Vertically Integrated Projects

Image Processing and Analysis

Lane Detection Team

Instructors – Prof. Edward Delp, Prof. Carla Zoltowski

Authors – Pranav Jagada, Tharm Lertviwatkul, Louis Liu, Ben Sukboontip, Rohit Tokala

December 4th, 2020

Table of Contents

Abstract

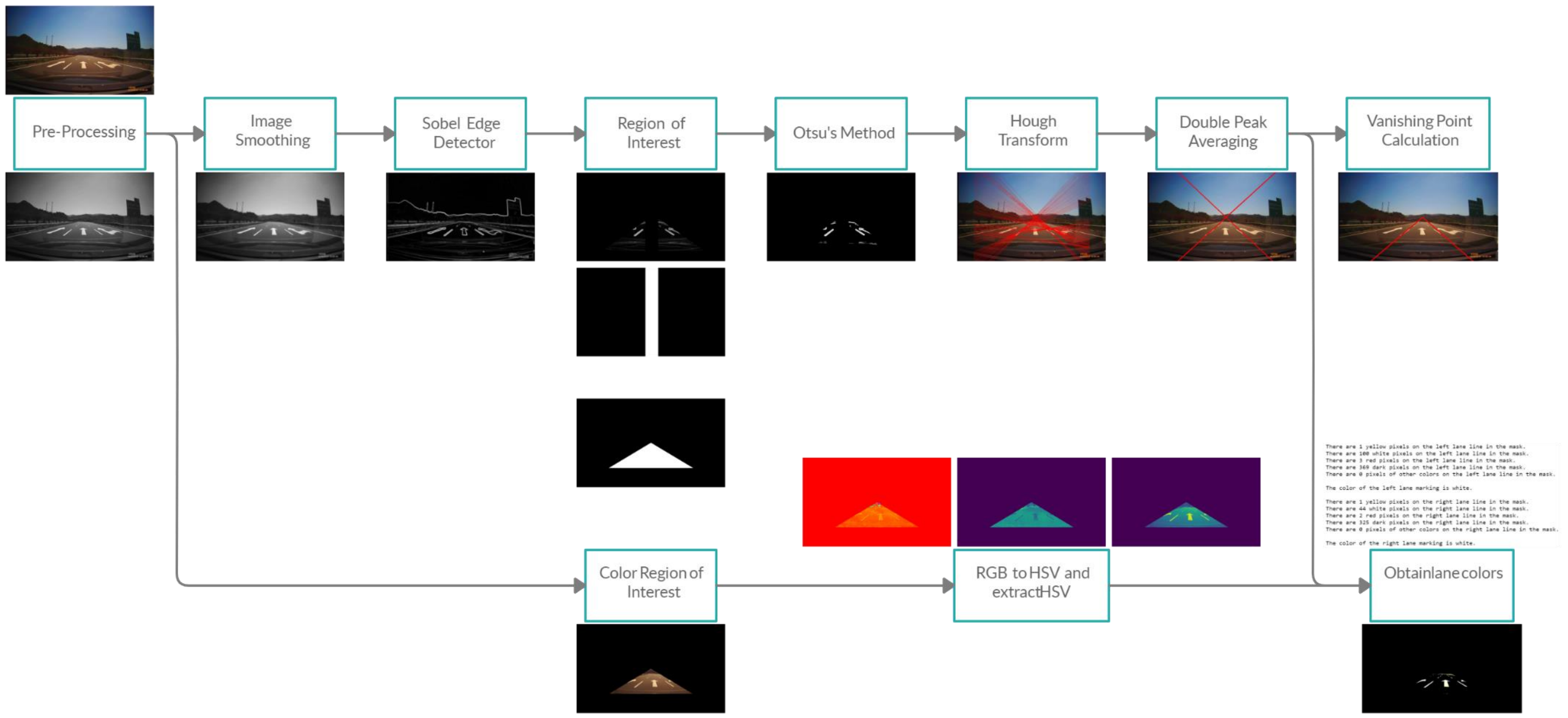
Lane detection is an integral part of any autonomous driving system. The most popular methods of lane detection rely on machine learning. The purpose of this project is to accurately detect lanes and their color without using machine learning principles. Specifically, we take images from the car's dashcam and detect the left and right lane markings of the lane the car is in. The main method used to extract the lanes from the image is the Hough transform, which is a popular robust method for detecting lines in any image. To further improve the results, we employ various pre-processing and post-processing methods. A Sobel edge detector is used to extract the edges in the image before the region of interest is selected. We use Otsu's thresholding to further extract the most prominent edges in the region of interest. Further masking is used to remove irrelevant information such as arrow marks on the road. After the Hough transform is applied, methods such as double peak averaging and a vanishing point calculation filter the output to the most essential information. To detect the color of the lanes, the RGB (Red, Green, Blue) image is converted to an HSV (Hue, Saturation, Value) image. The result is a process that, given a dashcam image, will detect straight lanes and their colors in perfect to moderate lighting conditions.

Project Motivation

Lane detection has been gaining traction in terms of research focus for the past several years due to the rise in popularity of self-driving cars. Due to the autonomous nature of self-driving cars, their research and development naturally go hand in hand with lane detection research. Most recent works in lane detection research tackle the problem using machine learning algorithms to leverage their massively large data sets to train models that perform accurately under both favorable and unfavorable conditions. However, due to the long training time and the need for large data sets, our research aims to develop a method that can detect lanes quickly and relatively accurately under favorable conditions.

Introduction

Our implementation uses many of the key features of image processing and analysis. The key principle of this is that an image is a two-dimensional matrix of pixels with the value of each pixel representing the intensity of light there. Each pixel value ranges from 0 to 255 (inclusive). Our implementation uses a variety of images from the Jiqing Expressway Dataset [1] to test the different features of our program. This project was written in Python.



```
There are 1 yellow pixels on the left lane line in the mask.
There are 100 white pixels on the left lane line in the mask.
There are 3 red pixels on the left lane line in the mask.
There are 369 dark pixels on the left lane line in the mask.
There are 0 pixels of other colors on the left lane line in the mask.

The color of the left lane marking is white.

There are 1 yellow pixels on the right lane line in the mask.
There are 44 white pixels on the right lane line in the mask.
There are 2 red pixels on the right lane line in the mask.
There are 325 dark pixels on the right lane line in the mask.
There are 0 pixels of other colors on the right lane line in the mask.

The color of the right lane marking is white.
```

Figure 1. Block diagram

As shown by the block diagram in Figure 1, an image is first resized to ensure that no image is too large. The image is then converted to grayscale since working with a color space is an unnecessary complication. After preprocessing, the image is then blurred using image smoothing to remove noise before it goes into the Sobel edge detector. After applying a region of interest, the edge-detected image undergoes thresholding using Otsu's method. The thresholded edge-detected image goes into the Hough transform to produce different clusters of possible lines. Using double peak averaging, one line is obtained for each lane, and the vanishing point is then calculated. For color detection, the resized image has a region of interest applied to it and is converted into the HSV color space. The color of each lane is then detected using the lines from the double peak averaging.

Implementation

Dataset Construction

We selected the Jiqing Expressway Dataset [1] based on a few criteria:

- The dataset must be mostly on highways.
- The dataset must be in good lighting and weather conditions.
- The dataset must be in low traffic conditions.
- The dataset must mostly contain straight lanes.
- Lane markings must be clearly visible.
- The video must be fairly stable.
- The dataset must contain labels of lane markings.

Those criteria were selected to provide an easier starting ground for this project. The Jiqing Expressway Dataset [1] meets all of those criteria, but it also contains enough images that posted a challenge and tested the limits of our implementation. The dataset contains multiple dashcam videos of highways in China. The lanes were labeled semiannually with dots in subsets.

Unfortunately, the labeling system is not fully correct and thus contains some faulty labels. In some cases, as shown in Figure 2, parts of the hood of the car are labeled as lanes when they just happen to connect to the actual lane. This mostly happens when the car is in the center lane. To detect faulty labels, slopes between each connected pair of dots were calculated. If the change in the slope is not gradual, then there must be a sharp turn and thus a false label. The false labels were then split into two sets based on the sign of the slope. Because lanes are distorted towards the vanishing point, lanes on the left will have negative slopes and lanes on the right will have positive slopes. The set with the correct slope was selected and used for evaluation. Figure 3 shows a comparison of before and after.

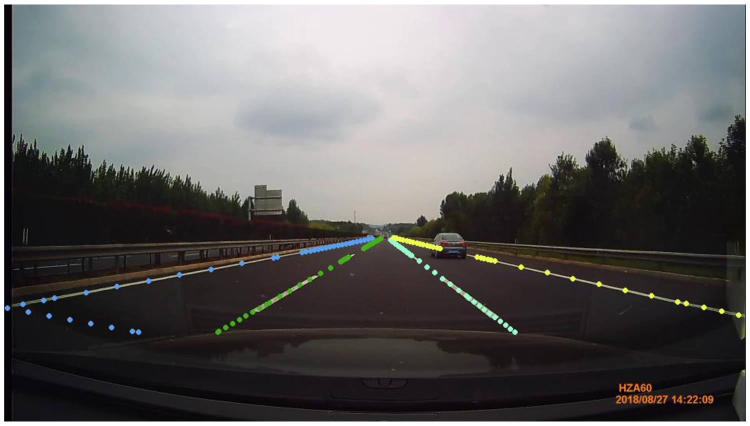


Figure 2. Dotted labels in the Jiqing Expressway Dataset [1]. Each color represents an individual lane.

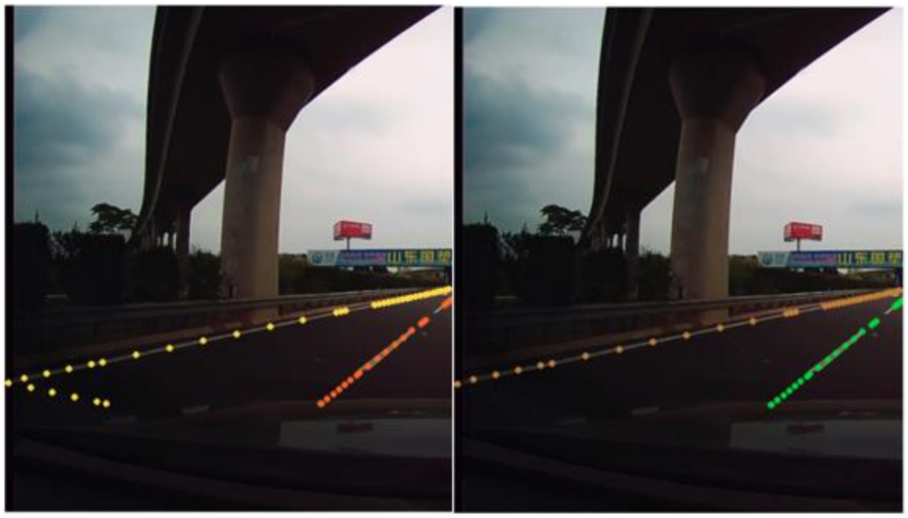


Figure 3. Comparison between faulty labels and processed labels

Pre-Processing

Before anything can be computed from the image, several things must be done to the image during pre-processing. Firstly, the image must be resized so that the height of the image is 512 pixels to ensure that all images are the same height. This is mainly done to ensure that no image is too large. As image size becomes larger, computation time greatly increases, so all images are initially resized to 512 pixels to ensure a relatively constant computation time.

Next, the images were converted using grayscale because we decided that the color representations of the images were not necessary features to accurately detect lanes. Moreover, later processes such as Sobel edge detection and the Hough transform work well with grayscale images. The grayscale conversion is done using the ITU-R BT.709 color coefficients shown in Equation 1, where Y is the grayscale value of the pixel computed using the three values of the red, green, and blue pixel values of the colored images. In the equation, the R, G, and B variables represent the red pixel value, green pixel value, and blue pixel value, respectively, of the specific color pixel while Y represents the grayscale value of the pixel presented by R, G, and B.

$$Y = 0.2126R + 0.715G + 0.0722B$$

Equation 1. ITU-R BT.709 color coefficients [2]

While the average color method computed using Equation 2 was considered, the ITU-R BT.709 color coefficients are used because of how human eyes perceive various colors differently.

$$Y = \frac{R + G + B}{3}$$

Equation 2. Averaging grayscale conversion [2]

The ITU-R BT.709 color coefficients take into account how our eyes see more green than blue and red to create coefficients that best represent how our eyes perceive color while the average color method does not take any of that into account. Therefore, the coefficients shown in Equation 1 are used for our grayscale conversion.

Image Smoothing

After the image is resized and converted to grayscale, we performed image smoothing to the grayscale image before passing it into the Sobel edge detection function. The purpose of image smoothing or blurring is to reduce the noise of the image – a low pass filter. When we pass an image into the Sobel edge detector, we want to take the partial derivatives in the *x* and *y* directions to compute the magnitude of the gradient to detect edges. However, the derivative of noise is more noise. We want to reduce the noise as much as possible, thus image smoothing is needed beforehand. Image smoothing makes the edges of the input image less defined, which reduces the unnecessary details and edges in the image, thus making it easier for the Sobel edge detector to detect the important edges in the image. If we use the grayscale image as the input to the Sobel edge detector without blurring it, there may be more unwanted edges detected. This would cause problems in later functions such as the Hough transform, where more Hough lines are detected, which can alter the clustering of lines in double peak averaging, resulting in the wrong lane lines detected.

We have chosen to use simple averaging smoothing as our smoothing algorithm because we have found it to be slightly easier to implement compared to other algorithms such as Gaussian smoothing and just as effective on our sample images. Simple averaging smoothing consists of performing a 2D-convolution on a window of pixels with a specified kernel size, where the elements of the kernel are all equal to $(\frac{1}{kernel\ size})^2$.

Kernel size can vary the performance of the Sobel edge detector. If the kernel size is too small, some unwanted details and edges may still be detected which causes problems in later functions, as mentioned. However, if the kernel size is too large, it may blur the image too much, causing Hough lines with inaccurate slopes to be generated. Therefore, we tested four different kernel sizes; 3, 5, 7, and 11, where all of which have appropriate padding accounted for to maintain the dimensions of the image. In the images below, a kernel size of 3 has little effect on the image, whereas a kernel size of 11 blurs the image too much. After running some sample images from preprocessing through the Hough transform, we found that in general, a kernel size of 5 generated the best results.

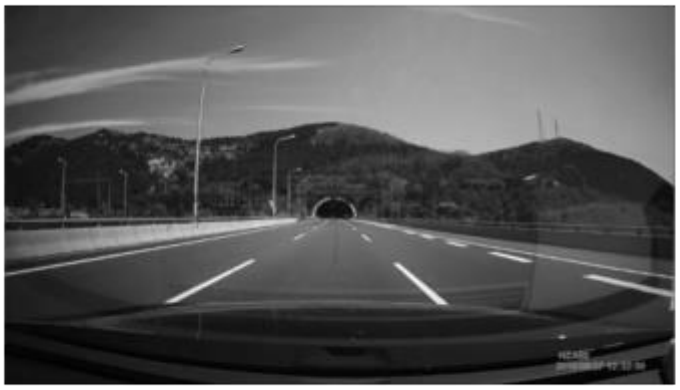


Figure 4.Smoothed image with 3x3 kernel

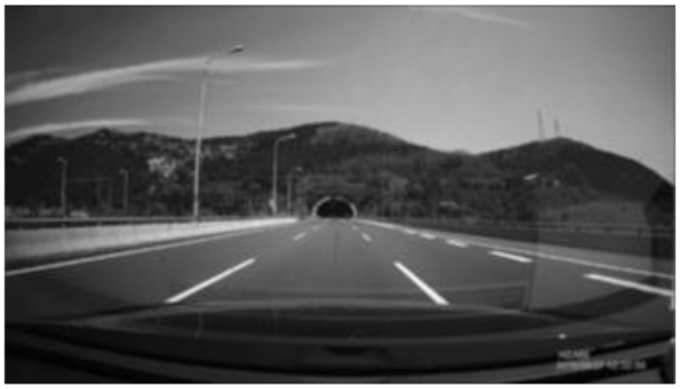


Figure 5.Smoothed image with 5x5 kernel

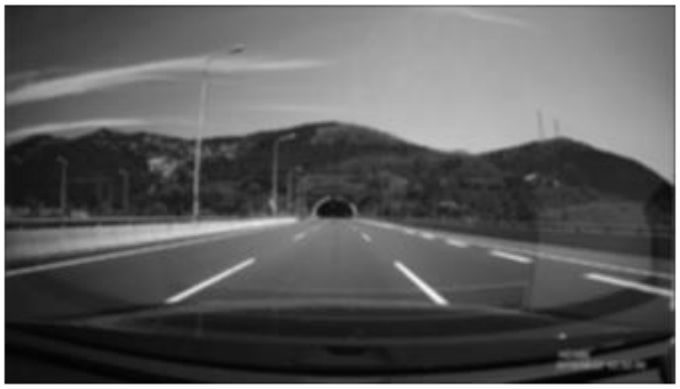


Figure 6.Smoothed image with 7x7 kernel

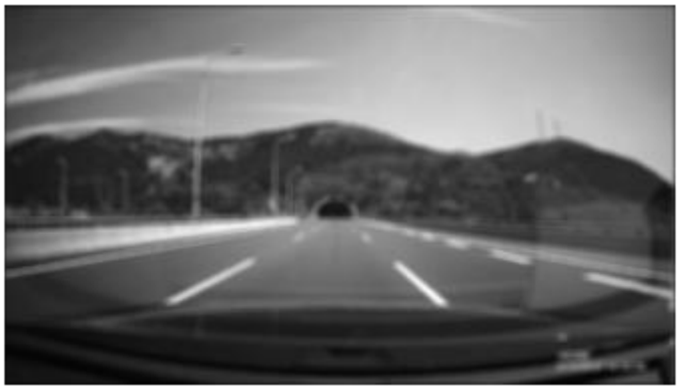


Figure 7. Smoothed image with 11x11 kernel

To perform simple averaging smoothing, we use Equation 3, where φ is the pixel value of the grayscale image, ranging from 0 to 255, and x and y are the column and row indices, respectively. A 2D-convolution is essentially the sum of the element-wise multiplication between each pixel value in the window and its corresponding kernel value for simple averaging smoothing because each of the elements in the kernel is $(\frac{1}{kernel\ size})^2$. ω, the resulting pixel value for the smoothed image, is the average pixel value of the window. When performing image smoothing, we are forming new images for each ω calculated. Furthermore, we use a stride of 1 to calculate each ω value.

$$\omega = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} * \frac{1}{25} * \begin{pmatrix} \varphi_{x-2,y-2} & \varphi_{x-1,y-2} & \varphi_{x,y-2} & \varphi_{x+1,y-2} & \varphi_{x+2,y-2} \\ \varphi_{x-2,y-1} & \varphi_{x-1,y-1} & \varphi_{x,y-1} & \varphi_{x+1,y-1} & \varphi_{x+2,y-1} \\ \varphi_{x-2,y} & \varphi_{x-1,y} & \varphi_{x,y} & \varphi_{x+1,y} & \varphi_{x+2,y} \\ \varphi_{x-2,y+1} & \varphi_{x-1,y+1} & \varphi_{x,y+1} & \varphi_{x+1,y+1} & \varphi_{x+2,y+1} \\ \varphi_{x-2,y+2} & \varphi_{x-1,y+2} & \varphi_{x,y+2} & \varphi_{x+1,y+2} & \varphi_{x+2,y+2} \end{pmatrix}$$

Equation 3. 2D convolution for image smoothing using 5x5 kernel [2]

To maintain the dimension of the image, we have to pad each border of the grayscale image with an appropriate number of columns or rows. The number of padded rows and columns on each border of the grayscale image before being smoothed is determined by Equation 4, where each of the elements in the extra rows and columns is zero. For example, when using a 5x5 kernel, 2 extra rows and columns are added to each border of the grayscale image to maintain the dimension of the smoothed image generated.

$$row\ or\ col\ num = \frac{Kernel\ width - 1}{2}$$

Equation 4. Number of padded rows and columns

Sobel Edge Detector

An edge in an image is defined as an area where there is an abrupt change in the grayscale value. To detect the edges, we must do a window operation to find the directional change in intensity value, or equivalently, the gradient. A high gradient value suggests that there is an abrupt change in the intensity value, signaling that there is an edge. By calculating the gradient in each direction, we are essentially taking the partial derivatives in the *x* and *y* directions. Similar to the image smoothing function, the edge enhanced image can be constructed by calculating two 2D-convolutions to compute the gradient in each of the *x* and *y* directions.

The Sobel edge detector uses two 2D-convolutions from Equations 5 and 6 below to calculate the gradients in the *x* and *y* directions, respectively. Notice that the 3x3 kernels used in the *x* and *y* directions are just a transpose of one another. *ω* is the grayscale value from the smoothed image calculated earlier, and *x* and *y* are the corresponding column and row indices of each pixel. Once the gradients in each of the directions are calculated, we then calculate the magnitude of gradient *G* using the Euclidean distance formula in Equation 7. The magnitude of the gradient is the total directional change of intensity value for each window. We have also used a stride of 1 for our convolution calculations. A new edge enhanced image is formed from the magnitude of gradient values calculated. With the appropriate smoothing kernel size, we have now formed an image with the most important edges in the image enhanced.

$$\frac{\partial\omega}{\partial x} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \omega_{x-1,y-1} & \omega_{x,y-1} & \omega_{x+1,y-1} \\ \omega_{x-1,y} & \omega_{x,y} & \omega_{x+1,y} \\ \omega_{x-1,y+1} & \omega_{x,y+1} & \omega_{x+1,y+1} \end{pmatrix}$$

Equation 5. Gradient in the x-direction [2]

$$\frac{\partial\omega}{\partial y} = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * \begin{pmatrix} \omega_{x-1,y-1} & \omega_{x,y-1} & \omega_{x+1,y-1} \\ \omega_{x-1,y} & \omega_{x,y} & \omega_{x+1,y} \\ \omega_{x-1,y+1} & \omega_{x,y+1} & \omega_{x+1,y+1} \end{pmatrix}$$

Equation 6. Gradient in the y-direction [2]

$$G = \sqrt{\left(\frac{\partial\omega}{\partial x}\right)^2 + \left(\frac{\partial\omega}{\partial y}\right)^2}$$

Equation 7. Magnitude of gradient [2]

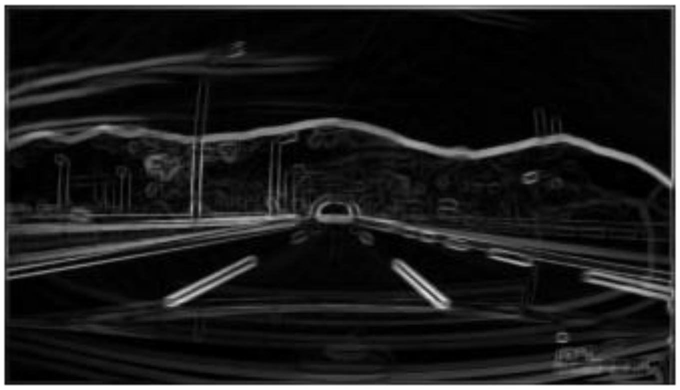


Figure 8. Sobel edge enhanced image

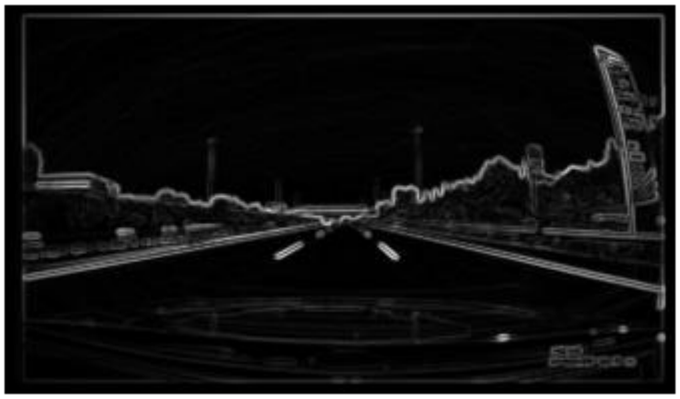


Figure 9. Sobel edge enhanced image

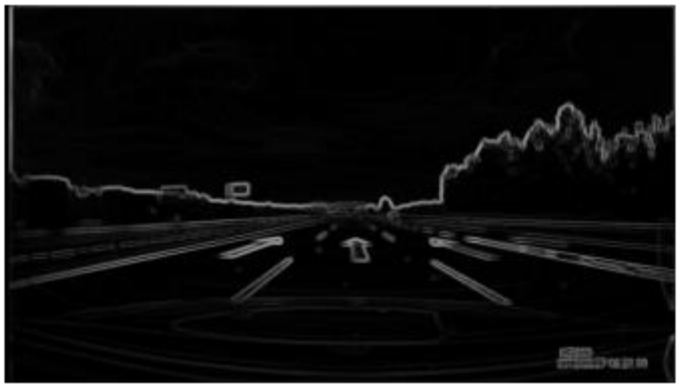


Figure 10. Sobel edge enhanced image

Masking

Almost all images taken from dash cameras of cars include information that is unnecessary to calculate lanes such as skies, signs, and even lane marking arrows on the road. Therefore, we used three different masking layers to eliminate as much noise as possible during our computation. Masking works by establishing a region of interest that we want for further computation and ignoring all information outside that region that could potentially create noise and affect our output.

A mask is first created by creating a black image with the same dimensions as the input image. Then, the value of each pixel in the mask that is inside the region of interest is set to 255 (white) to differentiate between pixels that are outside of the region of interest (black). Lastly, the mask is then compared to the input image by checking the index of each pixel of the input and seeing if its corresponding mask value is 0 or 255. If it is 0, the pixel in the input image is changed to black, and the input gets to retain its original pixel value if the corresponding mask value is 255.

The first masking layer is a simple triangle shown in Figure 11 that eliminates big, overarching noise such as the sky, other lanes, and the front of the car. This is done by establishing the middle of the image by dividing the height and width of the image by two and using that point as the top of the triangle. Next, that point is connected to the bottom-left corner and the bottom-right corner of the image to form a triangle. For each row index of a given pixel, the column index corresponding to the left and right side of the triangle is calculated using Equations 8 and 9 respectively. In these equations, h and w represent the height and width value of the image in pixel units. Moreover, y represents the row index of the given pixel while x_l represents the left column index of the triangle given y, and x_r represents the right column index of the triangle given y. For a pixel to be inside the region of interest, the column index of the pixel, given the row index y, must be both more than x_l and less than x_r. It should be noted that all values where h is less than half the height of the image are ignored and not inputted to either Equations 8 or 9.

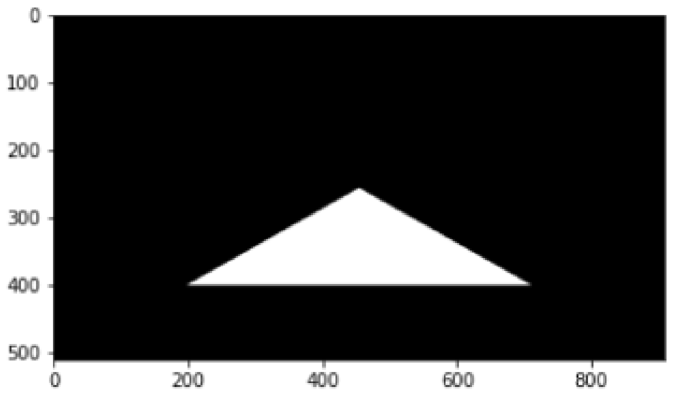


Figure 11. Region of interest 1

$$x_l = \frac{(\frac{h}{2} - y) * \frac{w}{2}}{\frac{h}{2}}$$

Equation 8. Left boundary of the first triangular region of interest

$$x_r = \frac{y * \frac{w}{2}}{\frac{h}{2}}$$

Equation 9. Right boundary of the first triangular region of interest

Lastly, to remove noise from the front of the car, all pixels with a row index above 400 are not in the region of interest. Therefore, the region of interest of the first mask is calculated by creating a triangle using Equations 8 and 9 and using pixels that are both within the triangle and having a column index between 256 (height of the image divided by 2) and 400.

Thresholding with Otsu's Method

Otsu's thresholding method is used to find the threshold value that separates the foreground of the image from the background. This method is implemented in our solution to filter the edges that were detected by the Sobel edge detector to remove the less prominent edges. The resultant image is an image where the white pixels represent edges, and all the other pixels are black.

This is done by iterating through all the possible pixel values in the image and calculating the variances for the two classes that are formed on either side of the threshold. There are two main methods for finding the optimum threshold value – minimizing within-class variance and maximizing between-class variance. Since both these methods give the same result, we used the minimizing within-class variance method.

$$P(i) = \frac{count(i)}{total\ number\ of\ pixels}$$

Equation 10. Calculation of probability of pixel value [3]

$$q_1(t) = \sum_{i=1}^{t} P(i), \qquad q_2(t) = \sum_{i=t+1}^{I} P(i)$$

Equation 11. Calculation of class weights [3]

$$\mu_1(t) = \sum_{i=1}^{t} \frac{i * p(i)}{q_1(t)}, \qquad \mu_2(t) = \sum_{i=t+1}^{I} \frac{i * p(i)}{q_2(t)},$$

Equation 12. Calculation of class means [3]

$$\sigma_1^2(t) = \sum_{i=1}^{t} \left(i - \mu_1(t)\right)^2 * \frac{p(i)}{q_1(t)}, \qquad \sigma_2^2(t) = \sum_{i=t+1}^{I} \left(i - \mu_2(t)\right)^2 * \frac{p(i)}{q_2(t)}$$

Equation 13. Calculation of class variances [3]

The first step is to iterate through all the threshold values from 1 to *I* (the maximum pixel value in the ROI) and calculate the probability of a pixel value *i* using equation 10. For each threshold value *t*, class weights $q_1(t)$ and $q_2(t)$ are calculated using Equation 11. Equations 12 and 13 are used to calculate the class means and class variances respectively. Once these are calculated, the in-class weighted variance $\sigma_w^2(t)$ for threshold *t* is calculated using Equation 14.

$$\sigma_w^2(t) = q_1(t) * \sigma_1^2(t) + q_2(t) * \sigma_2^2(t)$$

Equation 14. Calculation of within-class variance. [3]

After iterating through all the possible threshold values, the threshold that gives the least within-class variance is used as the final threshold value. All pixels in the image above the threshold value are set to white, and everything below is set to black. The resulting final image is a binary image where white pixels represent the detected edges.

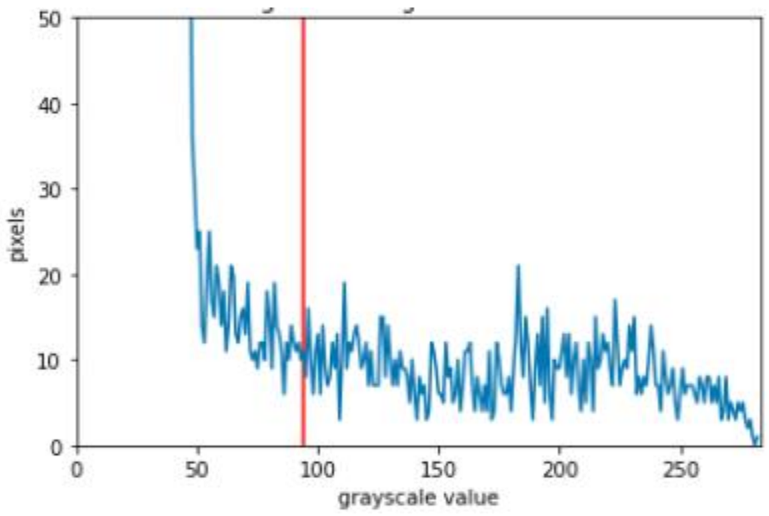


Figure 12. Histogram of the region of interest with the red line representing the final threshold

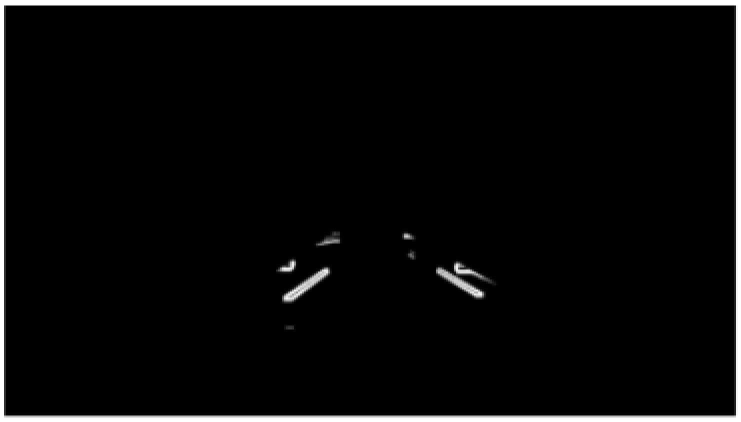


Figure 13. The region of interest after Otsu's thresholding

Hough Transform

Given an edge detected image, Hough transform can be used to represent the image in polar coordinates to find lines within the image. However, before diving into the Hough transform, we must first understand how lines can be represented in the accumulator space. Figure 14 demonstrates how lines in the Cartesian plane can be represented with a ρ and θ value instead of a linear equation containing x and y. Here, we see that any arbitrary line can be represented with a ρ and θ value where θ is the angle in degrees between the top of the image to the line that intersects the origin and is perpendicular to the line that is being represented. Moreover, ρ is computed in Equation 15 given θ and the x and y values of the point where the dashed line in Figure 14 intersects the line that is being represented. It should be noted that for each line, θ ranges from 0 to 180 instead of 360 because a negative ρ can be used to represent lines that have θ values between 180 and 360 degrees.

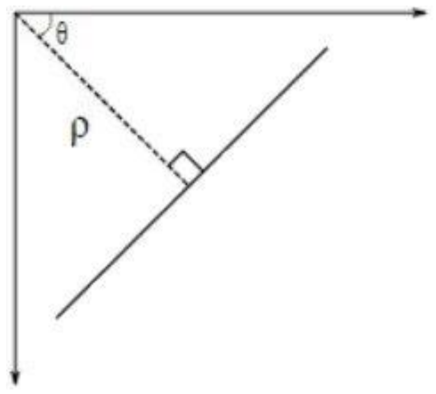


Figure 14. Line representation in accumulator space

$$\rho = x * cos(\theta) + y * sin(\theta)$$

Equation 15. Conversion from Cartesian plane to Hough space [4]

Now that we know how each line can be represented with ρ and θ, we can now begin the Hough transform calculation. Because the edge detected images contain either black or white pixels due to Otsu's thresholding, Hough transform goes through every white pixel in the edge detected image and draws 180 evenly spaced lines that surround the pixel. Figure 15 demonstrates 8 evenly spaced lines that could be drawn from a single pixel which is represented as a red dot in the middle of the figure. Each of these lines is then represented as ρ and θ values and then plotted on the accumulator space shown in Figure 16. The accumulator space has ρ as its x-axis and θ as its y-axis so that each point in the accumulator space is represented as a line. It should be noted that the size of the accumulator space is independent of the size of the input image. Initially, every pixel value in the accumulator space is set to 0 (black). Next, for each line that is generated from the white pixels in the edge detected image, the voted line would increase the pixel value of its corresponding pixel value in the accumulator space and cast its "vote." Therefore, whiter lines or the more voted lines in the accumulator space mean that the specific line is generated more than the darker lines or received more votes.

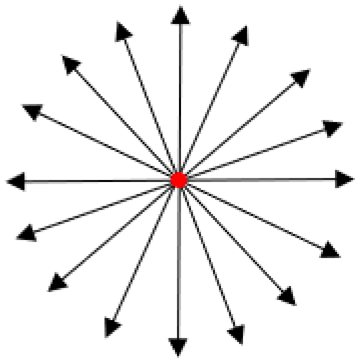


Figure 15. Possible lines represented in the accumulator space

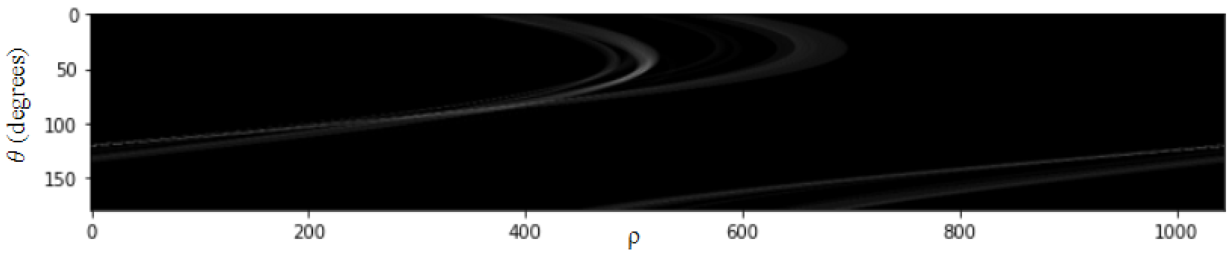


Figure 16. Accumulator space

Once the accumulator space is created and voting is done on all possible lines going through the detected edges, the next step is to convert the most prominent ρ and θ pixels into Cartesian lines. The threshold to select the most prominent lines took a lot of experimentation and testing. A threshold value of 50 was found to be optimal, i.e., every ρ and θ cell that has more than 50 votes is selected and stored in a list.

$$y = -\frac{cos(\theta)}{sin(\theta)} * x + \frac{\rho}{sin(\theta)}$$

Equation 16. Conversion from polar coordinates to Cartesian [4]

Equation 16 [4] is used to convert the selected ρ and θ pixels into the Cartesian plane, where x is the x-coordinate in the image, y is the y-coordinate in the image, ρ is the ρ value from the accumulator space, and θ is the θ value from the accumulator space. This equation gives an infinite line of a specific slope. These infinite lines formed are then limited to the boundaries of the image and overlaid on top of the input image. These lines indicate all the possible lane lines that the Hough transform has detected. Further processes are necessary to filter the lines to get the desired output.

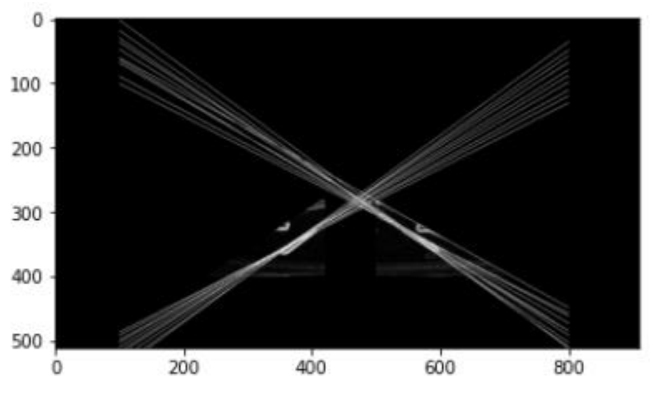


Figure 17. Hough transform lines overlaid on thresholded image

Since we are using a fixed threshold value to select the more prominent lines from the accumulator space, there are cases when there is a gap between the lane lines, and this causes the lane to not be detected. At this point, this is a case where the project does not work but is something that could be fixed in the future by utilizing a variable threshold method. As seen in Figure 18, though the lane line on the right is detected, the left lane marking could not be detected. A fully functioning top view transform will help solve this issue.

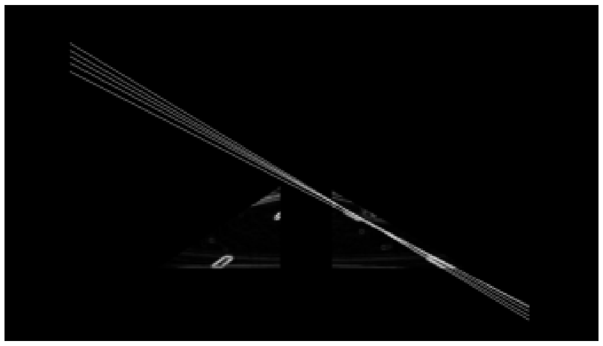


Figure 18. Case where the threshold is too low

Clustering

Due to the abundance of lines in the output of the Hough transform, we experimented with clustering to cluster the output lines into lanes. Because the output of the Hough transform can be represented as a plot where the axes are ρ and θ, a scatter plot can represent the Hough transform output where each point on the plot represents a line. Therefore, clustering the lines can just be a matter of clustering the points on the ρ and θ plot. Because most clustering algorithms need the number of clusters as parameters, they could not be used to cluster lanes because we do not know how many lanes or clusters there should be for each image. Therefore, hierarchical clustering has been proposed as a possible solution mainly because it can cluster multiple different lines into desirable lanes while also not needing to use the number of clusters as parameters.

Hierarchical clustering works by initially placing each line in a unique cluster that contains only one line. Then, the algorithm finds the two clusters that are closest together and combines the two clusters to create a new cluster. This process is repeated so that only a single cluster remains. Therefore, a dendrogram can be made to represent the process of combining different lines or points into a single cluster as shown in Figure 19.

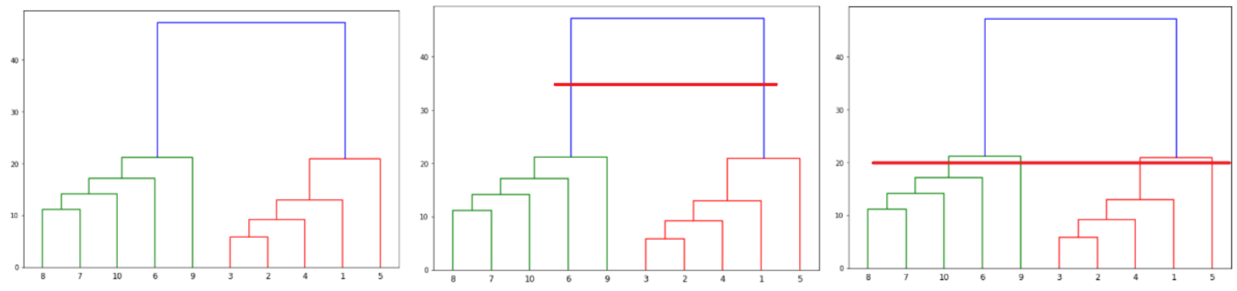


Figure 19. Dendrograms for hierarchical clustering

The dendrogram represents the different stages of the clustering. Therefore, whenever a horizontal line is drawn across the dendrogram as shown in Figure 19, clusters can be created as the result where the number of clusters would be the number of vertical lines intersecting the horizontal red line. For example, the middle figure in Figure 19 will result in two different clusters while the right figure will result in four different clusters. Now, we must determine which horizontal line we have to draw for our lane detection algorithm.

To initially test if a hierarchical clustering approach would work, the scikit-learn's Agglomerative Clustering function is applied to the Hough transform output. This function accepts a distance threshold as a parameter to determine the number of clusters by setting a minimum Euclidean distance between the clusters. Therefore, if the clusters are closer than the minimum threshold, the algorithm will proceed to join neighboring clusters until all clusters are more than the distance threshold apart. Now, the only problem is to find the correct distance threshold for the algorithm.

While simple images with little noise work with the distance threshold value of 600 as shown in Figure 20, problems arise for the hierarchical clustering approach when more noise is introduced so that the scatter plot cannot be objectively clustered as shown in Figure 21.

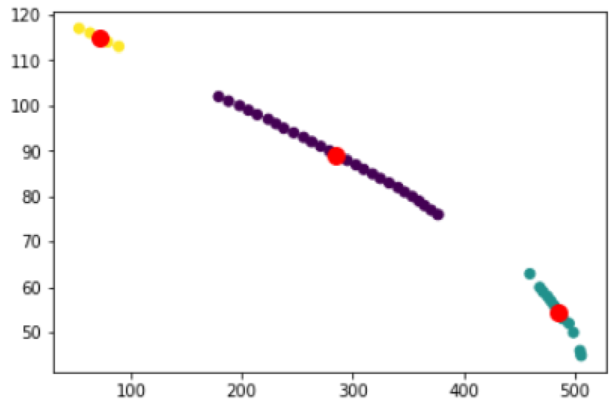


Figure 20. 600 distance threshold clustering case where clusters are clear

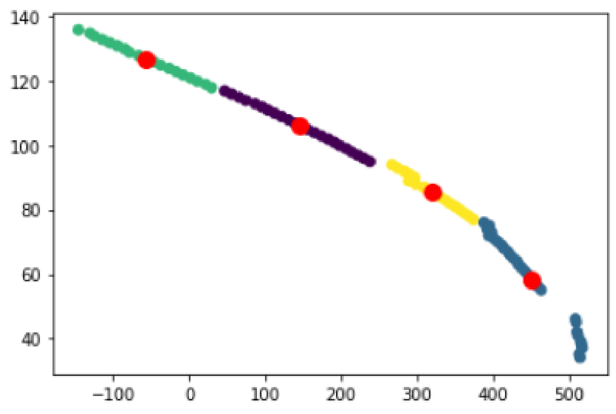


Figure 21. 600 distance threshold clustering case where clusters are not clear

Therefore, while the hierarchical clustering approach works well with favorable conditions with little noise, it struggles to cluster the Hough transform output that has many lines close together so that clusters could not be objectively defined.

Double Peak Averaging

Due to the natural weaknesses of hierarchical clustering, we developed our own method to clean the Hough Transform output more accurately called double peak averaging. For double peak averaging, ρ and θ values are first divided into histograms of 15 bins, which were determined experimentally to give the best result. Then, indices of local maximums in both histograms are searched and recorded. If the indices of local maximums match between the two histograms, then the ρ and θ values in those bins are averaged and outputted as shown in Figure 22.

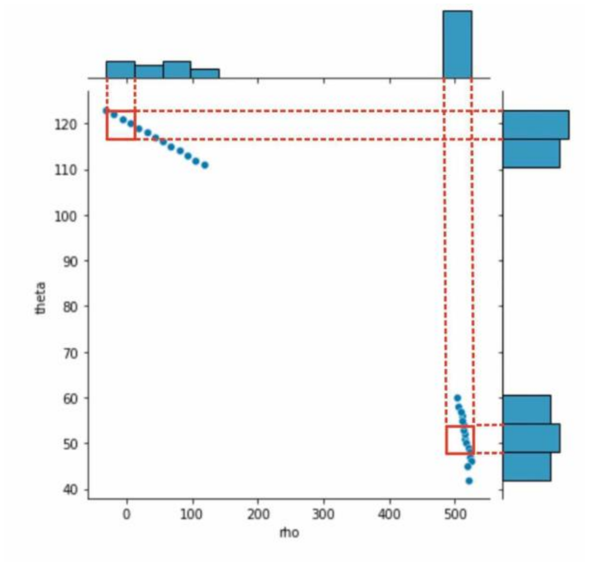


Figure 22. Double peak averaging simple cases

If there are continuous bins that contain local maximums with the same values in either the ρ or θ histogram, and if one or more local maximums are present in the other histogram's matching bins, then all values in the continuous bins are averaged and outputted as shown in Figure 23.

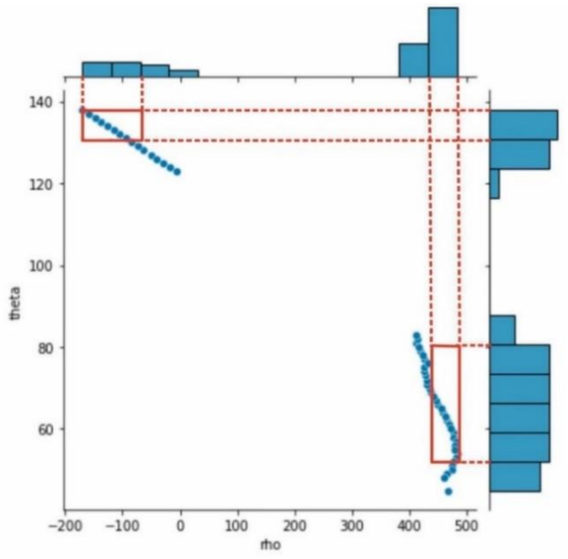


Figure 23. Double peak averaging edge case

Lane Marking Removal

As we tested more images, our collection of sample images got larger and larger. However, we only chose to test with straight lanes without other objects in the image that can be classified as an edge by the Sobel edge detector. If lane markings such as arrows are present, they will be detected by the Sobel edge detector as edges. If we do not apply an algorithm to remove the unwanted lane markings from the Sobel edge enhanced image, the Hough lines for those lane markings will be generated, causing extra clusters of Hough lines in the double peak averaging, thus extra lane lines will be present as the final output after double peak averaging.

Furthermore, the lane arrows are also present in neighboring lanes. Even with the triangular region of interest applied, in some sample images, the arrows are still sometimes within the region of interest if the car is not in the center of the lane, as shown in Figure 24. In most of our test cases, because the camera is on the left side of the car, it is always edging towards the left of the road, making the center of the lane somewhat to the right of the image. Thus, for example in Figure 24, the left lane marking is usually the lane marking that falls within the region of interest.

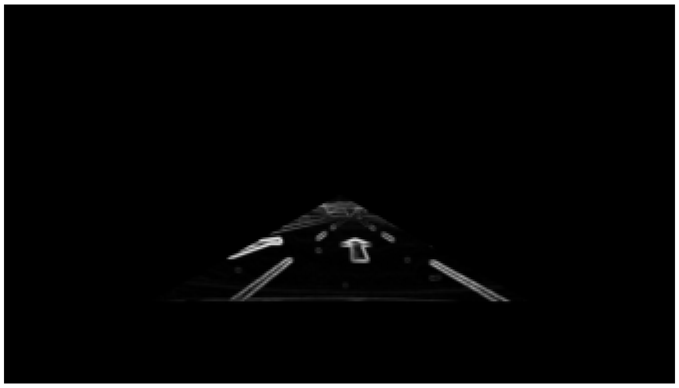


Figure 24. Region of interest 1 applied

We applied a second region of interest using Equations 17 and 18, which are the same equations for the first region of interest, but shifted down and to the right due to most of our test cases with lane markings having an issue where the left lane marking is within the region of interest. The second region of interest is shown in Figure 25. The purpose of the second region of interest is to eliminate the left lane marking in order to display only the two-lane edges and the central lane marking, if present. Furthermore, to eliminate the central lane marking from the image, we applied a third region of interest, shown in Figure 26, to the image. However, instead of having a triangular mask, the third region of interest is a rectangular mask where the left boundary is fixed at 420 pixels, and the right boundary is fixed at 500 pixels. We chose to apply a rectangular mask because, in some sample images like Figure 27, there may be objects such as tunnels or vehicles near the vanishing point, which may cause issues with the Hough transform later on. Thus, this final mask does not only eliminate the central lane marking, but it also eliminates any unwanted edges near the vanishing point.

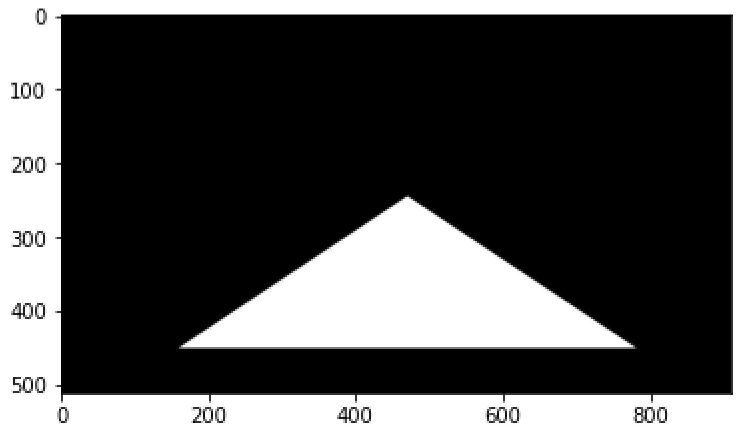


Figure 25. Region of interest 2

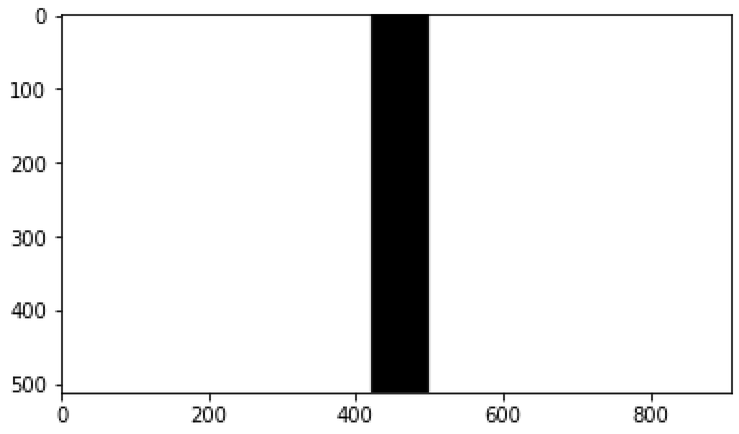


Figure 26. Region of interest 3

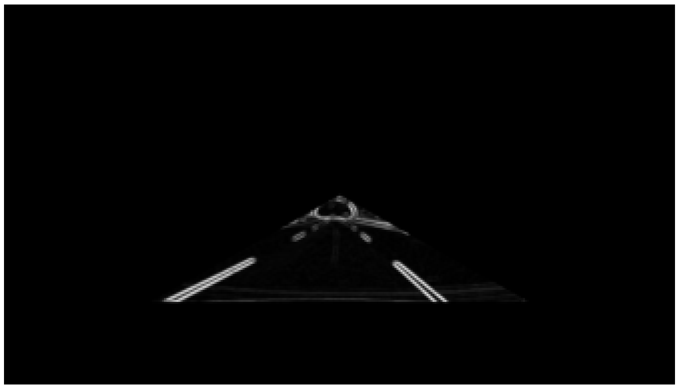


Figure 27. Output of region of interest 1 with edges near vanishing point

$$x_l = \frac{(h + 90 - y) * \frac{w}{2}}{\frac{h + 90}{2}} - 70$$

Equation 17. Left boundary of the second triangular region of interest

$$x_l = \frac{y * \frac{w}{2}}{\frac{h + 90}{2}} + 100$$

Equation 18. Right boundary of the second triangular region of interest

However, even with three masks, they are still in fixed positions and may not work for all cases. In Figure 28, we can clearly see that even with three regions of interest applied, depending on the sample image, a lane marking can still fall within all those regions of interest. Thus, our solution is to apply an algorithm to reduce the number of Hough lines after double peak averaging so that even with extra clusters of Hough lines from the extra lane markings, we can still eliminate those lines and output only the two correct lane lines.

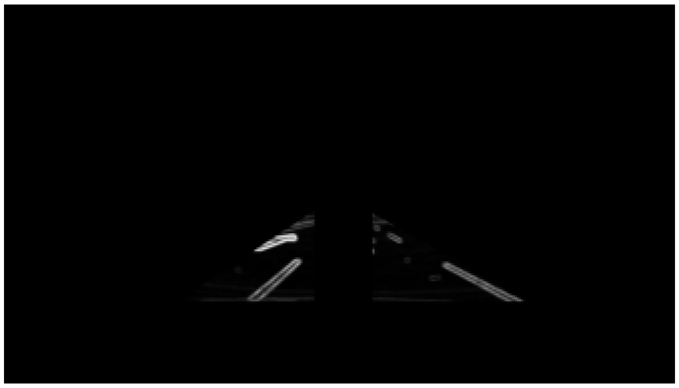


Figure 28. Output of region of interest 3 with the left arrow still detected

Vanishing Point Calculation

The vanishing point is defined as the point where the left and the right lanes converge on the horizon. In Figure 29, the vanishing point is the top corner of the red triangle. Next, the Hough transform output can be represented in a Cartesian plane using Equation 16 so that the lines can be drawn on the inputted images of roads. It should be noted that now, the lines are represented as a Cartesian line as shown in Equation 19 where m is the slope and c is the y-intercept of the line. Our initial approach to calculate the vanishing point is to group the lines into left and right lanes. This can be done by representing the left lanes with lines with negative slopes and right lanes with lines with positive slopes. This relationship can seem inverted because the origin of the image is at the top-left portion of the image while a normal Cartesian plane would generally place the origin at the bottom-left of the image.

$$y = m * x + c$$

Equation 19. Cartesian representation of a line

In addition to the grouping, all lines with slopes between -0.3 and 0.3 are ignored because it is impossible to have lanes that are perpendicular to the direction of the car. Therefore, all lines with slopes less than -0.3 are considered left lanes while all lanes with slopes more than 0.3 are considered right lanes. Next, the slopes and the y-intercepts of all left and right lanes are averaged to get an average left lane and an average right lane represented as Cartesian lines.

The problem with this averaging implementation is due to the heuristic nature of the implementation. This method fails to account for situations where the car could be changing lanes because three different lanes could be inside the region of interest, so the lanes may not be defined merely as left and right. Therefore, other methods of defining lanes were explored and this methodology was done on top of that.

The left and the right lines are inputted into a system of linear equations shown in Equation 20 to calculate the intersection of the two lanes which is also the vanishing point. In the equation, *x* and *y* represent the column index and the row index of the vanishing point, respectively. Moreover, m_l and m_r are the slopes of the left lane and right lane, respectively, while c_l and c_r represent the y-intercept of the left lane and right lane, respectively.

$$\begin{bmatrix} -m_l & 1 \\ -m_r & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_l \\ c_r \end{bmatrix}$$

Equation 20. System of equations for vanishing point calculation

After retrieving the coordinates for the indices for the vanishing point, we simply truncate all parts of the lines above the vanishing point in the image. Figure 29 shows how these calculations clean up our output and clearly trace the lanes on the road.

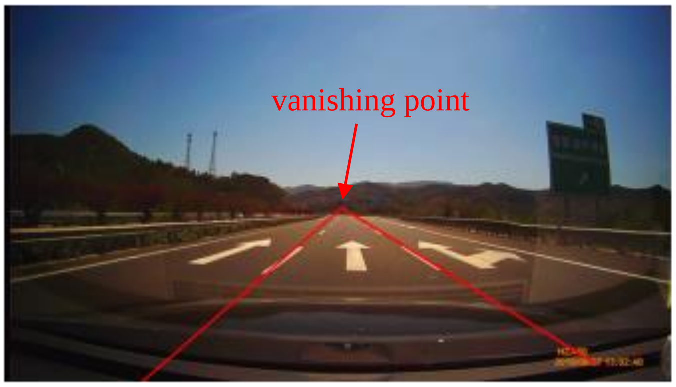


Figure 29. Lane detected image converging at the vanishing point

Lane Color Detection

In most digital images, color data is stored in the RGB space. This means that each pixel has three different values of red, green, and blue instead of just grayscale. Each color is 8-bit, so the value can range from 0 to 255. Because $(2^8)^3$ is about 16.7 million, there are about 16.7 million possible colors.

It is easy to see how some of these combinations of primary colors form different colors. For example, RGB of 255, 0, 0 is red because the red is maxed out and the green and blue are at zero. If all three are at zero, the pixel is black, and if all three are at 255, the image is white. When two of the channels are at 255 and one is at zero, the three secondary colors are formed. When blue is zero, we have red + green = yellow. When green is zero, we have red + blue = magenta. When red is zero, we have green + blue = cyan. Since lanes are either white or yellow, we could theoretically just count the pixels that have RGB of 255, 255, 255 (white) and the pixels that have RGB of 255, 255, 0 (yellow) and see which one has more pixels of that color. However, in practice, most pixels will not be perfectly white or yellow. Adjusting thresholds with three different primary colors is difficult to visualize and predict, so the RGB color space is not the best one for lane color detection.

While RGB is the most common color space, it is not the only one. The HSV color space uses the three parameters of hue, saturation, and value. Hue is the dominant wavelength of color. In layman's terms, it is what we think of when we think of color and the color wheel. Because of this, it is commonly represented as a color wheel ranging from 0 to 360 degrees, with pure red being zero degrees, pure green being 120 degrees, pure blue being 240 degrees, and all the other colors in between filling up the spectrum. Saturation is the brilliance or intensity of color. It ranges from 0 to 1 and commonly uses a percentage scale. A saturation of 100% represents full color, and as the saturation decreases from 100%, it becomes more tinted. At a high brightness, it becomes whitewashed. At a saturation of 0%, the color is entirely on the grayscale spectrum. The last parameter is value. Value represents the lightness or darkness of an image. Like saturation, it ranges from 0 to 1 and commonly uses a percentage scale. At a value of 100%, the pixel is at its lightest, and at a value of 0%, the pixel is black, regardless of the saturation and hue. A saturation of 0% and a value of 100% is white, regardless of the hue. Figure 30 shows the three spectrums with arrows showing how adjusting each parameter affects the color. Because the HSV color space is more intuitive for understanding color thresholding and coding it, we decided to work in the HSV color space for color detection.

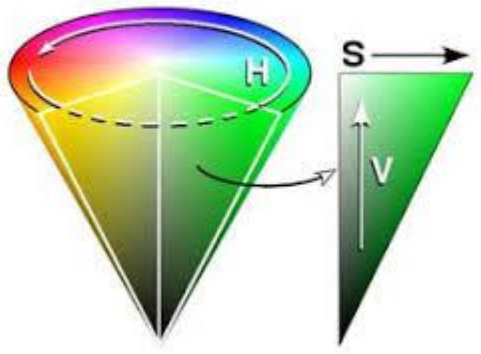


Figure 30. Representation of HSV color space [5]

To begin the process of color detection, the same region of interest from previous sections is used, but this time, the mask is a color image of white and black instead of grayscale white and black. It looks the same, but all the cropped-out pixels now have an RGB value of 0, 0, 0 instead of a grayscale value of 0.

The mask in Figure 32 is applied to the resized image (Figure 31) obtained from preprocessing to obtain the cropped color image (Figure 33).

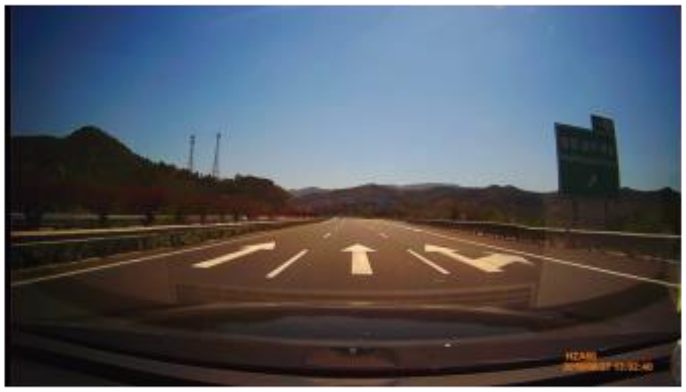


Figure 31. Resized color image

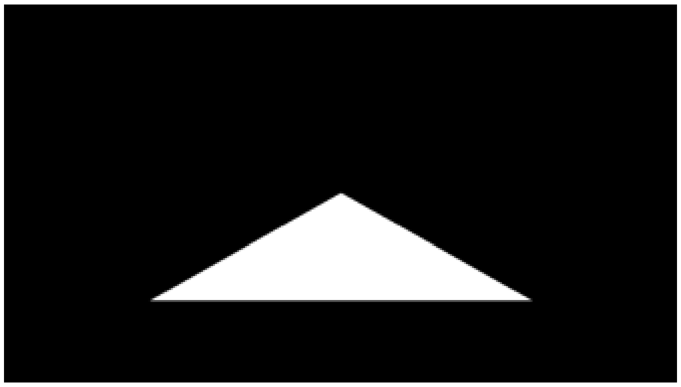


Figure 32. Color mask

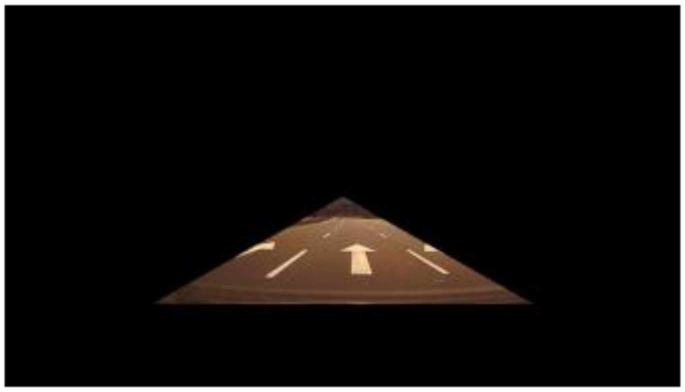


Figure 33. Cropped color image

The next step is to convert the image from the RGB color space to the HSV color space [6]. First, the red, green, and blue values need to be converted from a scale of 0 to 255 to a scale of 0 to 1.

$$R' = \frac{R}{255}$$

Equation 21. Red transformation [6]

$$G' = \frac{G}{255}$$

Equation 22. Green transformation [6]

$$B' = \frac{B}{255}$$

Equation 23. Blue transformation [6]

In Equations 21 through 23, *R*, *G*, and *B* represent the red, green, and blue values, respectively, of a pixel, and *R'*, *G'*, and *B'*, represent the red, green, and blue decimal values, respectively, of a pixel.

The next step is to figure out which of the primary colors is dominant and which are not.

$$C_{max} = \max\,(R', G', B')$$

Equation 24. Most prominent primary color [6]

$$C_{min} = \min\,(R', G', B')$$

Equation 25. Least prominent primary color [6]

$$\Delta = C_{max} - C_{min}$$

Equation 26. Difference in intensity between the most and least prominent primary color in a pixel [6]

Using Equations 24 through 26, we can figure out the hue, saturation, and value.

$$hue = \begin{cases} 0 & \Delta = 0 \\ (60^\circ * \frac{G' - B'}{\Delta}\ )\,mod\,360^\circ & C_{max} = R' \\ (60^\circ * \frac{B' - R'}{\Delta} + 120^\circ)\,mod\,360^\circ & C_{max} = G' \\ (60^\circ * \frac{R' - G'}{\Delta} + 240^\circ)\,mod\,360^\circ & C_{max} = B' \end{cases}$$

Equation 27. Hue calculation [6]

$$saturation = \frac{\Delta}{C_{max}}$$

Equation 28. Saturation calculation [6]

$$value = C_{max}$$

Equation 29. Value calculation [6]

Using Equations 27 through 29, we can extract each of the three channels as shown by the following images.

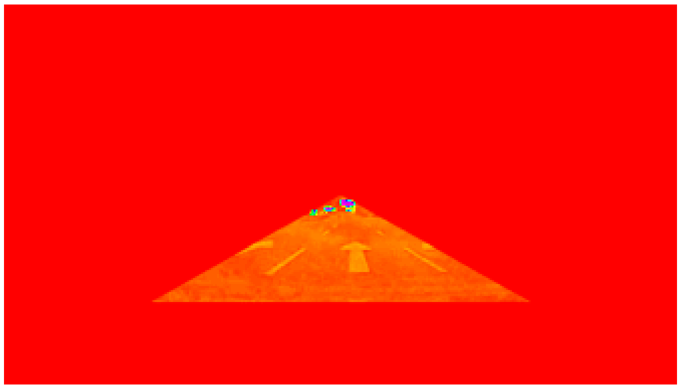


Figure 34. Hue channel

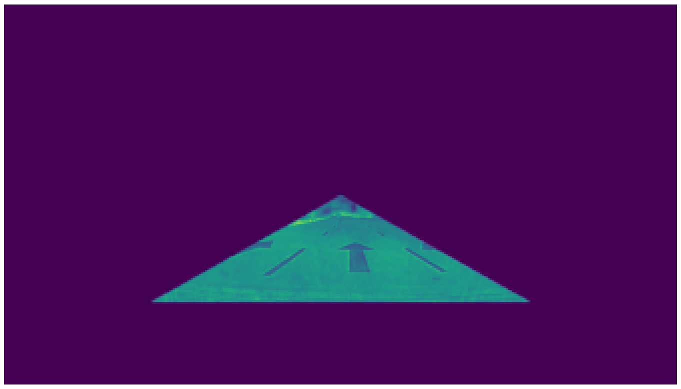


Figure 35. Saturation channel

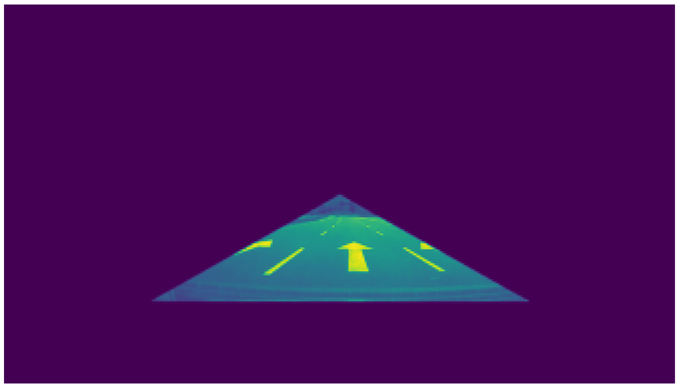


Figure 36. Value channel

In Figure 34, the hue is represented by the actual color if the saturation and value were at 100%. For example, any yellow pixels in the original image show up as yellow in the hue channel. In Figures 35 and 36, a high value or saturation is represented as yellow, and a value or saturation of zero is represented as purple.

The next step is to obtain the actual parameters that make up each color. To do that, histograms are made of each channel. The domain for these histograms includes the pixels that are both in the region of interest and on the lines (plus or minus one pixel to either side of the line) generated after calculating the vanishing point. Each histogram is then analyzed to figure out which values for hue, saturation, and value form what colors. The process was repeated with many different images to get a grasp of good parameters. Since image brightness greatly affects the value histogram, we decided it was best to use a variable value parameter as shown by Equation 30.

$$value\ parameter = argmax(value\ histogram\ of\ entire\ region\ of\ interest) + 0.25$$

Equation 30. Value Parameter calculation

The fixed offset of 0.25 in Equation 30 was obtained through trial and error. The methodology was as follows: The value must be enough to separate the pavement from lane markings, and the pavement will be much darker (lower value) than the markings. In the entire region of interest, the pavement pixels comprise most of the histogram domain, meaning that the value with the largest plurality of pixels is a value that applies to the pavement pixels. Through trial and error, making the value threshold to be 0.25 higher than the value with the most pixels resulted in a clean separation of the dark colors present in pavement and the light colors used in lane markings.

The histograms for the images in Figures 37 through 40 are shown below with a red line signifying the value parameter for that image.

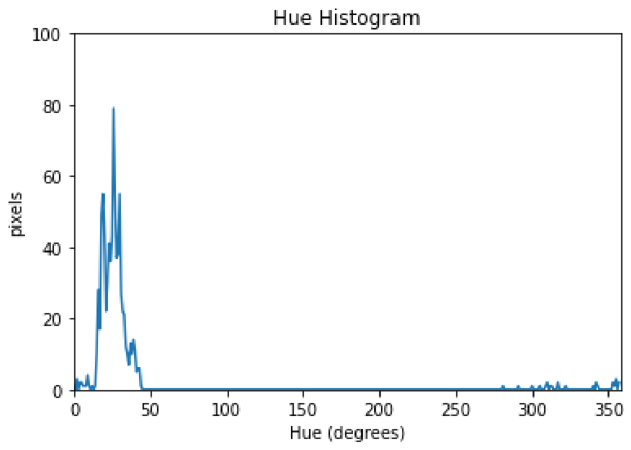


Figure 37. Hue histogram

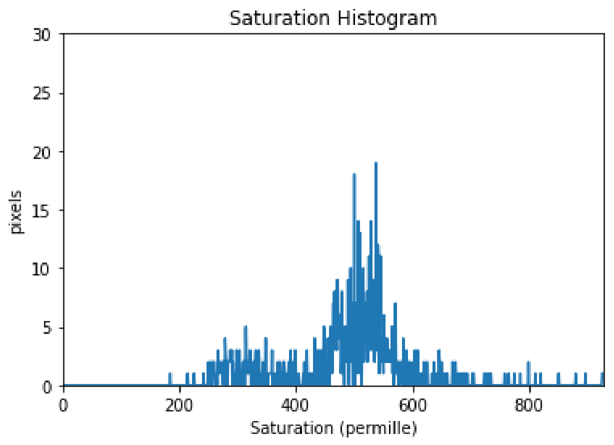


Figure 38. Saturation histogram

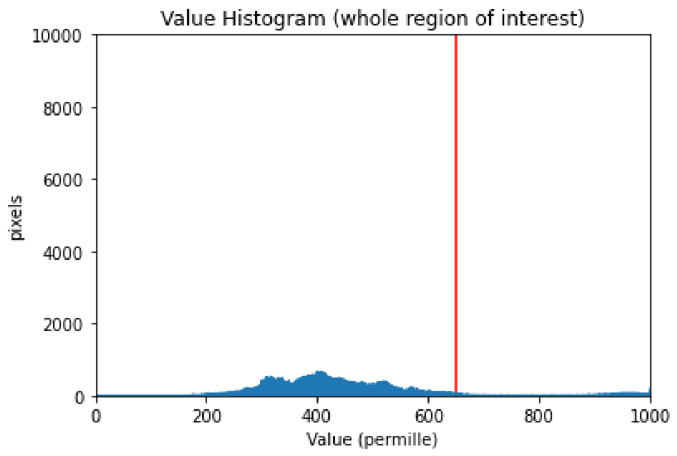


Figure 39. Value histogram for the entire region of interest

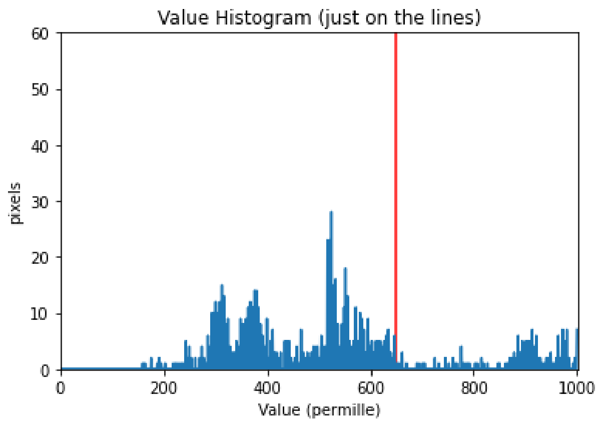


Figure 40. Value histogram for the pixels on the lines

By analyzing these histograms for multiple images, we were able to obtain parameters for color as shown below.

$$color_i = \begin{cases} dark & value < value\,parameter \\ white & value \geq value\,parameter,\ saturation \leq 0.40 \\ red & value \geq value\,parameter,\ saturation > 0.40,\ 0^\circ < hue \leq 30^\circ \\ yellow & value \geq value\,parameter,\ saturation > 0.40,\ 30^\circ < hue \leq 50^\circ \end{cases}$$

Equation 31. Color parameters for white and yellow

Using Equation 31, we find the color of each pixel on one of the lines (plus or minus one pixel above and below the line) and in the region of interest. If there are more white pixels, the lane color is white, and if there are more yellow pixels, the lane color is yellow. This process is repeated for the other lane line.

To verify that the color parameters are correct, we created a new image that converts all white pixels to pure white (RGB of 255, 255, 255) and all yellow pixels to pure yellow (RGB of 255, 255, 0). Any other pixels get turned to black (RGB of 0, 0, 0). We repeated this process with many different images to verify that all white pixels were being picked up as white and all yellow pixels were being picked up as yellow.

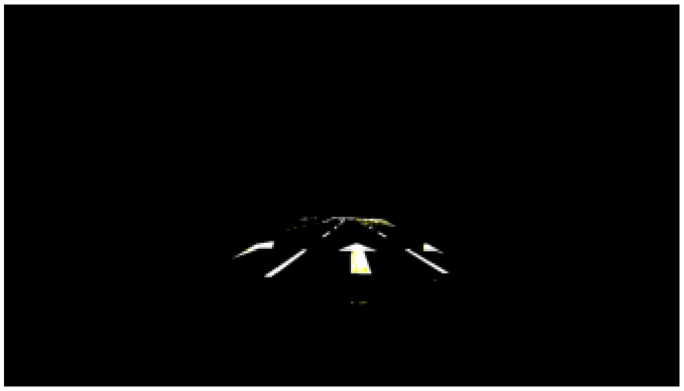


Figure 41. Just the whites and yellows

A few key details to note from the entire color detection process are shown in the figure below.

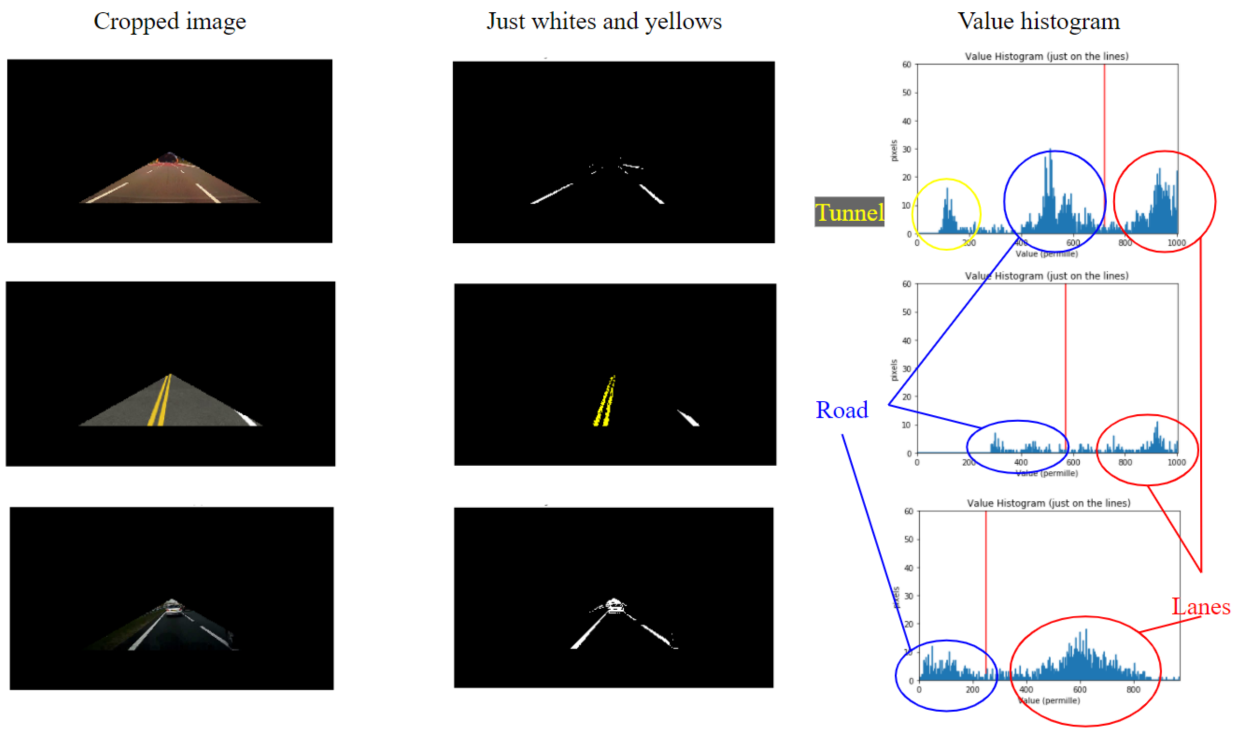


Figure 42. Color verification for three different images

In all three images in Figure 42, the value histogram on the right uses a domain of pixels that are just on the lane lines (including the pixel above and below) and in the region of interest. The red line represents the value parameter for that image. The top image shows how the super dark pixels of the tunnel have low values, the semi-dark pixels of the pavement have values that are still low but not as low, and the white pixels of the lane markings have high values. The value parameter separates that top cluster well as shown by the histogram and the image with just whites and yellows. The second sample image shows that the color parameters are detecting yellow. The third sample image shows why the variable value parameter is needed. This picture is much darker than the other images, so if the value parameter used for the other images were used for this one, it would detect many of the lane marking pixels as dark. The value histogram shows that a darker image means that the entire value histogram is essentially shifted to the left. Both clusters of lane pixels and pavement pixels are separated nicely into two clusters, so the value parameter works.

The current color detection implementation works for most white and yellow lanes on gray and dark-gray pavement. In the future, more testing should ideally be done, especially since not all road types were tested. For example, light-gray colored roads are common in many parts of the world, and green roads are also common (for example in bike lanes).

Vectorization

Modern Computer Architectures can leverage parallelization to increase computation speed. SIMD (Single Instruction, Multiple Data) is one component common in modern CPUs and GPUs used for parallelization, which is used when a single operation needs to be applied on a large dataset. Vectorization is the process of converting data into vectors and taking advantage of SIMD. By utilizing the Numpy library, we were able to vectorize some of our sub-processes where the same operation was applied to the entire image.

To vectorize the grayscale process, the image is sliced into the three channels (red, green, and blue). The channels are multiplied by their respective scalar TU-R BT.709 coefficients and summed directly. The masking process utilized a special elementwise XOR process where the image is preserved where the mask is true and set to 0 where the mask is false. Vectorized thresholding creates a Boolean mask of values above the threshold. Elements in the image are selected using the mask, and the values are set appropriately. Conversion from the RGB color space to the HSV color space was also similarly optimized by vectorization except for hue. To calculate the hue, a list of indices for each primary color was calculated and selected, and hue values of those indices were computed and set.

Using iPython magic commands, we tested the speed of the for-loop version and the vectorized version of our code. Values in Table TEMP are mean values calculated over 100 trials to ensure the most accurate result. Based on the results in Table 1, the vectorized version runs hundreds of times faster on average. In a real-time system such as autonomous driving, minimizing calculation time is needed.

Task	Average time using for loops (ms)	Average time using vectorization (ms)	Factor of speed-up
Grayscale conversion	4.45	3.04	1.463815789
Apply Gray Mask	758	0.874	867.2768879
Apply RGB Mask	494	1.21	408.2644628
Thresholding	294	2.05	143.4146341
RGB to HSV	4050	27.7	146.2093863
		Average Multiplier:	313.3258374

Table 1. Time advantage of vectorization over for loops

Top View Transformation

Our methodology can currently detect straight lanes in front of the car. However, it fails to detect the curvature of lanes because of how ρ and θ can only represent straight lines. To counter this, we decided to change the view of the input image from the front view to a virtual camera view called top view as shown in Figure 43. By doing this, the left and the right lane are now parallel to each other which allows for simpler computations towards detecting curved lanes. Figure 44 shows how a simple image of the front view of the camera can be transformed into a top view image so that the two sides of the lane are parallel to each other.

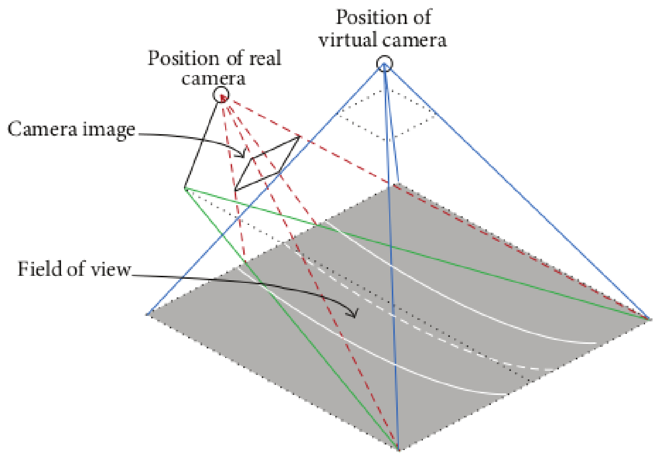


Figure 43.Front view and top view camera angle [7]

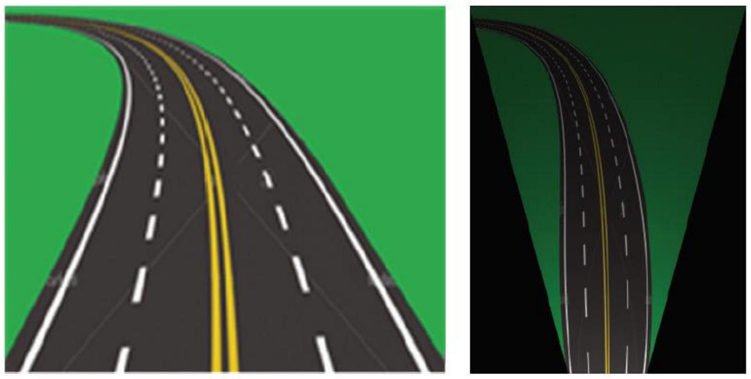


Figure 44.Front view and top view example [7]

To transform a front view image to a top view image, four main parameters are needed: H, α, θ_v, and θ_h. H is the height of the camera location in meters. α is the tilt of the camera angle in degrees. θ_v is the vertical view angle of the input camera in degrees. θ_h is the horizontal view angle of the input camera in degrees. To further understand these input parameters more clearly, Figure 45 shows how each variable is represented in context to the camera location. The top portion of Figure 45 shows the side view of the transformation while the bottom portion of Figure 45 shows the top view of the transformation.

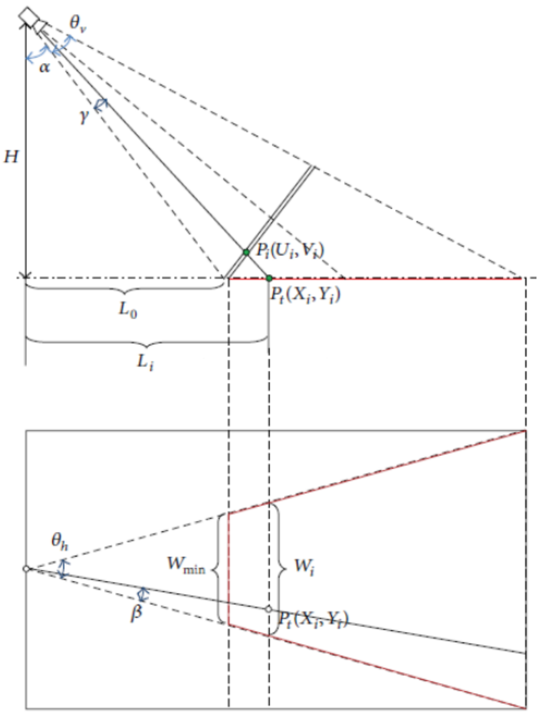


Figure 45. Mapping variable representations [7]

Before mapping, however, we must first convert *H* to pixels instead of meters so that *H* becomes H_{pixel}. To do this, coefficient *K* is calculated using Equation 32 through 34, which are intermediate steps to calculate *K*. *V* is the width of the front view image. L_{min} is the distance from the location of the camera to the point in the front view image closest to the camera in meters, and W_{min} is the minimum width of the top view image in meters. Next, *K* is multiplied by *H* as shown in Equation 35 to get the value of H_{pixel}.

$$L_{min} = H * tan(\alpha)$$

Equation 32. Distance from the camera to the part in the image closest to the camera in meters [7]

$$W_{min} = 2 * L_{min} * tan\left(\frac{\theta_h}{2}\right)$$

Equation 33. Minimum width of the top view image [7]

$$K = \frac{V}{W_{min}}$$

Equation 34. Coefficient to convert from H to H_{pixel} [7]

$$H_{pixel} = H * K$$

Equation 35. Conversion from H to H_{pixel} [7]

After getting H_{pixel}, we can now begin mapping each pixel value of the front view to the top view. To do this, we define U_i and V_i as the row index and column index, respectively, of pixel *i* in the front view image. We then define x_i and y_i as the row index and column index, respectively, of pixel *i* in the top view image. Therefore, for every pixel *i* in the top view image, which is represented with U_i and V_i, we can map that specific pixel with x_i and y_i to form a top view image. We used Equations 36 through 39 to calculate the value of x_i.

$$\gamma = \theta_v * \left(\frac{U - U_i}{U}\right)$$

Equation 36. Vertical angle between the bottom of the image to pixel i [7]

$$L_i = H_{pixel} * tan(\alpha + \gamma)$$

Equation 37. Distance from the camera to the part in the image closest to the camera in pixels [7]

$$L_0 = H_{pixel} * tan(\alpha)$$

Equation 38. Distance from the camera to the location at pixel i [7]

$$x_i = L_i - L_0$$

Equation 39. Row index of pixel i in the top view image [7]

Next, to calculate y_i, the column index of the pixel *i* in the top view image, Equation 40 through 41 is used. Here, β is the horizontal angle between the right side of the image to pixel *i*.

$$\beta = \theta_h * \left(\frac{V - V_i}{V}\right)$$

Equation 40. Horizontal angle between the right side of the image to pixel i [7]

$$y_i = L_i * tan(\theta_h - \beta)$$

Equation 41. Column index of pixel i in the top view image [7]

Now that we can convert from the front view of the image to the top view of the image, Figure 46 and 47 show the result of our implementation. Here, we cropped our input image because the camera angle of our dataset is looking straight ahead, causing too much of the sky to be included in the image. Because of this, the transformed image looks quite similar to the front view image. To counter this, we cropped portions of the input image so that the road makes up most of the input image.

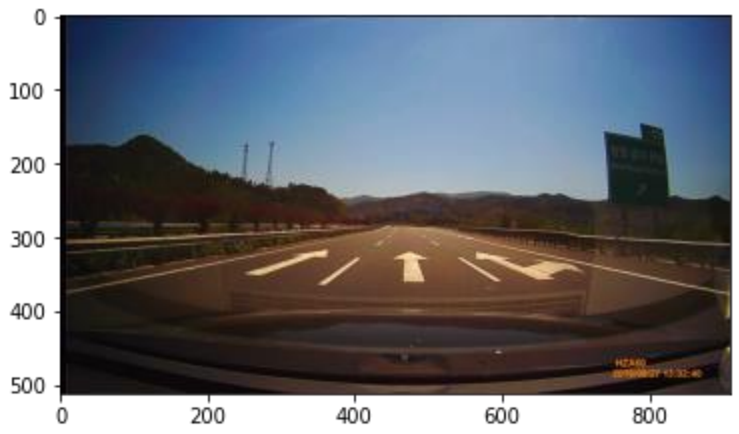


Figure 46. Mapping variable representations

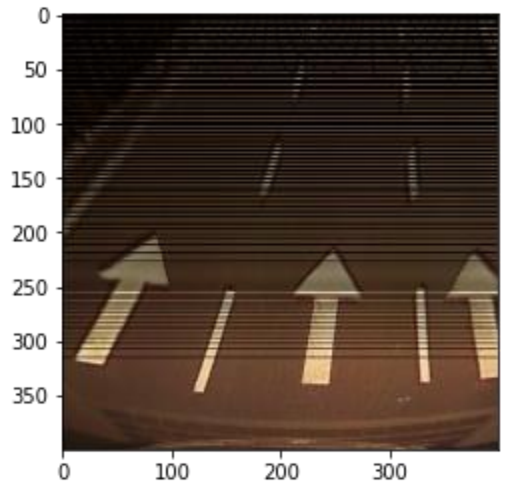
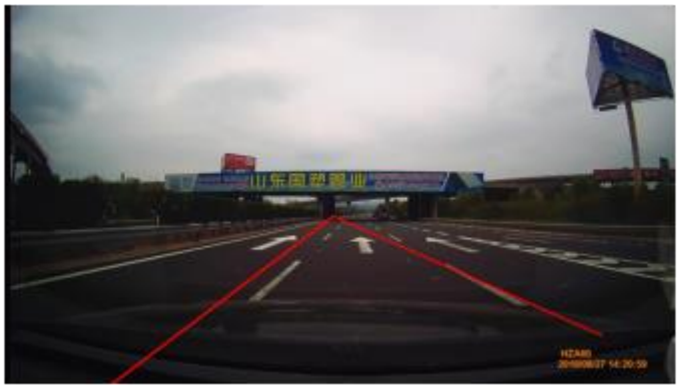
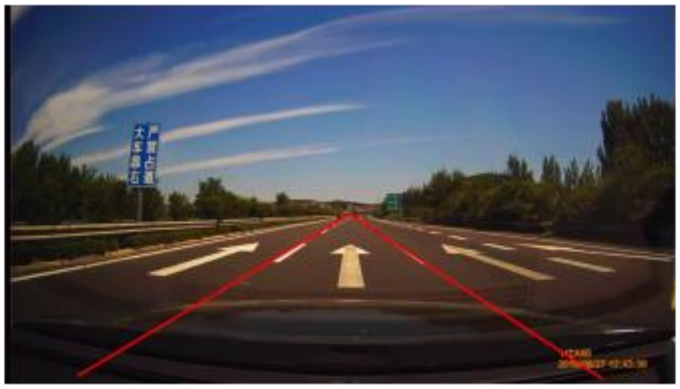
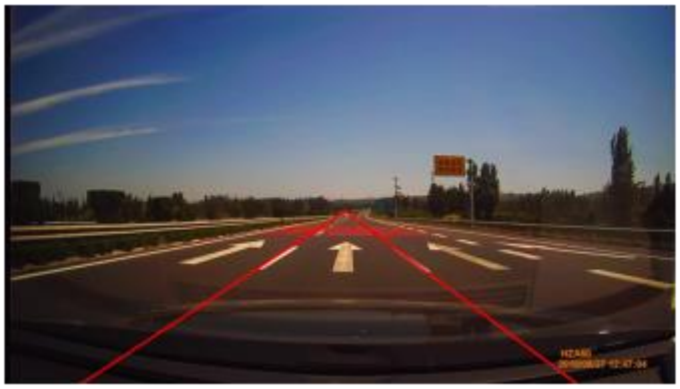
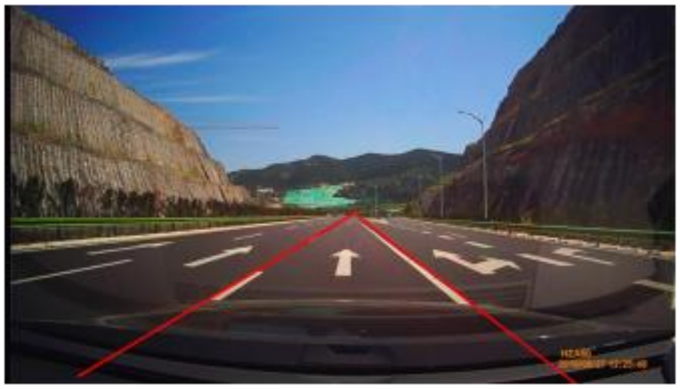
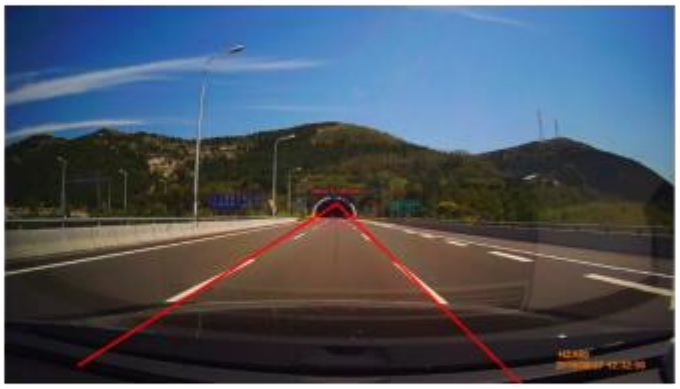
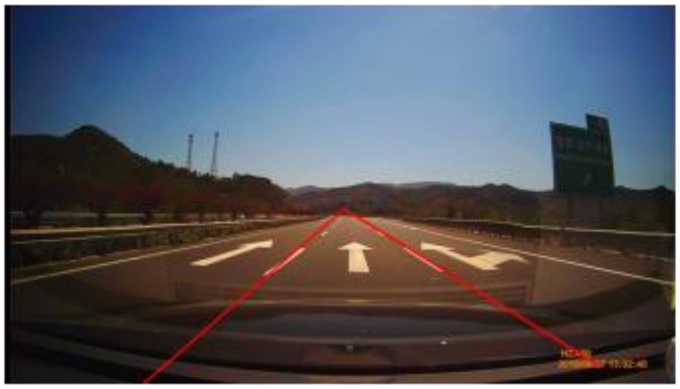
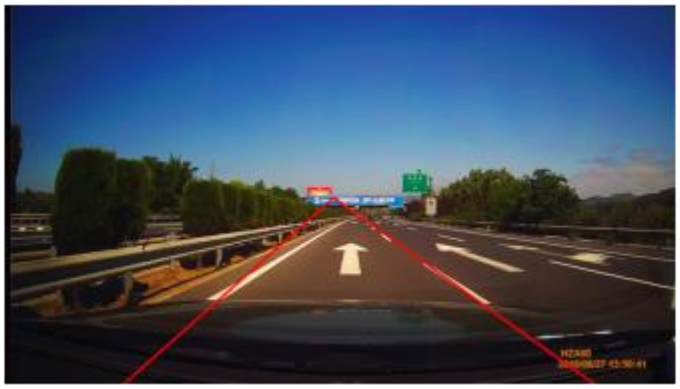


Figure 47. Mapping variable representations

In Figure 47, we can see that the top view image is stretched with many horizontal black lines in the image. We suspect that this is because the mapping between the front view image and the top view image is not 1-to-1. Because of this, different pixels in the front view image could be mapping to the same pixel in the top view image. Because, initially, our front view image is a matrix with all pixel values set to 0, the dark pixels are pixels where none of the pixels in the front view image maps to.

Evaluation of Results

Detected Lanes	Detected Colors (left/right)
	white/white
	white/white
	white/white

	white/white
	white/white
	white/white
	white/white

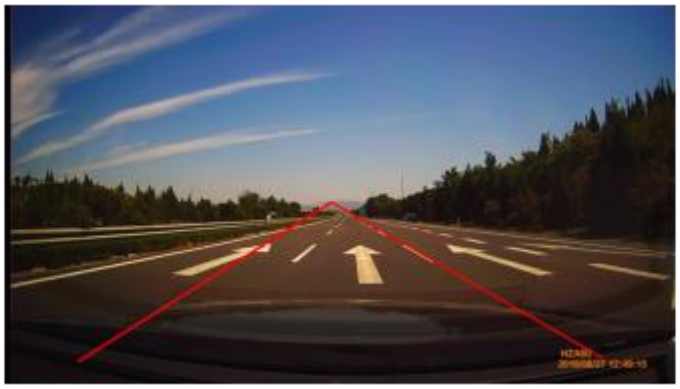
	white/white

Table 2. Results of implementation

Future Work

We spent a significant amount of time this semester learning about the basic image processing techniques such as Hough transform and Otsu's thresholding. The result that we have obtained can detect straight lanes in perfect to moderate lighting conditions. While this works in most of the situations a car is expected to be in, assumptions were made which prevented it from performing well in some situations. We assumed that the road is always black or dark gray, the only possible lane colors are white and yellow, the lanes are always straight, the number of lane lines detected is limited to 2, the car is in the middle of the lane, and that there is a considerable amount of space in front of the car.

Further improvement needs to be done on this process to increase its usability. The next steps would be to add support for curved lanes and better detection in worse lighting conditions. Adding support for curved lanes might mean using a method other than Hough transform or adding another method on top of it. In the project goals presented at the beginning of the semester, we wanted to have the lane information converted to equations which would then be passed to other systems in the autonomous driving system. We were not able to reach that part of the project yet, but it is something that could be worked on in the future.

Conclusion

From the results shown above, we see that our current implementation can detect lanes relatively accurately. Our method shows that we can detect lanes under favorable conditions without expensive computational resources. However, our current implementation fails when weather and lighting conditions change the visibility of lanes. Such an example would be lanes covered in snow where the road would have a similar color to the lanes, preventing the edge detector to detect the lanes themselves. Nonetheless, our team was able to develop a resource-efficient lane detection method that utilizes well-known algorithms such as the Sobel edge detector, Otsu's threshold, and Hough transform.

References

[1] S. Feng, J. Wang, D. Zhu and C. Yuan, "Jiqing Expressway Dataset," Shandong, 2019.

[2] E. Delp, "Image Processing Introduction," Purdue University, West Lafayette, 2020.

[3] N. Otsu, "A Threshold Selection Method from Gray-Level Histogram," *IEEE Transactions on Systems, Man, and Cybernetics,* February 1979.

[4] V. Chatzis and I. Pitas, "Fuzzy Cell Hough Transform For Curve Detection," *Pattern Recognition,* pp. 2031-2042, 4 February 1997.

[5] Carmody, "Portland State University," [Online]. Available: http://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Carmody_Visual&ColorModels.pdf. [Accessed 23 November 2020].

[6] "RapidTables," [Online]. Available: https://www.rapidtables.com/convert/color/rgb-to-hsv.html. [Accessed 23 October 2020].

[7] B. Dorj and D. J. Lee, "A Precise Lane Detection Algorithm Based on Top View Image Transformation and Least-Square Approaches," *Journal of Sensors,* 2016.