path to YOLO directory")

ap.add\_argument("-c", "--confidence", type=float, default=0.5,

help="minimum probability to filter weak detections")

ap.add\_argument("-t", "--threshold", type=float, default=0.3,

help="threshold when applyong non-maxima suppression")

args = vars(ap.parse\_args())

# load the COCO class labels our YOLO model was trained on

labelsPath = os.path.sep.join([args["yolo"], "coco.names"])

LABELS = open(labelsPath).read().strip().split("\n") #Changes here

# initialize a list of colors to represent each possible class label

np.random.seed(42)

COLORS = np.random.randint(0, 255, size=(len(LABELS), 3),

dtype="uint8")

weightsPath = os.path.sep.join([args["yolo"], "yolov3.weights"])

configPath = os.path.sep.join([args["yolo"], "yolov3.cfg"])

print("[INFO] loading YOLO from disk...")

net = cv2.dnn.readNetFromDarknet(configPath, weightsPath)

ln = net.getLayerNames() #Changes here

ln = [ln[i[0] - 1] for i in net.getUnconnectedOutLayers()]

# initialize the video stream, pointer to output video file, and

# frame dimensions

vs = cv2.VideoCapture(args["input"])

writer = None

(W, H) = (None, None)

try:

prop = cv2.cv.CV\_CAP\_PROP\_FRAME\_COUNT if imutils.is\_cv2() \

else cv2.CAP\_PROP\_FRAME\_COUNT

total = int(vs.get(prop))

print("[INFO] {} total frames in video".format(total))

except:

print("[INFO] could not determine # of frames in video")

print("[INFO] no approx. completion time can be provided")

total = -1

cnt = 0

fno = 0

# ------------------FRAME PART-----------------------------------------

counter = 0

while True:

start1 = time.time()

(grabbed, frame) = vs.read()

if(cnt%2!=0):

cnt+=1

continue

fno+=1

print("Frame No:", fno)

if not grabbed:

break

# if the frame dimensions are empty, grab them

if W is None or H is None:

(H, W) = frame.shape[:2]

blob = cv2.dnn.blobFromImage(frame, 1 / 255.0, (416, 416),

swapRB=True, crop=False)

net.setInput(blob)

start = time.time()

layerOutputs = net.forward(ln)

end = time.time()

boxes = []

confidences = []

classIDs = []

# loop over each of the layer outputs

for output in layerOutputs:

# loop over each of the detections

for detection in output:

# extract the class ID and confidence (i.e., probability)

# of the current object detection

scores = detection[5:]

classID = np.argmax(scores)

confidence = scores[classID]

# filter out weak predictions by ensuring the detected

# probability is greater than the minimum probability

if confidence > args["confidence"]:

box = detection[0:4] \* np.array([W, H, W, H])

(centerX, centerY, width, height) = box.astype("int")

x = int(centerX - (width / 2))

y = int(centerY - (height / 2))

# update our list of bounding box coordinates,

# confidences, and class IDs

boxes.append([x, y, int(width), int(height)])

confidences.append(float(confidence))

classIDs.append(classID)

# apply non-maxima suppression to suppress weak, overlapping

# bounding boxes

idxs = cv2.dnn.NMSBoxes(boxes, confidences, args["confidence"],

args["threshold"])

# ensure at least one detection exists

if len(idxs) > 0:

idArray = []

for j in idxs.flatten():

if classIDs[j]==2 or classIDs[j]==0:

idArray.append(j)

flag = 0

for j in idArray:

for k in idArray:

if k==j:

continue

(x1, y1) = (boxes[j][0], boxes[j][1])

(w1, h1) = (boxes[j][2], boxes[j][3])

(x2, y2) = (x1+w1, abs(y1-h1))

(x3, y3) = (boxes[k][0], boxes[k][1])

(w2, h2) = (boxes[k][2], boxes[k][3])

(x4, y4) = (x3+w2, abs(y3-h2))

if(x1>x4 or x3>x2):

flag = 1

counter = 0

if(y1 < y4 or y3 < y2):

flag = 1

counter = 0

if flag == 0:

counter+=1

break

if flag == 0:

break

# loop over the indexes we are keeping

if(counter==4):

print("CRASH ALERT")

cv2.imwrite("output/Crash{}.jpg".format(fno), frame)

counter=0

for i in idxs.flatten():

# extract the bounding box coordinates

(x, y) = (boxes[i][0], boxes[i][1])

(w, h) = (boxes[i][2], boxes[i][3])

# draw a bounding box rectangle and label on the frame

color = [int(c) for c in COLORS[classIDs[i]]]

cv2.rectangle(frame, (x, y), (x + w, y + h), color, 2)

text = "{}: {:.4f}".format(LABELS[classIDs[i]],

confidences[i])

cv2.putText(frame, text, (x, y - 5),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, color, 2)

print(f"Box[{i}]: {x} {y} {w} {h} Labels[{i}]: {LABELS[classIDs[i]]} classIDs[{i}]: {classIDs[i]} AveragePrecision[{i}]: {confidences[i]}")

end1 = time.time()

cv2.imwrite("output/frame{}.jpg".format(fno), frame)

print("Complete time for algorithm", (end1-start1))

cnt+=1

print("[INFO] cleaning up...")

print("[Msg] Accidents frames analyised")

vs.release()

program = "wwd\_analysis.py "+args["input"]+" TRUE"

print("[INFO] Wrong Driving Analysiss Started")

subprocess.call("python "+program)

print("[INFO] Making All Frames into a video File")

subprocess.call("python playCrash.py")

print("[INFO] Video FIle Playing Started")

subprocess.call("python PlayVideo.py")

**wwdanalysis.py**

import sys

import numpy as np

import cv2

import math

import matplotlib.pyplot as plt

from numpy.linalg import inv

import time

import operator

import random

import shutil, os

from scipy import signal

######## Setting most important parameters #######

show\_images\_flag = False

if(len(sys.argv)<2):

sys.exit("please provide the name of video to be analyzed")

video = sys.argv[1]

if(len(sys.argv)>2):

show\_images\_flag = sys.argv[2]

####### To create an empty sub\_final folder everytime we run this script #######

if not os.path.exists('sub\_final'):

os.mkdir('sub\_final')

shutil.rmtree('sub\_final')

os.makedirs('sub\_final')

####### This code can be used to generate gray frames from given video #######

print("extracting frames from image")

cap = cv2.VideoCapture(video)

count = 0

while(cap.isOpened()):

ret, frame = cap.read()

if(ret):

gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

else:

break

# cv2.imshow('frame',gray)

cv2.imwrite("sub\_final/frame%d.jpg" % count, gray)

count = count + 1

if cv2.waitKey(1) & 0xFF == ord('q'):

break

cap.release()

cv2.destroyAllWindows()

####### The above code to extract the frames from the images was given in the official documentation of opencv #######

####### Reference: https://docs.opencv.org/3.1.0/dd/d43/tutorial\_py\_video\_display.html #######

sub\_part\_of\_image1 = str(int(count\*0.65))

sub\_part\_of\_image2 = str(int(count\*0.65)+1)

path\_of\_image1 = 'sub\_final/frame'+ sub\_part\_of\_image1 + '.jpg'

path\_of\_image2 = 'sub\_final/frame'+ sub\_part\_of\_image2 + '.jpg'

####### Takes a matrix as input and converts range of matrix to 0 to 255 range #######

####### This function is usually used for printing purposes (better vizualization) #######

def change\_range(old\_array):

new\_array = np.zeros(old\_array.size).reshape(old\_array.shape[0],old\_array.shape[1])

max\_intensity = old\_array.max()

min\_intensity = old\_array.min()

for j in range(0, old\_array.shape[1]):

for i in range(0, old\_array.shape[0]):

new\_array[i][j] = ((old\_array[i][j]-min\_intensity)\*(255/(max\_intensity - min\_intensity)))

return new\_array

####### This function takes in an image matrix and smoothes its with k dimension kernal #######

def smooth\_image(img,k):

# creating a new image matrix in which smoothed values will be stored

a = img.copy()

#k = dimension of the filter

for i in range(math.floor(k/2), (img.shape[0]-math.ceil(k/2))):

for j in range(math.floor(k/2), (img.shape[1]-math.ceil(k/2))):

i\_lower = i-int(math.floor(k/2))

i\_upper = i+int(math.ceil(k/2))

j\_lower = j-int(math.floor(k/2))

j\_upper = j+int(math.ceil(k/2))

sub\_img\_flat = img[i\_lower:i\_upper,j\_lower:j\_upper].ravel().copy()

a[i][j] = np.sum(sub\_img\_flat)/(k\*k)

return a

####### This function used for reporducing edge strength and edge orientation matrix for given smoothed #######

####### matrix image. This function returns edge strenght matrix (mapped from 0 to 255) and edge #######

####### orientation matrix mapped from -90 to 90 #######

def edge\_detection(a):

img = a

# creating a new image matrix in which edge-y values will be stored

c = np.zeros(img.size).reshape(img.shape[0],img.shape[1])

c\_display = np.zeros(img.size).reshape(img.shape[0],img.shape[1])

# creating a new image matrix in which edge-x values will be stored

d = np.zeros(img.size).reshape(img.shape[0],img.shape[1])

d\_display = np.zeros(img.size).reshape(img.shape[0],img.shape[1])

# applying x and y edges on the image and storing the values in another matrix images

for j in range(1, img.shape[1]-1):

for i in range(1, img.shape[0]-1):

c[i][j] = ((-0.5)\*int(a[i-1][j]) + (0.5)\*int(a[i+1][j]) + (0)\*(int(a[i][j])))

d[i][j] = ((-0.5)\*int(a[i][j-1]) + (0.5)\*int(a[i][j+1]) + (0)\*(int(a[i][j])))

c\_display = change\_range(c)

# creating a new image matrix in whihc edge matrix values will be stored

e = np.zeros(img.size).reshape(img.shape[0],img.shape[1])

e\_display = np.zeros(img.size).reshape(img.shape[0],img.shape[1])

# finding edge magnitude values from edge x and edge y values

for j in range(1, img.shape[1]-1):

for i in range(1, img.shape[0]-1):

e[i][j] = math.sqrt((c[i][j]\*c[i][j]) + (d[i][j]\*d[i][j]))

e\_display = change\_range(e)

# creating a new image matrix in whihc edge orientation matrix values will be stored

f = np.zeros(img.size).reshape(img.shape[0],img.shape[1])

# just trying to avoid divide by zero

d[d == 0] = 0.0000000001

# finding edge orientation values from edge x and edge y values

for j in range(1, img.shape[1]-1):

for i in range(1, img.shape[0]-1):

f[i][j] = math.degrees(math.atan(c[i][j]/d[i][j]))

f\_display = change\_range(f)

return e\_display, f

####### This is a threshold function which takes in a matrix and threshold level number #######

####### and outputs all the values greater than threshold level, retains the values in matrix #######

def threshold(e\_display, level):

image = np.zeros((e\_display.shape[0], e\_display.shape[1]))

for i in range(0, e\_display.shape[0]):

for j in range(0, e\_display.shape[1]):

if(e\_display[i][j]<level):

image[i][j] = 0

else:

image[i][j] = e\_display[i][j]

return image

####### This is a histogram function used for display images only as it can sample for 255 bins only #######

####### Output is an array of size 255 which consists of number of pixels w.r.t. index of array #######

def histogram(img):

bins = 255

img\_flat = img.ravel()

max\_intensity = img\_flat.max()

min\_intensity = img\_flat.min()

hist = np.zeros(bins)

for element in img\_flat:

bin\_id = int(round((bins-1)\*((element - min\_intensity)/(max\_intensity - min\_intensity))))

hist[bin\_id] += 1

return hist

####### This is a median filter with kernal size of kXk #######

def medianfilter(img,k):

# creating a new image matrix in which smoothed values will be stored

a = img.copy()

#k is dimension of the filter

for i in range(math.floor(k/2), (img.shape[0]-math.ceil(k/2))):

for j in range(math.floor(k/2), (img.shape[1]-math.ceil(k/2))):

i\_lower = i-int(math.floor(k/2))

i\_upper = i+int(math.ceil(k/2))

j\_lower = j-int(math.floor(k/2))

j\_upper = j+int(math.ceil(k/2))

sub\_img\_flat = img[i\_lower:i\_upper,j\_lower:j\_upper].ravel().copy()

a[i][j] = np.max(sub\_img\_flat)

return a

####### custom made gaussian blur kernel which inputs image, kernel size of 1-D and sigma########

def GaussianBlur\_custom(image, size, sigma):

a = image.copy()

gaussian\_kernal = signal.get\_window(('gaussian',sigma),size\*size)

for i in range(math.floor(size/2), (image.shape[0]-math.ceil(size/2))):

for j in range(math.floor(size/2), (image.shape[1]-math.ceil(size/2))):

i\_lower = i-int(math.floor(size/2))

i\_upper = i+int(math.ceil(size/2))

j\_lower = j-int(math.floor(size/2))

j\_upper = j+int(math.ceil(size/2))

sub\_img\_flat = image[i\_lower:i\_upper,j\_lower:j\_upper].ravel().copy()

value = [a\*b for a,b in zip(sub\_img\_flat,gaussian\_kernal)]

a[i][j] = np.sum(value)/np.sum(gaussian\_kernal)

return a

# reading a first image file in black and white mode

print("Reading and resizing the first image")

img1 = cv2.imread(path\_of\_image1,0)

# resizing image to 200 X 400 size image

img1 = cv2.resize(img1, (400,200) )

# displaying all the image properties

print("Shape of First Image: ", img1.shape)

print("Minimum Intensity: ", img1.min())

print("Maximum Intensity: ", img1.max())

if(show\_images\_flag):

plt.imshow(img1)

plt.title("First Image")

plt.show()

# reading a second image file in black and white mode

print("Reading and resizing the second image")

img2 = cv2.imread(path\_of\_image2,0)

# resizing image to 200 X 400 size image

img2 = cv2.resize(img2, (400,200) )

# displaying all the image properties

print("Shape of Second Image: ", img2.shape)

print("Minimum Intensity: ", img2.min())

print("Maximum Intensity: ", img2.max())

if(show\_images\_flag):

plt.imshow(img2)

plt.title("Second Image")

plt.show()

####### Preparing image for performing hour transform on them #######

####### so that we will be able to divide two sides of the road #######

####### and then look into optical flow. I will do an hough transform #######

####### on only one image as it is enough to determine the boundaries #######

####### of the left and right road Assumption: there is no drastic #######

####### movement of camera in between both the image #######

print("Preparing image for Hough Transform")

# smoothing

print("Smoothing the image")

smooth\_image1 = smooth\_image(img1,3)

# detecting edges

print("Doing edge detection")

edge\_output1, orientation\_output1 = edge\_detection(smooth\_image1)

# thresholding

image1 = threshold(edge\_output1, 0)

if(show\_images\_flag):

print("Output after edge detection")

# displaying smoothed, edge detected and thresholded image

plt.imshow(image1)

plt.title("Smoothed, Edge Detected and Thresholded Image1")

plt.show()

# creating a histogram to analyze about the thresholded image

hist = histogram(image1)

# print("Computing histogram of edge detected image")

# #visualizing the histogram

# if(show\_images\_flag):

# y\_hist = np.arange(len(hist))

# plt.bar(y\_hist, hist)

# plt.title("Histogram of Thresholded Image1")

# plt.show()

#plotting the edges on hough space

start = time.time()

print("Starting hough transform")

large\_dim = int(math.sqrt(math.pow(image1.shape[0],2)+math.pow(image1.shape[1],2)))

accumulator = np.zeros((large\_dim, 180))

for i in range(0, image1.shape[0]):

for j in range(0, image1.shape[1]):

if(image1[i][j] > 0):

for k in range(0, 180):

r = (i\*np.cos(np.deg2rad(k))) + (j\*np.sin(np.deg2rad(k)))

# considering in weight of edges as incremental factor in hough space accumulator

accumulator[int(np.round(r))][k] = accumulator[int(np.round(r))][k] + (image1[i][j]/255.0)

end = time.time()

print("time taken for hough transform", end - start) # usually takes around 2.5 to 3 minutes for 200 X 400 image size

# displaying accumulator

if(show\_images\_flag):

print("Plotting Hough Space")

accumulator\_display = change\_range(accumulator)

plt.imshow(accumulator\_display)

plt.title("Hough space for Image 1 after edge detection and thresholding")

plt.show()

# finding local maximum peaks in accumulator

print("Finding the local maximuma and thresholded peaks")

r\_list = []

theta\_list = []

####### 2 ways of finding peaks and I am using both the ways #######

####### First Way: find the pixel which has more intensity pixel than all #######

####### of its 8N neighbourhood #######

####### Second Way: thresholding (I find this to be very useful) #######

for i in range(1, accumulator.shape[0]-1):

for j in range(1, accumulator.shape[1]-1):

if(accumulator[i][j]>accumulator[i-1][j] and

accumulator[i][j]>accumulator[i-1][j+1] and

accumulator[i][j]>accumulator[i-1][j-1] and

accumulator[i][j]>accumulator[i+1][j] and

accumulator[i][j]>accumulator[i+1][j-1] and

accumulator[i][j]>accumulator[i+1][j+1] and

accumulator[i][j]>accumulator[i][j-1] and

accumulator[i][j]>accumulator[i][j+1] and

accumulator[i][j]>50):

r\_list.append(i)

theta\_list.append(j)

print("Number of lines found", len(r\_list))

print("Rho List", r\_list)

print("Theta List", theta\_list)

# filtering required middle dividers and filter them

print("Filtering only required middle lines and finding the best divider line")

req\_theta\_list = []

req\_r\_list = []

for i in range(0, len(theta\_list)):

if(theta\_list[i] > 87 and theta\_list[i] < 94 and r\_list[i] > ((image1.shape[1]/2)-25) and r\_list[i] < ((image1.shape[1]/2)+25) ):

req\_r\_list.append(r\_list[i])

req\_theta\_list.append(theta\_list[i])

print(req\_r\_list, req\_theta\_list)

if len(req\_theta\_list)==0:

print("In he video Lines are not found")

exit()

right\_theta\_angle = round(sum(req\_theta\_list)/len(req\_theta\_list))

right\_r\_distance = round(sum(req\_r\_list)/len(req\_r\_list))

print(right\_r\_distance, right\_theta\_angle)

divider\_column = right\_r\_distance

#displaying those lines on an empty image

r\_list = [right\_r\_distance]

theta\_list = [right\_theta\_angle]

lines\_image = np.zeros((image1.shape[0],image1.shape[1]))

for k in range(0, len(r\_list)):

for i in range(1, image1.shape[0]):

x = i

y = ((-np.cos(np.deg2rad(theta\_list[k]))/np.sin(np.deg2rad(theta\_list[k])))\*x) + ((r\_list[k])/np.sin(np.deg2rad(theta\_list[k])))

if(math.isnan(y)):

lines\_image[x][x] = 255

continue

y = int(np.round(y))

if(y>=0 and y< image1.shape[1]):

lines\_image[x][y] = 255

if(show\_images\_flag):

plt.imshow(lines\_image)

plt.title("The DIVIDER LINE plotted on an empty image")

plt.show()

####### Reality: We should place our camera somewhere in the middle of the #######

####### road and display a red light (Our main goal) #######

####### Corner Case: the road can be curved #######

####### (We do not look into such cases, thats a different study altogether) #######

# To make masks

# Left Mask can be used for vehicles driving on the left side of the road

# Right Mask can be used for vehicles driving on the right side of the road

print("Implementing the right and left maskes")

maskl = np.zeros((img1.shape[0], img1.shape[1]))

for i in range(0, lines\_image.shape[0]):

for j in range(0, lines\_image.shape[1]):

if(lines\_image[i][j] == 255):

break

else:

maskl[i][j] = 1

maskr = np.ones((img1.shape[0], img1.shape[1]))

for i in range(0, lines\_image.shape[0]):

for j in range(0, lines\_image.shape[1]):

if(lines\_image[i][j] == 255):

break

else:

maskr[i][j] = 0

if(show\_images\_flag):

plt.imshow(maskr)

plt.title("Right Mask")

plt.show()

plt.imshow(maskl)

plt.title("Left Mask")

plt.show()

# Analyzing left side of the road

# Applying Left Mask to first and second images

print("Applying Left Mask to first and second images")

img1l = img1\*maskl

img2l = img2\*maskl

if(show\_images\_flag):

plt.imshow(img1l)

plt.title("Image 1 with Left Mask")

plt.show()

plt.imshow(img2l)

plt.title("Image 2 with Left Mask")

plt.show()

# Applying Right Mask to first and second images

print("Applying Right Mask to first and second images")

img1r = img1\*maskr

img2r = img2\*maskr

if(show\_images\_flag):

plt.imshow(img1r)

plt.title("Image 1 with Right Mask")

plt.show()

plt.imshow(img2r)

plt.title("Image 2 with Right Mask")

plt.show()

####### OPTICAL FLOW starts from here using Lucas Kalade technique #######

####### care has been taken that the u and v displacement between frames is very small #######

print("Applying Guassian bluring on all the four images")

# img1l = cv2.GaussianBlur(img1l,(3,3),2)

img1l = GaussianBlur\_custom(img1l, 3, 2)

if(show\_images\_flag):

plt.imshow(img1l)

plt.title("Gaussian Blurred 1st Left Image")

plt.show()

# img2l = cv2.GaussianBlur(img2l,(3,3),2)

img2l = GaussianBlur\_custom(img2l, 3, 2)

if(show\_images\_flag):

plt.imshow(img2l)

plt.title("Gaussian Blurred 2nd Left Image")

plt.show()

# img1r = cv2.GaussianBlur(img1r,(3,3),2)

img1r = GaussianBlur\_custom(img1r, 3, 2)

if(show\_images\_flag):

plt.imshow(img1r)

plt.title("Gaussian Blurred 1st Right Image")

plt.show()

# img2r = cv2.GaussianBlur(img2r,(3,3),2)

img2r = GaussianBlur\_custom(img2r, 3, 2)

if(show\_images\_flag):

plt.imshow(img2r)

plt.title("Gaussian Blurred 2nd Right Image")

plt.show()

####### finding x gradient, y gradient and time gradient for both the frames #######

####### I am finding gradients for both the images and averaging it #######

####### as suggested by one of the professors in UCF #######

print("Applying LUCAS KANADE OPTICAL FLOW on both the images on both the sides of the road")

print("Applying derivative averaging for both the images on right and left sides using loberts masks")

start\_time = time.time()

zero\_img\_matrix = np.zeros(img1.size,float).reshape(img1.shape[0],img1.shape[1])

x\_gradient\_img1l = zero\_img\_matrix.copy()

y\_gradient\_img1l = zero\_img\_matrix.copy()

x\_gradient\_img2l = zero\_img\_matrix.copy()

y\_gradient\_img2l = zero\_img\_matrix.copy()

x\_gradient\_imgl = zero\_img\_matrix.copy()

y\_gradient\_imgl = zero\_img\_matrix.copy()

t\_gradient\_img1l = zero\_img\_matrix.copy()

t\_gradient\_img2l = zero\_img\_matrix.copy()

t\_gradient\_imgl = zero\_img\_matrix.copy()

for i in range(0, img1.shape[0]-2):

for j in range(0, img1.shape[1]-2):

x\_gradient\_img1l[i][j] = (-1\*(img1l[i][j])) + (-1\*(img1l[i+1][j])) + (1\*(img1l[i][j+1])) + (1\*(img1l[i+1][j+1]))

x\_gradient\_img2l[i][j] = (-1\*(img2l[i][j])) + (-1\*(img2l[i+1][j])) + (1\*(img2l[i][j+1])) + (1\*(img2l[i+1][j+1]))

y\_gradient\_img1l[i][j] = (-1\*(img1l[i][j])) + (1\*(img1l[i+1][j])) + (-1\*(img1l[i][j+1])) + (1\*(img1l[i+1][j+1]))

y\_gradient\_img2l[i][j] = (-1\*(img2l[i][j])) + (1\*(img2l[i+1][j])) + (-1\*(img2l[i][j+1])) + (1\*(img2l[i+1][j+1]))

t\_gradient\_img1l[i][j] = (-1\*(img1l[i][j])) + (-1\*(img1l[i+1][j])) + (-1\*(img1l[i][j+1])) + (-1\*(img1l[i+1][j+1]))

t\_gradient\_img2l[i][j] = (+1\*(img2l[i][j])) + (+1\*(img2l[i+1][j])) + (+1\*(img2l[i][j+1])) + (+1\*(img2l[i+1][j+1]))

x\_gradient\_imgl = x\_gradient\_img1l + x\_gradient\_img2l

y\_gradient\_imgl = y\_gradient\_img1l + y\_gradient\_img2l

t\_gradient\_imgl = t\_gradient\_img1l + t\_gradient\_img2l

# x\_gradient\_imgl = cv2.GaussianBlur(x\_gradient\_imgl,(3,3),2)

x\_gradient\_imgl = GaussianBlur\_custom(x\_gradient\_imgl, 3, 2)

# y\_gradient\_imgl = cv2.GaussianBlur(y\_gradient\_imgl,(3,3),2)

y\_gradient\_imgl = GaussianBlur\_custom(y\_gradient\_imgl, 3, 2)

# t\_gradient\_imgl = cv2.GaussianBlur(t\_gradient\_imgl,(3,3),2)

t\_gradient\_imgl = GaussianBlur\_custom(t\_gradient\_imgl, 3, 2)

x\_gradient\_img1r = zero\_img\_matrix.copy()

y\_gradient\_img1r = zero\_img\_matrix.copy()

x\_gradient\_img2r = zero\_img\_matrix.copy()

y\_gradient\_img2r = zero\_img\_matrix.copy()

x\_gradient\_imgr = zero\_img\_matrix.copy()

y\_gradient\_imgr = zero\_img\_matrix.copy()

t\_gradient\_img1r = zero\_img\_matrix.copy()

t\_gradient\_img2r = zero\_img\_matrix.copy()

t\_gradient\_imgr = zero\_img\_matrix.copy()

for i in range(0, img1.shape[0]-2):

for j in range(0, img1.shape[1]-2):

x\_gradient\_img1r[i][j] = (-1\*(img1r[i][j])) + (-1\*(img1r[i+1][j])) + (1\*(img1r[i][j+1])) + (1\*(img1r[i+1][j+1]))

x\_gradient\_img2r[i][j] = (-1\*(img2r[i][j])) + (-1\*(img2r[i+1][j])) + (1\*(img2r[i][j+1])) + (1\*(img2r[i+1][j+1]))

y\_gradient\_img1r[i][j] = (-1\*(img1r[i][j])) + (1\*(img1r[i+1][j])) + (-1\*(img1r[i][j+1])) + (1\*(img1r[i+1][j+1]))

y\_gradient\_img2r[i][j] = (-1\*(img2r[i][j])) + (1\*(img2r[i+1][j])) + (-1\*(img2r[i][j+1])) + (1\*(img2r[i+1][j+1]))

t\_gradient\_img1r[i][j] = (-1\*(img1r[i][j])) + (-1\*(img1r[i+1][j])) + (-1\*(img1r[i][j+1])) + (-1\*(img1r[i+1][j+1]))

t\_gradient\_img2r[i][j] = (+1\*(img2r[i][j])) + (+1\*(img2r[i+1][j])) + (+1\*(img2r[i][j+1])) + (+1\*(img2r[i+1][j+1]))

x\_gradient\_imgr = x\_gradient\_img1r + x\_gradient\_img2r

y\_gradient\_imgr = y\_gradient\_img1r + y\_gradient\_img2r

t\_gradient\_imgr = t\_gradient\_img1r + t\_gradient\_img2r

# x\_gradient\_imgr = cv2.GaussianBlur(x\_gradient\_imgr,(3,3),2)

x\_gradient\_imgr = GaussianBlur\_custom(x\_gradient\_imgr, 3, 2)

# y\_gradient\_imgr = cv2.GaussianBlur(y\_gradient\_imgr,(3,3),2)

y\_gradient\_imgr = GaussianBlur\_custom(y\_gradient\_imgr, 3, 2)

# t\_gradient\_imgr = cv2.GaussianBlur(t\_gradient\_imgr,(3,3),2)

t\_gradient\_imgr = GaussianBlur\_custom(t\_gradient\_imgr, 3, 2)

end\_time = time.time()

print("Time taken to apply gradient averaging and applying gaussian blur", end\_time - start\_time)

####### Displaying X, Y and time gradient graphs #######

if(show\_images\_flag):

print("Showing averaged X, Y and time gradient of left and right images")

plt.imshow(change\_range(x\_gradient\_imgl))

plt.title("Averaged X gradient of Left images")

plt.show()

plt.imshow(change\_range(y\_gradient\_imgl))

plt.title("Averaged Y gradient of Left images")

plt.show()

plt.imshow(change\_range(t\_gradient\_imgl))

plt.title("Averaged Time gradient of Left images")

plt.show()

plt.imshow(change\_range(x\_gradient\_imgr))

plt.title("Averaged X gradient of Right images")

plt.show()

plt.imshow(change\_range(y\_gradient\_imgr))

plt.title("Averaged Y gradient of Right images")

plt.show()

plt.imshow(change\_range(t\_gradient\_imgr))

plt.title("Averaged Time gradient of Right images")

plt.show()

####### compute u and v using lucas kanade method, i am not using matrix inverse #######

####### I am using the least mean squares

start\_time = time.time()

print("Determining U and V matrices for the left side of the road images")

####### For left side of the road #######

ul = zero\_img\_matrix.copy()

vl = zero\_img\_matrix.copy()

uv\_magl = zero\_img\_matrix.copy()

uv\_dirl = zero\_img\_matrix.copy()

for i in range(1, img1.shape[0]-2):

for j in range(1, divider\_column):

if(maskl[i][j] == 1):

fx = [x\_gradient\_imgl[i-1][j-1],x\_gradient\_imgl[i-1][j],x\_gradient\_imgl[i-1][j+1],

x\_gradient\_imgl[i][j-1],x\_gradient\_imgl[i][j],x\_gradient\_imgl[i][j+1],

x\_gradient\_imgl[i+1][j-1],x\_gradient\_imgl[i+1][j],x\_gradient\_imgl[i+1][j+1]]

fy = [y\_gradient\_imgl[i-1][j-1],y\_gradient\_imgl[i-1][j],y\_gradient\_imgl[i-1][j+1],

y\_gradient\_imgl[i][j-1],y\_gradient\_imgl[i][j],y\_gradient\_imgl[i][j+1],

y\_gradient\_imgl[i+1][j-1],y\_gradient\_imgl[i+1][j],y\_gradient\_imgl[i+1][j+1]]

ft = [t\_gradient\_imgl[i-1][j-1],t\_gradient\_imgl[i-1][j],t\_gradient\_imgl[i-1][j+1],

t\_gradient\_imgl[i][j-1],t\_gradient\_imgl[i][j],t\_gradient\_imgl[i][j+1],

t\_gradient\_imgl[i+1][j-1],t\_gradient\_imgl[i+1][j],t\_gradient\_imgl[i+1][j+1]]

sum\_fx2 = np.sum(np.square(fx))

sum\_fy2 = np.sum(np.square(fy))

sum\_fxy = np.sum([a\*b for a,b in zip(fx,fy)])

sum\_fxy\_2 = np.square(sum\_fxy)

denom = sum\_fx2\*sum\_fy2-sum\_fxy\_2

if(denom == 0.0):

print(i,j)

continue

sum\_fxt = np.sum([a\*b for a,b in zip(fx,ft)])

sum\_fyt = np.sum([a\*b for a,b in zip(fy,ft)])

ul[i][j] = ((-sum\_fy2\*sum\_fxt) + (sum\_fxy\*sum\_fyt))/denom

vl[i][j] = ((sum\_fxt\*sum\_fxy) - (sum\_fx2\*sum\_fyt))/denom

print("Applying median filter on left u and v")

ul = medianfilter(ul,3)

vl = medianfilter(vl,3)

for i in range(1, img1.shape[0]-2):

for j in range(1, divider\_column):

uv\_magl[i][j] = math.sqrt((ul[i][j]\*ul[i][j])+(vl[i][j]\*vl[i][j]))

uv\_dirl[i][j] = math.degrees(math.atan(ul[i][j]/(vl[i][j]+0.0001)))

print("Determining U and V matrices for the right side of the road images")

####### For right side of the road #######

ur = zero\_img\_matrix.copy()

vr = zero\_img\_matrix.copy()

uv\_magr = zero\_img\_matrix.copy()

uv\_dirr = zero\_img\_matrix.copy()

for i in range(1, img1.shape[0] - 2):

for j in range(divider\_column, img1.shape[1] - 2):

if(maskr[i][j] == 1):

fx = [x\_gradient\_imgr[i-1][j-1],x\_gradient\_imgr[i-1][j],x\_gradient\_imgr[i-1][j+1],

x\_gradient\_imgr[i][j-1],x\_gradient\_imgr[i][j],x\_gradient\_imgr[i][j+1],

x\_gradient\_imgr[i+1][j-1],x\_gradient\_imgr[i+1][j],x\_gradient\_imgr[i+1][j+1]]

fy = [y\_gradient\_imgr[i-1][j-1],y\_gradient\_imgr[i-1][j],y\_gradient\_imgr[i-1][j+1],

y\_gradient\_imgr[i][j-1],y\_gradient\_imgr[i][j],y\_gradient\_imgr[i][j+1],

y\_gradient\_imgr[i+1][j-1],y\_gradient\_imgr[i+1][j],y\_gradient\_imgr[i+1][j+1]]

ft = [t\_gradient\_imgr[i-1][j-1],t\_gradient\_imgr[i-1][j],t\_gradient\_imgr[i-1][j+1],

t\_gradient\_imgr[i][j-1],t\_gradient\_imgr[i][j],t\_gradient\_imgr[i][j+1],

t\_gradient\_imgr[i+1][j-1],t\_gradient\_imgr[i+1][j],t\_gradient\_imgr[i+1][j+1]]

sum\_fx2 = np.sum(np.square(fx))

sum\_fy2 = np.sum(np.square(fy))

sum\_fxy = np.sum([a\*b for a,b in zip(fx,fy)])

sum\_fxy\_2 = np.square(sum\_fxy)

denom = sum\_fx2\*sum\_fy2-sum\_fxy\_2

if(denom == 0.0):

print(i,j)

continue

sum\_fxt = np.sum([a\*b for a,b in zip(fx,ft)])

sum\_fyt = np.sum([a\*b for a,b in zip(fy,ft)])

ur[i][j] = ((-sum\_fy2\*sum\_fxt) + (sum\_fxy\*sum\_fyt))/denom

vr[i][j] = ((sum\_fxt\*sum\_fxy) - (sum\_fx2\*sum\_fyt))/denom

print("Applying median filter on right u and v")

ur = medianfilter(ur,3)

vr = medianfilter(vr,3)

for i in range(1, img1.shape[0]-2):

for j in range(divider\_column, img1.shape[1]-2):

uv\_magr[i][j] = math.sqrt((ur[i][j]\*ur[i][j])+(vr[i][j]\*vr[i][j]))

uv\_dirr[i][j] = math.degrees(math.atan(ur[i][j]/(vr[i][j]+0.0001)))

end\_time = time.time()

print("Complete time taken to find u and v matrices for both the sides of the road and apply median filter on it", end\_time - start\_time)

####### plotting uv direction and magnitude graph of the left side of the road #######

if(show\_images\_flag):

print("Showing the UV direction, magnitude of both left and right side images of the road")

plt.imshow(change\_range(uv\_dirl))

plt.title("UV direction of left side of the road")

plt.show()

plt.imshow(change\_range(uv\_magl))

plt.title("UV Magnitute of left side of the road")

plt.show()

plt.imshow(change\_range(uv\_dirr))

plt.title("UV direction of right side of the road")

plt.show()

plt.imshow(change\_range(uv\_magr))

plt.title("UV Magnitute of right side of the road")

plt.show()

####### looking at the left side of the road analysis of UV direction #######

magnitude\_threshold = 10

print("Analyzing the histogram of the UV directions of the left side of the road")

angles\_list = []

for i in range(0, img1.shape[0]):

for j in range(0, divider\_column):

if(uv\_magl[i][j]> magnitude\_threshold):

angles\_list.append(uv\_dirl[i][j])

data = angles\_list

# fixed bin size

bins = np.arange(-200, 200, 5) # fixed bin size

if(show\_images\_flag):

plt.xlim([min(data)-5, max(data)+5])

plt.hist(data, bins=bins, alpha=0.5)

plt.title('UV orientation of left side of the bin (fixed bin size)')

plt.xlabel('UV orientation X (bin size = 5)')

plt.ylabel('count')

plt.show()

####### Determining the degree of the vehicle on the left side of the vehicle #######

bins\_for\_histogram = 5

max\_x = 90 + bins\_for\_histogram

min\_x = -90 + bins\_for\_histogram

bin\_list = []

for i in range(min\_x, max\_x, 5):

bin\_list.append(i)

countprev = 0

value\_list = []

for i in range(len(bin\_list)):

count = 0

for j in range(len(data)):

if(data[j] < bin\_list[i]):

count += 1

value\_list.append(count - countprev)

countprev = count

# weighted\_sum\_list = [a\*b for a,b in zip(bin\_list,value\_list)]

# weighted\_angle\_value\_left = sum(weighted\_sum\_list)/sum(value\_list)

# max\_degree\_left = weighted\_angle\_value\_left

max\_index, max\_value = max(enumerate(value\_list), key=operator.itemgetter(1))

max\_degree\_left = bin\_list[max\_index]

####### Right side of the road analaysis #######

print("Analyzing the histogram of the UV directions of the right side of the road")

angles\_list = []

for i in range(0, img1.shape[0]):

for j in range(divider\_column, img1.shape[1]):

if(uv\_magr[i][j]> magnitude\_threshold):

angles\_list.append(uv\_dirr[i][j])

data = angles\_list

# fixed bin size

bins = np.arange(-200, 200, 5) # fixed bin size

if(show\_images\_flag):

plt.xlim([min(data)-5, max(data)+5])

plt.hist(data, bins=bins, alpha=0.5)

plt.title('UV orientation of right side of the bin (fixed bin size)')

plt.xlabel('UV orientation X (bin size = 5)')

plt.ylabel('count')

plt.show()

####### Determining the degree of the vehicle on the left side of the vehicle #######

bins\_for\_histogram = 1

max\_x = 90 + bins\_for\_histogram

min\_x = -90 + bins\_for\_histogram

bin\_list = []

for i in range(min\_x, max\_x, 5):

bin\_list.append(i)

countprev = 0

value\_list = []

for i in range(len(bin\_list)):

count = 0

for j in range(len(data)):

if(data[j] < bin\_list[i]):

count += 1

value\_list.append(count - countprev)

countprev = count

# weighted\_sum\_list = [a\*b for a,b in zip(bin\_list,value\_list)]

# weighted\_angle\_value\_right = sum(weighted\_sum\_list)/sum(value\_list)

# max\_degree\_right = weighted\_angle\_value\_right

max\_index, max\_value = max(enumerate(value\_list), key=operator.itemgetter(1))

max\_degree\_right = bin\_list[max\_index]

print("FINAL RESULTS")

left\_correct = False

right\_correct = False

if(max\_degree\_left < 0):

print("The vehicle on the LEFT SIDE OF THE ROAD IS IN THE CORRECT DIRECTION")

left\_correct = True

else:

print("The vehicle on the LEFT SIDE OF THE ROAD IS IN THE WRONG DIRECTION")

if(max\_degree\_right > 0):

print("The vehicle on the RIGHT SIDE OF THE ROAD IS IN THE CORRECT DIRECTION")

right\_correct = True

else:

print("The vehicle on the RIGHT SIDE OF THE ROAD IS IN THE WRONG DIRECTION")