PROJECT REPORT

LANE DETECTION

USING IMAGE PROCESSING

LANE DETECTION

**INTRODUCTION:**

The market of self-driving cars is growing at a very rapid pace. An autonomous car has to be able to pick up every small detail that surrounds it. It has to be able to drive flawlessly in unpredictable environments. One of the important module of self-driving cars is the lane detection which I our focus for this project.

**PROBLEM STATEMENT:**

To design a lane detection pipeline which is able to detect the current lane of the car.

**EXPLANATION:**

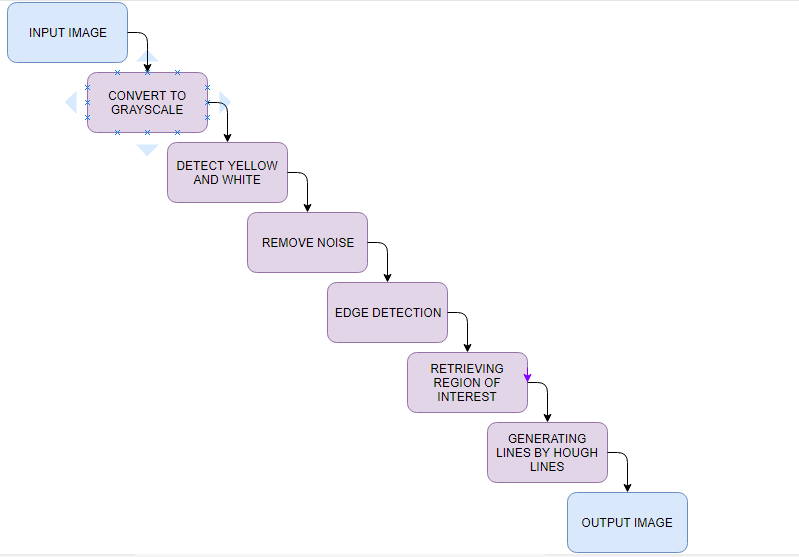
For the purpose of this project, a path identification and lane detection pipeline is proposed as a component of the computer vision segment for a self-driving vehicle. By utilizing a video feed input of a self-driving car, the pipeline will identify where the path is so the auto can utilize its area to abstain from escaping it.

**SOFTWARES AND LIBRARIES:**

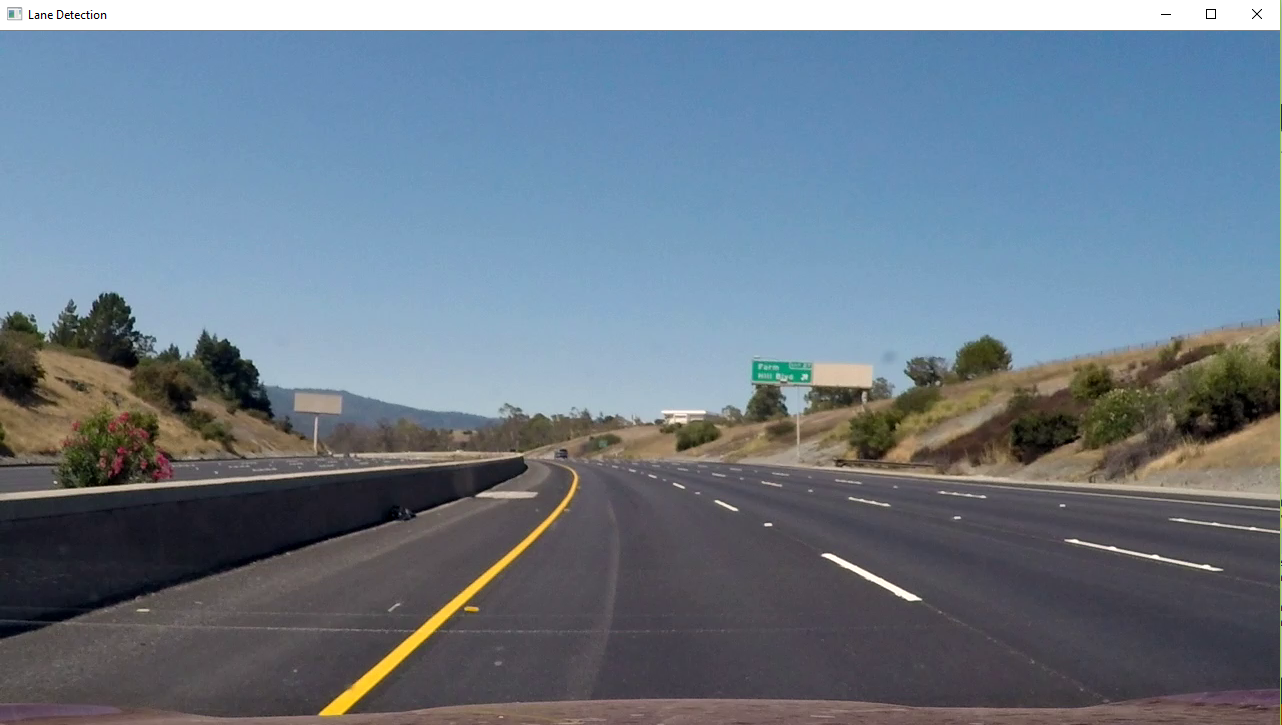
The task is produced utilizing C++, Visual Studio and OpenCV library.

**SOLUTION:**

No we will describe the details of the algorithm that is utilized in order to accomplish the lane detection task. The algorithm consists of the following pipeline:



Original image:



**STEP 1: TO GRAY SCALE**

The first and the foremost step involved in the lane detection algorithm we proposed id the conversion of the image from RGB color space to GRAY scale. In this part we have utilized the built-in opencv function to convert a rgb image to gray scale. It uses the following conversion

**Y = 0.299 R + 0.587 G + 0.114 B**

Code:

Mat change\_in\_grayscale(Mat image) {

Mat gray\_image;

cvtColor(image, gray\_image, COLOR\_BGR2GRAY);

return gray\_image;

}

Output:



**STEP 2: DETECT THE YELLOW AND WHITE COLOR OBJECTS**

After we have the grayscale version of the input image we now detect the regions of the image corresponding to the yellowish and white color as these are the most prominent colors that are used to draw lanes. We check for yellow region first and then white. Then we perform bitwise or of both before taking the bitwise and with the image we have after step1 to get the image with the regions that contain yellow and white colors. We use following spectrum values for yellow color:

**Low: 0, 60, 60**

**High: 30, 255, 255**

Code:

Mat detect\_yellow\_white(Mat image) {

Scalar yellow\_low = Scalar(0, 60, 60);

Scalar yellow\_high = Scalar(30, 255, 255);

Mat mask1, mask2, mask3;

Mat yelwhite;

inRange(image, yellow\_low, yellow\_high, mask1);

inRange(image, 200, 255, mask2);

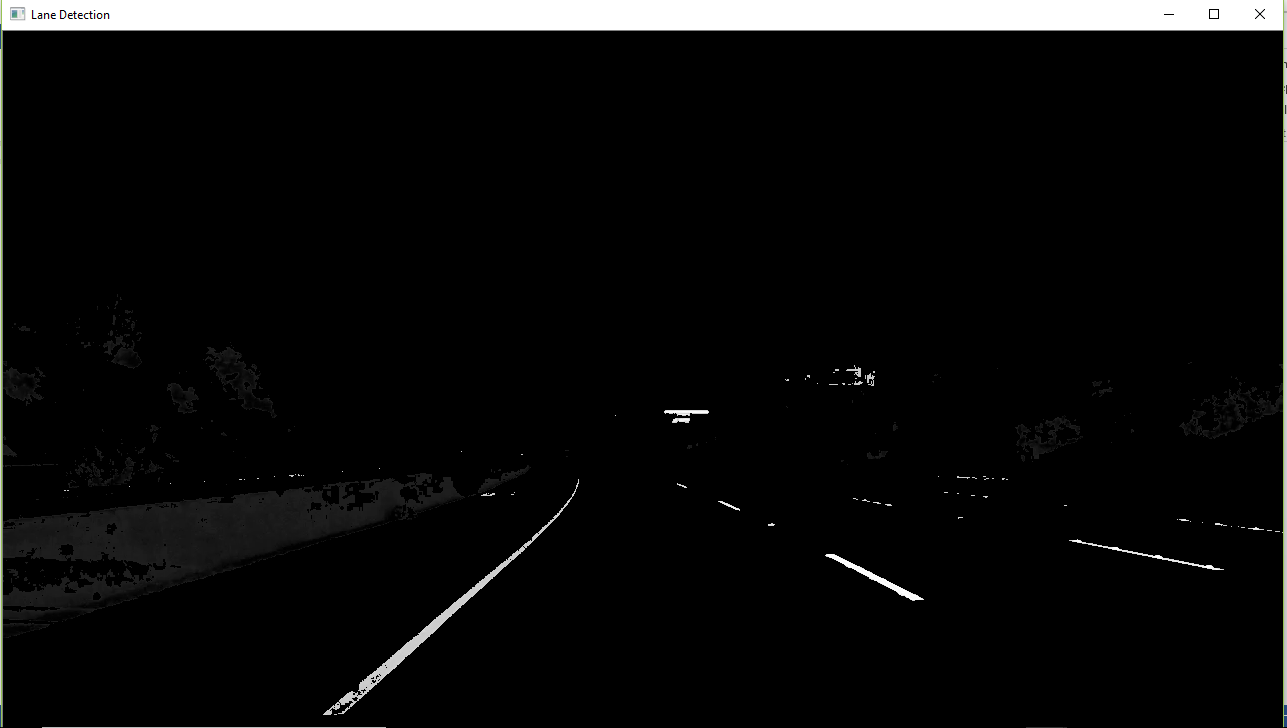
bitwise\_or(mask1, mask2, mask3);

bitwise\_and(image, mask3, yelwhite);

return yelwhite;

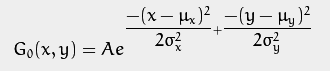
}

Output:



**STEP 3: REMOVING NOISE**

After detecting the regions of the image that contain yellowish or white color in them. We now remove the noise from the image. We utilize the Gaussian Blur filter of kernel size 5. This blurs the image decreasing the noise of the image. The formula for Gaussian blur is as follows:



where \mu is the mean (the peak) and \sigmarepresents the standard deviation (per each of the variables x and y)

Code:

Mat clean\_image(Mat image) {

Mat cleaned\_image;

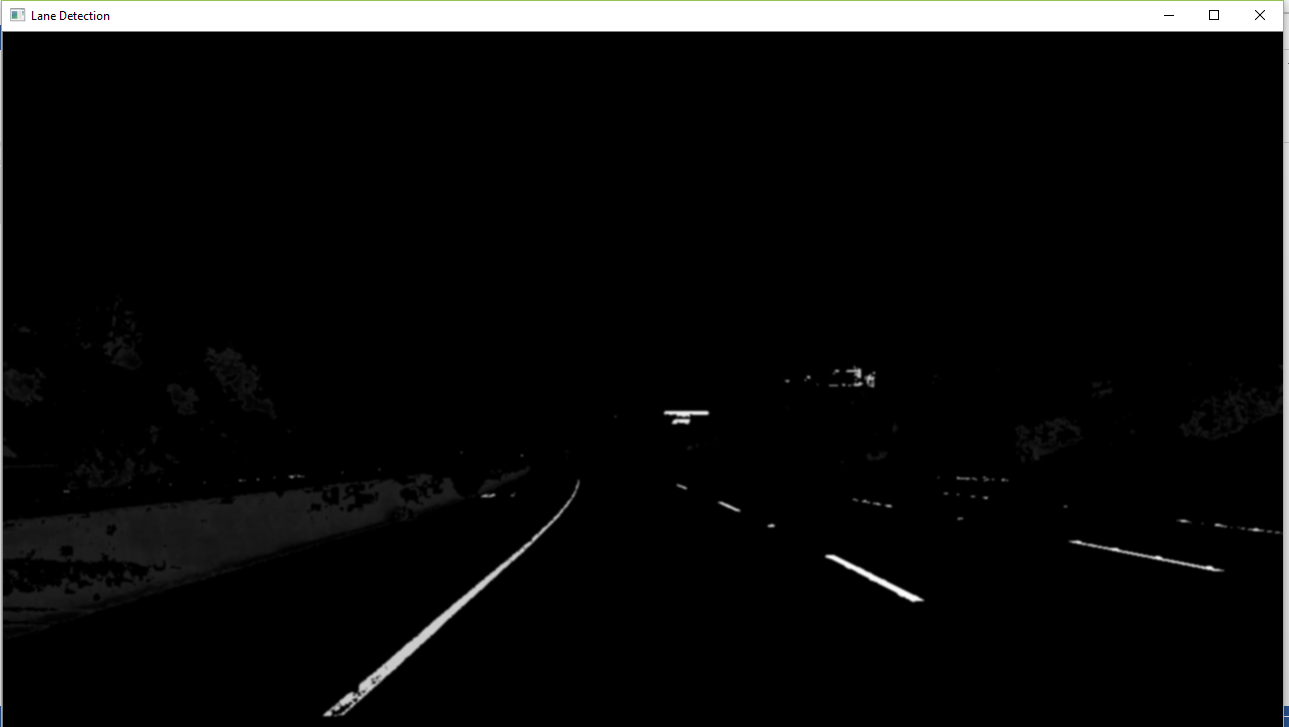
Size blur\_kernel\_size(5,5);

GaussianBlur(image, cleaned\_image, blur\_kernel\_size, 0, 0);

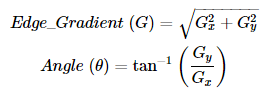
return cleaned\_image;

}

Output:



**STEP 4: EDGE DETECTION**

After using blurring filter to remove the noise from the image, we use canny edge detector to detect the edges in the image. We used opencv built-in function for this task. Canny edge detector has these steps. First is does noise reduction then finds the intensity gradient by following formula:

Then it does non maximum suppression and then thresholding.

Code:

Mat detect\_edges(Mat image) {

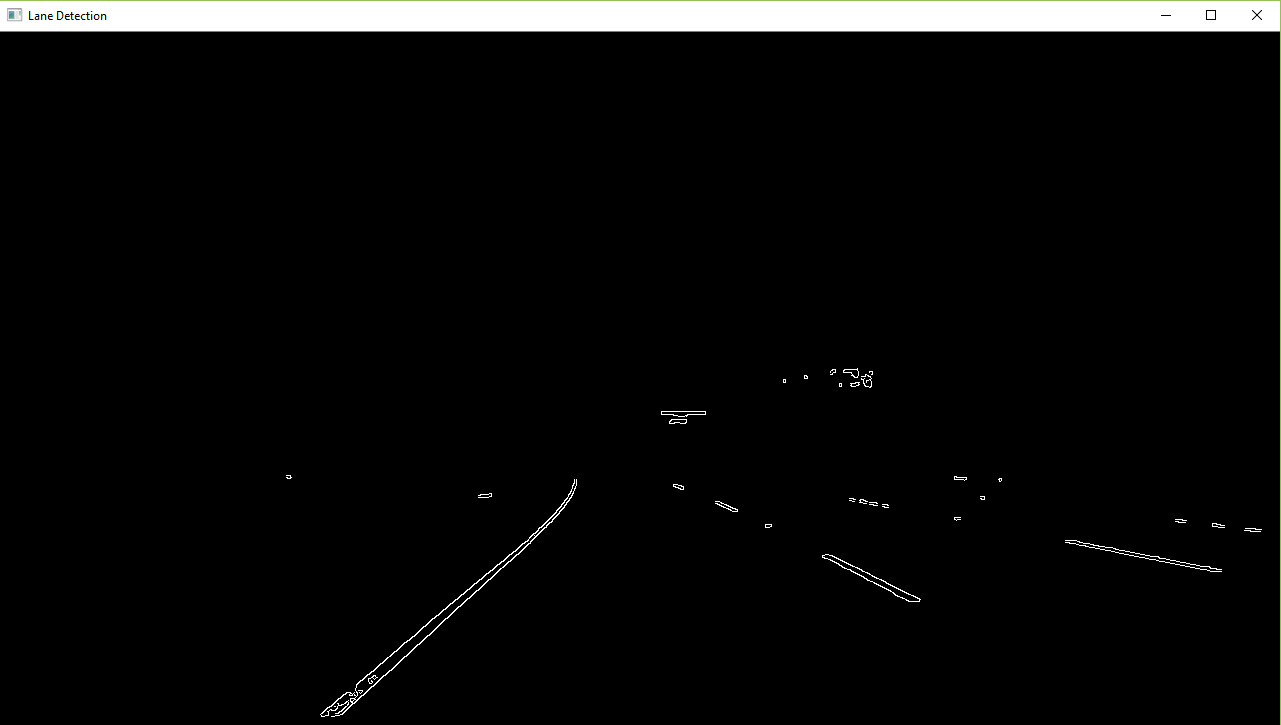
Mat cannyed\_image;

Canny(image, cannyed\_image, 200, 300);

return cannyed\_image;

}

Output:



**STEP 5: RETRIEVING THE REGION OF INTEREST**

We now have the edges in the whole image but all this is not important to us. Our region of interest lies in the lower triangle of the image. We apply the mask of this triangle to the image to get only the edges that lies in our region of interest. Formula for our triangle vertices is as follows:

**Vertex 1: (50, height - 50)**

**Vertex 2: (width / 2, height / 2)**

**Vertex 3: (width - 50, height - 50)**

Code:

Mat crop\_image\_to\_roi(Mat image) {

Mat cropped\_img;

int type\_of\_image = image.type();

int height = image.size().height;

int width = image.size().width;

Point vertices[3] = { Point(50, height-50), Point(width/2, height/2), Point(width-50, height-50)};

Mat region\_mask = Mat::zeros(Size(width,height), type\_of\_image);

fillConvexPoly(region\_mask, vertices, 3, Scalar(255, 0, 0));

bitwise\_and(image, region\_mask, cropped\_img);

return cropped\_img;

}

Output:



**STEP 6: GENERATING LINES USING HOUGH LINES:**

In this step we use the image generated by various image processing techniques and apply mathematics on it.

1. First we apply built in hough lines algorithm in the opencv library. Hough transform is a helpful technique in finding the shape even if it is distorted. A line can be represented as following in parametric form:

****

Hough transform uses this line with ρ (rho) and θ (theta) to calculate the possible lines in the image.

1. After this we detect slopes of all lines that are detected using he slope formula:

***m = (y1 - y0) / (x1 - x0)***

1. Now, we remove the lines that are too straight (have a slope less than 0.3) as they cannot be our lanes.
2. Next we separate the right and left lanes on the basis of the direction of slope and the distance from midpoint using following condition:

Right lane: slope **>** 0 **AND** start point X **>** midpoint **AND** end Point X **>** midpoint

Left lane: slope **<** 0 **AND** start point X **<** midpoint **AND** end Point X **<** midpoint

Code:

int rightStartY = image.size().height;

double rightStartX;

double rightEndX;

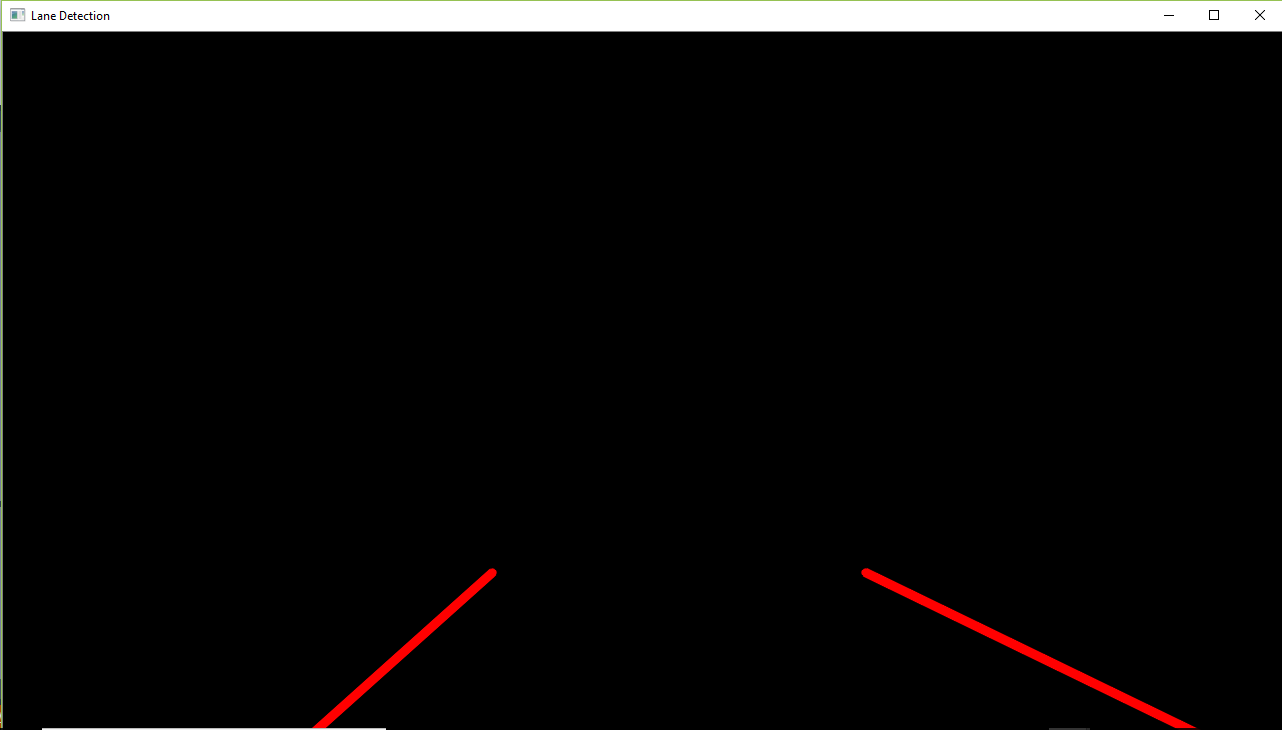
double leftStartX;

double leftEndX;

// calculating hough lines

HoughLinesP(image, lanelines, rho\_hough, theta\_hough, hough\_thresh, min\_length, min\_gap);

Output:



**STEP 7: DRAWING THE LANES ONTO THE ORIGINAL IMAGE:**

Now we have the lanes, we just have to draw them on the original image and fill the space between it for visual showing.

Code:

void drawLanes(vector<Point> lanePoints, Mat image) {

vector<Point> poly\_points;

Mat drawLanePoints;

……

Scalar laneColor = Scalar(0, 0, 0);

Scalar lineColor = Scalar(0, 0, 255);

int line\_thickness = 7;

fillConvexPoly(drawLanePoints, poly\_points, laneColor, 16, 0);

addWeighted(drawLanePoints, 0.3, image, 0.7, 0, image);

// Plot both lines of the lane boundary

line(image,line1start,line1end,lineColor,line\_thickness,16,0);

line(image,line2start,line2end,lineColor,line\_thickness,16,0);

// Show the final drawLanePoints image

namedWindow("Lane Detection", CV\_WINDOW\_AUTOSIZE);

imshow("Lane Detection", image);

}

Output:



**CONCLUSION:**

The project was successful. We were able to detect the lanes with good accuracy. Video output shows that the lanes are detected smoothly. Our pipeline proved to be correct and delivered the results.

**FUTURE IMPROVEMENTS:**

Although the project is a success, it only detect straight lines. It is a difficult topic to deal with the curved lines. We might have to use interpolation in order to do this. Another thing it lacks is in the detection of steep roads as we assume that the region of interest is in the center of the image. Increasing the frames per second of the output can be another improvement.