Lane Detection for urban and non-urban environments

Project Report

Submitted for:

ECE 59500: Introduction to 2D and 3D image processing

Submitted on:

04/27/2017

Submitted by :

Sreeram Venkitachalam

MS Electrical and Computer engineering

IUPUI

1. Objective

The objective of the project is to develop a lane detection algorithm which can be used in interstate highways as well as busy urban roads. The current algorithms work accurately for the highways but cannot work as well in the roads which have high traffic or other obstacles in the path. The algorithm should also be able to detect the side of the roads were proper lanes are not present by small tweaks in the algorithm.

2. Introduction

An autonomous car is a vehicle that is capable of sensing its environment and navigating without human input [2]. One of the key features of autonomous driving is Lane detection. Lane detection is, therefore extensively researched at present for improving the accuracy and precision of the algorithm. At present lane detection algorithms are accurate for interstate highways or Non-urban roads. However, the same algorithm may not work for urban roads. This is due to the presence of lot of objects, vehicles and other disturbances which make the algorithm unable to detect the lane edges on the roads. The purpose of this project is to make an algorithm and design flow for lane detection, study and compare the results with urban and inter-state roads. As suggested in [3] it will be difficult to detect the adjacent lanes as there maybe obstacles blocking the view of the lane edges.

Apart from Lane detection, the algorithm should also detect edges of the roads were lanes are not available. This will help in the betterment of autonomous driving in urban environment.

2.1.Work statement

The project will be done in Visual Studio 2013 with OpenCV 2.4.10. The present idea is to use a system with the following specification; The processor (64 bit) of the system will be Intel Core i7 – 7500 with CPU clock of 2.70GHz. The RAM of the system will be 16GB.

The project was done in Microsoft Visual Studio 2015 with OpenCV 3.1. The software was installed on a PC with the following specifications: - Intel Core i7 – 7500 processor was used with CPU clock of 2.70 GHz. The RAM of the PC used was 16GB. The advantage of using OpenCV 3.x as compared to 2 .x is that it will have most functions added in a single library file namely, ***opencv_world310d.lib.*** The Project uses OpenCV with visual studio. The First step in setting up the solution involves the inclusion of additional include libraries for OpenCV add the set of libraries required (*C:\opencv3_1\opencv\build\include*). As I am using OpenCV 3.1 with Visual studio 2015, I have changed the additional dependencies as required for OpenCV 3.1 (*C:\opencv3_1\opencv\build\x64\vc14\lib;*). The solution works on x64; therefore, it is to be made sure that the program is run on the x64. In the properties of the project, in the C/C++ and the Linker tabs make sure to include the required OpenCV library location as additional directories. ***opencv_world310.lib and opencv_world310d.lib*** are the additional Libraries to be added on the Linker input in the properties of the Project. This varies for each OpenCV version. Once the project is setup, kindly place the address of the video in the Command Argument in the property window of the project. This will complete the process for the setup.

3. Related work

In [1], the authors design a new method for detection of Lane for autonomous vehicle using Non-Uniform B-spline interpolation (NUBS). The author uses image enhancement techniques to remove the noise. Then the image is checked for edges and thinned so that the boundaries are well defined for the NUBS interpolation technique. The control points obtained from the NUBS interpolation is used for drawing the boundaries of the Lane. The correct control points are selected sing the vector lane technique. Finally, a mathematical model is introduced for estimating the left and right curvatures of the lane. In the conclusion, the author discusses his result and suggests that the algorithm works even in the case of severe noise and can be applied for lane detection in Autonomous Guided Vehicles.

Paper [3] discusses a method for lane detection in structured roads (Highways). Flow of the algorithm is as follows.

a. Preprocessing
b. Central lane detection
c. Simplified perspective transformation
d. Adjacent lanes estimation
e. Adjacent lanes detection

The author divides his algorithm into 2 parts

a. The case of a straight line
   Central lane is detected using Hough transformation and then simplified perspective transformation is applied to the detection result for estimating the adjacent lanes. Preprocessing is done to find out the vanishing point. The ROI is given as the image below the vanishing point. Those lane markings that are detected by obstacles in the adjacent lanes can be estimated using the affine transformation.

b. The case of curve
   The following will be the work flow for the curves:
   i. Edge detection
   ii. Perspective transformation
   iii. Central Lane detection
   iv. Adjacent lane estimation

The author performs a perspective transformation of camera installation parameters. Then we obtain a central lane marks by scanning techniques and fitting these points with a circular curve. The cameras pitch yaw and roll angles are required, which will be obtained from the knowledge of vanishing point and the image center. With the knowledge of the pitch yaw roll we can find the perspective transformation using the formula.

In paper [4], the authors analyses the result of lane detection using 2 methods namely, receiver operating Characteristic (ROC) curve and Detection Error Tradeoff (DET) curve. This paper was studied to get an idea of the performance measuring parameters which is to be tested for checking the accuracy and precision of the algorithm implemented.

The authors in paper [5] discuss their algorithm of detecting the Lane markers in urban areas. The paper uses inverse perspective mapping which is a top view based mapping technique. The image filtering is

performed and the threshold is adjusted so that only the highest values of edges remain. Straight line is then drawn using the Hough Transform flowed by the RANSAC line fitting step, and then a novel RANSAC spline fitting step is performed to refine the detected straight lines and correctly detect the curves.

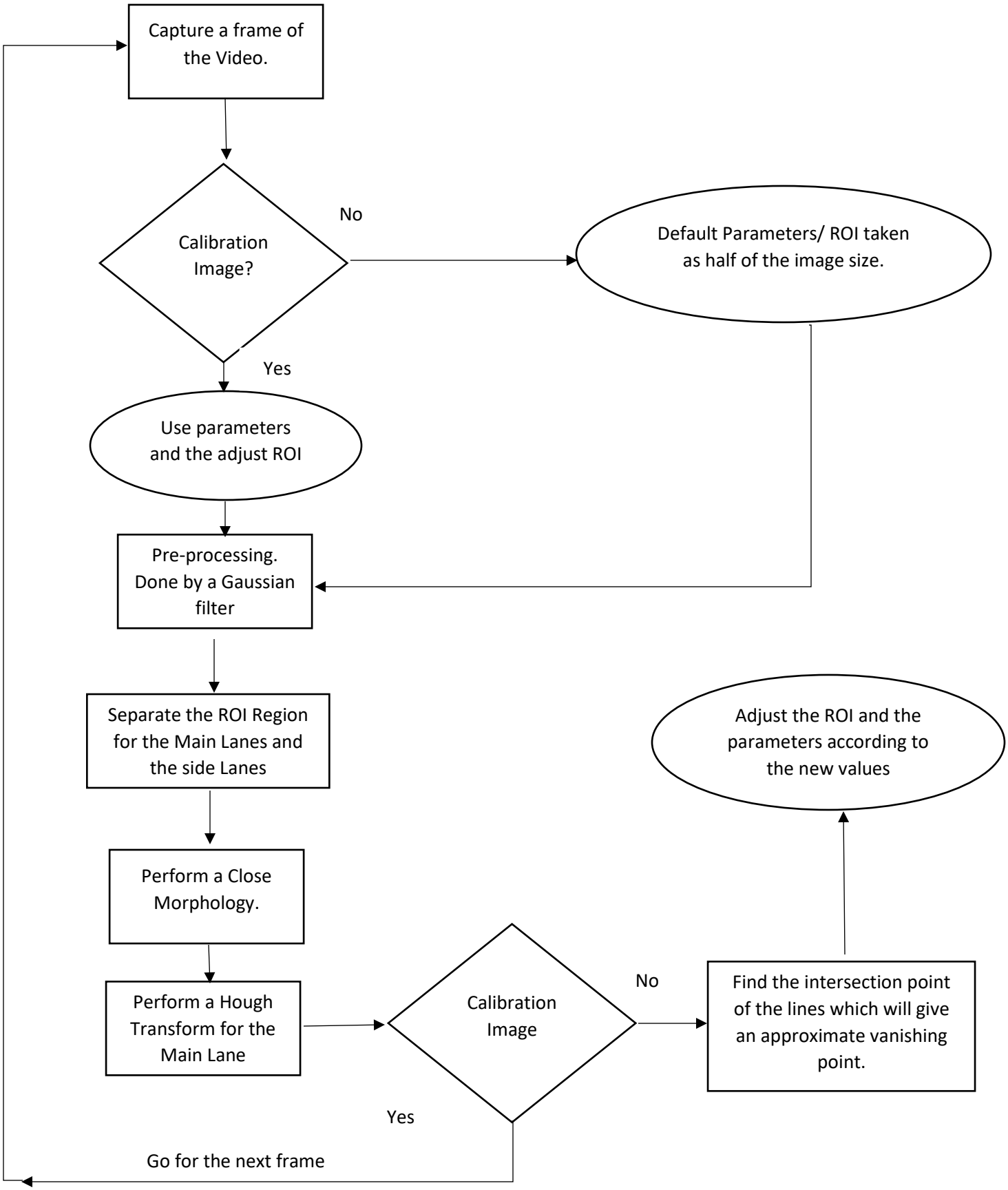
These papers present parts of the work which is planned to be done in the project. In this project as stated in the design flow it is planned to test and compare the efficiency of the Hough transform and the RANSAC algorithm separately. It is also planned that the project will also test a mixture of the Hough and the RANSAC together.

4. Project Design flow

The design flow of the project is given in the next page. The flow and different steps of the project is explained below using code snippets as well

a. Capture Image frame from video.
Image data can be captured from a video using the function *VideoCapture*. This function captures each frame of the video and processing can be done on the same. The function in the program is responsible for opening the video to be tested and to save output video. There was a reference taken for the development of this code. The reference was taken from [7]. The code snippet is given below:

```
        while (1)
        {
                Mat frame;
                capture >> frame;

                if (frame.empty())
                {
                        cout << "#### Info: Frame empty #####";
                        break;
                }

                _merge = frame.clone();
                _merge_pavement = frame.clone();

                FramePreproceesing(frame, Main_Lane, Main_Lane_NoBlur, Sub_Lane);
                hough(frame, Main_Lane, Main_Lane_NoBlur, Sub_Lane);

                outputVideo << _merge;

        }
return 0;
```


Capture a frame of the Video.
Calibration Image?
No
Default Parameters/ ROI taken as half of the image size.
Yes
Use parameters and the adjust ROI
Pre-processing. Done by a Gaussian filter
Separate the ROI Region for the Main Lanes and the side Lanes
Adjust the ROI and the parameters according to the new values
Perform a Close Morphology.
Perform a Hough Transform for the Main Lane
Calibration Image
No
Find the intersection point of the lines which will give an approximate vanishing point.
Yes
Go for the next frame

b. Condition for calibration
The frame is checked with respect to the calibration flag.

c. ROI/ Parameters.
If the images are calibration images, then the default parameters and the ROI is taken and the image is sent for preprocessing. If not, the previously calibrated values are sent in to the next step for preprocessing the image with respect to a modified ROI and parameters. The ROI parameters are given a default value of half the frame size. The code snippet for the same is given below:

```
ROI_paveY = frame_calib.rows / 2;
ROI_paveX = frame_calib.cols / 2;
ROI_Y = frame_calib.rows / 2;
```

d. Preprocessing the image.
It was planned that preprocessing will be done to remove the noise present in the images. Smoothening the image with a Gaussian 7x7 kernel helped in improving the efficiency of the Canny Edge Detector. The command for the same is given below:

```
GaussianBlur(src, src, Size(7, 7), 1.5, 1.5);
```

The images are then passed through Close Morphological operation so that there are no edges caused due to small shadows of on the road (Example: Shadows of wires on the road). The command lines for the same is given below:

```
morphologyEx(Main_Lane, Main_Lane, 3, element);
```

e. Selecting the region of interest.
The region of interest was selected based on the previously saved values (default for calibration images and calibrated values for the other frames). The region of interest will be different for the main lane and the side lanes. These are saved as different variables and sent in for edge detection. The region of interest is selected to increase the efficiency of the program. The code snippet for the same is given below:

```
Rect roi(ROI_X, ROI_Y_NextFrame, src.cols - ROI_X - 100, src.rows - ROI_Y_NextFrame);
Main_Lane = src(roi);
Main_Lane_NoBlur = _merge(roi);

Rect roi_pave(0, ROI_paveY, ROI_paveX, src.rows - ROI_paveY);
Sub_Lane = _merge(roi_pave);I_Y_NextFrame, src.cols - ROI_X - 100, src.rows - ROI_Y_NextFrame);
```

f. Edge Detection and Lane detection
The edge detection will be made using the Canny Edge Detector. There are separate Canny functions for side lane and the main lane. For the canny we will be able to modify the kernel size and the threshold values. These will be fixed once the calibration is done. The code snippet for the Canny Edge detector is given below:

```
Mat EdgeDetected = _image.clone();

blur(_image, EdgeDetected, Size(2 * kernel_size + 1, 2 * kernel_size + 1));
Canny(EdgeDetected, EdgeDetected, lowThresholdside, lowThresholdside*ratio, 2 * kernel_size + 1);

namedWindow("Canny_Filter_side", CV_WINDOW_AUTOSIZE);
imshow("Canny_Filter_side", EdgeDetected);

dest = EdgeDetected.clone();
```

The output of the Canny edge detector will go in to the Standard Hough transform. At the start a default threshold values are given in to the algorithm and with a calculated guess that the Lane and the roads will have a proper Edge, the number of lines that must be available on the Hough transform is made as 10. A step input increase in the threshold is given until the number of lines reduce to 10. The code for the Standard Hough Transform is given below, the code has some references from [8] for finding only the required Hough lines.

```
do
	{
		HoughLines(dest, MainLane_Lines, 1, CV_PI / 180, hough_threshold_main);
		hough_threshold_main = hough_threshold_main + 5;

	} while (MainLane_Lines.size() > 10);

	for (size_t i = 0; i < MainLane_Lines.size(); i++)
	{
		float rho = MainLane_Lines[i][0], theta = MainLane_Lines[i][1];

		if ((theta > CV_PI / 180 * 10 && theta < CV_PI / 180 * 80) || (theta > CV_PI / 180 *
100 && theta < CV_PI / 180 * 170))
		{
			double a = cos(theta), b = sin(theta);
			double x0 = a*rho, y0 = b*rho;

			pt1[Line_increment].x = cvRound(x0 + 1000 * (-b));
			pt1[Line_increment].y = cvRound(y0 + 1000 * (a));
			pt2[Line_increment].x = cvRound(x0 - 1000 * (-b));
			pt2[Line_increment].y = cvRound(y0 - 1000 * (a));
			line(Main_Lane_NoBlur, pt1[Line_increment], pt2[Line_increment], Scalar(0, 0,
255), 3, CV_AA);
			Line_increment++;
		}
	}
```

If the image is a calibration image then the intersection point of the lines are calculated. This is considered as the vanishing point. The ROI is then adjusted for each of the new frames. The code snippet for finding the intersection point is given below. This referenced from [8][9] [10].

```
        Point2f x = o2 - o1;
        Point2f d1 = p1 - o1;
        Point2f d2 = p2 - o2;

        float cross = d1.x*d2.y - d1.y*d2.x;
        if (abs(cross) < /*EPS*/1e-8)
                return false;

        double t1 = (x.x * d2.y - x.y * d2.x) / cross;
        r = o1 + d1 * t1;
        return true;
```

If the images are not of the calibration images then next set of ROIs is also taken for the Side Lanes and using Hough transforms the side lane is also drawn. The algorithm for the Side Lane is similar to the main lane as a result the code snippet is not put here.

For the detection of dashed lanes, the Probabilistic Hough transform is used. The values here are not calibrated and fixed values are used *threshold* = 40, *minLineLength* =20, *maxLineGap* =100. The obtained lines are then drawn on to the ROI image if the lines satisfy the angle criteria. The code snippet for the same is given below:

```
HoughLinesP(dest, lines, 1, 2 * CV_PI / 180, 40, 20, 100);

for (size_t i = 0; i < lines.size(); i++)
{
        float rho = lines[i][0], theta = lines[i][1];
        // Point P1 is represented as (x1,y1) and P2 is represented as (x2,y2)
        MainLane_intersectionpoint_houghP[0].x = lines[i][0];         // x1
        MainLane_intersectionpoint_houghP[0].y = lines[i][1];         // y1
        MainLane_intersectionpoint_houghP[1].x = lines[i][2];         // x2
        MainLane_intersectionpoint_houghP[1].y = lines[i][3];         // y2
        float angle = atan2(MainLane_intersectionpoint_houghP[0].y -
        MainLane_intersectionpoint_houghP[1].y, MainLane_intersectionpoint_houghP[0].x -
        MainLane_intersectionpoint_houghP[1].x);
        angle = 180 * angle / CV_PI;
        //cout << angle << endl;
        if ((angle < -100 && angle > -160) || (angle > 100 && angle < 160))  // Angle limits
        {

line(Main_Lane_NoBlur, MainLane_intersectionpoint_houghP[0], MainLane_intersectionpoint_houghP[1]
Scalar(0, 0, 255), 3, CV_AA);

        }
}
```

The ROI Image is then merged with the full image and then displayed. This value is also saved into *Output_Video.avi*

4.1.Project Schedule

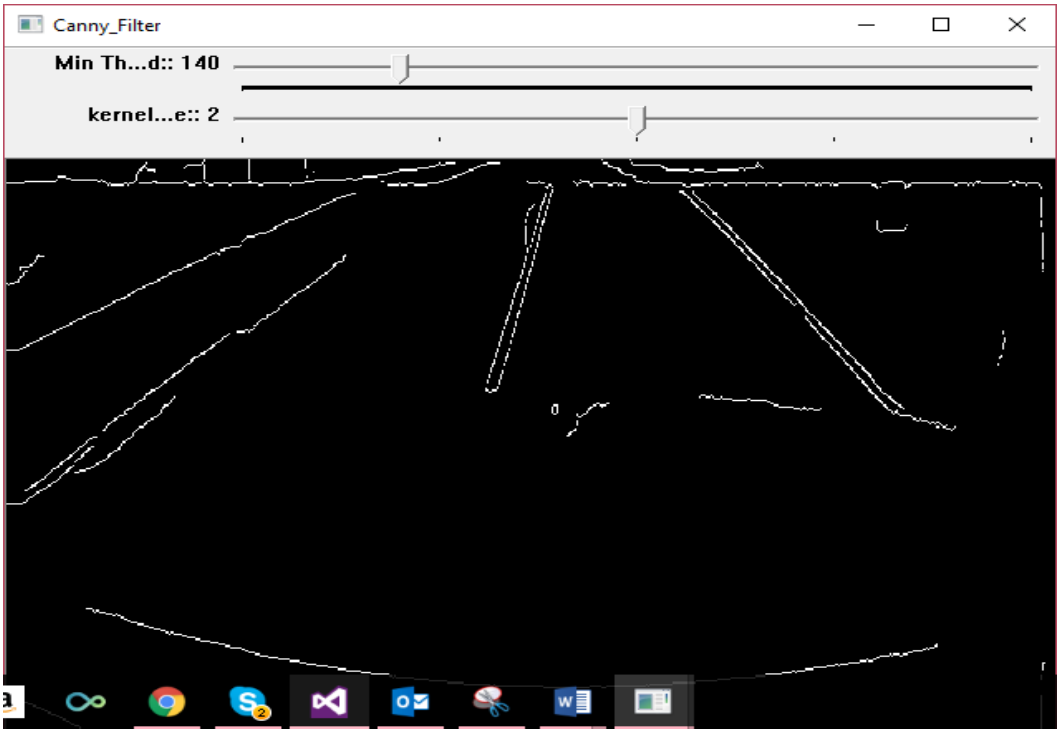
The project was completed according to the schedule below:

Task to be completed	Task details	Date of completion Expected	Date of completion
Region of interest	Region of interest is taken so that the edge detection and the Hough lines are taken only for the road.	04/17/2017	04/17/2017
Smoothening/ Preprocessing	To check the betterment of the edge detection algorithm we apply smoothening for the process.	04/20/2017	04/20/2017
Edge Detection and setting the threshold	Use of Canny Edge detector for the Hough transform.	04/20/2017	04/20/2017
Application of Hough Line transform	To the Edge detection the Canny edge detector has to be applied and the image should be calibrated for the urban roads. This was done by selecting a random image and adjusting the parameters.	04/24/2017	04/24/2017
Try with RANSAC Algorithm (If time permits)	Have not tried the algorithm with a rough code[5][6].	04/27/2017	Did not perform
Test and optimization	Trying the algorithm with different test cases optimization of code and obtaining results for different environments.	04/29/2017	04/26/2017
Final Project presentation and report submission		04/30/2017	Report 04/27/2017

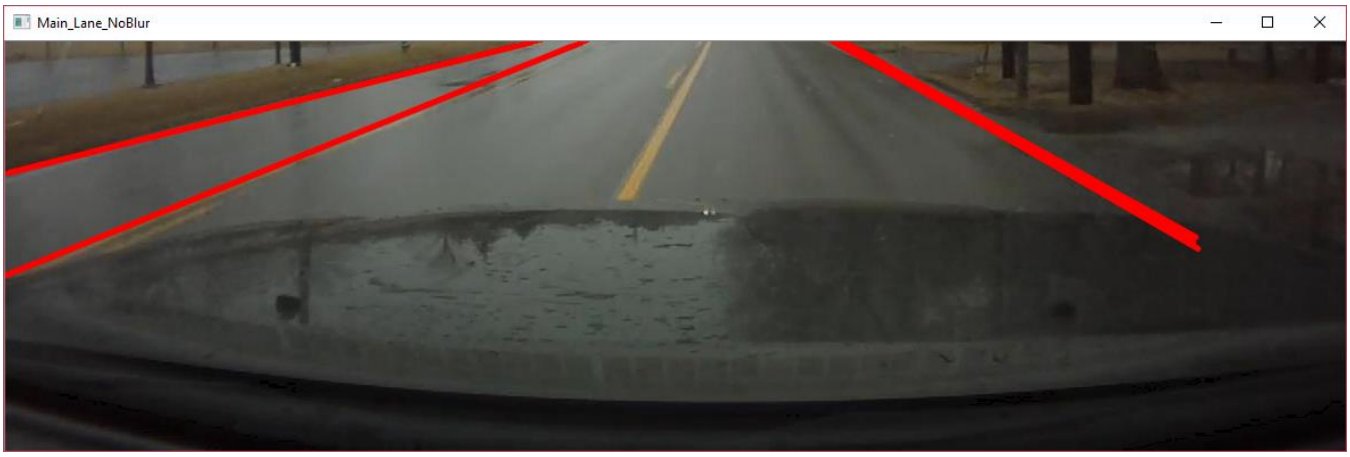
5. Results

The 1st output of the algorithm is to get the image/ frame from the video. The output example is shown below:

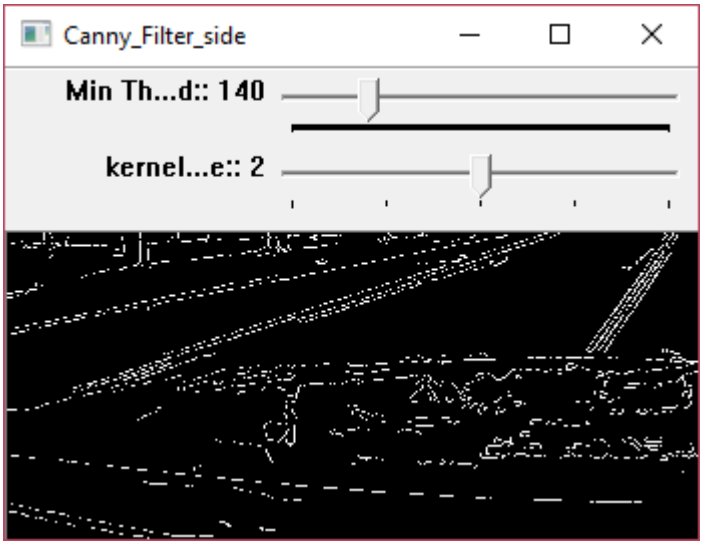
The next output is will be that of the Canny Edge detector. The Canny is done for the specific region of interest. The image for the same is shown below:



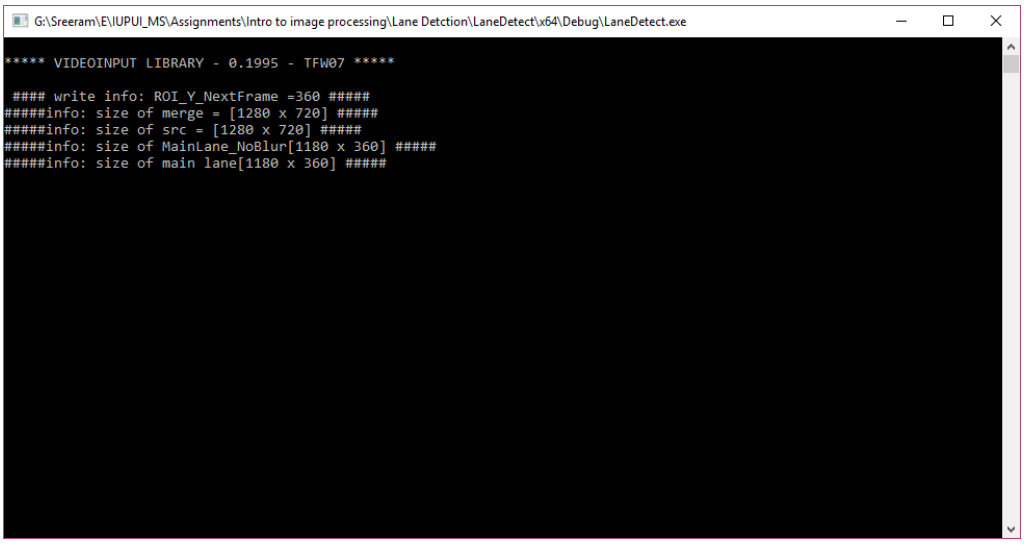
After this the next image displayed will be image *MainLane_Noblur* which will have the Lanes shown (which does not include the Probabilistic Hough Line segments).



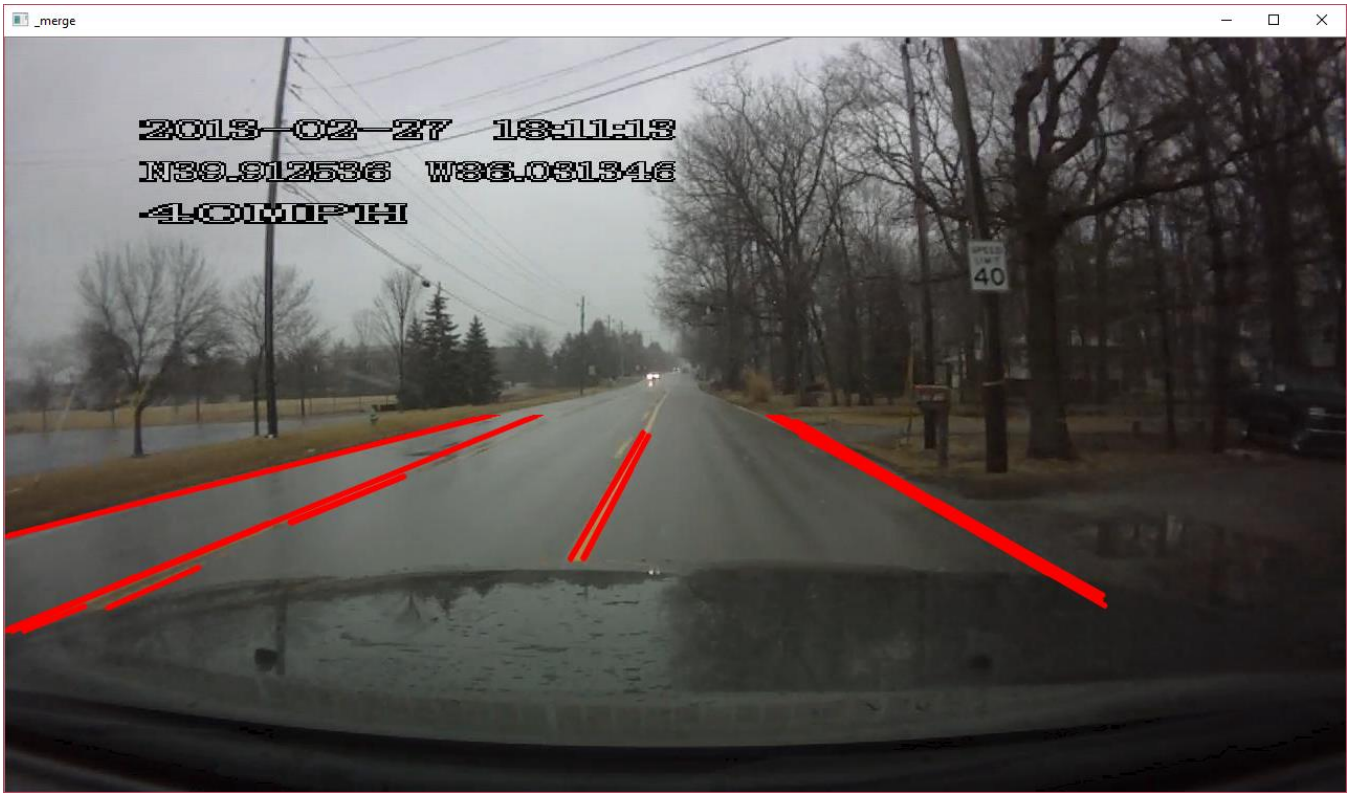
Along with this Canny of the side lanes will also be asked to be calibrated.



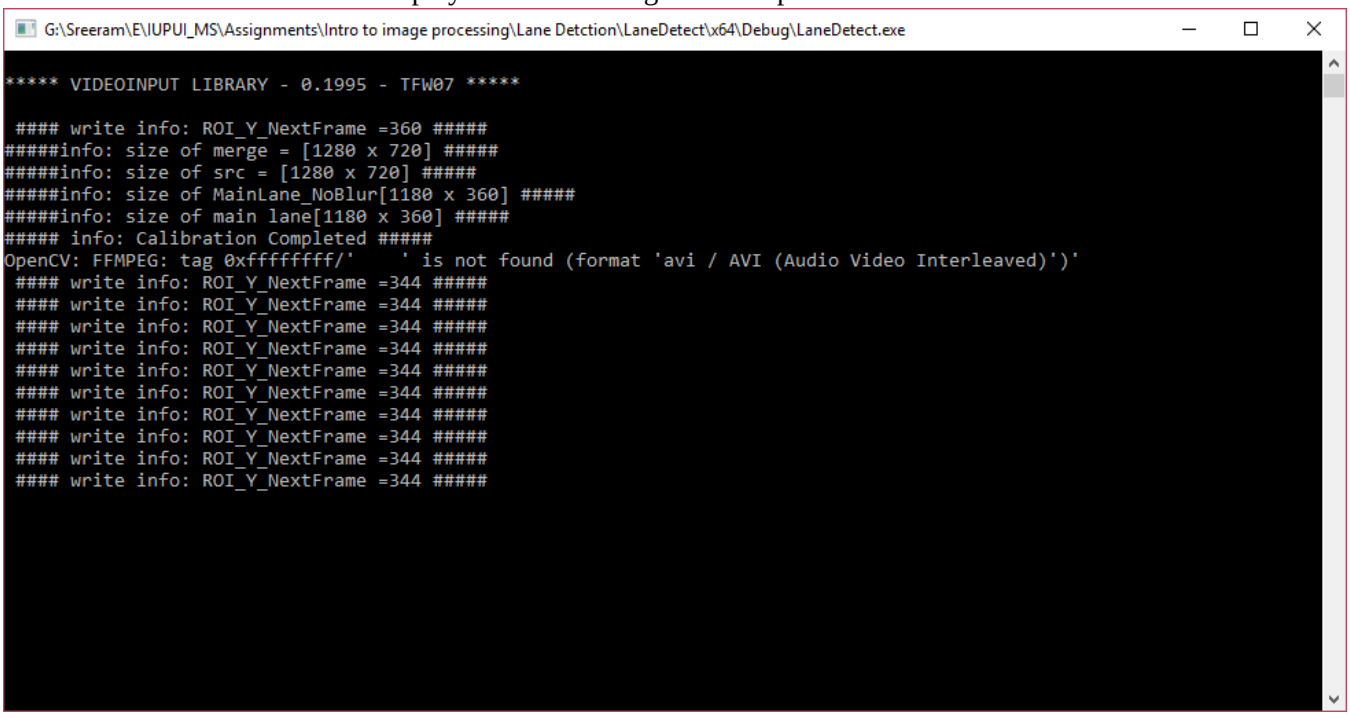
The size of the different matrices also will be put in the output window.



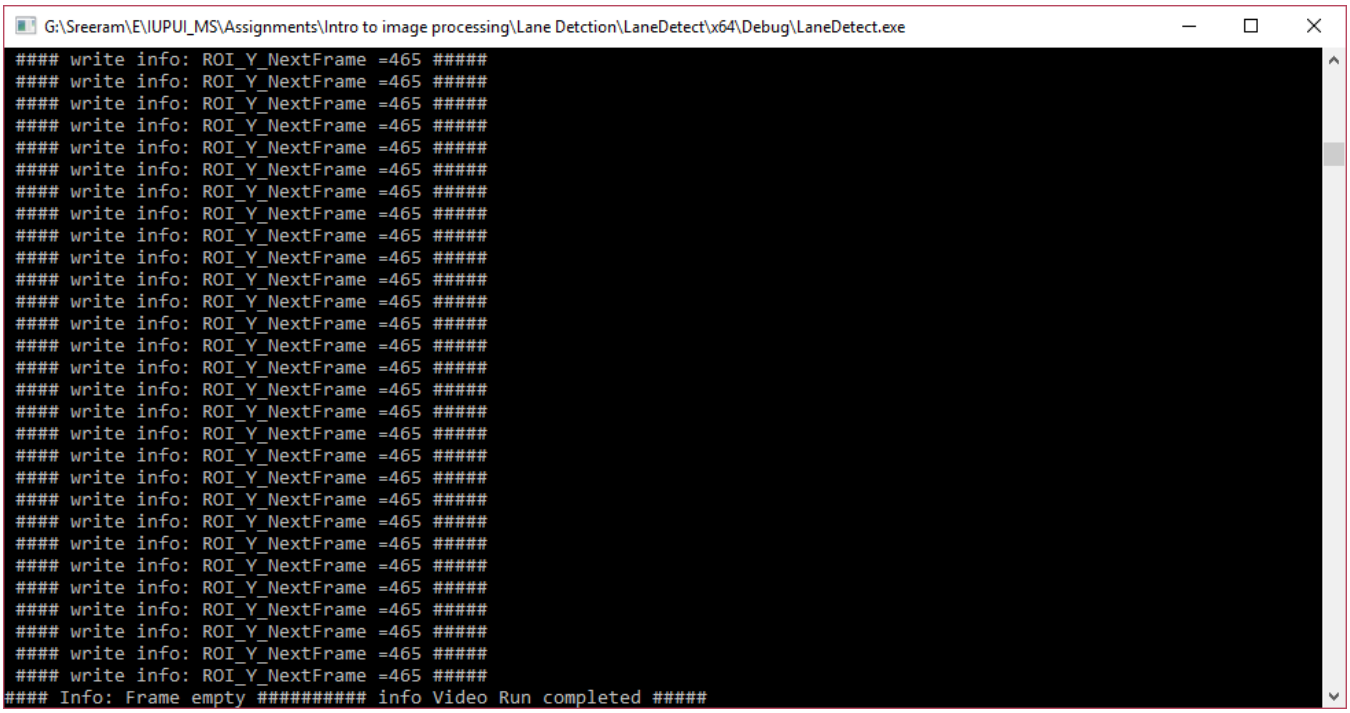
In the next step the Hough Lines P will also be added to the ROI and ROI image was added to the original image and the final image will be obtained.



Once the calibration is completed the video saving option will pop up and once completed the output window will have the new ROI displayed for each image. The output is shown below:



Once the Video is completed the following message will be obtained.



The video will be saved in project directory.

The tests were done in different weather conditions and road conditions

1. Normal bright whether conditions: -
   The initial test cases used had bright clear images. The were no objects disturbing the lanes. Straight Lane was used instead of dashed Lanes. The video output is attached along with the report. An image output frame is given below.

Fig: test1.avi

Fig: test4.avi

2. Rain conditions: -
   The weather is rainy. The lanes are dashed lines. There were objects in the field of view. Unfortunately, the objects lines were also detected and there were false positive cases present in the video frames. The video output is attached along with the report (test8.avi). A part of the frame is given below.

Fig: test8.avi

3. Snow Conditions: -
   Whether condition is snowfall. No dashed lines and there is a small object in between the frames. Half the video has the vehicle out of the lane as a result it did not detect, at many times, the center lane. The output video (test10.avi) is attached along with the report. An image of the same video is attached below.

Fig: test10.avi

4. Shadowed roads: -
   The algorithm does not detect the lanes properly if the roads are heavily disturbed by shadows. But will work perfectly for small shadows. This can be seen in the output video test2.avi wherein the program initially completely missed the lane because of the shadow but was then able to detect the lanes once the shadow area was lessened.

Fig: test2.avi

5. Faded road conditions.
   Weather is clear and bright but the lanes are faded. The algorithm completely failed in this condition. This can be seen in test12.avi.

Fig: test12.avi

6. Urban conditions: -

The output video can be found in test3.avi (Note: This video was a 640 x 480). The algorithm is working sufficiently well although the Hough line segments were throwing off errors on occasions. The HoughLinesP may have to be improved. The image for the same is seen below:

Fig: test3.avi

6.1. Challenges.

The algorithm was very slow as the processing time for a single frame was around 500ms (except for test3.avi which had lesser dimensions). There were also problems with the shadow, part of which was then improved using the Morphological close function. The standard Hough transform were easy to control, however the Probabilistic Hough which was used to find the dashed lanes were very difficult to control and was prone to errors caused by other objects like nearby cars. Initially it was decided that the vanishing point will be calculated using the RANSAC algorithm as discussed in [8]. However, it was not possible to do so in the given time framework. Thus, the vanishing point was calculated by the intersection of the Hough lanes detected. For this the lanes detected should be mainly of only the lanes on the road. As a result, proper Canny and lesser number of Hough lines are required. This is the main reason for the increase in processing time of each frame.

6. Recommendations for future work

The future work will mainly depend on improving the processing time. This can be done by analyzing the road lanes better. For example, to remove the background images which contains a combination of yellow lanes and white lanes, the image can be initially masked with a yellow mask to the image, which only takes the yellow lanes. The same can be done for the image with

white mask to get the white lanes. These can be super imposed on the Main image. This should be done as a part of the preprocessing. This should help in reduction of many of the unwanted edges, if yellow is not available in the background. This is but just one way to improve the algorithm. The HoughLineP can be avoided if the ROI is increased and the threshold is decreased for the image. But this case will not work if the environment is urban. So, any future work required will depend on how to make a better algorithm for preprocessing the data for enhancing only the road lane edges.

In future, the program should decide the threshold and kernel value for the images by itself so that the calibration part of the program can also be automated.

7. Conclusion

The project was done with the aim creating a program for Detection of Lane in both urban and non-urban environments. The implementation was successful though it has some drawbacks. There is room for improvement, which if worked on, can definitely make the program better detect lanes for the urban as well as the non-urban environment.

8. References:

1. Truong, Quoc-Bao, and Byung-Ryong Lee. "New lane detection algorithm for autonomous vehicles using computer vision." In *Control, Automation and Systems, 2008. ICCAS 2008. International Conference on*, pp. 1208-1213. IEEE, 2008.
2. Gehrig, Stefan K.; Stein, Fridtjof J. (1999). *Dead reckoning and cartography using stereo vision for an autonomous car*. IEEE/RSJ International Conference on Intelligent Robots and Systems. **3**. Kyongju. pp. 1507–1512.
3. Jiang, Yan, Feng Gao, and Guoyan Xu. "Computer vision-based multiple-lane detection on straight road and in a curve." In *Image Analysis and Signal Processing (IASP), 2010 International Conference on*, pp. 114-117. IEEE, 2010.
4. Vishwakarma, Sunil Kumar, and Divakar Singh Yadav. "Analysis of lane detection techniques using openCV." In *India Conference (INDICON), 2015 Annual IEEE*, pp. 1-4. IEEE, 2015.
5. Aly, Mohamed. "Real time detection of lane markers in urban streets." *Intelligent Vehicles Symposium, 2008 IEEE*. IEEE, 2008.
6. Wang, Hong, and Qiang Chen. "Real-time lane detection in various conditions and night cases." *Intelligent Transportation Systems Conference, 2006. ITSC'06. IEEE*. IEEE, 2006.
7. http://docs.opencv.org/2.4/doc/tutorials/highgui/video-write/video-write.html
8. https://marcosnietoblog.wordpress.com/2011/12/27/lane-markings-detection-and-vanishing-point-detection-with-opencv
9. http://docs.opencv.org/trunk/d6/d6e/group__imgproc__draw.html#ga482fa7b0f578fcdd8a174904592a6250
10. http://stackoverflow.com/questions/7446126/opencv-2d-line-intersection-helper-function

9. Appendices

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/imgproc/imgproc_c.h"
#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <opencv2/videoio/videoio.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui/highgui.hpp>

#include <stdlib.h>
#include <stdio.h>
#include <iostream>

using namespace std;
using namespace cv;

int hough(Mat &src, Mat &Main_Lane, Mat &Main_Lane_NoBlur, Mat &Sub_Lane);
int video(string);
void Merge_Function(Mat &, Mat &);
bool intersection(Point2f, Point2f, Point2f, Point2f, Point2f &);
int FramePreproceesing(Mat&, Mat &, Mat &, Mat &);
void CannyThreshold_side(int, void*);
void CannyThreshold(int, void*);


int Calib_flag = 0;
int ROI_paveY = 360;
int ROI_paveX = 300;
int ROI_Y = 350;
int ROI_Y_NextFrame = 350;
int ROI_X = 0;
int ypoint = 0;
int xpoint = 0;
int lowThreshold = 0;
int lowThresholdside = 0;
int kernel_size = 3, ratio = 3;

Mat _merge;
Mat _image, dest;
Mat _merge_pavement;

int main(int argc, char *argv[])
{

	const string VideoFileName = argv[1];

	int return_cmdVideo = video(VideoFileName);
	cout << "##### Info: Program completed #####";

	return return_cmdVideo;
}

int video(string VideoFileName)
{
```

```
	/************ The calibration is performed by taking the 1st frame of the video. ***************/


	VideoCapture calib(VideoFileName);

	if (!calib.isOpened())
	{
		cout << "##### ERROR: error at videocapture calib - could not open file" << VideoFileName << "#####" << endl;
		return -1;
	}

	/**************** initializing default values for region of interest. This will be modified later***************/

	Mat frame_calib;
	calib >> frame_calib;

	//cout << frame_calib.rows / 2;

	ROI_paveY = frame_calib.rows / 2;
	ROI_paveX = frame_calib.cols / 2;
	ROI_Y = frame_calib.rows / 2;

	_merge = frame_calib.clone();
	Mat Main_Lane = _merge.clone();
	Mat Main_Lane_NoBlur = _merge.clone();
	Mat Sub_Lane = _merge.clone();

	FramePreproceesing(frame_calib, Main_Lane, Main_Lane_NoBlur, Sub_Lane);
	hough(frame_calib, Main_Lane, Main_Lane_NoBlur, Sub_Lane);
	cout << "##### info: Calibration Completed #####" << endl;
	waitKey(0);


	/**************** END of initializing default values for region of interest***************/

	/********************** END of calibration  ************************/

	Calib_flag = 1;

	/********************************** Main video run ***********************************/

	Size S = Size(frame_calib.cols, frame_calib.rows);                    // size of of the video frame for updating the output video
	VideoCapture capture(VideoFileName);

	if (!capture.isOpened())
	{
		cout << "##### ERROR: error at videocapture capture - could not open file";
		return -1;
	}

	VideoWriter outputVideo;
	outputVideo.open("Output_Video.avi", -1, capture.get(CV_CAP_PROP_FPS), S, true);
```

```
	if (!outputVideo.isOpened())
	{
		cout << "##### ERROR: error at VideoWriter outputVideo - could not open file to write";
		return -1;
	}

	while (1)
	{
		Mat frame;
		capture >> frame;

		if (frame.empty())
		{
			cout << "#### Info: Frame empty #####";
			break;
		}

		_merge = frame.clone();
		_merge_pavement = frame.clone();

		FramePreproceesing(frame, Main_Lane, Main_Lane_NoBlur, Sub_Lane);
		hough(frame, Main_Lane, Main_Lane_NoBlur, Sub_Lane);

		/*** Reference taken from http://docs.opencv.org/2.4/doc/tutorials/highgui/video-write/video-write.html ***/

		outputVideo << _merge;                                          // saving the merged data into a new .avi file
	}

	cout << "##### info Video Run completed #####";
	waitKey(0);

	/********************************** End Main video run **********************************/
	return 0;

}

void CannyEdgeSeg(Mat& image, int)
{
	_image = image.clone();
	dest = image.clone();

	cvtColor(image, _image, CV_BGR2GRAY);
	namedWindow("Canny_Filter", CV_WINDOW_NORMAL);

	if (Calib_flag == 0)
				// Only for initial calibration
	{
		createTrackbar("Min Threshold:", "Canny_Filter", &lowThreshold, 700, CannyThreshold, &kernel_size);
		CannyThreshold(lowThreshold, 0);
		createTrackbar("kernel_size:", "Canny_Filter", &kernel_size, 4, CannyThreshold, &lowThreshold);
		CannyThreshold(kernel_size, 0);
```

```
		waitKey(0);
	}
	else
					// For the normal video run
	{
		CannyThreshold(lowThreshold, 0);
	}

}

void CannyThreshold(int, void*)
{
	Mat EdgeDetected = _image.clone();
	blur(_image, EdgeDetected, Size(2 * kernel_size + 1, 2 * kernel_size + 1));
	Canny(EdgeDetected, EdgeDetected, lowThreshold, lowThreshold*ratio, 2 *
kernel_size + 1);
	imshow("Canny_Filter", EdgeDetected);
	dest = EdgeDetected.clone();
}

void CannyEdgeSeg_side(Mat& image_side, int)
{
	_image = image_side.clone();			// initializing _image
	dest = image_side.clone();			// initializing dest

	cvtColor(image_side, _image, CV_BGR2GRAY);
	namedWindow("Canny_Filter_side", CV_WINDOW_NORMAL);

	if (Calib_flag == 0)				// Only for initial calibration
	{
		createTrackbar("Min Threshold:", "Canny_Filter_side", &lowThresholdside,
700, CannyThreshold_side, &kernel_size);
		CannyThreshold_side(lowThresholdside, 0);
		createTrackbar("kernel_size:", "Canny_Filter_side", &kernel_size, 4,
CannyThreshold_side, &lowThresholdside);
		CannyThreshold_side(kernel_size, 0);
		waitKey(0);
	}
	else						// For the normal
video run
	{
		CannyThreshold_side(lowThresholdside, 0);
	}

}

void CannyThreshold_side(int, void*)
{
	Mat EdgeDetected = _image.clone();
	blur(_image, EdgeDetected, Size(2 * kernel_size + 1, 2 * kernel_size + 1));
	Canny(EdgeDetected, EdgeDetected, lowThresholdside, lowThresholdside*ratio, 2 *
kernel_size + 1);
	namedWindow("Canny_Filter_side", CV_WINDOW_AUTOSIZE);
	imshow("Canny_Filter_side", EdgeDetected);
	dest = EdgeDetected.clone();
}
int FramePreproceesing(Mat &src, Mat &Main_Lane, Mat &Main_Lane_NoBlur, Mat &Sub_Lane)
{
```

```
	/******* For removing noise if any. The processing also helped further in edge detection. The canny worked better as compared to one without the Blur*******/

	GaussianBlur(src, src, Size(7, 7), 1.5, 1.5);

	/***************END of GaussianBlur **************/

	if (Calib_flag == 0)
	{
		ROI_Y_NextFrame = ROI_Y;
					// For the initial frame for calibration a default ROI is taken as the vanishing point is not known.
	}

	if (((src.cols - ROI_X - 100) < 0) || ((src.rows - ROI_Y_NextFrame) < 0) || (ROI_X < 0) || (ROI_Y_NextFrame < 0) || (ROI_Y < 0))
	{
		cout << "##### Error: The Value of ROI is not in the limits of the image. ROI cannot be obtained";
		return -1;
	}

	cout << " #### write info: ROI_Y_NextFrame =" << ROI_Y_NextFrame<<" #####" << endl;
	Rect roi(ROI_X, ROI_Y_NextFrame, src.cols - ROI_X - 100, src.rows - ROI_Y_NextFrame);

	// Initially the Region Of Interest is taken as the bottom half of the image. It is then adjusted according to the vanishing point which is calculated in the calibration process.

	Main_Lane = src(roi);

	Main_Lane_NoBlur = _merge(roi);

	Rect roi_pave(0, ROI_paveY, ROI_paveX, src.rows - ROI_paveY);
		// Left side lane and pavement detection


	Sub_Lane = _merge(roi_pave);

	int operation = 2;
					// for Close function morphology

	Mat element = getStructuringElement(0, Size(2 * 15 + 1, 2 * 15 + 1), Point(15, 15));//  A 31 x 31 Matrix is taken for the morphological operation

	// A closing operation is performed for removing the false positives which causes unwanted lanes. Eg: removal of shadows

	morphologyEx(Main_Lane, Main_Lane, 3, element);

	return 0;
}

int hough(Mat &src,Mat &Main_Lane, Mat &Main_Lane_NoBlur, Mat &Sub_Lane)
{
```

```
	Point2f MainLane_intersectionpoint;                                    // to find
the vanishing point during calibration
	vector<Vec2f> MainLane_Lines, SubLanes_Lines;          // Detects the number of
lines in the image.
	Point pt1[10000], pt2[10000];                                          // Maximum
lines that can be taken into consideration is 10000. This is done to gt the approximation
of the vanishing point.

	int hough_threshold_main = 30;                                         // The
initial threshold for the Lane in which the vehicle is present.
	int hough_threshold_sub = 50;                                          // The left
Sub Lane for which the vehicle is present

	int Line_increment=0;                                                          // For
saving the number of lines that are generated by the hough Transform.

	CannyEdgeSeg(Main_Lane, Calib_flag);


	/************        Performs the Hough Transform for getting the Main Lanes
	               ***************/
	/************ The algorithm starts with the initial or default threshold and
varies the          ***************/
	/************ threshold with a step of 5 for each of the run until the number of
lines         ***************/
	/************ generated becomes less than 10. This is in assumption that the road
and the       ***************/
	/************ lane have a good edge and this can be detected even with less number
of threshold ************/

	do
	{
		HoughLines(dest, MainLane_Lines, 1, CV_PI / 180, hough_threshold_main);
	// gets the number of hough lines for the Main Lane

		//cout << "MainLane_Lines.size() = " << MainLane_Lines.size() << endl;

		hough_threshold_main = hough_threshold_main + 5;

	} while (MainLane_Lines.size() > 10);

	/*            The theta is checked for each of the lines generated and those which
fall in the required range is drawn to the main image         */
	/* Reference from : https://marcosnietoblog.wordpress.com/2011/12/27/lane-
markings-detection-and-vanishing-point-detection-with-opencv/ */

	for (size_t i = 0; i < MainLane_Lines.size(); i++)
	{
		float rho = MainLane_Lines[i][0], theta = MainLane_Lines[i][1];

		// 73 to 78 will work for left lane detection but very crude. To get a
wider angle of the lanes 10 to 80 degree and 100 to 170 degree is taken

		if ((theta > CV_PI / 180 * 10 && theta < CV_PI / 180 * 80) || (theta >
CV_PI / 180 * 100 && theta < CV_PI / 180 * 170))
		{
			double a = cos(theta), b = sin(theta);                         // The
formula used is: rho = x0 * cos (theta) + y0 * sin (theta)
```

```
                        double x0 = a*rho, y0 = b*rho;

                        /************* For getting the end points of the line segment
*************/

                        pt1[Line_increment].x = cvRound(x0 + 1000 * (-b));
                        pt1[Line_increment].y = cvRound(y0 + 1000 * (a));
                        pt2[Line_increment].x = cvRound(x0 - 1000 * (-b));
                        pt2[Line_increment].y = cvRound(y0 - 1000 * (a));
                        line(Main_Lane_NoBlur, pt1[Line_increment], pt2[Line_increment],
Scalar(0, 0, 255), 3, CV_AA);
                        Line_increment++;

                        /************************END: end points of the line segment
************************/
                }
        }
        imshow("Main_Lane_NoBlur", Main_Lane_NoBlur);

        /********* A rough estimate of calculating the vanishing point for the image. This
is called only during th calibration phase ********/
        /*********    Reference:
http://docs.opencv.org/trunk/d6/d6e/group__imgproc__draw.html#ga482fa7b0f578fcdd8a1749045
92a6250                 ********/

        if (Calib_flag == 0)
        {
                int i_loop = 0;
                int j_loop = 0;
                do
                {
                        intersection(pt1[i_loop], pt2[i_loop], pt1[i_loop + 1], pt2[i_loop +
1], MainLane_intersectionpoint);

                        xpoint = (int)MainLane_intersectionpoint.x;
                        ypoint = (int)MainLane_intersectionpoint.y;

                        i_loop++;
                        j_loop++;
                } while (ypoint >= src.rows / 2);

                ROI_Y_NextFrame = Main_Lane_NoBlur.rows + ypoint + 30;        // The
intersection point is taken as the vanishing point. So the image is reduced to 30 pixel
less than the vanishing point

                                // NOTE: This is a rough approximation only of the vanishing
point.

                cout << "#####info: size of merge = " << size(_merge)<<" #####" << endl;
                cout << "#####info: size of src = " << size(src) << " #####" << endl;
                cout << "#####info: size of MainLane_NoBlur" << size(Main_Lane_NoBlur) << "
#####" << endl;
                cout << "#####info: size of main lane" << size(Main_Lane) << " #####" <<
endl;
                drawMarker(_merge, MainLane_intersectionpoint, 0, 20, 1, 8);
        }
```

```
	/********* END: A rough estimate of calculating the vanishing point for the image. This is called only during th calibration phase ********/

	/************ HoughLinesP are required t0 get the images which have dashed Lanes *************/

	vector<Vec4i> lines;
	// Line segments for HoughTransformP.
	Point2f MainLane_intersectionpoint_houghP[2];                        // TO get the intersection point

	// Apply canny edge

	HoughLinesP(dest, lines, 1, 2 * CV_PI / 180, 40, 20, 100);    // The threshold = 40,minLineLength =20, maxLineGap =100

	for (size_t i = 0; i < lines.size(); i++)
	{

		float rho = lines[i][0], theta = lines[i][1];

		// Point P1 is represented as (x1,y1) and P2 is represented as (x2,y2)

		MainLane_intersectionpoint_houghP[0].x = lines[i][0];         // x1
		MainLane_intersectionpoint_houghP[0].y = lines[i][1];         // y1
		MainLane_intersectionpoint_houghP[1].x = lines[i][2];         // x2
		MainLane_intersectionpoint_houghP[1].y = lines[i][3];         // y2

		float angle = atan2(MainLane_intersectionpoint_houghP[0].y - MainLane_intersectionpoint_houghP[1].y, MainLane_intersectionpoint_houghP[0].x - MainLane_intersectionpoint_houghP[1].x);
		angle = 180 * angle / CV_PI;
		//cout << angle << endl;
		if ((angle < -100 && angle > -160) || (angle > 100 && angle < 160))  // Angle limits
		{
			line(Main_Lane_NoBlur, MainLane_intersectionpoint_houghP[0], MainLane_intersectionpoint_houghP[1], Scalar(0, 0, 255), 3, CV_AA);
		}
	}

	/************ END: HoughLinesP are required t0 get the images which have dashed Lanes *************/

	// Calculation of hough transform for Side Lanes

	CannyEdgeSeg_side(Sub_Lane, Calib_flag);

	do
	{
		HoughLines(dest, SubLanes_Lines, 1, CV_PI / 180, hough_threshold_sub);
		hough_threshold_sub = hough_threshold_sub + 10;
	// The step update is 10 for the threshold

		//cout << "SubLanes_Lines.size() = "<< SubLanes_Lines.size() << endl;

	} while (SubLanes_Lines.size() > 10);
```

```
	/*********    Reference:
http://docs.opencv.org/trunk/d6/d6e/group__imgproc__draw.html#ga482fa7b0f578fcdd8a1749045
92a6250                *********/

	for (size_t i = 0; i < SubLanes_Lines.size(); i++)
	{

		float rho = SubLanes_Lines[i][0], theta = SubLanes_Lines[i][1];
		// 73 to 78 will work for left lane detection but very crude// normal lane
detction works with 55 to 60
		if ((theta > CV_PI / 180 * 70 && theta < CV_PI / 180 * 80) || (theta >
CV_PI / 180 * 120 && theta < CV_PI / 180 * 130))
		{
			double a = cos(theta), b = sin(theta);
			double x0 = a*rho, y0 = b*rho;
			pt1[Line_increment].x = cvRound(x0 + 1000 * (-b));
			pt1[Line_increment].y = cvRound(y0 + 1000 * (a));
			pt2[Line_increment].x = cvRound(x0 - 1000 * (-b));
			pt2[Line_increment].y = cvRound(y0 - 1000 * (a));
			line(src, pt1[Line_increment], pt2[Line_increment], Scalar(0, 0,
255), 3, CV_AA);
			Line_increment++;
		}
	}

	Merge_Function(Main_Lane_NoBlur, Sub_Lane);
	return 0;

	/********************************** Hough Transform
End***********************************/

}

/***************  Finds the intersection of two lines, or returns false. The lines are
defined by (o1, p1) and (o2, p2). ***************/
/*************** http://stackoverflow.com/questions/7446126/opencv-2d-line-intersection-
helper-function ***************/

bool intersection(Point2f o1, Point2f p1, Point2f o2, Point2f p2, Point2f &r)
{
	Point2f x = o2 - o1;
	Point2f d1 = p1 - o1;
	Point2f d2 = p2 - o2;

	float cross = d1.x*d2.y - d1.y*d2.x;
	if (abs(cross) < /*EPS*/1e-8)
		return false;

	double t1 = (x.x * d2.y - x.y * d2.x) / cross;
	r = o1 + d1 * t1;
	return true;
}

/*************** END: Intersection function**************/

/******************* For merging of the image with the Lane to the normal
image*****************/
void Merge_Function(Mat& Main_Lane, Mat& Sub_Lane)
```

```
{
	int i = 0;
	int j = 0;

	if (Calib_flag == 0)
	{
		for (i = 0; i < Main_Lane.rows; i++)
		{
			for (j = 0; j < Main_Lane.cols; j++)
			{
				_merge.at<cv::Vec3b>(i + ROI_Y, j + ROI_X)[0] =
Main_Lane.at<cv::Vec3b>(i, j)[0];
				_merge.at<cv::Vec3b>(i + ROI_Y, j + ROI_X)[1] =
Main_Lane.at<cv::Vec3b>(i, j)[1];
				_merge.at<cv::Vec3b>(i + ROI_Y, j + ROI_X)[2] =
Main_Lane.at<cv::Vec3b>(i, j)[2];
			}

		}
	}
	else
	{
		for (i = 0; i < Main_Lane.rows; i++)
		{
			for (j = 0; j < Main_Lane.cols; j++)
			{
				_merge.at<cv::Vec3b>(i + ROI_Y_NextFrame, j + ROI_X)[0] =
Main_Lane.at<cv::Vec3b>(i, j)[0];
				_merge.at<cv::Vec3b>(i + ROI_Y_NextFrame, j + ROI_X)[1] =
Main_Lane.at<cv::Vec3b>(i, j)[1];
				_merge.at<cv::Vec3b>(i + ROI_Y_NextFrame, j + ROI_X)[2] =
Main_Lane.at<cv::Vec3b>(i, j)[2];
			}

		}
	}
	imshow("_merge", _merge);
	waitKey(1);

}
```