A

Project Report on

**Self-Driving Car Lane Detection System**

Submitted in partial fulfillment of completion of the course

Advanced Diploma in IT, Networking and Cloud

Submitted by:

**Rana Karmakar**

**Basudev Gharai**

**Laxmi Pandey**

**Damini Gurjar**

**Kamlesh Debnath**

Under Guidance of:

**Somya Mahanty (IBM)**

**Madhukar Wadekar (Edunet)**

|  |  |  |
| --- | --- | --- |
| IBM-Logo - Chicago Innovation | DGT introduces high end diploma courses - digitalLEARNING Magazine | Edunet Foundation-Delhi- CSR Organization profile |

Year 2022

Abstract

Acknowledgement

Team Composition and Workload Division

Table of Contents

1. Introduction to Problem
2. Literature Review
3. Proposed Solution
4. Requirements

4.1 Technology Stack

4.2 Hardware

4.3 Software

4.4 Deployment Environment

5. User Requirements

6. Design Documentation

7. Implementation Details

8. Testing

9. Deployment

10. Future Scope

11. Conclusion

Appendix A Project Code

Appendix B Screenshot of Project

Appendix C abbreviation

References

1. Opencv Documentation
2. Youtube.com (Tutorial)
3. Stackoverflow
4. Github
5. Wikipedia

Instructions:

1. Font- Arial
2. Main /Title Heading- 16 (bold, center aligned)
3. Heading-14 (bold)
4. Sub heading-12 (bold)
5. Normal text-12
6. Text Alignments- Justified
7. Image/Screenshot/Table Alignments- Center
8. Caption below Images/screenshot/table - Centre, Font size 8
9. References to be numbered in square box like [1] ….
10. Any code to be attached as appendix at the end like Appendix A, Appendix B …
11. Screenshots of project can also be attached as appendix

**Literature Review**

**Reviewing Literature**

First a picture of the road is acquired with the assistance of a camera attached on the vehicle. Next one may reduce the processing time by translating the image to a grayscale image. Next, the existence of disturbance captured in the image will interrupt the accurate detection of the edges, so one can activate filters to get rid of noises. Some of the filters which can be used are bilateral filter, gaussian filter, trilateral filter. There upon in order to produce an edged image, an edge detector can be used which makes use of canny filter to get the edges by using machine generated thresholding. Line detector can then use it for the purpose of detection. It will generate a left side and right side segments of the lane boundary. As a result, yellow and the white lanes are obtained using the RGB color codes. Techniques that are used for detecting the lanes plays a compelling part in technologically intelligent transport setup. Methods that one may make use of have been studied in this paper. Many of them resulted in inappropriate conclusions. Hence, other enrichments can also be included in the present approach in a way to increase the efficiency of the setup. In the coming future, one can change the current Hough Transformation so that it can sum up curved and straight roads respectively. This approach cannot give accurate results in poor environmental conditions like on hazy, cloudy, rainy and stormy days, therefore one needs to make amendments in it.

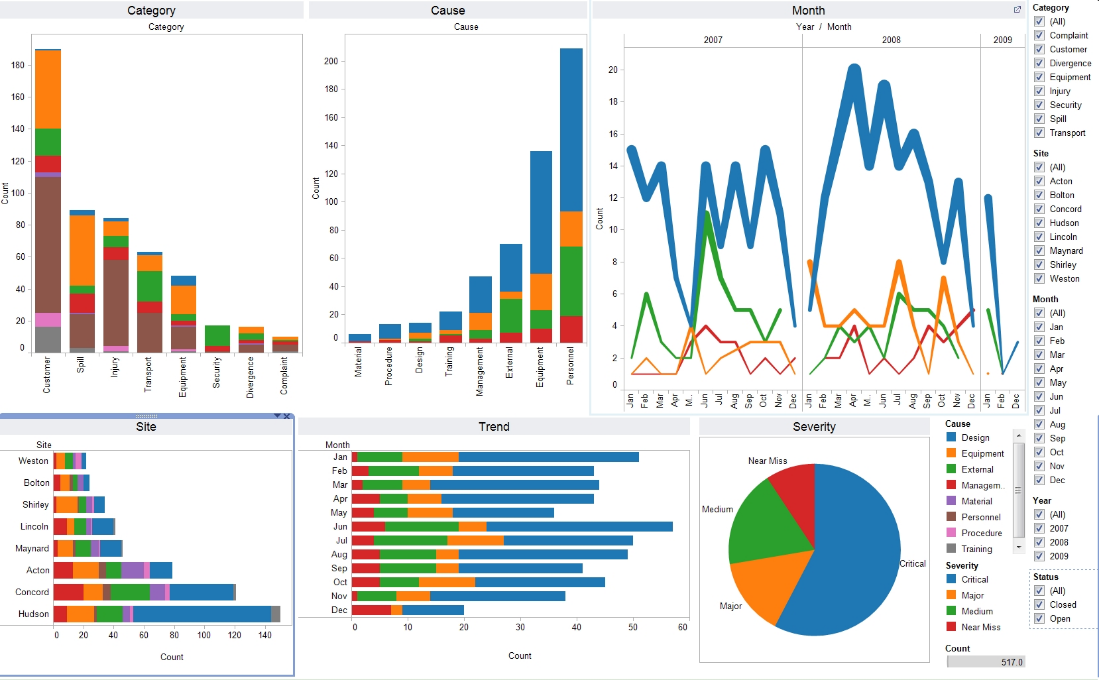


Image:1 Data visualization of electronics manufacturing market Reference: [data\_visualization\_python.jpg (1658×1026) (wp.com)](https://i1.wp.com/www.simplifiedpython.net/wp-content/uploads/2018/07/data_visualization_python.jpg?resize=1658%2C1026&ssl=1)

Abstract

For vehicles to be able to drive by themselves, they need to understand their surrounding world like human drivers, so they can navigate their way in streets, pause at stop signs and traffic lights, and avoid hitting obstacles such as other cars and pedestrians. Based on the problems encountered in detecting objects by autonomous vehicles an effort has been made to demonstrate lane detection using **OpenCV** library. The reason and procedure for choosing grayscale instead of colour, detecting edges in an image, selecting region of interest, applying Hough Transform and choosing polar coordinates over Cartesian coordinates has been discussed.

During the driving operation, humans use their optical vision for vehicle manoeuvring. The road lane marking, act as a constant reference for vehicle navigation. One of the prerequisites to have in a self-driving car is the development of an Automatic Lane Detection system using an algorithm.

Acknowledgement

This report is written during ADIT-IBM Internship project at the Faculty of ADIT-IBM, National Skill Training Institute, Bhubaneswar. We would like to thank Somya Mohanty (IBM) and Madhukar Wadekar (EduNet) for giving us the opportunity to work in the department on challenging project which allow us to pursue different ideas, eventually leading to this thesis report. We would like to express our gratitude to several individuals who supported in completion of thesis project including, Deepneel Majumder for providing help in choosing relevant image processing algorithm and methods and providing some ideas about how these methods are suitable for different hardware and software architecture. Finally, we would like to thanks our supervisor Brindaban Das (Assistant Director, NSTI Bhubaneswar), for providing valuable review and feedback comments, the time he spent for discussing problems and new idea for achieving goals of the project. Last but not the least, we would like to express our heart-felt gratitude to our family members for their support and encouragement.

Team Composition and Workload Division

* Data Collection and Pre-processing – Laxmi Pandey and Damini Gurjar
* Algorithm design and Code Building – Rana Karmakar
* Testing – Kamlesh Devnath
* Deployment – Basudev Ghadai

Introduction to Problem

With the rapid development of society, automobiles have become one of the transportation tools for people to travel. In the narrow road, there are more and more vehicles of all kinds. As more and more vehicles are driving on the road, the number of victims of car accidents is increasing every year. How to drive safely under the condition of numerous vehicles and narrow roads has become the focus of attention. Advanced driver assistance systems which include lane departure warning (LDW), Lane Keeping Assist, and Adaptive Cruise Control (ACC) can help people analyse the current driving environment and provide appropriate feedback for safe driving or alert the driver in dangerous circumstances. This kind of auxiliary driving system is expected to become more and more perfect. However, the bottleneck of the development of this system is that the road traffic environment is difficult to predict. After investigation, in the complex traffic environment where vehicles are numerous and speed is too fast, the probability of accidents is much greater than usual. In such a complex traffic situation, road colour extraction and texture detection as well as road boundary and lane marking are the main perceptual clues of human driving.

Lane detection is a hot topic in the field of machine learning and computer vision and has been applied in intelligent vehicle systems. The lane detection system comes from lane markers in a complex environment and is used to estimate the vehicle’s position and trajectory relative to the lane reliably. At the same time, lane detection plays an important role in the lane departure warning system.

Literature Review

The objective of the literature review is to find and explore the benefits of lane detection algorithms and also what are the different problems in existing algorithms and techniques. The main goal of this literature review is to find the gaps in existing research and methods and also what will be the possible solutions to overcome these holes.

Proposed Solution

To detect white markings in the lane, first, we need to mask the rest part of the frame. We do this using frame masking. The frame is nothing but a NumPy array of image pixel values. To mask the unnecessary pixel of the frame, we simply update those pixel values to 0 in the NumPy array.

After making we need to detect lane lines. The technique used to detect mathematical shapes like this is called Hough Transform. Hough transformation can detect shapes like rectangles, circles, triangles, and lines.

Requirements

**Technology Stack –**

**Software Configuration**

These are the Software Configurations that are required.

• Operating System: Windows 10/8/7 (incl. 64-bit), Mac OS, Linux

* Language: Python 3
* Libraries: OpenCv, Numpy, Matplotlib, Pandas, OS

• IDE: Jupyter Notebook/ Spyder 3

• Framework: Tkinter

**Hardware Configuration**

These are minimum Hardware configurations that are required.

• Processor: Intel core 2 duo or higher.

• RAM: 2 GB or higher

• HDD: 256 GB or higher

• Monitor: 1024 x 768 minimum screen resolution.

• Keyboard: US en Standard Keyboard.

User Requirements

Since this an autonomous self-driving car, the user does not directly interact with the car. Rather, the car may have some in-dash computer system, like those found in modern day cars.

The in-dash systems, apart from providing entertainment features and vital information like speed and engine RPM, provide navigation aids like GPS as well. They, thus, act as an interface between the user and the car. Such in-dash systems and modifications required for them to work with our system, however, are beyond the scope of this project.

Design Documentation

When contrasting a self-driving car, one of the most important things that the vehicle needs to do is determine the outline of the lane it is traveling in. Lane markings are a key piece of information a driver uses while on the road to determine the direction of travel and stay safe while driving, as the markings show where the driver can drive legally. Thus, it is vitally important that a self-driving vehicle also be able to determine lane markings, and thereby the safe lane of travel.

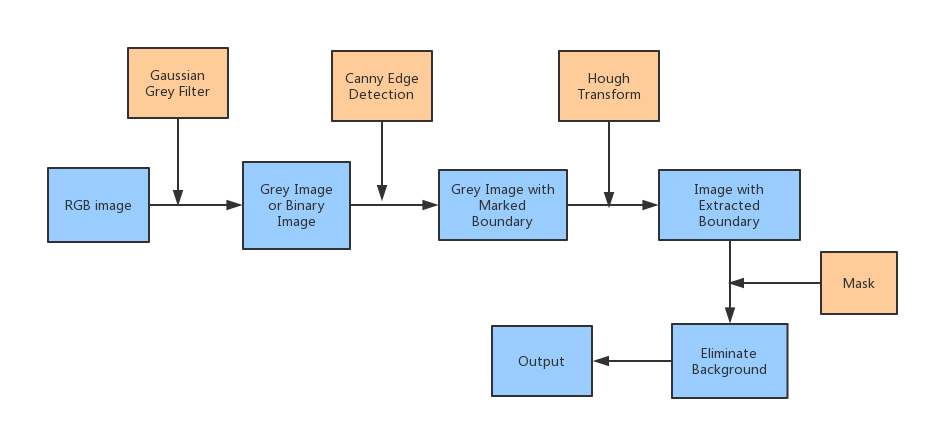
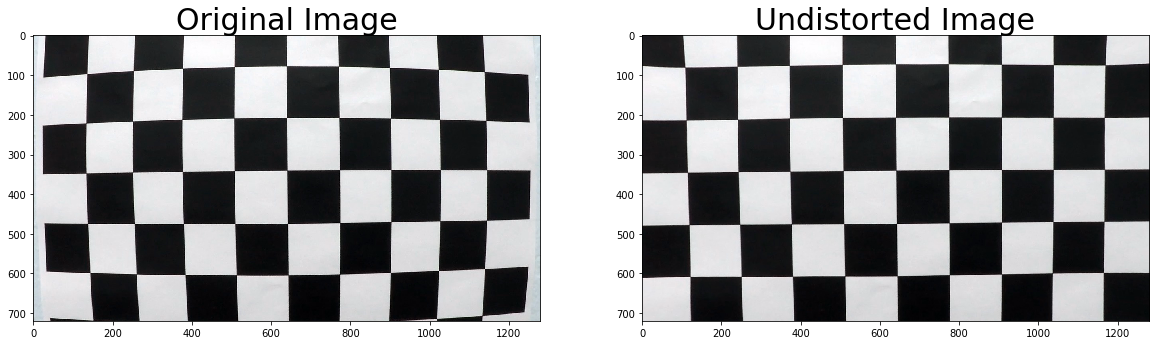


Figure 1 Work Flow Design

Implementation Details

**Distortion Correction**

The camera used in the test video was used to take 20 pictures of a checkerboard, which were used to generate the distortion model. We begin by converting the image to grayscale, then applying the [*cv2.findChessboardCorners*](http://cv2.findchessboardcorners/)*()* function. We already know that this chessboard is a 2 dimensional object with exclusively straight lines, so we can apply some transformations to the detected corners to align them properly. I used *the cv2.CalibrateCamera()* to get the distortion coefficients and the camera matrix. The camera has been calibrated. Then we used cv2.undistort() to correct the rest of your input data. You can see the difference between the original image of the checkerboard and the corrected image below:

Figure 2 Image Distortion

Here's the distortion correction applied to an image of the road. You might not be able to notice the slight difference, but it can have a huge impact on image processing



Figure 3 Undistorted Image

### Perspective Warp

### Detecting curved lanes in camera space is not very easy. What if we could get a bird’s eye view of the lanes? That can be done by applying a perspective transformation on the image. Here's what it looks like:

****

Figure 4 Prospective warp

cv2.getPerspectiveTransform() function to get the transformation matrix, and cv2.warpPerspective() to apply it to an image.

### Sobel Filtering

### we can use a method similar to our edge detector, this time to filter out the road. Lane lines typically have a high contrast to the road, so we can use this to our advantage. The ****Canny**** edge detector previously used in version 1 makes use of ****Sobel Operator**** to get the gradient of an image function. We'll be using this to detect areas of high contrast to filter lane markings and ignore the road.



Figure 5 Sobal Filter

### Histogram Peak Detection

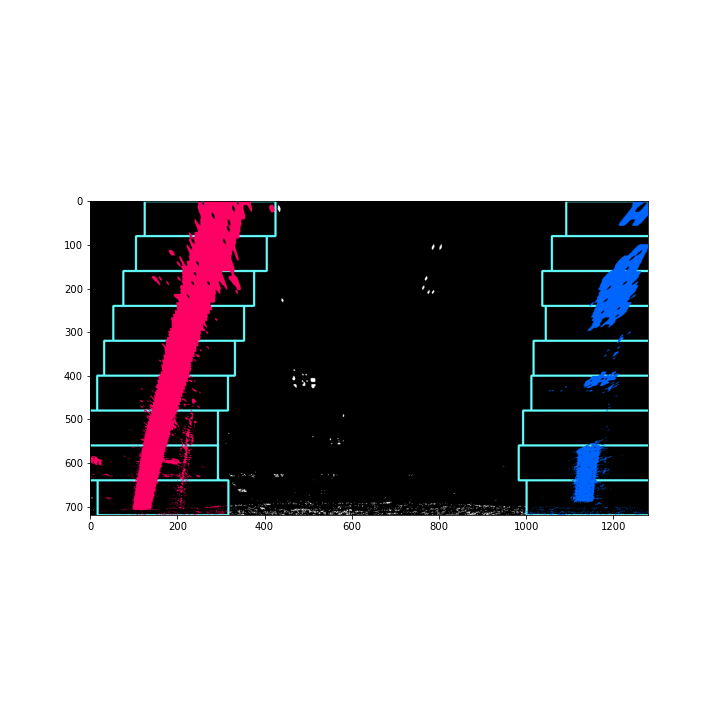
We'll be applying a special algorithm called the **Sliding** **Window** **Algorithm** to detect our lane lines.



Figure 6Histogram Peak Detection

### Sliding Window Search

The sliding window algorithm will be used to differentiate between the left and right lane boundaries so that we can fit two different curves representing the lane boundaries.

The algorithm itself is very simple. Starting from the initial position, the first window measures how many pixels are located inside the window. If the amount of pixels reaches a certain threshold, it shifts the next window to the average lateral position of the detected pixels. If not enough pixels are detected, the next window starts in the same lateral position. This continues until the windows reach the other edge of the image.

The pixels that fall within the windows are given a marker. In the images below, the blue marked pixels represent the right lane, and the red ones represent the left:

Figure 7 Sliding Window Search

### Curve Fitting

The rest of the project is really easy. We apply polynomial regression to the red and blue pixels individually using np.polyfit(), and then the detector is mostly done!

Here's what the curves look like:



Figure 8 Curve Fitting

### Overlay Detected Lane

Here's the final part of the detection system, the user interface! We simply create an overlay which fills in the detected portion of the lane, and then we can finally apply it to video. Once put through the video pipeline, we can see the following output:

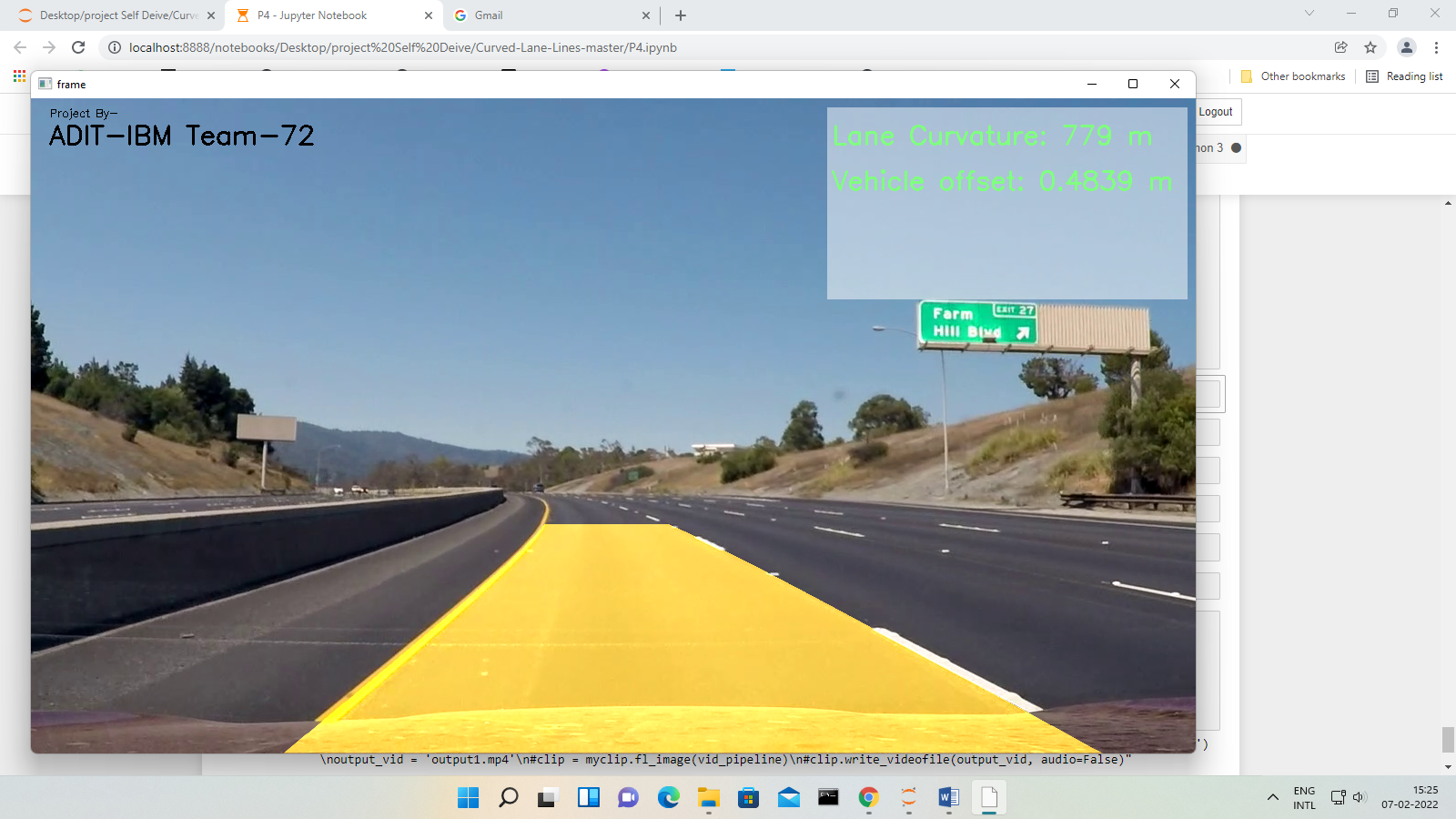
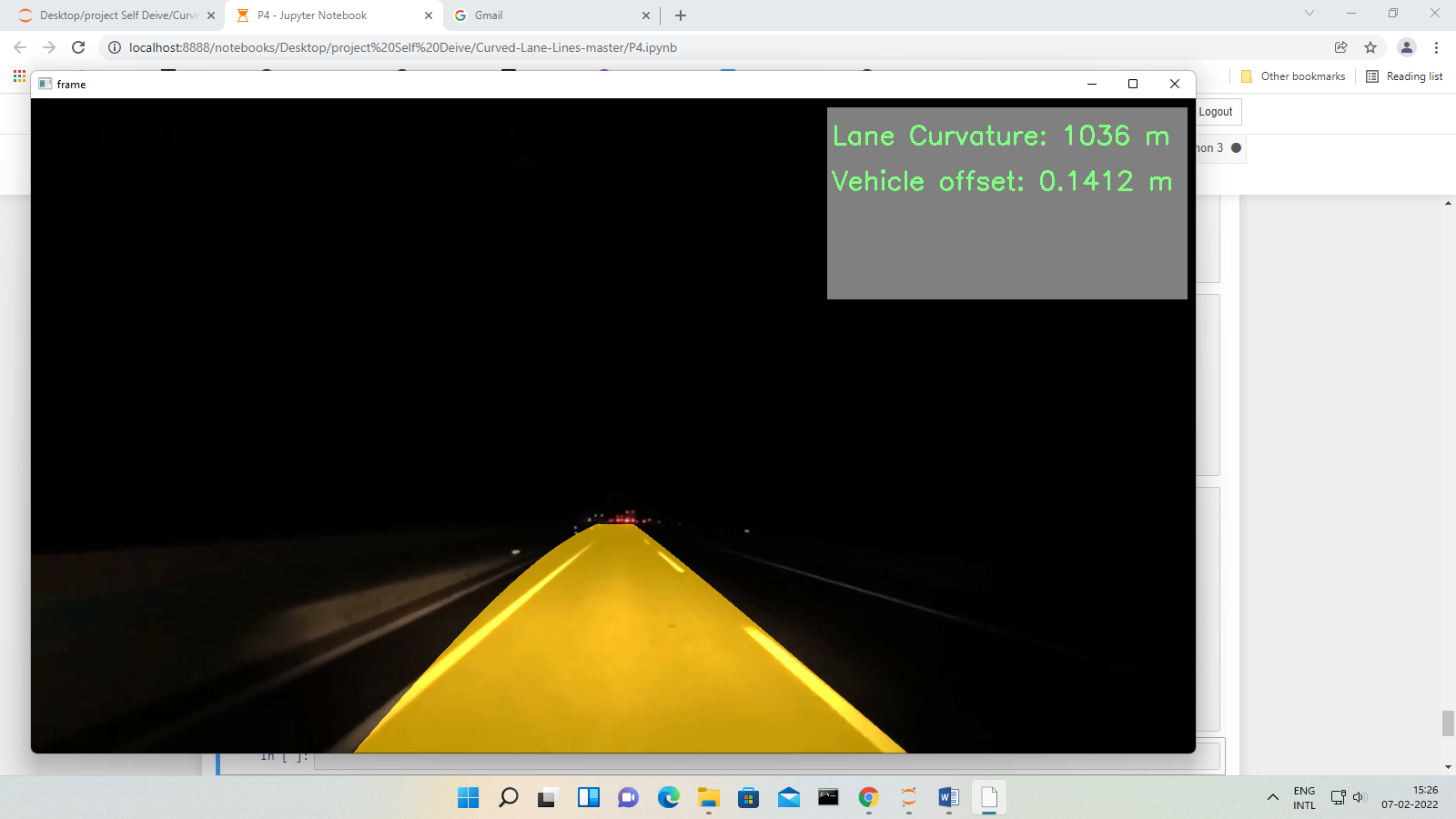


Figure 9 Final Output Night

Figure 10 Final Output

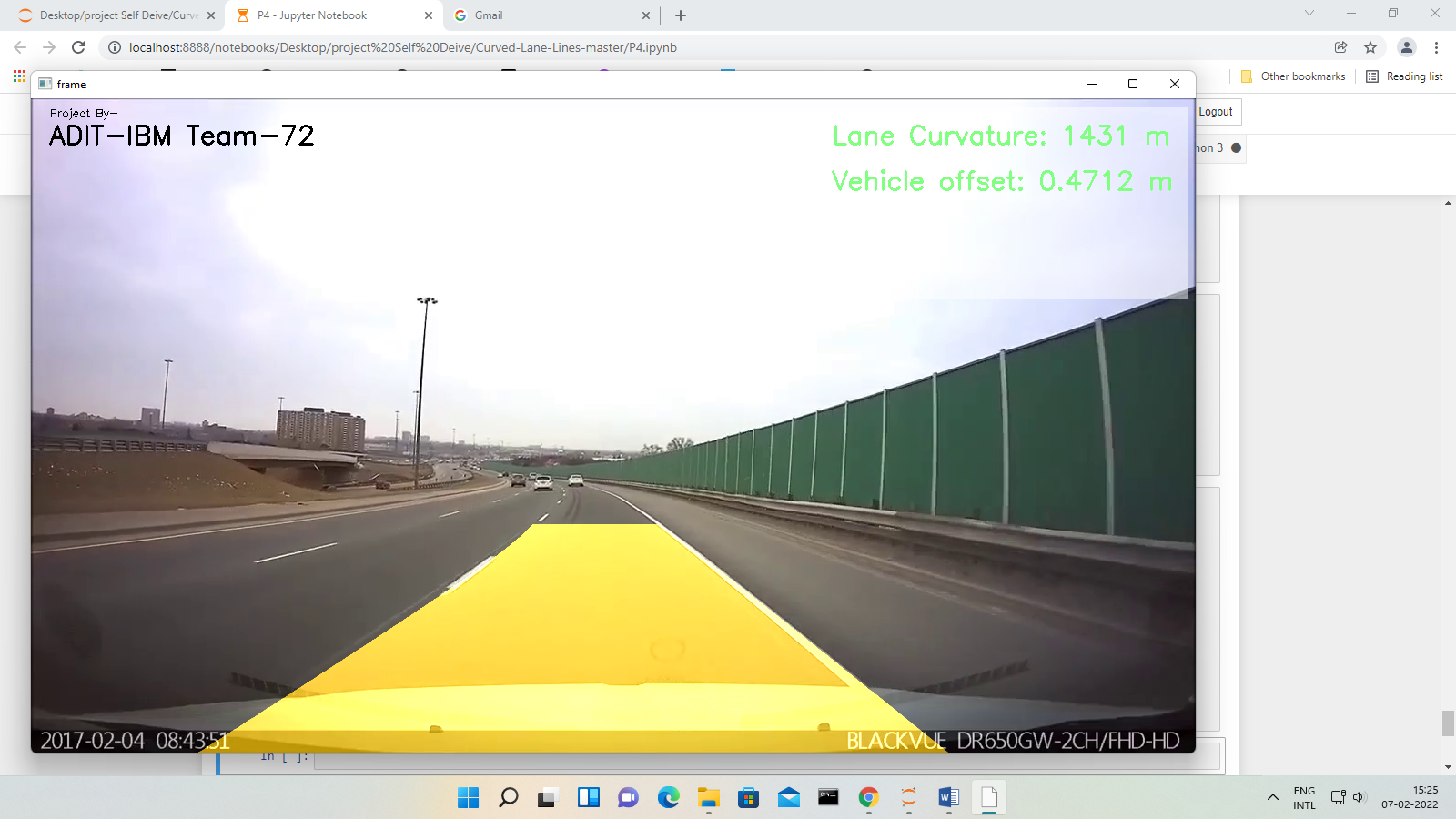
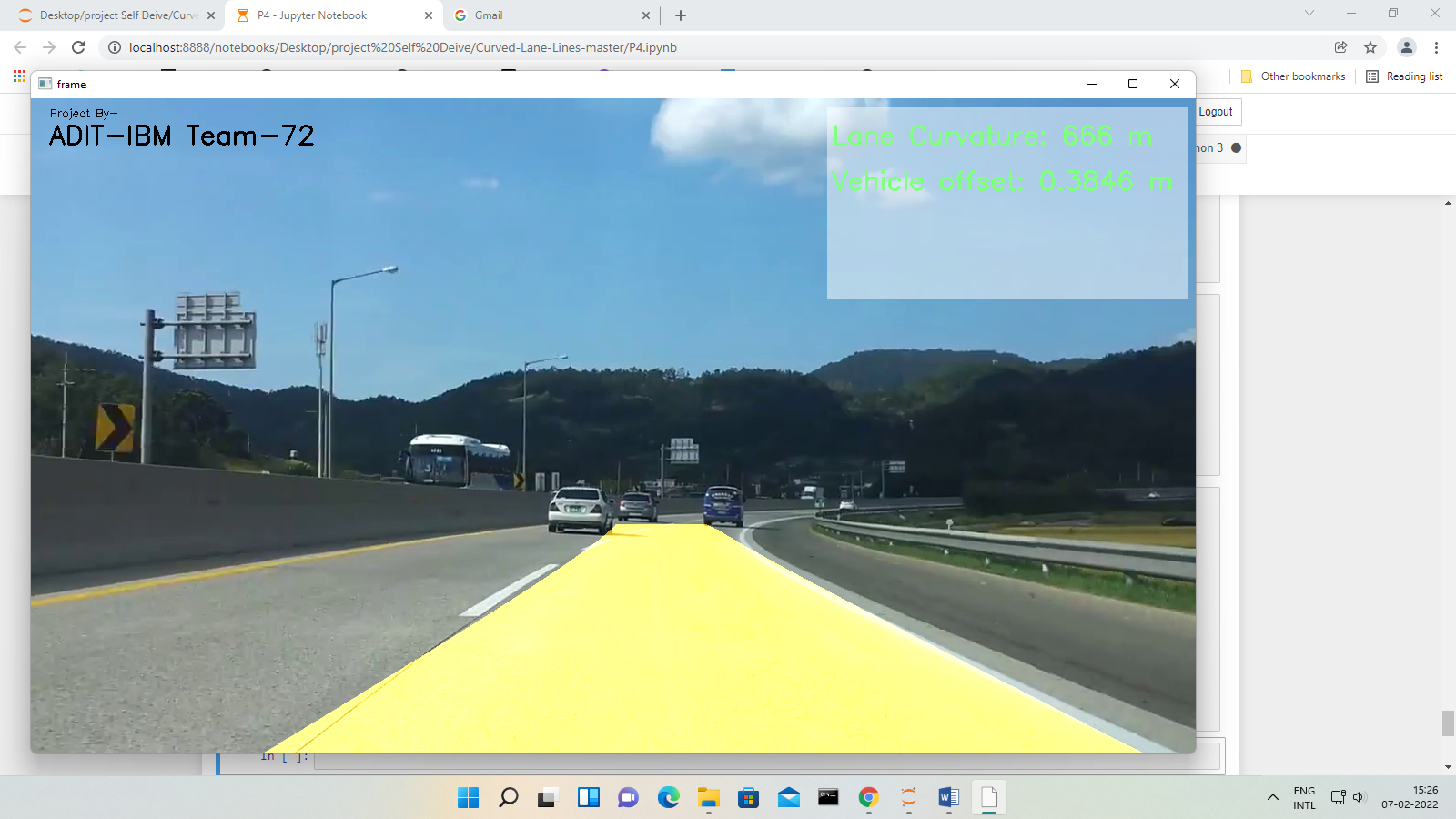


Figure 11Final Output

Figure 12 Final Output

Testing

We tested it on various Road and condition like foggy weather and Night. It will work fine. Some examples are shown below

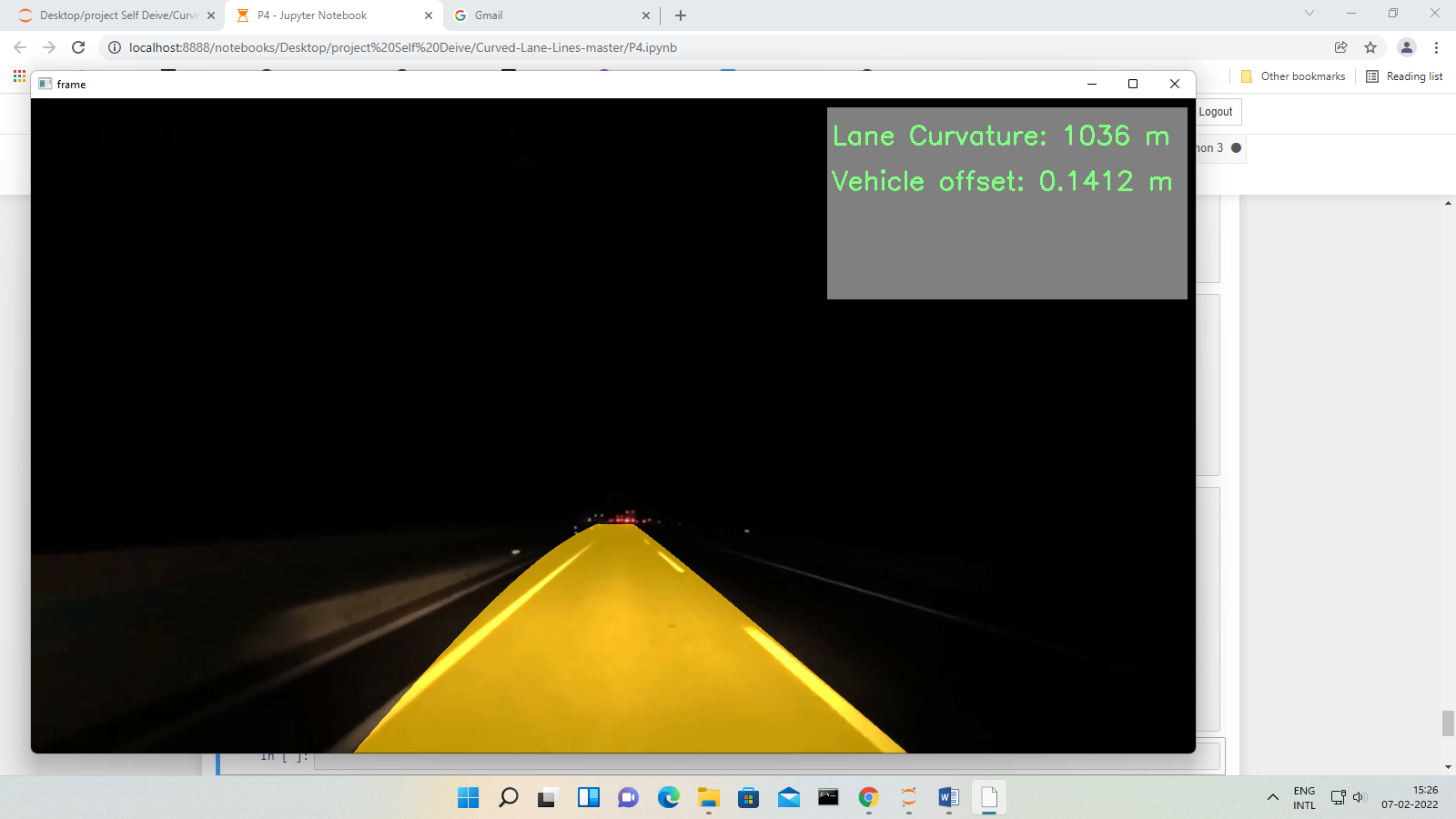


Figure 13 Night Camera

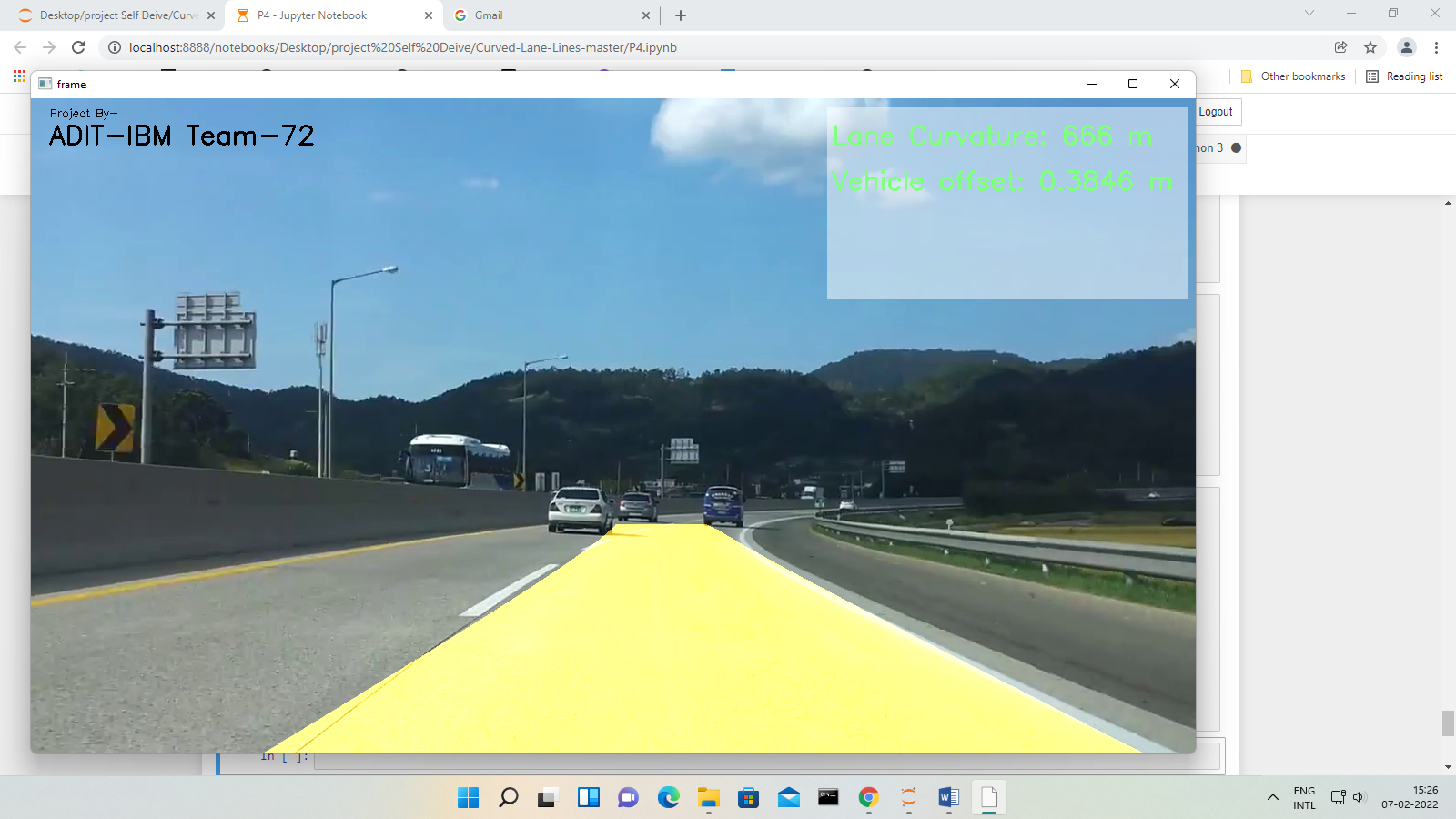


Figure 14 Testing



Figure 15 Testing

Deployment

It is ready to deployed on any cloud platform line AWS, Azure, GCP and IBMCloud.

Project Scope

This model can be updated and tuned with more efficient mathematical modelling, whereas the classical OpenCV approach is limited and no upgrade is possible as the approach is not efficient It is unable to give accurate results on the roads which do not have clear markings present on the roads. Also it cannot work for all climatic conditions This technology is increasing the number of applications such as traffic control, traffic monitoring, traffic flow, security etc.

Conclusion

In this project, we proposed a new lane detection pre-processing and ROI selection methods to design a lane detection system. The main idea is to add white extraction before the conventional basic pre-processing. Edge extraction has also been added during the pre-processing stage to improve lane detection accuracy. We also placed the ROI selection after the proposed pre-processing. Compared with selecting the ROI in the original image, it reduced the nonlane parameters and improved the accuracy of lane detection. Currently, we only use the Hough transform to detect straight lane and EKF to track lane and do not develop advanced lane detection methods. In the future, we will exploit a more advanced lane detection approach to improve the performance.

Appendix A

Project Code

import numpy as np

import pandas as pd

import cv2

import os

import glob

import matplotlib.pyplot as plt

import pickle

%matplotlib inline

def undistort\_img():

# Prepare object points 0,0,0 ... 8,5,0

obj\_pts = np.zeros((6\*9,3), np.float32)

obj\_pts[:,:2] = np.mgrid[0:9, 0:6].T.reshape(-1,2)

# Stores all object points & img points from all images

objpoints = []

imgpoints = []

# Get directory for all calibration images

images = glob.glob('camera\_cal/\*.jpg')

for indx, fname in enumerate(images):

img = cv2.imread(fname)

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

ret, corners = cv2.findChessboardCorners(gray, (9,6), None)

if ret == True:

objpoints.append(obj\_pts)

imgpoints.append(corners)

# Test undistortion on img

img\_size = (img.shape[1], img.shape[0])

# Calibrate camera

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img\_size, None,None)

dst = cv2.undistort(img, mtx, dist, None, mtx)

# Save camera calibration for later use

dist\_pickle = {}

dist\_pickle['mtx'] = mtx

dist\_pickle['dist'] = dist

pickle.dump( dist\_pickle, open('camera\_cal/cal\_pickle.p', 'wb') )

def undistort(img, cal\_dir='camera\_cal/cal\_pickle.p'):

#cv2.imwrite('camera\_cal/test\_cal.jpg', dst)

with open(cal\_dir, mode='rb') as f:

file = pickle.load(f)

mtx = file['mtx']

dist = file['dist']

dst = cv2.undistort(img, mtx, dist, None, mtx)

return dst

img = cv2.imread('camera\_cal/calibration1.jpg')

dst = undistort(img)

#plt.savefig("savefig\_img/distroteding.png")

# Visualize undistortion

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))

ax1.imshow(img)

ax1.set\_title('Original Image', fontsize=30)

ax2.imshow(dst)

ax2.set\_title('Undistorted Image', fontsize=30)

img = cv2.imread('test\_images/test3.jpg')

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

dst = undistort(img)

# Visualize undistortion

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))

ax1.imshow(img)

ax1.set\_title('Original Image', fontsize=30)

ax2.imshow(dst)

ax2.set\_title('Undistorted Image', fontsize=30)

def pipeline(img, s\_thresh=(100, 255), sx\_thresh=(15, 255)):

img = undistort(img)

img = np.copy(img)

# Convert to HLS color space and separate the V channel

hls = cv2.cvtColor(img, cv2.COLOR\_RGB2HLS).astype(np.float)

l\_channel = hls[:,:,1]

s\_channel = hls[:,:,2]

h\_channel = hls[:,:,0]

# Sobel x

sobelx = cv2.Sobel(l\_channel, cv2.CV\_64F, 1, 1) # Take the derivative in x

abs\_sobelx = np.absolute(sobelx) # Absolute x derivative to accentuate lines away from horizontal

scaled\_sobel = np.uint8(255\*abs\_sobelx/np.max(abs\_sobelx))

# Threshold x gradient

sxbinary = np.zeros\_like(scaled\_sobel)

sxbinary[(scaled\_sobel >= sx\_thresh[0]) & (scaled\_sobel <= sx\_thresh[1])] = 1

# Threshold color channel

s\_binary = np.zeros\_like(s\_channel)

s\_binary[(s\_channel >= s\_thresh[0]) & (s\_channel <= s\_thresh[1])] = 1

color\_binary = np.dstack((np.zeros\_like(sxbinary), sxbinary, s\_binary)) \* 255

combined\_binary = np.zeros\_like(sxbinary)

combined\_binary[(s\_binary == 1) | (sxbinary == 1)] = 1

return combined\_binary

def perspective\_warp(img,

dst\_size=(1280,720),

src=np.float32([(0.43,0.65),(0.58,0.65),(0.1,1),(1,1)]),

dst=np.float32([(0,0), (1, 0), (0,1), (1,1)])):

img\_size = np.float32([(img.shape[1],img.shape[0])])

src = src\* img\_size

# For destination points, I'm arbitrarily choosing some points to be

# a nice fit for displaying our warped result

# again, not exact, but close enough for our purposes

dst = dst \* np.float32(dst\_size)

# Given src and dst points, calculate the perspective transform matrix

M = cv2.getPerspectiveTransform(src, dst)

# Warp the image using OpenCV warpPerspective()

warped = cv2.warpPerspective(img, M, dst\_size)

return warped

def inv\_perspective\_warp(img,

dst\_size=(1280,720),

src=np.float32([(0,0), (1, 0), (0,1), (1,1)]),

dst=np.float32([(0.43,0.65),(0.58,0.65),(0.1,1),(1,1)])):

img\_size = np.float32([(img.shape[1],img.shape[0])])

src = src\* img\_size

# For destination points, I'm arbitrarily choosing some points to be

# a nice fit for displaying our warped result

# again, not exact, but close enough for our purposes

dst = dst \* np.float32(dst\_size)

# Given src and dst points, calculate the perspective transform matrix

M = cv2.getPerspectiveTransform(src, dst)

# Warp the image using OpenCV warpPerspective()

warped = cv2.warpPerspective(img, M, dst\_size)

return warped

def get\_hist(img):

hist = np.sum(img[img.shape[0]//2:,:], axis=0)

return hist

img = cv2.imread('test\_images/test3.jpg')

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

dst = pipeline(img)

dst = perspective\_warp(dst, dst\_size=(1280,720))

# Visualize undistortion

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))

ax1.imshow(img)

ax1.set\_title('Original Image', fontsize=30)

ax2.imshow(dst, cmap='gray')

ax2.set\_title('Warped Image', fontsize=30)

left\_a, left\_b, left\_c = [],[],[]

right\_a, right\_b, right\_c = [],[],[]

def sliding\_window(img, nwindows=9, margin=150, minpix = 1, draw\_windows=True):

global left\_a, left\_b, left\_c,right\_a, right\_b, right\_c

left\_fit\_= np.empty(3)

right\_fit\_ = np.empty(3)

out\_img = np.dstack((img, img, img))\*255

histogram = get\_hist(img)

# find peaks of left and right halves

midpoint = int(histogram.shape[0]/2)

leftx\_base = np.argmax(histogram[:midpoint])

rightx\_base = np.argmax(histogram[midpoint:]) + midpoint

# Set height of windows

window\_height = np.int(img.shape[0]/nwindows)

# Identify the x and y positions of all nonzero pixels in the image

nonzero = img.nonzero()

nonzeroy = np.array(nonzero[0])

nonzerox = np.array(nonzero[1])

# Current positions to be updated for each window

leftx\_current = leftx\_base

rightx\_current = rightx\_base

# Create empty lists to receive left and right lane pixel indices

left\_lane\_inds = []

right\_lane\_inds = []

# Step through the windows one by one

for window in range(nwindows):

# Identify window boundaries in x and y (and right and left)

win\_y\_low = img.shape[0] - (window+1)\*window\_height

win\_y\_high = img.shape[0] - window\*window\_height

win\_xleft\_low = leftx\_current - margin

win\_xleft\_high = leftx\_current + margin

win\_xright\_low = rightx\_current - margin

win\_xright\_high = rightx\_current + margin

# Draw the windows on the visualization image

if draw\_windows == True:

cv2.rectangle(out\_img,(win\_xleft\_low,win\_y\_low),(win\_xleft\_high,win\_y\_high),

(100,255,255), 3)

cv2.rectangle(out\_img,(win\_xright\_low,win\_y\_low),(win\_xright\_high,win\_y\_high),

(100,255,255), 3)

# Identify the nonzero pixels in x and y within the window

good\_left\_inds = ((nonzeroy >= win\_y\_low) & (nonzeroy < win\_y\_high) &

(nonzerox >= win\_xleft\_low) & (nonzerox < win\_xleft\_high)).nonzero()[0]

good\_right\_inds = ((nonzeroy >= win\_y\_low) & (nonzeroy < win\_y\_high) &

(nonzerox >= win\_xright\_low) & (nonzerox < win\_xright\_high)).nonzero()[0]

# Append these indices to the lists

left\_lane\_inds.append(good\_left\_inds)

right\_lane\_inds.append(good\_right\_inds)

# If you found > minpix pixels, recenter next window on their mean position

if len(good\_left\_inds) > minpix:

leftx\_current = np.int(np.mean(nonzerox[good\_left\_inds]))

if len(good\_right\_inds) > minpix:

rightx\_current = np.int(np.mean(nonzerox[good\_right\_inds]))

# if len(good\_right\_inds) > minpix:

# rightx\_current = np.int(np.mean([leftx\_current +900, np.mean(nonzerox[good\_right\_inds])]))

# elif len(good\_left\_inds) > minpix:

# rightx\_current = np.int(np.mean([np.mean(nonzerox[good\_left\_inds]) +900, rightx\_current]))

# if len(good\_left\_inds) > minpix:

# leftx\_current = np.int(np.mean([rightx\_current -900, np.mean(nonzerox[good\_left\_inds])]))

# elif len(good\_right\_inds) > minpix:

# leftx\_current = np.int(np.mean([np.mean(nonzerox[good\_right\_inds]) -900, leftx\_current]))

# Concatenate the arrays of indices

left\_lane\_inds = np.concatenate(left\_lane\_inds)

right\_lane\_inds = np.concatenate(right\_lane\_inds)

# Extract left and right line pixel positions

leftx = nonzerox[left\_lane\_inds]

lefty = nonzeroy[left\_lane\_inds]

rightx = nonzerox[right\_lane\_inds]

righty = nonzeroy[right\_lane\_inds]

# Fit a second order polynomial to each

left\_fit = np.polyfit(lefty, leftx, 2)

right\_fit = np.polyfit(righty, rightx, 2)

left\_a.append(left\_fit[0])

left\_b.append(left\_fit[1])

left\_c.append(left\_fit[2])

right\_a.append(right\_fit[0])

right\_b.append(right\_fit[1])

right\_c.append(right\_fit[2])

left\_fit\_[0] = np.mean(left\_a[-10:])

left\_fit\_[1] = np.mean(left\_b[-10:])

left\_fit\_[2] = np.mean(left\_c[-10:])

right\_fit\_[0] = np.mean(right\_a[-10:])

right\_fit\_[1] = np.mean(right\_b[-10:])

right\_fit\_[2] = np.mean(right\_c[-10:])

# Generate x and y values for plotting

ploty = np.linspace(0, img.shape[0]-1, img.shape[0] )

left\_fitx = left\_fit\_[0]\*ploty\*\*2 + left\_fit\_[1]\*ploty + left\_fit\_[2]

right\_fitx = right\_fit\_[0]\*ploty\*\*2 + right\_fit\_[1]\*ploty + right\_fit\_[2]

out\_img[nonzeroy[left\_lane\_inds], nonzerox[left\_lane\_inds]] = [255, 0, 100]

out\_img[nonzeroy[right\_lane\_inds], nonzerox[right\_lane\_inds]] = [0, 100, 255]

return out\_img, (left\_fitx, right\_fitx), (left\_fit\_, right\_fit\_), ploty

def get\_curve(img, leftx, rightx):

ploty = np.linspace(0, img.shape[0]-1, img.shape[0])

y\_eval = np.max(ploty)

ym\_per\_pix = 30.5/720 # meters per pixel in y dimension

xm\_per\_pix = 3.7/720 # meters per pixel in x dimension

# Fit new polynomials to x,y in world space

left\_fit\_cr = np.polyfit(ploty\*ym\_per\_pix, leftx\*xm\_per\_pix, 2)

right\_fit\_cr = np.polyfit(ploty\*ym\_per\_pix, rightx\*xm\_per\_pix, 2)

# Calculate the new radii of curvature

left\_curverad = ((1 + (2\*left\_fit\_cr[0]\*y\_eval\*ym\_per\_pix + left\_fit\_cr[1])\*\*2)\*\*1.5) / np.absolute(2\*left\_fit\_cr[0])

right\_curverad = ((1 + (2\*right\_fit\_cr[0]\*y\_eval\*ym\_per\_pix + right\_fit\_cr[1])\*\*2)\*\*1.5) / np.absolute(2\*right\_fit\_cr[0])

car\_pos = img.shape[1]/2

l\_fit\_x\_int = left\_fit\_cr[0]\*img.shape[0]\*\*2 + left\_fit\_cr[1]\*img.shape[0] + left\_fit\_cr[2]

r\_fit\_x\_int = right\_fit\_cr[0]\*img.shape[0]\*\*2 + right\_fit\_cr[1]\*img.shape[0] + right\_fit\_cr[2]

lane\_center\_position = (r\_fit\_x\_int + l\_fit\_x\_int) /2

center = (car\_pos - lane\_center\_position) \* xm\_per\_pix / 10

#print("left - ",left\_curverad,"Right - ", right\_curverad,"Center - ", center)

# Now our radius of curvature is in meters

return (left\_curverad, right\_curverad, center)

def draw\_lanes(img, left\_fit, right\_fit):

ploty = np.linspace(0, img.shape[0]-1, img.shape[0])

color\_img = np.zeros\_like(img)

left = np.array([np.transpose(np.vstack([left\_fit, ploty]))])

right = np.array([np.flipud(np.transpose(np.vstack([right\_fit, ploty])))])

points = np.hstack((left, right))

cv2.fillPoly(color\_img, np.int\_(points), (0,200,255))

inv\_perspective = inv\_perspective\_warp(color\_img)

inv\_perspective = cv2.addWeighted(img, 1, inv\_perspective, 0.7, 0)

return inv\_perspective

#%matplotlib gtk

plt.figure(figsize=(10,10))

out\_img, curves, lanes, ploty = sliding\_window(dst)

plt.imshow(out\_img)

#plt.savefig("savefig\_img/sliding window.png")

plt.figure(figsize=(10,7))

plt.plot(curves[0], ploty, color='green', linewidth=1)

plt.plot(curves[1], ploty, color='green', linewidth=1)

#plt.savefig("savefig\_img/CurvatureLines.png")

print(np.asarray(curves).shape)

curverad=get\_curve(img, curves[0],curves[1])

print(curverad)

img\_ = draw\_lanes(img, curves[0], curves[1])

plt.figure(figsize=(12,10))

plt.imshow(img\_, cmap='hsv')

f, (ax1, ax2, ax3, ax4, ax5) = plt.subplots(5, 1, figsize=(10, 20))

f.tight\_layout()

ax1.imshow(img)

# plt.savefig("savefig\_img/DrawLines.png")

ax1.set\_title('Original', fontsize=20)

ax2.imshow(dst)

#plt.savefig("savefig\_img/original.png")

ax2.set\_title('Filter+Perspective Tform', fontsize=20)

ax3.imshow(out\_img)

#plt.savefig("savefig\_img/Filter+Perspective Tform.png")

ax3.plot(curves[0], ploty, color='yellow', linewidth=30)

ax3.plot(curves[1], ploty, color='yellow', linewidth=30)

ax3.set\_title('Sliding window+Curve Fit', fontsize=20)

ax4.imshow(img\_)

#plt.savefig("savefig\_img/Sliding window+Curve Fit.png")

ax4.set\_title('Overlay Lanes', fontsize=20)

#plt.savefig("savefig\_img/Overlay Lanes.png")

#plt.subplots\_adjust(left=0., right=1, top=0.9, bottom=0.)

def vid\_pipeline(img):

global running\_avg

global index

img\_ = pipeline(img)

img\_ = perspective\_warp(img\_)

out\_img, curves, lanes, ploty = sliding\_window(img\_, draw\_windows=True)

curverad =get\_curve(img, curves[0], curves[1])

lane\_curve = np.mean([curverad[0], curverad[1]])

img = draw\_lanes(img, curves[0], curves[1])

font = cv2.FONT\_HERSHEY\_SIMPLEX

fontColor = (0, 255, 0)

fontSize=0.95

cv2.putText(img, 'Lane Curvature: {:.0f} m'.format(lane\_curve), (880, 50), font, fontSize, fontColor, 2)

cv2.putText(img, 'Vehicle offset: {:.4f} m'.format(curverad[2]), (880, 100), font, fontSize, fontColor, 2)

cv2.putText(img, "Project By-", (20, 20), font, .4, (0,0,0), 1)

cv2.putText(img, "ADIT-IBM Team-72 ", (20, 50), font, .95, (0,0,0), 2)

#plt.savefig("savefig\_img/CurvatureImage.png")

return img

def car\_detection(img):

grey = cv2.cvtColor(img,cv2.COLOR\_BGR2GRAY)

blur = cv2.GaussianBlur(grey,(5,5),0)

dilated = cv2.dilate(blur,np.ones((3,3)))

kernel = cv2.getStructuringElement(cv2.MORPH\_ELLIPSE, (2, 2))

closing = cv2.morphologyEx(dilated, cv2.MORPH\_CLOSE, kernel)

car\_cascade\_src = 'cars.xml'

car\_cascade = cv2.CascadeClassifier(car\_cascade\_src)

cars = car\_cascade.detectMultiScale(closing, 1.1, 1)

cnt = 0

for (x,y,w,h) in cars:

cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

cnt += 1

return img, cnt

def dashboard(img):

shapes = np.zeros\_like(img, np.uint8)

cv2.rectangle(shapes, (875, 10), (1270, 220), (255,255,255), cv2.FILLED)

out = img.copy()

alpha = 0.5

mask = shapes.astype(bool)

out[mask] = cv2.addWeighted(img, alpha, shapes, 1 - alpha, 0)[mask]

return out

cap = cv2.VideoCapture("videos/challenge\_video.mp4")

while cap.isOpened():

ret, frame = cap.read()

if ret:

frames = vid\_pipeline(frame)

#frames1, \_ = car\_detection(frames)

frames2 = dashboard(frames)

cv2.imshow('frame', frames2)

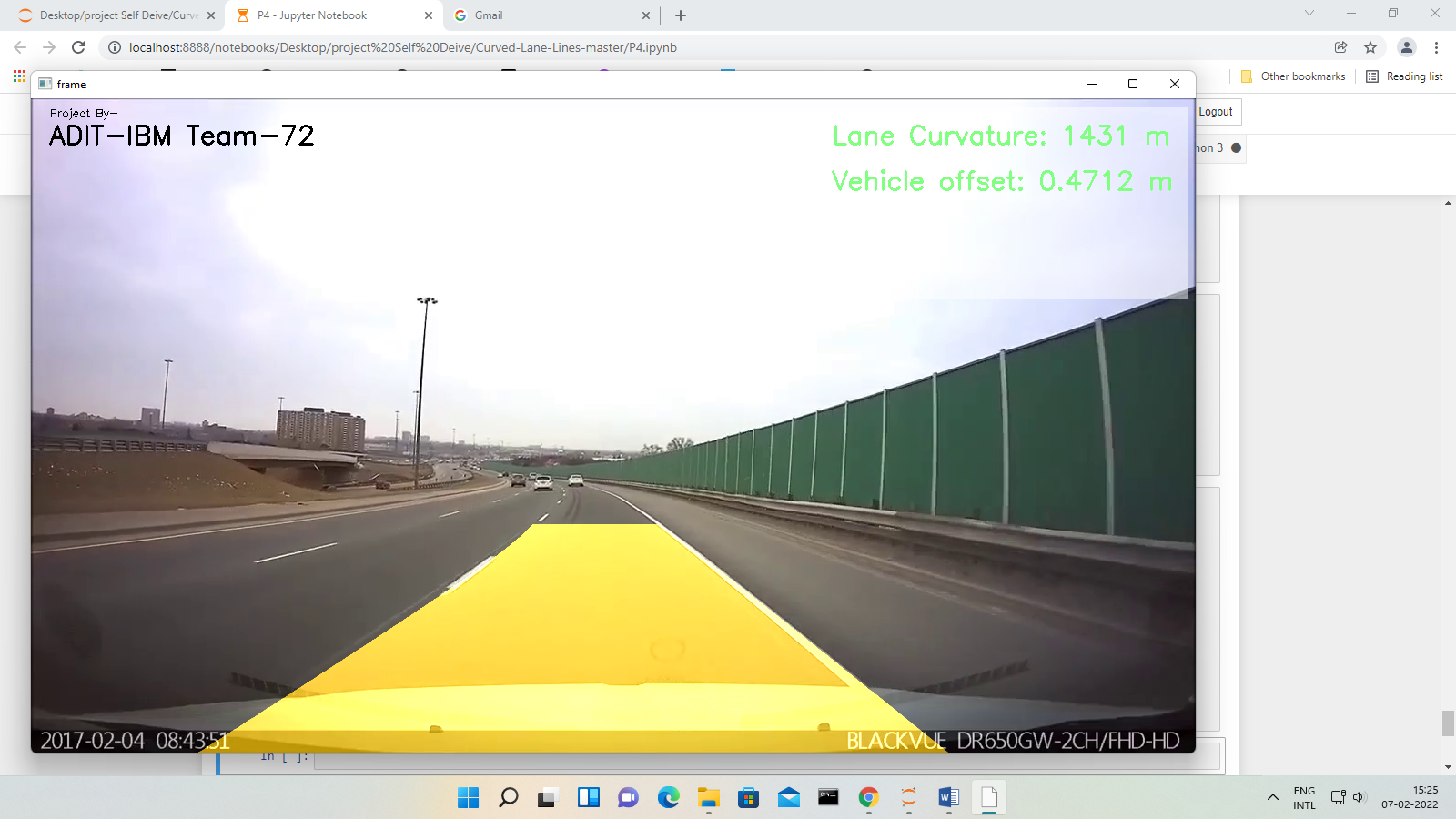
k = cv2.waitKey(30) & 0xff

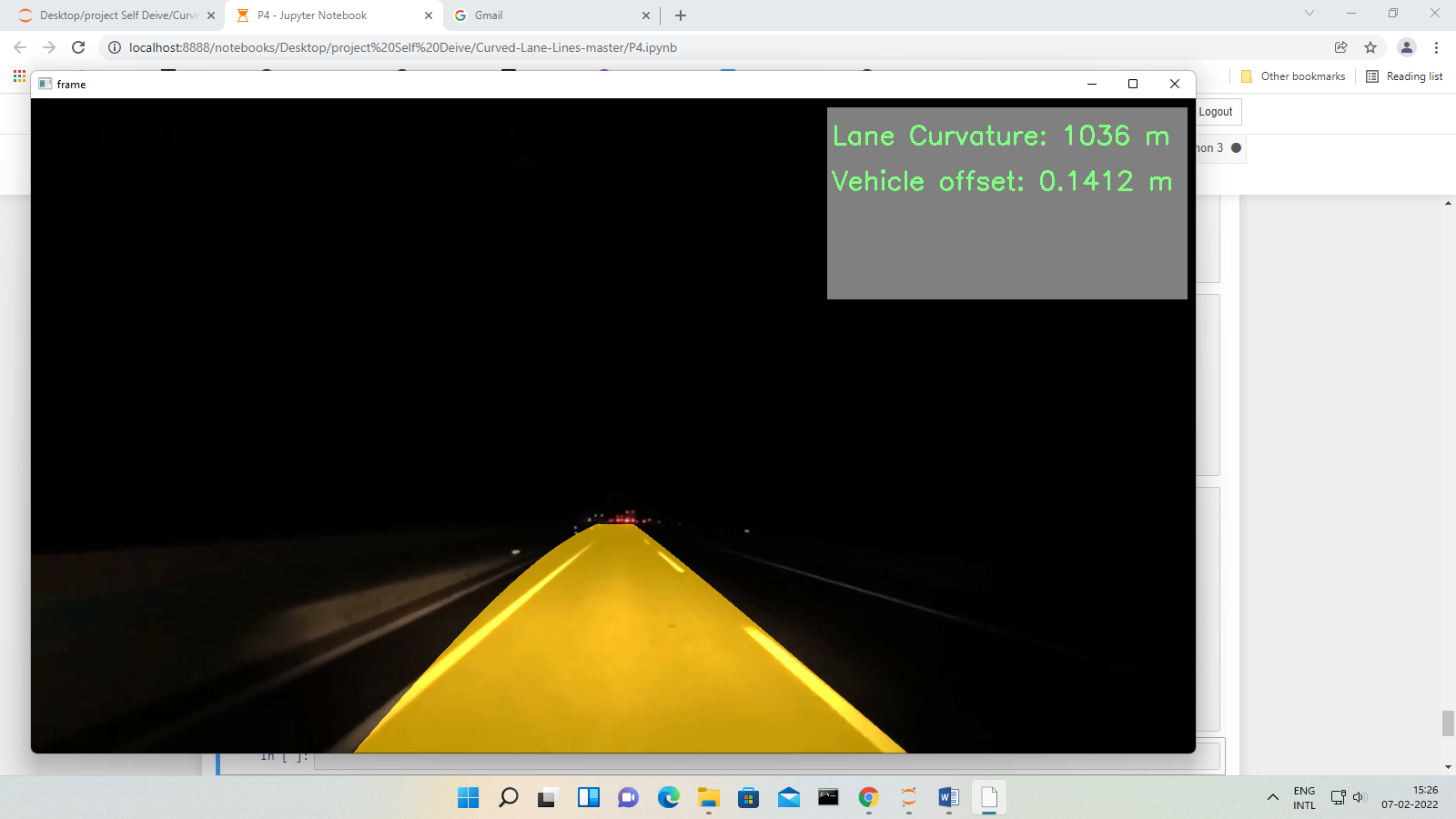
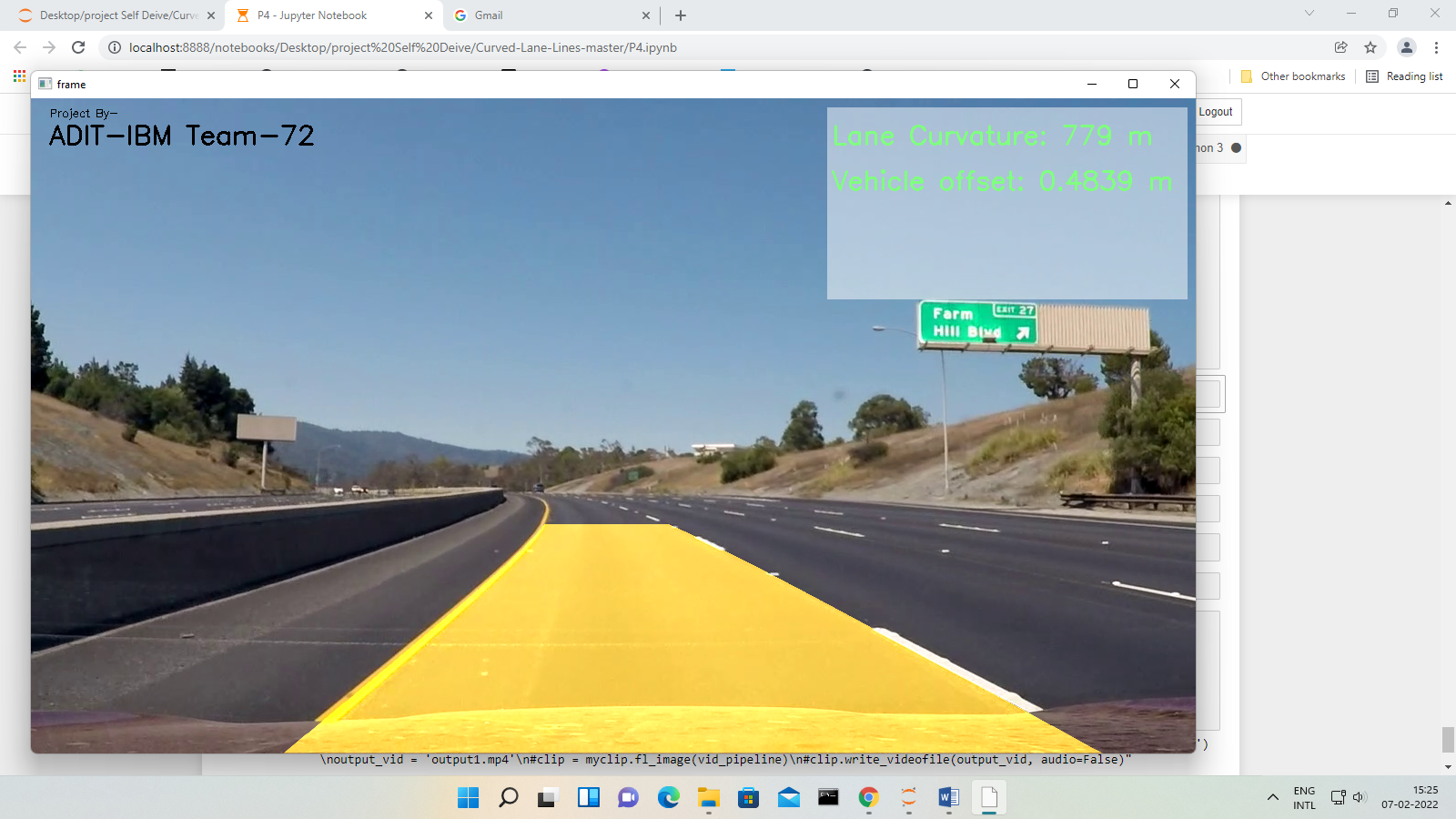
if k == 27:

break

cap.release()

cv2.destroyAllWindows()

Screenshot of the Projects



Abbreviation

ADAS Advance Driver Assistance System

AI Artificial Intelligence

APS Active-Pixel Sensor

AVB Audio/Video Bridging

CAN Controller Area Network

CCD Charge-Coupled Device

CMOS Complementary Metal-Oxide Semiconductor

CNS Car Navigation Systems.

CV Computer Vision

EVS Embedded Vision Systems

FPGA Field Programmable Gate Array

FPS Frame Per Second

GPS Global Positioning System

GPU Graphics Processing Unit

HMI Human Machine Interface

HT Hough Transform

ITS Intelligent Transport System

LDA Lane Detection Algorithm

LDM Local Dynamic Map

LDW Lane Detection Warning System

LLIP Low-Level Image Processing

ML Machine Learning

MOST Media Oriented Systems Transport

OpenCL Open Computing Language

OpenCV Open Computer Vision

ROI Region of Interest

SoC Systems-on-chip

V2V Vehicle-to-Vehicle