ADVANCED COMPUTER VISION

LANE DETECTION ALGORITHM

- ANIRUDH NARAYANAN(an9425)

INTRODUCTION

Self-driving cars, also known as autonomous vehicles, are vehicles that can operate without a human driver. Self-driving cars are revolutionizing the automotive industry by offering a safer, more efficient, and convenient mode of transportation. These autonomous vehicles rely on sophisticated software and sensors to detect and interpret road conditions, traffic signs, and other obstacles. By doing so, they have the potential to drastically reduce the number of accidents caused by human error, which is currently the leading cause of road fatalities. In addition to improving road safety, self-driving cars can also help reduce traffic congestion, lower carbon emissions, and provide greater mobility to people who are unable to drive.

Lane detection is an essential aspect of the development of self-driving cars. It involves identifying and tracking the lanes on the road, allowing the vehicle to remain within them. Self-driving cars use various sensors and algorithms to detect the lanes, which enables them to make decisions regarding steering, acceleration, and braking based on this information. Sensor fusion, which combines data from multiple sensors such as cameras, LiDAR, and radar, is one such technique used for lane detection. Once the vehicle detects that it is veering out of its lane, it can make steering adjustments to stay within the lane. Lane detection is just one of several components that work together to ensure that self-driving cars can safely and efficiently navigate the roads.

DIFFERENT APPROACHES

TRADITIONAL TECHNIQUES:

Traditional lane detection techniques rely on computer vision algorithms to analyze images or video frames captured by cameras mounted on the car. These algorithms detect lane markings by identifying edges and lines in the image and fitting them to a mathematical model of a lane. This process involves several techniques such as Hough Transform, Canny Edge Detection, and Sobel Edge Detection. While traditional lane detection techniques are widely used and effective to some extent, they have limitations. For example, they may not work well in adverse weather conditions or in situations where the lane markings are faded or obscured.

MODERN TECHNIQUES:

Advanced lane detection techniques , utilize machine learning algorithms that can learn to detect lanes from training data. These algorithms are trained on large datasets of images and video frames with annotated lane markings. Some of the most popular techniques used for advanced lane detection include Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs). These techniques can detect lanes more accurately and effectively than traditional techniques, and they can also be adapted to different driving conditions and scenarios. However, they require a large amount of training data and can be computationally intensive, which may be a limiting factor in some applications.

METHODOLOGY

I have decided to use the traditional computer vision techniques for lane detection in my project this semester. The project was implemented using c++ and opencv. The project did not require extra c++ or opencv libraries to be downloaded. The steps involved in detecting the lanes involved a few computer vision image processing steps such as converting the image to grayscale, applying gaussian blur to smoothen the image, which would help us find edges using canny edges. Then using the edges,we find hough lines. Now, we fit a line using the fitLine() function in opencv which uses linear regression to estimate the best-fit line for a set of 2D points in an image. This project also contains a second implementation in which uses homography to convert the input frame to a bird's eye point of view and then the same operations are performed on the converted frame to detect the lanes and then the lanes are projected back on to the dashcam point of view.

While implementing my project I saw that the video I was running was a little shaky, so my lines were moving around no matter how much I fine tuned my program. So I tried to run my code on another video which seemed much more stable and the output was considerably better. The video also had curves on the road, while I did not have specific implementation such as finding hough circles, my program was detecting the lines decently and up to a good distance in the frame. The output is added after I go over my implementation for my original video.

Now, I will go over both the implementations side by side and show intermediate results in order to demonstrate the effectiveness of each approach.

I :

Read in the input frame

The first step is reading in the video file and processing the input frame by frame. So here is the first frame that the program receives.

II :

Convert frame to Bird’s eye POV (only for homography)

Here, we perform homography on the input frame in order to create a bird’s eye point of view to perform the image processing steps on.

This is the bird’s eye point of view that I got. As we can see there is radial distortion present in the image. I did not have access to the distortion coefficients in order to get a better picture after homography, but I believe this result should help improve my previous implementation.

III :

IMAGE PROCESSING :

The image processing involves the following 5 steps:

STEP 1: Convert the input frame to grayscale

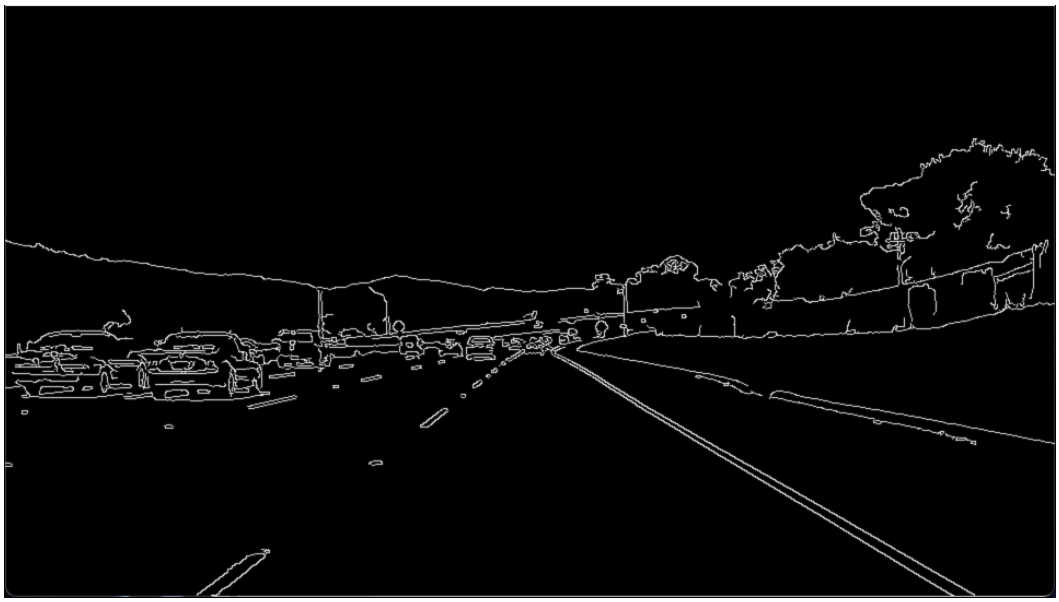
STEP 2: Apply Gaussian Blur on the grayscale image using a disk of size (9,9).

STEP 3: Use Canny Edges to find edges from the smoothened image.

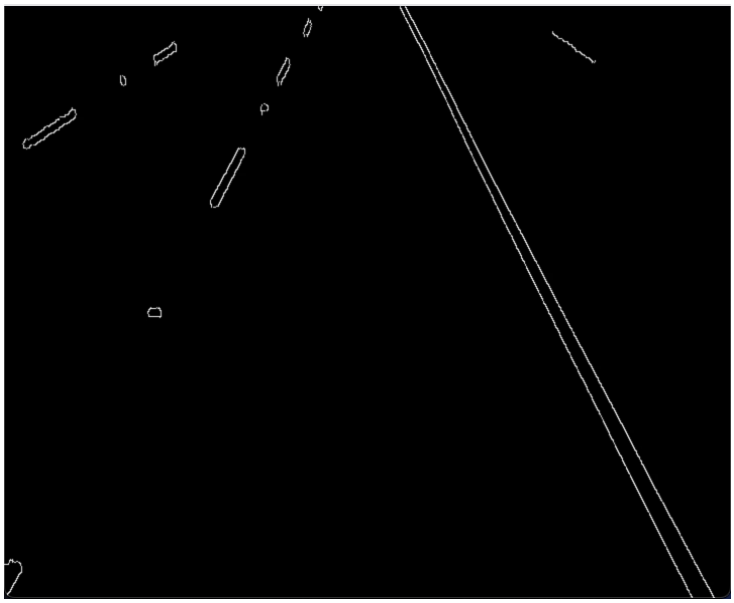
(continued)

STEP 3 OUTPUT:

DASHCAM TECHNIQUE



BIRD'S EYE POV

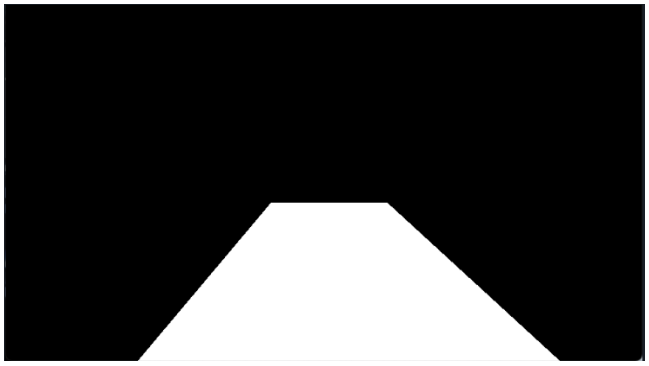


STEP 4:

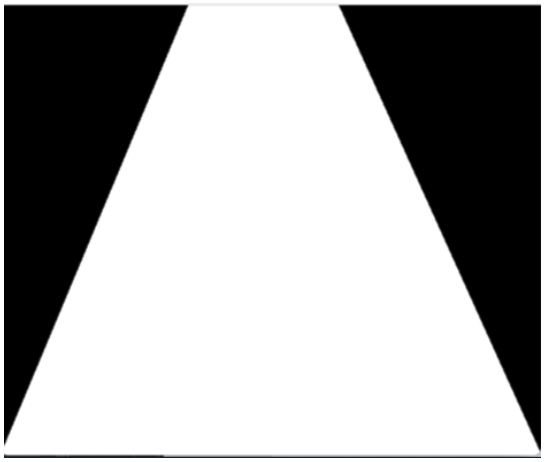
As we can see from the results of step 3, we are getting good edges for both the implementations, but we also have a lot of redundant information. These redundant edges will cause the hough lines to find lines maybe somewhere in the trees for the dashcam point of view or the adjacent lane in the bird's eye point of view. In order to tackle it, I decided to use a fixed mask. The coordinates of the mask were decided based on the amount of region I decided contained necessary information.

Here are the masks I used for each implementation.

DASHCAM TECHNIQUE:

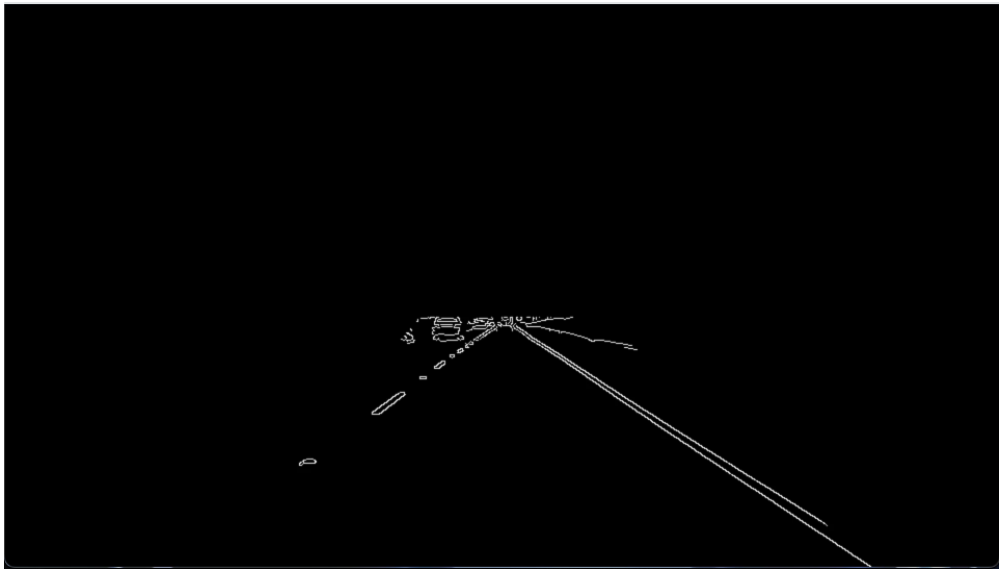


BIRD'S EYE POV:

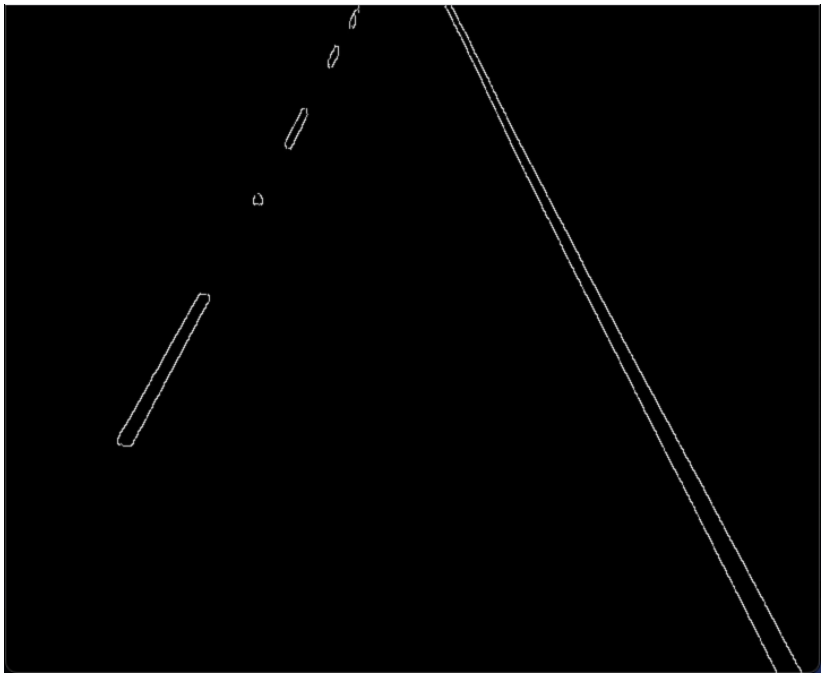


The masks seen above are applied to the image containing the edges and this is the result that is obtained.

DASHCAM TECHNIQUE:



BIRD'S EYE POV:

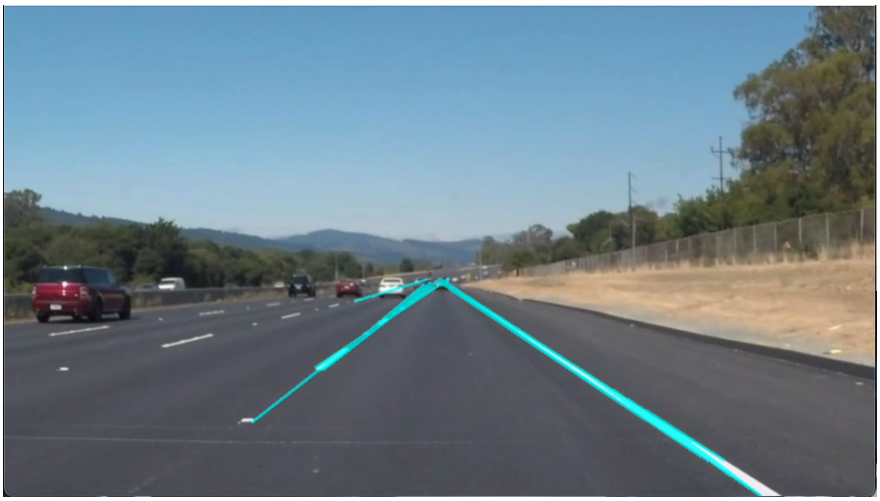


STEP 5:

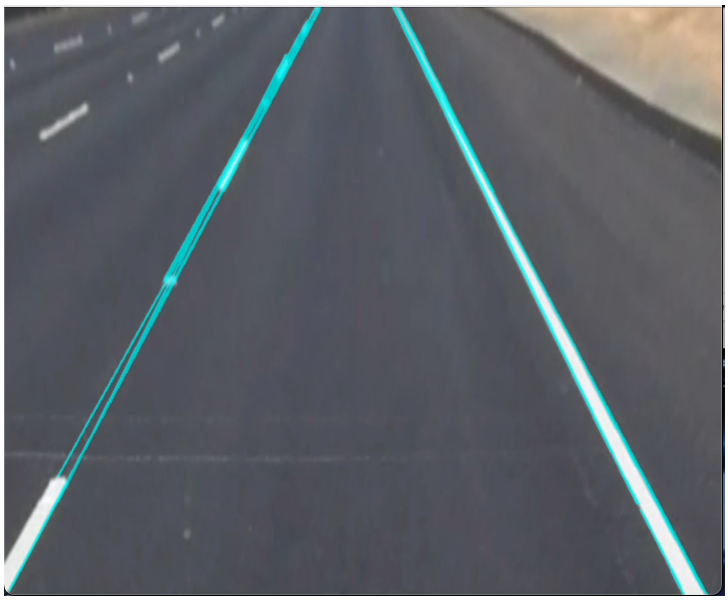
The next step involves finding hough lines from the edge images above.

Here the outputs for each technique.

DASHCAM TECHNIQUE:



BIRD'S EYE POV:



IV:

MATH

Now that we have the hough lines, we will remove some redundant lines and use some linear operations to create 2 lanes for the current lane the car is driving on.
Here are the steps involved in reaching the final lanes.

STEP 1: Find angle of all the lines using the slope of each line and eliminate lines if they do not all within a certain range.

ANGLE RANGE USED:

DASHCAM TECHNIQUE : 15 - 80 Degrees

BIRD'S EYE POV : 40 - 90 Degrees

STEP 2 : Based on the value of the slope, divide the lines into 2 sets, one containing the lines with negative slopes and the other with lines with positive slopes.

(**STEP 3 - 7** are performed for both the positive and negative sets obtained in **STEP 2**.)

STEP 3: Find the median of the slopes and eliminate lines that do not fall within a 1 standard deviation from the median.

STEP 4: The next step involves finding the slope for each line and then finding the Y-Intercept using the slope and then finding the X-Intercept based on the slope and Y-Intercept. We find **X-Intercepts** and **Y-Intercept** for **y = 0** using the **slope intercept form**.

STEP 5: This step involves finding the median of the X-Intercept from the previous step and eliminating lines which do have X-Intercept within a 1 standard deviation of the median.

STEP 6: In this step we store one point from each line in a Matrix.

STEP 7: In this step, we fit a line using the fitLine() function in opencv which uses linear regression to estimate the best-fit line for a set of 2D points in an image. The fitLine() function takes in the set of points as input and returns the slope and y-intercept of the line that best fits the points, along with the line direction vector. **CODE:**

FOR POSITIVE LINES:

```
fitLine(points, FitedLine, DIST_L2, 0, 0.01, 0.01);                                    // Fit a line to the points
int t0 = (0-FitedLine[3])/FitedLine[1];                                                // Compute the intersection with the top edge of the image
int t1 = (frame1.rows -FitedLine[3])/FitedLine[1];                                     // Compute the intersection with the bottom edge of the im
Point p0 = Point(FitedLine[2], FitedLine[3]) + Point(t0 * FitedLine[0], t0 * FitedLine[1]);    // First endpoint of the line
Point p1 = Point(FitedLine[2], FitedLine[3]) + Point(t1 * FitedLine[0], (t1 * FitedLine[1]));  // Second endpoint of the line
line(frame1, p0, p1, Scalar(0, 255, 0), 6);                                             // Draw the line
```

FOR NEGATIVE LINES:

```
fitLine(pointsN, FitedLineN, DIST_L2, 0, 0.01, 0.01)                                   // Fit a line to the points
int t0N = (0-FitedLineN[3])/FitedLineN[1];                                             // Compute the intersection with the
int t1N = (frame1.rows -FitedLineN[3])/FitedLineN[1];                                  // Compute the intersection with the
Point p0N = Point(FitedLineN[2], FitedLineN[3]) + Point(t0N * FitedLineN[0], t0N * FitedLineN[1]);    // First endpoint of the line
Point p1N = Point(FitedLineN[2], FitedLineN[3]) + Point(t1N * FitedLineN[0], t1N * FitedLineN[1]);    // Second endpoint of the line
line(frame1, p0N, p1N, Scalar(0, 255, 0), 6);                                          // Draw the line
```

Here, for both the lines, we calculate the t0 where y=0, which represents a x-coordinate on the line, And t1 represents a x-coordinate on the line where y = number of rows in the frame, i.e. bottom of the frame.

Using this information and the output from fitLine, we can plot the lines on the lane.

V :

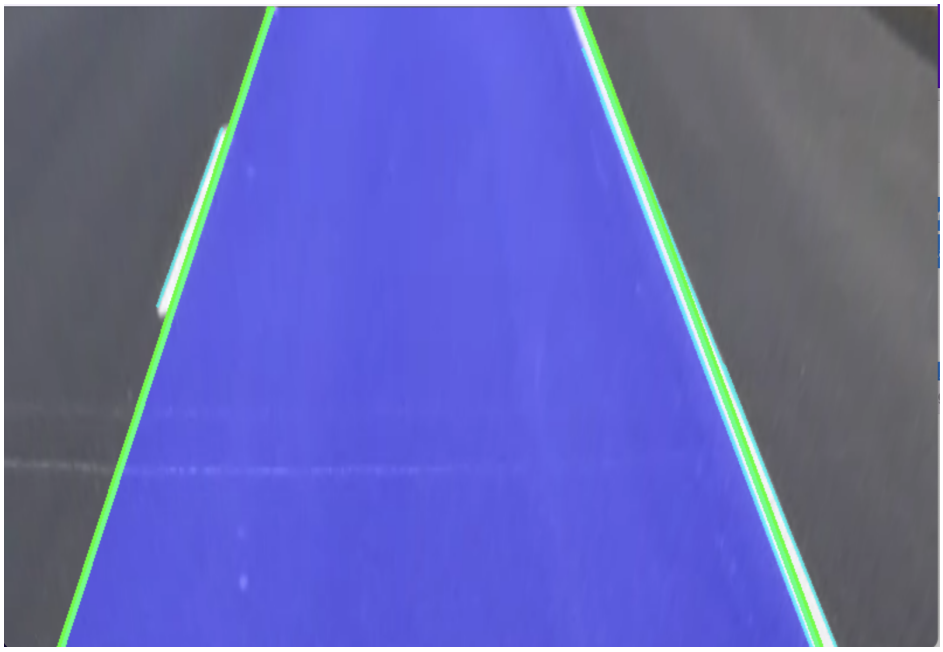
RESULTS

Following the above steps, here is the final result for each both mentioned in the project

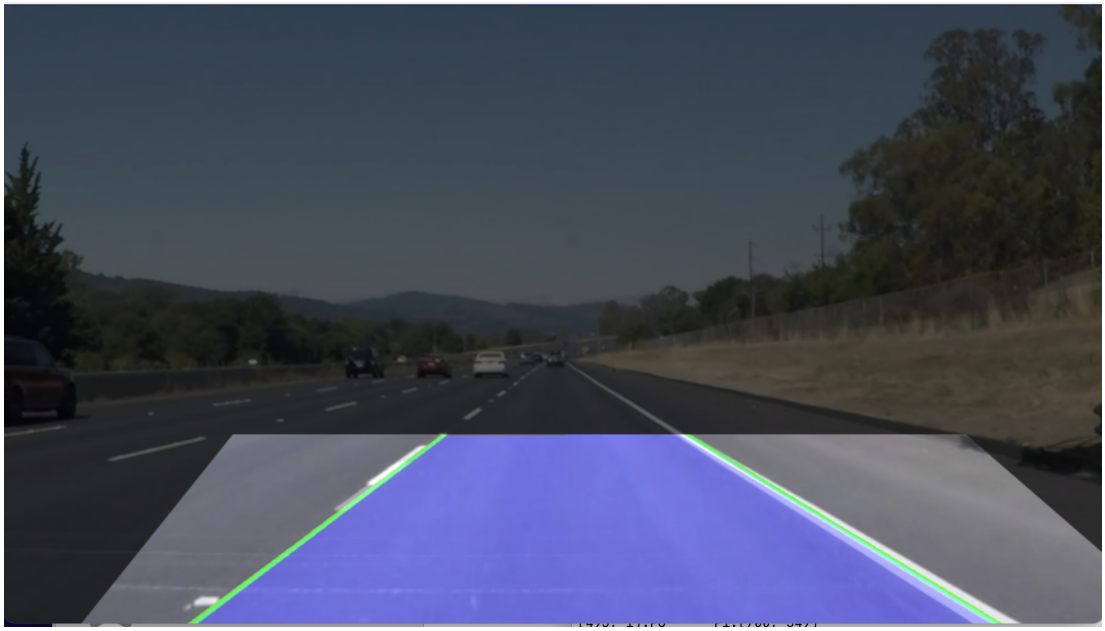
DASHCAM TECHNIQUE:

(continued)

BIRD'S EYE POV:



BIRD'S EYE POV CONVERTED TO DASHCAM POV:



VI :

IMPROVING BY REMEMBERING

The above implementation worked really well, except in a few frames the lines jump around a lot as shown in the results. In order to tackle this I decided to store the lane coordinates from the previous frame and initialized a jump threshold of 50 pixels. If the new pixels move by more than 3 pixels but less than the threshold, then we get the direction of the change by subtracting the new x-value from the old x-value. If the result is negative, then we reduce the x-coordinate from the previous frame by 1 pixel and perform the vice-versa if the result is positive.

This really improved my algorithm and stopped the lines from jumping around. In a certain test video that I ran, initially the left lane started a little to the left of the line, but it corrected itself within a few frames and steadily followed the line from there on.

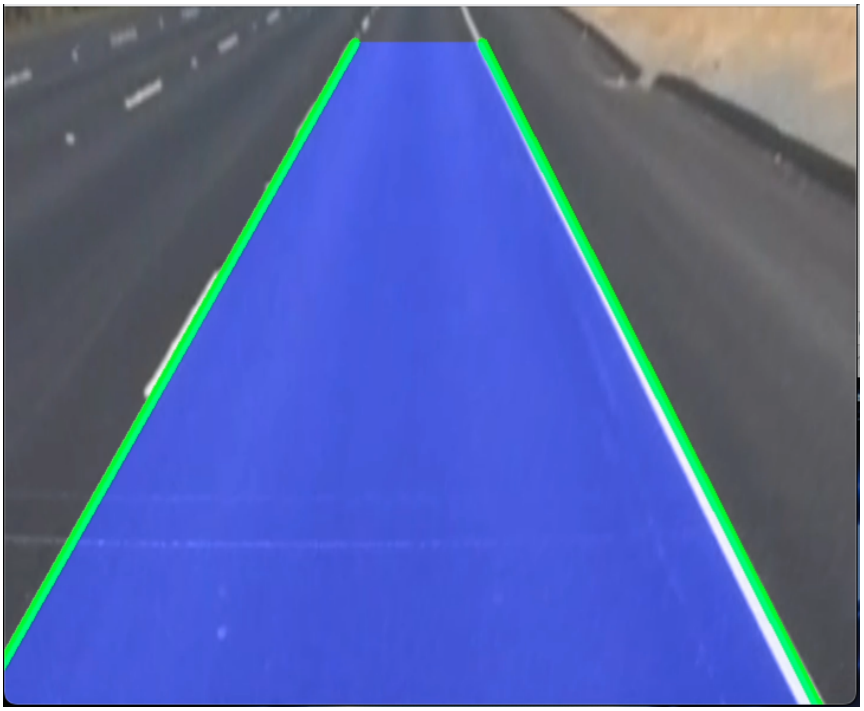
Here are the results after applying the above update to the program.

RESULT:

DASHCAM TECHNIQUE:

(continued)

BIRD'S EYE POV:

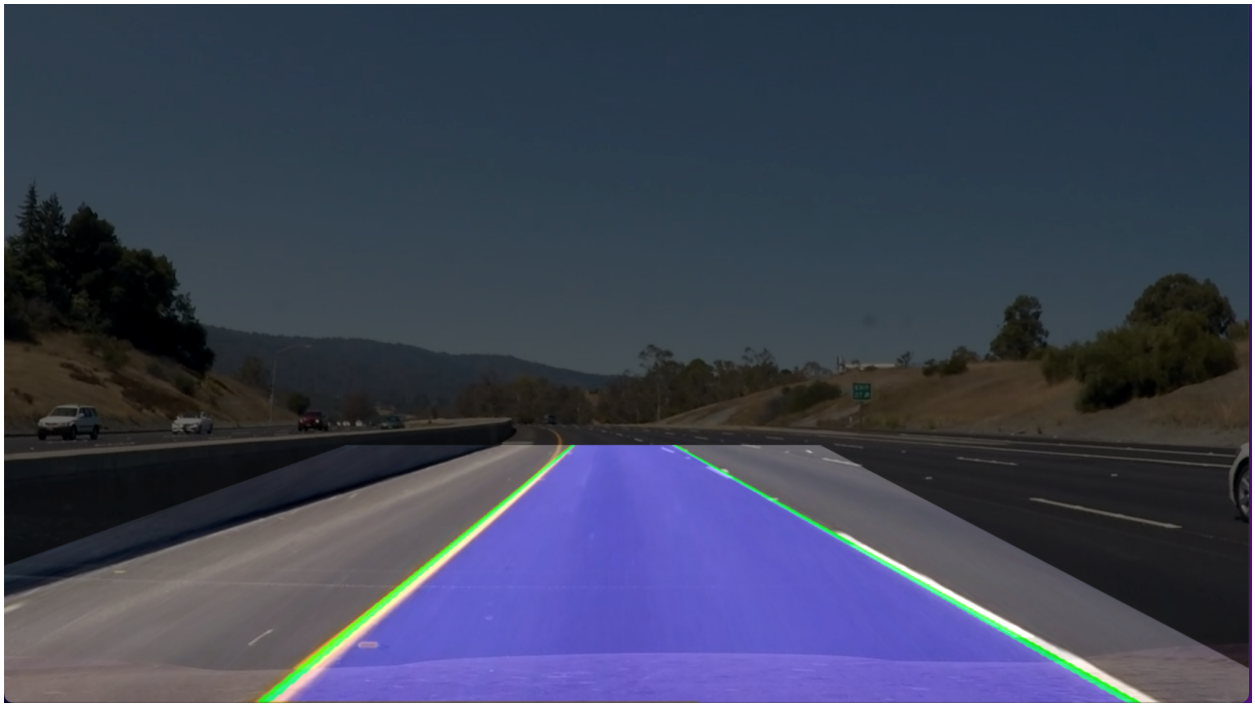


BIRD'S EYE POV CONVERTED TO DASHCAM POV:

As we can see from the results of the homography implementation, the lines are not converting perfectly from each view. I think this is because I did not have access to the camera matrix of the camera used for the video. I had to play around with the pixel values a bit to match the lanes while converting the lanes back to the dashcam point of view.

VIDEO 2:

OUTPUT:



(continued)

VII :

CONCLUSION AND LEARNING

Based on the different approaches I have tried to implement, it was pretty evident that converting a frame to the bird's eye point of view really helps in identifying the lanes. This is pretty clear from the image processing steps, where we can see that output of the bird's eye pov was better in each stage from the first edges image to the final output video with the lane detection shown. I would like to work further on the project in my own time to use regression to fit a line to the frame and maybe implement some other aspects of a self-driving car.

While implementing my project I saw that the video I was running was a little shaky, so my lines were moving around no matter how much I fine tuned my program. So I tried to run my code on another video which seemed much more stable and the output was considerably better.

Implementing the project using opencv in c++ was really hard to begin with. It took a really long time to set everything up, but once I started working on the project I really got comfortable with opencv. Even though I was running the code from the terminal, I was using waitKey statements to debug my code and see where the math was going wrong. Overall I learnt a lot from implementing this project and this might give the confidence and knowledge base I need to work on my capstone project.

VIII :

REFERENCES

1. S. K. Vishwakarma, Akash and D. S. Yadav, "Analysis of lane detection techniques using openCV," *2015 Annual IEEE India Conference (INDICON)*, New Delhi, India, 2015, pp. 1-4, doi: 10.1109/INDICON.2015.7443166.

2. Shin, B.-S., Tao, J., & Klette, R. (2011). A superparticle filter for lane detection. Computer Vision and Image Understanding, 115(2), 262-277. doi: 10.1016/j.cviu.2010.09.005