Lane Detection and Distance Estimation With a Monocular Camera

by

Biram Bawo and Moses Chuka Ebere

A report presented in fulfilment of the
EE 417 Computer Vision Term Project requirement

Course Instructor: Prof. Dr. Mustafa Ünel

Faculty of Engineering and Natural Sciences
Department of Mechatronics Engineering
January 12, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing demand for fully autonomous vehicles has led scientists and inventors to come up with hardware and software architectures to ensure safe and sustainable autonomous driving experiences. In the latter half of the twentieth century, manufacturers accentuated the need for more sophisticated hardware solutions; however, within the last few years, the focus has shifted to cutting-edge software technologies to foster an end-to-end driverless vehicle experience. These technologies have so far sought to unpack what we have come to know about driving in a bid to automate a spectrum of driving tasks - from the most trivial to the most complex. A prime example of such a task would be perfectly aligned vehicle parking which now has numerous Park Assist System solutions from renowned manufacturers. Beyond this, other complex cases surface while a vehicle is in transit. Of all the complexities that researchers seek to control and automate in a moving vehicle, lane detection, and localization estimation emerge as fundamental challenges. In fact, data collated and analyzed by the US Insurance Institute for Highway Safety suggests that the systems that tackle the aforementioned challenges have had a direct impact on reducing crash statistics [1]. Companies working on solving the Full Self-Driving problem (FSD) are making giant strides by using sophisticated neural network architectures to perform detection, segmentation and depth estimation, all from the same model [2]. However, such models require outrageous amounts of computational power to run, and this makes it difficult for constrained start-ups to catch up in the race to solving FSD; hence, the motivation behind this project.

Chapter 2

Problem Definition

In this project, we estimate distances between objects in front of a moving vehicle and also detect lanes on which the vehicle runs. This first requires calibrating the camera of the vehicle to ascertain its intrinsic and extrinsic parameters. The overarching goal is to address the cardinal challenges of lane detection and distance estimation using fast and robust algorithms that do not require a lot of computing power and can run on edge devices like micro-controllers (a Raspberry Pi in this case).

To properly develop a holistic understanding of the project at hand, it is imperative to start this study with a clear and concise definition of a lane. A lane is a region bounded by two-line features upon which single-way traffic is allowed to flow. For the sake of simplicity, the lanes referred to in this paper would be clearly defined by white lines on the freeway.

Over the years, lane detection has become increasingly important due to the gradual transition from manufacturing traditional automobiles that require little to no level of automation to the research and development of fully autonomous vehicles. According to the US Department of Transportation's National Highway Traffic Safety Administration (NHTSA), there are six automation levels (from level 0 to level 5), of which levels 1 through 5 require an Advanced Driver Assistance System (ADAS) or an Automated Driving System (ADS) - in higher levels [3]. Lane detection finds its application in a plethora of areas across the above-mentioned automation spectrum, especially in levels 3, 4, and 5.

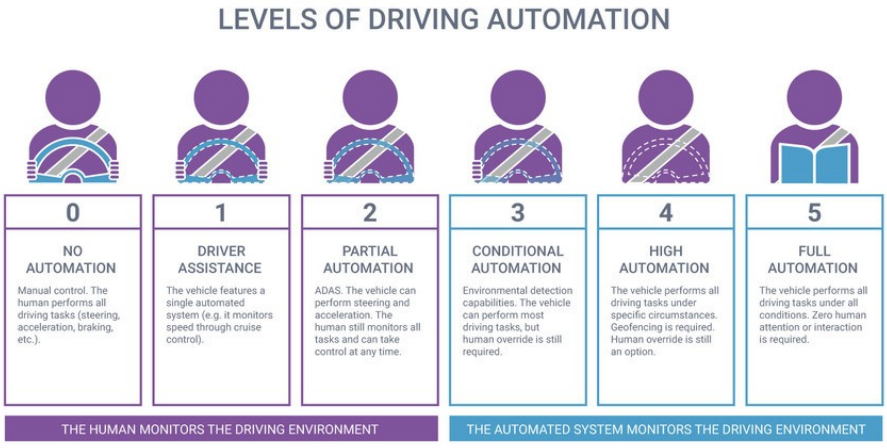


Figure 2.1: SAE Automation Levels [4]

The process of lane detection could be viewed as a "calibration process" for autonomous vehicles in the sense that a crisp identification of the lanes on a motorway is a key precursor for the decision-making process of automated driving systems. A poignant example of this is the case of Simultaneous Localization and Mapping (SLAM) where detected lanes could serve as landmarks for the SLAM algorithm [5], and the alignment of the vehicle with the lane(s) could be used for accurate pose estimation [6]. Equipped

with such information amongst other sensor measurements, a near-pristine map of the road network could be constructed. The authors of [7] identified three broad categories of lane detection algorithms namely, model-based algorithms, feature-based algorithms, and learning-based algorithms. The vital importance of robust lane detection algorithms has been underscored by a host of researchers over the years. However, the likes of [8] indicated that the constraint on computational resources is a limiting factor for achieving real-time lane detection. In the foregoing sections, we implement a feature-based algorithm that achieves near real-time detection without being computationally demanding.

A system underpinned by a sophisticated lane detection algorithm can only go a long way in solving the complex problem of fully autonomous driving. This is why distance estimation is also considered a primary issue of concern in this discourse. In SAE level 0 where the driver is in complete control of the car, deciphering the distance from the car to oncoming traffic or other obstacles is a task that is completely dependent on the strength of the human eyes. In this case, perception plays an important role in estimating distance. However, as we go higher on the autonomy scale, the goal is to use plausible solutions in tackling this problem; therefore, it is imperative that the solution in question not only performs like the human eye but also offers much more flexibility and robustness. More technically and generally referred to as localization estimation, the challenge of distance estimation has been approached differently by researchers across the years. The upshot of this is a dichotomy of solutions proposed in the literature - active solutions and passive solutions. While the active solutions explore the use of sensors (like LIDAR and ultrasonic-based sensors, to name a few) that inevitably emit energy to the environment to measure some response or reaction based on the age-old time-of-flight principle, passive solutions employ non-energy-emitting sensors [9]. For a system constrained to implementing only computer vision solutions (which are passive solutions), a stereo set of cameras could be used in the ego vehicle. This allows a relatively easy elimination of depth ambiguity - a major computer vision problem - through the knowledge of the disparity between simultaneously taken image/video frames of the different cameras involved. Using a monocular camera (like in our case), on the other hand, proves to be a more daunting approach to solving the distance estimation problem. For this reason, reams of publications have been released that postulate different methods (some of which are novel) to approach this. Some of these methods include but are not limited to, road geometry-based methods, object geometry and orientation-based methods [10], deep learning methods, etc. On the grand scale, depth estimation in vehicles is an integral part of different computer architectures for autonomous driving [11] like emergency braking systems, lane change assistance systems, path planning, collision avoidance systems, Dynamic High Beam (DHB) assist systems, etc.

2.1 Problem Formulation

2.1.1 Object Detection and Distance Estimation

Object detection involves finding the presence of an object in an image and localizing the object with a bounding box. Distance estimation, on the other hand, is the process of estimating the distance between an object in the real-world and the optical center of the camera. As of today, several object detection algorithms exist, some of which use classical computer vision techniques [12] and others use methods involving machine Learning [13, 14]. We approach the problem by trying out both methods before discussing the downsides of each approach.

The proposed distance estimation method in [10] makes use of geometric representations, and we adapt this approach to estimating distances between a target object and the camera. Using [10] requires knowing the intrinsic camera parameters, and for that reason, we have to calibrate our camera first.

2.1.2 Lane Detection

As stated in the problem definition, the approach we take to this task is purely from a computer vision point of view. We seek to detect lane markings on a freeway using image features with the following key characteristics:

1. Local: these features can be observed around neighborhood pixels. In the implementation of the algorithm, a region of interest will be "isolated" within the image frame under consideration.
2. Meaningful: the features must be proper representations of real aspects of the real scene. Lines would directly translate to lanes within the region of interest.
3. Detectable: based on the above 2 points, the features must be computationally detectable using algorithms. We make use of the probabilistic Hough Line Transform function in OpenCV for this.

Given an image of a road taken from a camera mounted on the dashboard of an ego vehicle, there are key factors and assumptions to be mindful of in formulating the lane detection algorithm:

Figure 2.2: A sample image from the KITTI Road Dataset [15]

1. For a moving vehicle on a freeway, there are two white lines (lane markings) on either side.
2. The camera is mounted in the car such that it is between the two white lines (preferably at the center).
3. Due to perspective projection, the lanes meet at the horizon.
4. Following the above, the region of interest is usually chosen in such a way that it excludes every detail outside the lanes. This implies that we end up with a triangular area, which could be extended to an n-sided polygon for convenience.

Since the feature of interest is lane marking, the seemingly arduous detection problem becomes a case of line detection.

2.1.3 Line Detection Using Hough Transform

The Hough Transform, which was initially introduced by Paul Hough in 1959, was based on the notion that a straight line with the general equation in (2.1) could be mapped to a point represented by its slope and intersect parameters in a different coordinate space [16].

$$y = mx + c \tag{2.1}$$

Figure 2.3: A sample image from the CULane Dataset [17]

This method, however, was limited in the sense that it could not accommodate special cases like when vertical lines (with infinite slopes) are to be detected (For instance, the lane in Figure 2.3 - assuming it is perfectly vertical - would not be detectable using the slope-intercept parameter space). This translated to an unbounded parameter space [18] which made discretization impossible. To remedy this problem,

R. Duda and P. Hart in their 1972 paper suggested that the polar equation of a line in (2.2) be used such that the parameters of interest become the orientation of the line and its perpendicular distance from the origin [19].

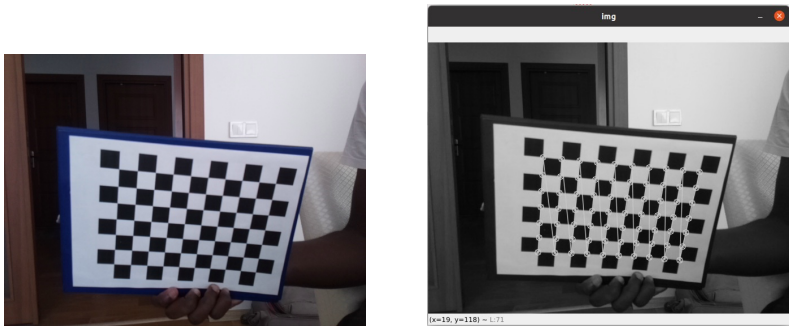
$$\rho = xcos\theta + ysin\theta \tag{2.2}$$

Going by the rho-theta parameterization, lines in the image space would map to sinusoids in the parameter space. The cornerstone idea behind this is that we count the number of sinusoids that intersect at the same point in the accumulator space, and the line(s) of best fit is/are chosen as the line(s) with cumulative intersections that exceed a user-defined threshold.

Chapter 3

Methods

3.1 Camera Calibration



(a) Calibration Grid Map (b) Image corners Map

Figure 3.1: Camera Calibration

We used Zhang's method [20] to calibrate our camera model. This was because of the ease and flexibility it provides in calibrating a camera. Rather than two or three orthogonal planar views, the method only requires a single planar pattern from a few (at least two) different orientations. Either the camera or the plane can be moved to generate a few different orientations of the image. Given a camera model, our goal here is to learn the camera matrix.

$$C = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

3.2 Distance Estimation

In estimating the distance between a target object and the camera, we employ the technique used by Chu, Ji, Guo, Li and Wang [10]. Their method adapted a geometry model of detecting distance to an object using monocular vision as shown in Figure 3.2.

If P is a point on the target object, h is the height from the optical center of the camera to ground, (x_0, y_0) is the intersection of the principal axis and the image plane, (x, y) are the coordinates, α is the camera tilt angle, then the horizontal distance d of the target object from the camera can be estimated

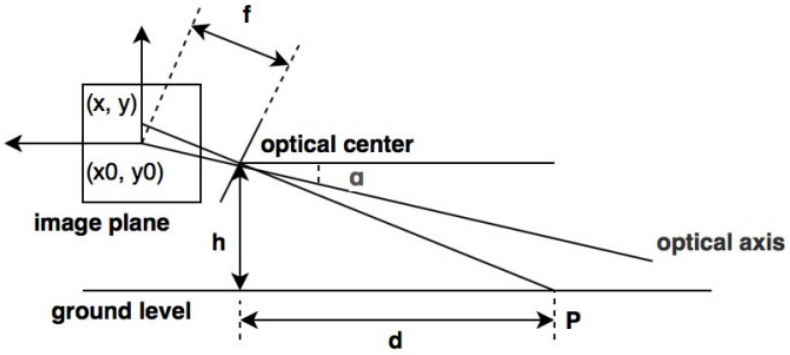


Figure 3.2: Geometry Method for Distance Estimation

by:

$$d = \frac{h}{tan(\alpha + arctan(\frac{(y-y_0)}{f}))} \tag{3.1}$$

$$u = \frac{x}{dx} + u_0 \quad v = \frac{y}{dy} + y_0 \tag{3.2}$$

Let $x_0 = y_0 = 0$. Then

$$d = \frac{h}{tan(\alpha + arctan(\frac{(v-v_0)}{a_y}))} \tag{3.3}$$

where $a_y = f/dy$

The parameters a_y and v_0 are obtained from the camera calibration matrix

$$C = \begin{bmatrix} a_x = f_x & 0 & u_0 = c_x \\ 0 & a_y = f_y & v_0 = c_y \\ 0 & 0 & 1 \end{bmatrix}$$

The height of the center of the camera to the ground, h, was measured manually, and the coordinate of the target object in the image framed was passed from each image frame.

But in order to be able to apply the above formula, we must first detect our target object from the camera with its bounding box. For this object detection phase, we tried two methods: Object detection using contours and Haar feature-based cascade classifier method.

3.2.1 Contour-based Object Detection

OpenCV makes it really easy to find contours in an object using the findContours() and drawContours() functions. The contours in our images were found by:
1. Converting the input image to grayscale
2. Applying a 7x7 Gaussian filter to remove noise and blur the image
3. Detecting edges using Canny Edge detector
4. Dilating the image to individual elements and join boundary points of objects in the image
5. We then apply OpenCV's findContours() and drawContours() functions to get the countours in the image. We set a certain threshold of the area of detected objects to draw the bounding boxes.

3.2.2 Haar Feature-Based Cascade Classifiers Method

We trained a Haar cascade classifier to detect and return the bounding boxes of our target object which in our case in a traffic sign as shown in Figure 3.3. The training data consisted of positive and negative data. The positive training data contains 100 cropped images of the target object we want to detect, whereas the negative data consists of 152 images of anything other that our target images.

(a) Positive Image Sample Map

(b) Negative Image Sample Map

Figure 3.3: Sample Training Data for Haar Cascade Training

3.3 Lane Detection

The flowchart in Figure 3.4 summarizes the steps involved in the lane detection algorithm.

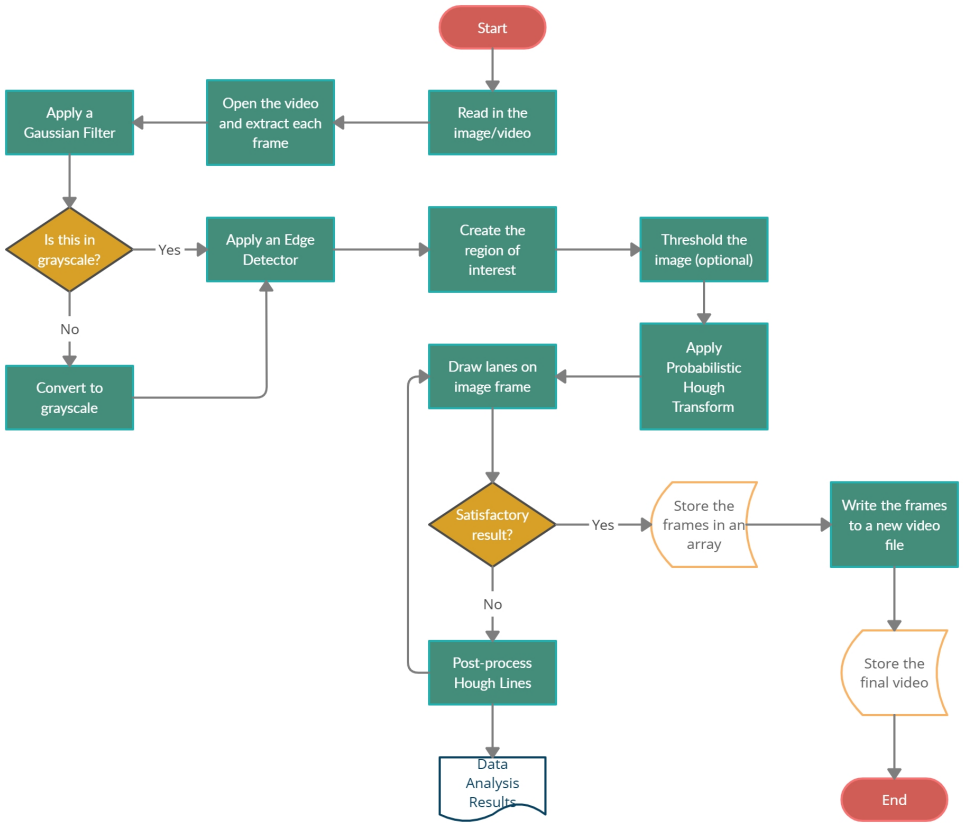


Figure 3.4: Flowchart for Lane Detection

3.3.1 Image Smoothing

Since we are unsure about the exact distribution of the noise in each image frame, it is not out of place to assume that the noise is normally distributed and apply a Gaussian filter. We apply this filter because it is based on a simple linear convolution and it prevents the further propagation of noise through the image.

$$L(x, y; \sigma) = [I * G_\sigma](x, y) \tag{3.4}$$

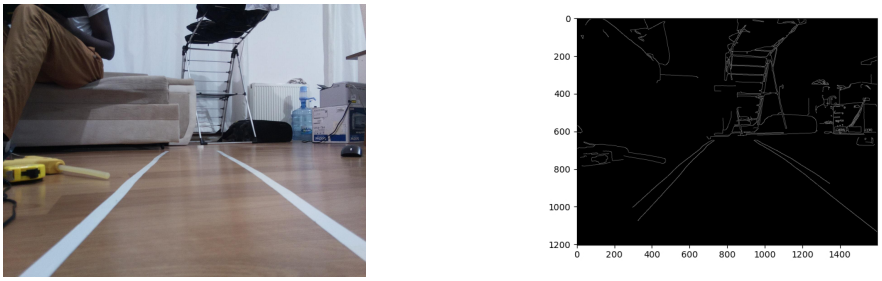
Prior to applying the Gaussian filter, the image is converted to grayscale to avoid having to deal with multiple image channels simultaneously. This significantly reduces the computational time.

3.3.2 Edge Detection

The Edge detection algorithm of choice is the Canny Edge Detector [21]. We opted for this optimal edge detector because of how well it manages noise smoothing, edge enhancement, and edge localization. More significantly, the Canny algorithm allows us to implement **Hysteresis Thresholding** to eliminate both false positives and false negatives.

3.3.3 Region of Interest

Observe the edge image (Figure 3.5(b)) of a make-shift lane for a Raspberry Pi-based wheeled robot.



(a) RGB Image

(b) Edge Image (Using Canny Edge Detector)

Figure 3.5: Determining the Region of Interest

It is apparent that any attempt to detect the lanes (without any machine learning implementation) would be directly affected by the background edges in the image. This necessitates the identification of a region of interest. For simplicity, the camera is assumed to be mounted at the center of the dashboard of a car almost centrally located between two white lane markings. This makes it easier to estimate the location of the region of interest.

3.3.4 Apply Probabilistic Hough Transform and Implement Data Analysis to Find the Line of Best Fit

We chose the probabilistic Hough Transform over the standard one because it returns the coordinates (x,y) of the line, not the parameters (ρ,θ) from (2.2). This detects multiple Hough lines for each lane marking. Now, we apply some data analysis techniques to determine the line of best fit for the final output lanes.

3.3.5 Data Analysis

We make use of the convenient Pandas library in Python for this process [22]. Pandas is a sophisticated library that offers myriads of features that could be leveraged for data manipulation. Within this library, we use the "DataFrame" object which is ideal for indexing.

Data frames were created for all the detected Hough lines in two groups - lines with negative slopes (representing the left lane) and lines with positive slopes (representing the right lane). Statistical measures of central tendency and dispersion (mean, median, interquartile range, skewness) and visualization methods (Box plot, scatter plot) were applied to identify and eliminate outliers.

Chapter 4

Implementation & Results

The intrinsic parameters of our camera module was found using OpenCV:

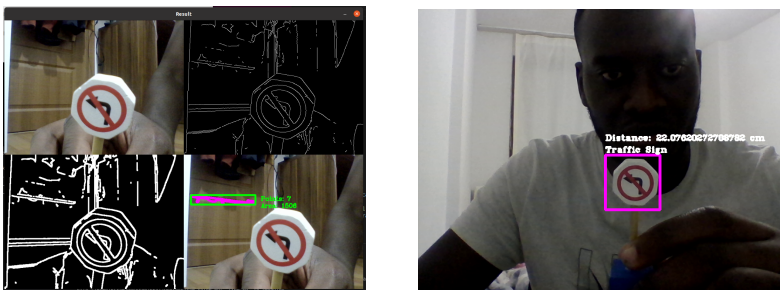
$$C = \begin{bmatrix} a_x = 494.255 & 0 & u_0 = 313.871 \\ 0 & a_y = 494.66 & v_0 = 240.683 \\ 0 & 0 & 1 \end{bmatrix}$$

The determined parameters were used to estimate distance of objects in front of the camera, with objects and lanes detected using with the proposed methods tried and tested on the dataset we created (and other widely available road datasets). The results obtained are shown and discussed in the following sections.

4.1 Object Detection

The first sub-image in Figure 4.1(a) shows the object we want to detect using contours[12]. The second sub-image in the same figure shows the edges detected using the Canny Edge Detector, and we can see the edges clearly highlighted both for the object we are interested in and that of objects in the background. The third sub-image (bottom-left) shows the dilated image of detected edges which kind of neutralizes the weak edges and highlights the strong edges. Finally, when we look at the output with bounding box of detected object, we see that the edges of background objects are very strong, and this is where we get the bounding box, rather than on our target object in front.

Figure 4.1(b) shows the result of using Haar cascades[23, 24] to detect objects. It is observed that desired object was accurately detected within a distance range of 10 to 120cm.



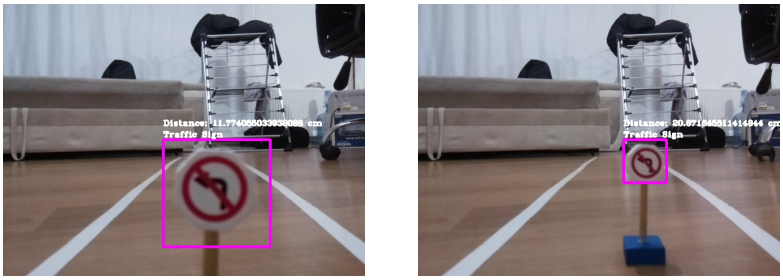
(a) Contour-based object Detection Map

(b) Haar Feature-based Object Detection Map

Figure 4.1: Object Detection

4.2 Distance Estimation

The method proposed by [10] shows great promise in estimating the distance of objects in front of a camera. The required parameters, which were obtained from the camera calibration matrix and manual measuring of the height and camera tilt angle, would inevitably include some degree of error; however, we observe that the error is not so large as to overly skew the data relative to actual measurements. Furthermore, with the real measurements known as in Table 4.1, we can always map the camera measurements to the actual measurements to get estimates that are as close to the real ones as possible.



(a) Camera Distance = 11.7cm, Actual Distance = 10cm Map

(b) Camera Distance = 20.87cm, Actual Distance = 20cm Map

Figure 4.2: Distance Estimation

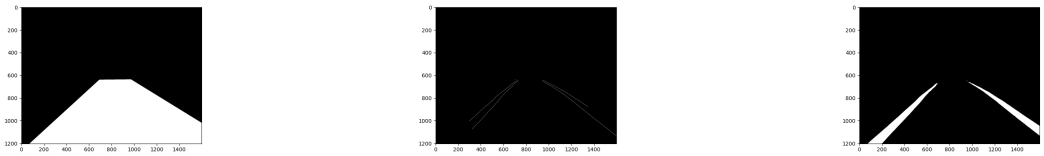
Order	Actual Distance (cm)	Camera Distance (cm)
1	10	11.7
2	20	20.87
3	25	26.0
4	30	29.0
5	40	43.3
6	50	47.1

Table 4.1: Real Measurements and Measurements from Camera

The results of Table 4.1 show the real and camera measurements of distances of objects from the camera center. It is observed that objects that are closer to the camera have more accurate measurement results than ones that are farther away. Thus, in addition to sensor measurements, the method proposed by [10] can be used by real world autonomous vehicles for obstacle avoidance when objects come too close to a moving vehicle.

4.3 Lane Detection

From the flowchart in 3.4, after obtaining the edge image of the reference frame (3.5(a)), we end up with 3.5(b). The subsequent implementations to detect the lanes are as follows:



(a) Image Mask

(b) Region of Interest

(c) Threshold Result (without edge detection)

Figure 4.3: Extracting Region of Interest

A mask with the same resolution as the original frame is created (Figure 4.3(a)). This is used to cut out the region of interest by the **logical bitwise AND operation** (Result: Figure 4.3(b)). This is an element-by-element binary operation where a 1 is returned if and only if 1 is operated on by another 1. Another approach would be to simply threshold the region of interest (as opposed to finding the edges) as seen in Figure 4.3(c). The downside of this approach is that a manually defined threshold is not adaptive; hence, it may fail under different conditions.

After applying the probabilistic Hough Transform, the detected lines are shown in Figure 4.4(a). To obtain clearly detected lanes, it is imperative to reduce the number of lines to the line of best fit on either side. This is done by a post-processing procedure that involves performing *data analysis* on all the lines detected to choose the optimal one.

(a) Detected Hough Lines

(b) Multiple False Positives

Figure 4.4: Detecting Hough Lines for Lane Detection

From Figure 4.4(a), it is clear that randomly choosing one of the detected lines may not necessarily yield an ideal outcome. In some extreme cases, some detected lines may be considered as outliers when compared to the others. This is the case in Figure 4.4(b); hence, we would need clever methods to get rid of the outliers. For instance, the horizontal line in Figure 4.4(b) is clearly an outlier, so using the previously-mentioned statistical tools (mean, median, interquartile range, skewness), it could be singled out and eliminated. For this procedure, we make use of the dataframes created for the positively- and negatively-sloping lines using the Pandas library.

Post-processing the Hough Lines in Figure 4.4(b)

Interquartile Range: This is a statistical measure of dispersion that corresponds to the difference between the 75th percentile and the 25th percentile.

$$IQR = Q_3 - Q_1 \tag{4.1}$$

```
Interquartile Range for the left lines
 slope          0.119489
intercept     120.120935
```

(a) Left Lines

```
Interquartile Range for the right lines
 slope          0.149029
intercept     133.693080
```

(b) Right Lines

Figure 4.5: Interquartile Ranges

To identify the outliers, we apply a logical function in tandem with the IQR such that the outliers would return a true value while inliers will return a false value. The result of this operation on the same set of left and right lines can be seen in Figure 4.6. We can see that two lines in the set of left lines are outliers (one of the lines has an inlying slope).

Skewness: Generally, skewness values outside the range -1 $\leq$ *skew* $\leq$ 1 are considered too high, and such are pointers to the presence of outliers. The values below confirm our conclusion from the IQR analysis.

	slope	intercept
0	False	False
1	False	False
2	False	False
3	False	False
4	False	False
5	False	False
6	False	False
7	False	False
8	True	True
9	False	False
10	False	False
11	False	False
12	False	False
13	False	False
14	False	False
15	True	False

(a) Left Lines

	slope	intercept
0	False	False
1	False	False
2	False	False
3	False	False
4	False	False
5	False	False
6	False	False
7	False	False
8	False	False
9	False	False
10	False	False
11	False	False
12	False	False
13	False	False

(b) Right Lines

Figure 4.6: Outlier Identification using IQR

```
Skew for the left lines 2.6944046991725172
```

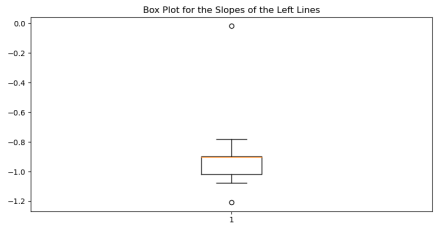
(a) Left Lines

```
Skew for the right lines 0.485847678308044
```

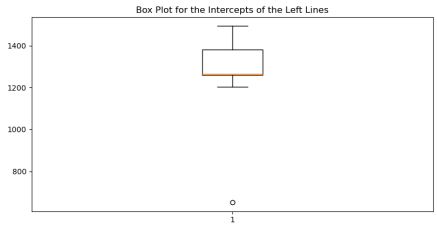
(b) Right Lines

Figure 4.7: Outlier Identification using Skewness Value

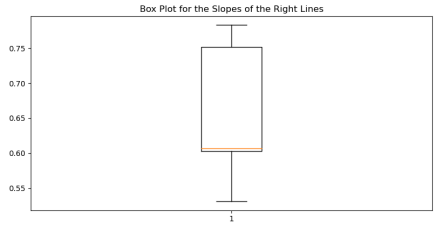
Visualizing the Outliers



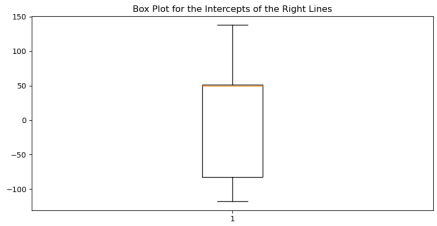
(a)



(b)



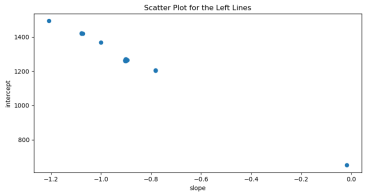
(c)



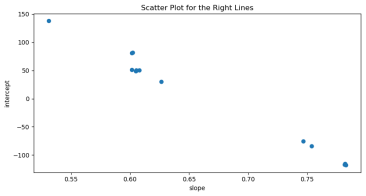
(d)

Figure 4.8: Visualizing Outliers using Box Plots

Consistent with the previous observations, outliers exist in the left lines dataset.



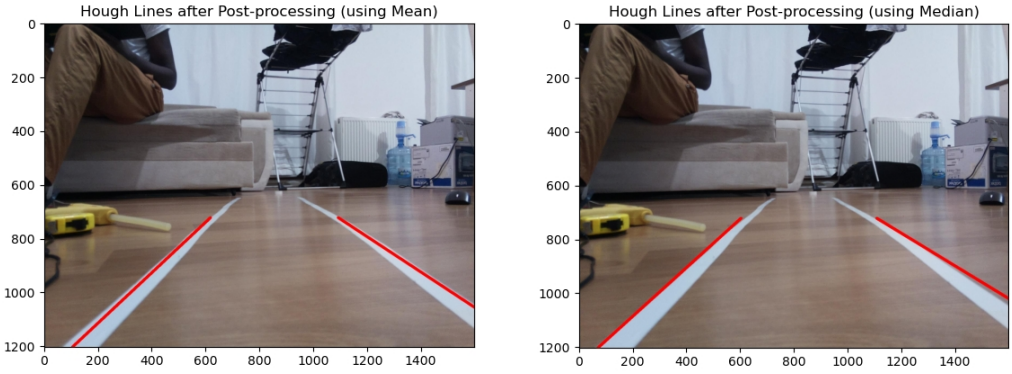
(a)



(b)

Figure 4.9: Visualizing Outliers using Scatter Plots

Handling Outliers There are several ways to eliminate outliers; however, for the sake of brevity, we implemented quantile-based flooring and capping. This is predicated on the concept that the minimum value in a dataset is floored at a specific percentile; correspondingly, the maximum value is capped at a certain percentile. For the lines in Fig. 4.4(b), we applied a 25th percentile flooring and a 75th percentile capping. This eliminates the outliers we observed in the previous sections leaving us with data from which we can take the mean or median slope and intercept to construct the final lanes (i.e., the lines of best fit) as seen in Figures 4.10(a) & (b). Clearly, using the mean to obtain the final lanes produces better results compared to using the median.

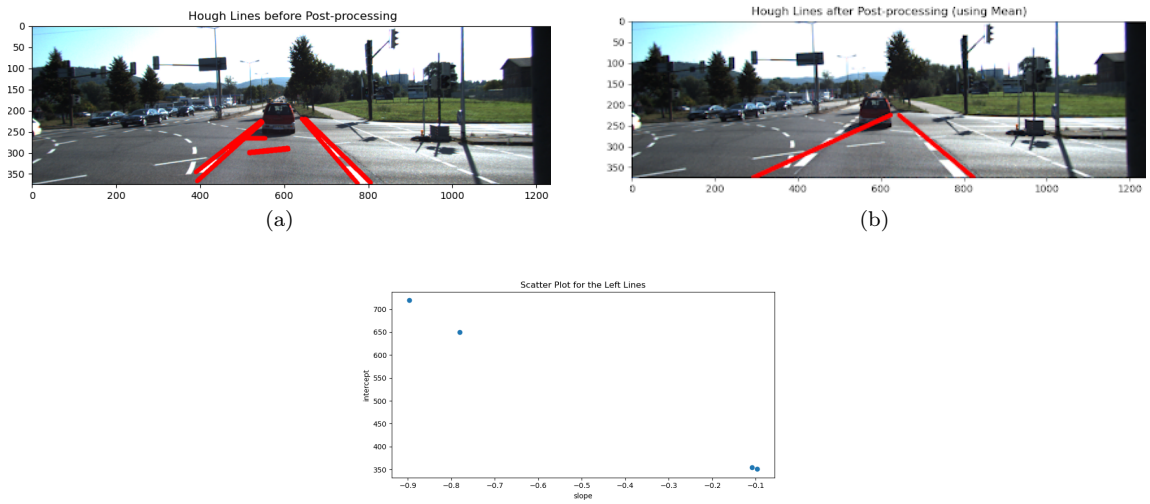


(a) (b)

Figure 4.10: Detected Lanes after Post-processing

Performance of the Lane Detection Algorithm on Other Datasets

Using the candidate image from the KITTI Road Dataset in Figure 2.2,



(a) (b)

(c)

Figure 4.11: Results from the KITTI Road Dataset

The results in Figure 4.11 show the loopholes in the algorithm. As we can see above, there are four detected lines (two of which are clearly not our desired lines). Just by looking at the scatter plot, it is impossible to tell which lines are the outliers. Hence, the mean lines chosen for the final output are not the results we want (especially for the left lane).
The lesson here is that the more the detected lines, the better the statistical elimination process and final results. Here, we can explore two plausible solutions:

1. Alter the criteria for the lines detected: this implies that we adjust the parameters for the probabilistic Hough Transform function. A more acceptable result is shown in Figure 4.12.

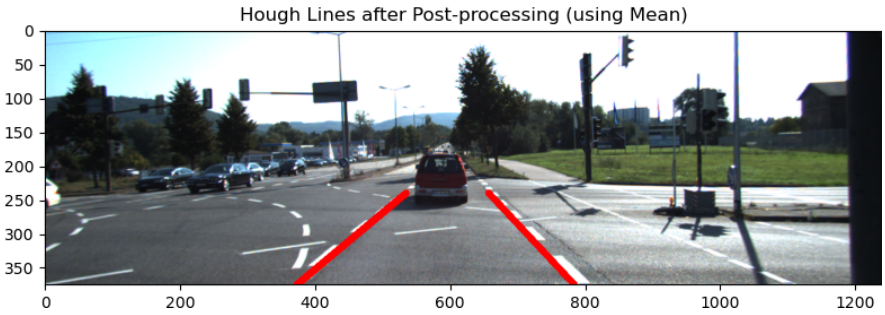
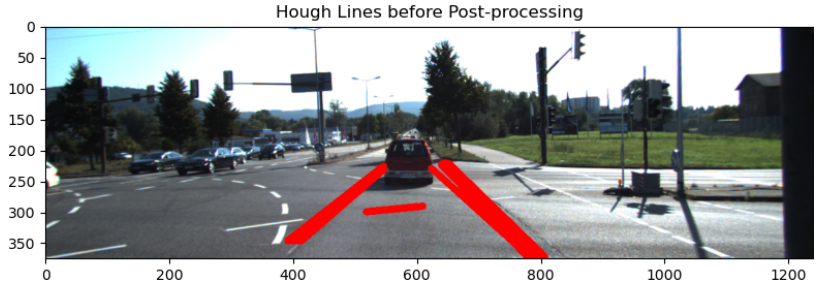
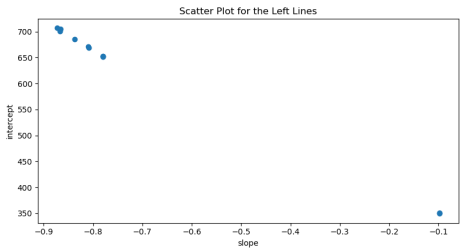


Figure 4.12: A More Plausible Result

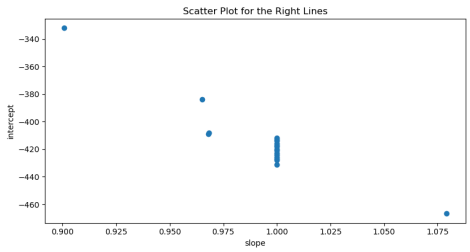
2. Replace the edge detection step with simple image thresholding. This increases the number of Hough lines detected (Figure 4.13(a)); hence, making it more difficult to skew the results (Figure 4.13(d)). Observe that the outliers are easily identifiable from the scatter plots, and by extension, they are easy to sift out.



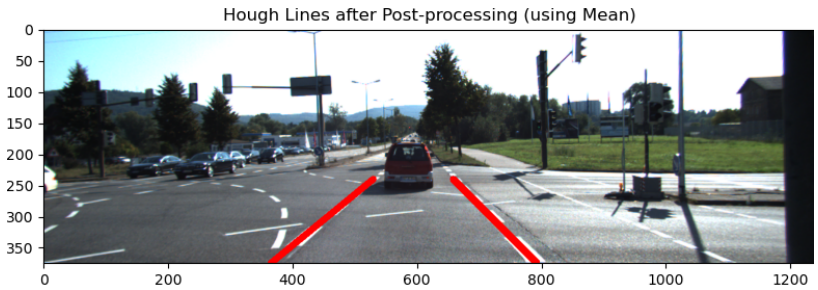
(a)



(b)



(c)



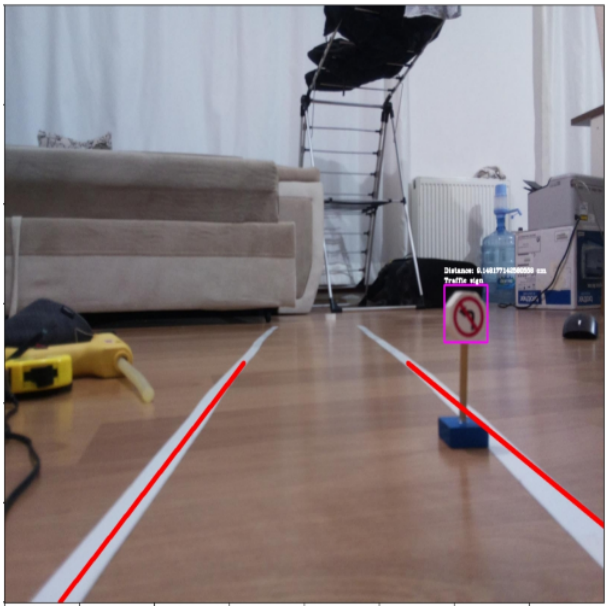
(d)

Figure 4.13: Post-processed Results from the KITTI Road Dataset

4.4 Putting It All Together

Figure 4.14 shows the result of putting everything we have done together. Given an input image 4.14(a), our algorithms are able process and run lane detection, target object detection and distance estimaton algorithms on a 1GB RAM raspberry-pi 3 in near real-time.

(a) Input Image



(b) Output Image

Figure 4.14: Lane Detection, Object Detection and Distance Estimation all put together

Chapter 5

Discussion

It is observed that although contour-based object detection does a good job in detecting contours, finding contours exactly belonging to the shape of an object is a challenging problem. Given an image, we obtain different contours describing all objects present in the image - this includes objects within and without our scope of interest. Furthermore, since the bounding boxes detected by the contour-based method are highly subjective and quite lacking in robustness, we decided to look for another object detection approach that will perhaps give us the bounding boxes we need. Deep Learning based methods were an option, but we chose to work with methods that are even easier to implement, yet robust and suitable for edge device deployment. We chose the Haar Feature-based method, and after training 100 positive image samples, the result is shown in Figure 4.1. Although the Haar feature-based method is not invariant to rotation, we chose it because our target object will be positioned on a fixed ground in the real world, and we are not worried about rotation. Our concern is the robustness of the bounding boxes that are produced, and the results observed from our experiments show just that. The height of the bounding boxes is one of the parameters needed for the distance estimation algorithm in 3.3, thus the performance of the algorithm correlates with the bounding boxes detected.

The Distance Estimation algorithm works to precise accuracy when the object is close to the camera and this can be used in addition to active sensors in obstacle avoidance algorithms for autonomous robots.

The approach taken to solve the lane detection problem is also strictly from a traditional computer vision perspective. This singular fact affects the degree of robustness and flexibility of the developed algorithm. We established that the seemingly complex problem is narrowed down to a relatively straightforward case of line detection using Hough Transform; however, as we saw in 3.5(b), directly applying Hough Transform to an edge image would lead to multiple detections, most of which are not representative of our desired lines/lanes. For this reason, a region of interest was identified and extracted. As much as this helps us streamline the detection task, choosing the region of interest is a manual process which requires **apriori knowledge** of the road network, the camera pose, etc. In real-time applications, such information may or may not be readily available. This is where more sophisticated techniques like machine learning and deep learning techniques would outperform a traditional CV technique like ours.

In our implementation, after detecting the Hough Lines, we were faced with the dilemma of designing an algorithm that would not just select one of the detected lines but also produce a line that best fits the lanes in the 3D scene. This is where we came up with the clever data analysis techniques to get rid of detected lines that do not align with the lanes in the 3D scene. However, we encountered some minor challenges when we tested this on road networks with multiple lane markings like in Figure 2.2, or with objects that obstruct the lanes in the scene (like the car partially does in the same figure). The challenges surface when the number of false positives and negative exactly match or are close to the number of acceptable detections (as seen in Figure 4.11(c)). Like we mentioned in the previous section, in this case, sifting out outliers becomes even more challenging. We went ahead to proffer solutions to cases of this nature; however, in the real world, it may not feasible to address each unique case differently while maintaining real-time performance.

One other issue that may arise (on rare occasions) is a case where the lane markings are broken lines that are far apart. Here, some lane detection algorithms could experience flickering in the video output due to video frames that return no detected lines. As mentioned before, learning-based techniques would almost seamlessly fix this; however, some traditional CV solutions also exist in the literature - some of which require some level of preprocessing on individual video frames.

In conclusion, based on all the results and submissions above, we opine that camera-based lane detection and distance estimation systems should be accompanied by other active-sensor based methods to offer a broad range of advantages that properly accommodate the failure or sub-par performance of any subsystem in solving FSD. In essence, **redundancy** should be a top priority in designing computer architectures for autonomous driving systems.

Bibliography

[1] Jessica B Cicchino. "Effects of lane departure warning on police-reported crash rates". In: *Journal of safety research* 66 (2018), pp. 61–70.

[2] Ravi Teja Mullapudi et al. "Hydranets: Specialized dynamic architectures for efficient inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8080–8089.

[3] *Automated vehicles for safety*. URL: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[4] *The 6 Levels of Vehicle Autonomy Explained*. URL: https://www.synopsys.com/automotive/autonomous-driving-levels.html.

[5] Kevin Meier, Soon-Jo Chung, and Seth Hutchinson. "Visual-inertial curve simultaneous localization and mapping: Creating a sparse structured world without feature points". In: *Journal of Field Robotics* 35.4 (2018), pp. 516–544.

[6] Xingxing Li et al. "Vehicle Pose Estimation Method for Lane Line Detection Through Monocular Camera". In: *IOP Conference Series: Earth and Environmental Science*. Vol. 769. 4. IOP Publishing. 2021, p. 042012.

[7] Swapnil Waykole, Nirajan Shiwakoti, and Peter Stasinopoulos. "Review on Lane Detection and Tracking Algorithms of Advanced Driver Assistance System". In: *Sustainability* 13.20 (2021), p. 11417.

[8] Oshada Jayasinghe et al. "SwiftLane: Towards Fast and Efficient Lane Detection". In: *arXiv preprint arXiv:2110.11779* (2021).

[9] Abdelmoghit Zaarane et al. "Distance measurement system for autonomous vehicles using stereo camera". In: *Array* 5 (2020), p. 100016.

[10] Chu Jiangwei et al. "Study on method of detecting preceding vehicle based on monocular camera". In: *IEEE Intelligent Vehicles Symposium, 2004*. IEEE. 2004, pp. 750–755.

[11] Shaoshan Liu et al. "Computer architectures for autonomous driving". In: *Computer* 50.8 (2017), pp. 18–25.

[12] Joseph Schlecht and Björn Ommer. "Contour-based object detection." In: *BMVC*. 2011, pp. 1–9.

[13] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "Yolov4: Optimal speed and accuracy of object detection". In: *arXiv preprint arXiv:2004.10934* (2020).

[14] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448.

[15] Jannik Fritsch, Tobias Kuehnl, and Andreas Geiger. "A new performance measure and evaluation benchmark for road detection algorithms". In: *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*. IEEE. 2013, pp. 1693–1700.

[16] Paul VC Hough. "Machine analysis of bubble chamber pictures". In: *Proc. of the International Conference on High Energy Accelerators and Instrumentation, Sept. 1959*. 1959, pp. 554–556.

[17] Xingang Pan et al. "Spatial as deep: Spatial cnn for traffic scene understanding". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[18] *Hough transform*. Oct. 2021. URL: https://en.wikipedia.org/w/index.php?title=Hough_transform&oldid=1050916528.

[19] Richard O Duda and Peter E Hart. "Use of the Hough transformation to detect lines and curves in pictures". In: *Communications of the ACM* 15.1 (1972), pp. 11–15.

[20] Zhengyou Zhang. "A flexible new technique for camera calibration". In: *IEEE Transactions on pattern analysis and machine intelligence* 22.11 (2000), pp. 1330–1334.

[21] John Canny. "A computational approach to edge detection". In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), pp. 679–698.

[22] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[23] Paul Viola and Michael J Jones. "Robust real-time face detection". In: *International journal of computer vision* 57.2 (2004), pp. 137–154.

[24] Rainer Lienhart and Jochen Maydt. "An extended set of haar-like features for rapid object detection". In: *Proceedings. international conference on image processing*. Vol. 1. IEEE. 2002, pp. I–I.

Chapter 6

Appendix

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import statistics
from scipy.stats import linregress
import pandas as pd
import math

path = "haarcascades/cascade_trafficsign.xml"  # PATH OF THE CASCADE
color = (255, 0, 255)

# LOAD THE CLASSIFIERS DOWNLOADED
cascade = cv2.CascadeClassifier(path)


class DistanceToCamera(object):
    def __init__(self):
        # camera params
        self.alpha = 3.0 * math.pi / 180  # degree measured manually
        self.v0 = 240.68330464  # from camera matrix
        self.ay = 494.86623672  # from camera matrix

    def calculate(self, v, h):
        # compute and return the distance from the target point to the camera
        d = h / math.tan(self.alpha + math.atan((v - self.v0) / self.ay))
        if d > 0:
            return d


d_to_object = DistanceToCamera()


def smooth(img):
    gaussian_filtering = cv2.GaussianBlur(img, (3, 3), 0)
    return gaussian_filtering


def convert_to_grayscale(img):
    BW = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    return BW
```

```
def detect_edges(img):
    # Apply Hysteresis Thresholding: H = 150; L = 50.
    e_image = cv2.Canny(img, 50, 150)
    return e_image


def ROI(img):
    # Find the total number of rows and columns of the image.
    row = img.shape[0]
    column = img.shape[1]
    # Create a trapezoidal region of interest that encompases the two lanes of interest.
    r_o_i = np.array([[
        (0, row), (column, row), (round(0.9*column), round(0.3*row)),
        (round(0.1*column), round(0.3*row))
    ]])
    # Create a mask (fully black) with the same dimension as the original image.
    mask = np.zeros_like(img)
    # Create a white region - the same shape as our region of interest - out of the mask.
    cv2.fillConvexPoly(mask, r_o_i, 255)
    # Implement a bitwise AND operation on the mask and the image to carve out
    # the region of interest
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image


def apply_threshold(img):
    # Apply a threshold on the region of interest to eliminate pixels that don't
    # represent the lane. Since the lane is almost white, we can set a lower
    # threshold not too far from 255.
    _, result = cv2.threshold(img, 170, 255, cv2.THRESH_BINARY)
    return result


def find_lines(img):
    # Use Hough Transform to extract lines in the image.
    # Carefully discretize the Hough Space to optimze accuracy.
    # If the precision is too low, there'll be lots of false positives.
    # If the precision is too high, there'll be lots of false negatives.
    # Set a threshold to determine which lines should be drawn.
    # Reject any lines below 40, and join lines with a gap of 5 between them.
    # lines = cv2.HoughLinesP(img, 1, np.pi/180, 30, maxLineGap=200)
    lines = cv2.HoughLinesP(
        img, 1, np.pi / 180, 70, np.array([]), minLineLength=20, maxLineGap=50
    )
    return lines


def display_lines(img, lines):
    # line_image = np.zeros_like(img)
    if lines is not None:
        for line in lines:
            # x1, y1, x2, y2 = line[0]
            # print(line)
            x1, y1, x2, y2 = line.reshape(4)
            # Draw a blue line (BGR) with thickness = 10
            cv2.line(img, (x1, y1), (x2, y2), (255, 0, 0), 10)
    return img
```

```
def construct_the_lanes(img, slope_and_intercept):
    final_slope, final_intercept = slope_and_intercept
    # Apply the equation of a line, y = mx+c, to construct the left and right
    # lane using the mean slope and intercept.
    y1 = img.shape[0]  # Since the first component contains the image rows (max = bottom)
    # The lanes would extend from the bottom to 3 fifths of the image height.
    y2 = int(y1 * 0.6)
    x1 = int((y1 - final_intercept) / final_slope)
    x2 = int((y2 - final_intercept) / final_slope)
    # Return the coordinates as a 1D array
    return np.array([x1, y1, x2, y2])


def data_analysis_of_lines(img, lines):
    left_lanes_array = []
    right_lanes_array = []
    left_slope_array = []
    right_slope_array = []
    left_intercept_array = []
    right_intercept_array = []
    for line in lines:
        x1, y1, x2, y2 = line.reshape(4)
        # Obtain the slope and intercept of each line using linear regression
        x = [x1, x2]
        y = [y1, y2]
        slope, intercept, r_value, p_value, std_err = linregress(x, y)
        # Recall that the x-y plane of an image is different. The origin is at
        # the top-left corner. Therefore, lines on the left would have a
        # negative slope (+ve for lines on the right).
        # Separate the left and right lines
        if slope < 0:
            left_lanes_array.append((slope, intercept))
            left_slope_array.append(slope)
            left_intercept_array.append(intercept)
        else:
            right_lanes_array.append((slope, intercept))
            right_slope_array.append(slope)
            right_intercept_array.append(intercept)

    # Create Pandas Dataframes for the arrays obtained above for statistical
    # calculations.
    df_left = pd.DataFrame(
        {"slope": left_slope_array, "intercept": left_intercept_array}
    )

    df_right = pd.DataFrame(
        {"slope": right_slope_array, "intercept": right_intercept_array}
    )
    #######
    ####### DATA ANALYSIS ON THE HOUGHLINES FOUND TO CHECK AND ELIMINATE OUTLIERS.
    ###### USING MEASURES OF STATISTICAL DISPERSION
    # This is used to find the interquartile range.
    Q1 = df_left.quantile(0.25)
    Q3 = df_left.quantile(0.75)
    IQR = Q3 - Q1
    print("Interquartile Range for the left lines \n", IQR)
```

```
Q1_ = df_right.quantile(0.25)
Q3_ = df_right.quantile(0.75)
IQR_ = Q3_ - Q1_
print("Interquartile Range for the right lines \n", IQR_)

# Use the interquartile range to check which array components are OUTLIERS.
# True denotes the presence of an OUTLIER.
print((df_left < (Q1 - 1.5 * IQR)) | (df_left > (Q3 + 1.5 * IQR)))
print((df_right < (Q1_ - 1.5 * IQR_)) | (df_right > (Q3_ + 1.5 * IQR_)))

# Use the skew of the array to check which array components are OUTLIERS.
print("Skew for the left lines", df_left['slope'].skew())
print(df_left['slope'].describe())
print(df_left['intercept'].describe())

print("Skew for the right lines", df_right['slope'].skew())
print(df_right['slope'].describe())
print(df_right['intercept'].describe())
#
# ###### THE FOLLOWING COULD BE USED TO VISUALLY IDENTIFY OUTLIERS.
# ## Box Plot
plt.boxplot(df_left["slope"])
plt.title('Box Plot for the Slopes of the Left Lines')
plt.show()
plt.boxplot(df_left["intercept"])
plt.title('Box Plot for the Intercepts of the Left Lines')
plt.show()
plt.boxplot(df_right["slope"])
plt.title('Box Plot for the Slopes of the Right Lines')
plt.show()
plt.boxplot(df_right["intercept"])
plt.title('Box Plot for the Intercepts of the Right Lines')
plt.show()

#### Scatter Plot
## Left
fig, ax = plt.subplots(figsize=(12,6))
ax.scatter(df_left['slope'], df_left['intercept'])
ax.set_xlabel('slope')
ax.set_ylabel('intercept')
ax.set_title('Scatter Plot for the Left Lines')
plt.show()

## Right
figg, aax = plt.subplots(figsize=(12,6))
aax.scatter(df_right['slope'], df_right['intercept'])
aax.set_xlabel('slope')
aax.set_ylabel('intercept')
aax.set_title('Scatter Plot for the Right Lines')
plt.show()

### USE THE INTERQUARTILE RANGE TO ELIMINATE OUTLIERS
df_left["slope"] = np.where(
    df_left["slope"] < df_left["slope"].quantile(0.25),
    df_left["slope"].quantile(0.25),
    df_left["slope"],
)
```

```
df_left["slope"] = np.where(
    df_left["slope"] > df_left["slope"].quantile(0.75),
    df_left["slope"].quantile(0.75),
    df_left["slope"],
)

df_left["intercept"] = np.where(
    df_left["intercept"] < df_left["intercept"].quantile(0.25),
    df_left["intercept"].quantile(0.25),
    df_left["intercept"],
)
df_left["intercept"] = np.where(
    df_left["intercept"] > df_left["intercept"].quantile(0.75),
    df_left["intercept"].quantile(0.75),
    df_left["intercept"],
)

df_right["slope"] = np.where(
    df_right["slope"] < df_right["slope"].quantile(0.25),
    df_right["slope"].quantile(0.25),
    df_right["slope"],
)
df_right["slope"] = np.where(
    df_right["slope"] > df_right["slope"].quantile(0.75),
    df_right["slope"].quantile(0.75),
    df_right["slope"],
)

df_right["intercept"] = np.where(
    df_right["intercept"] < df_right["intercept"].quantile(0.25),
    df_right["intercept"].quantile(0.25),
    df_right["intercept"],
)
df_right["intercept"] = np.where(
    df_right["intercept"] > df_right["intercept"].quantile(0.75),
    df_right["intercept"].quantile(0.75),
    df_right["intercept"],
)

# Show the new dataframes
# print(df_left['slope'].describe())
# print(df_left['intercept'].describe())
#
# print(df_right['slope'].describe())
# print(df_right['intercept'].describe())

### These conditional statements are used to check for frames without
# detections and skip them to avoid errors.
if len(right_lanes_array) == len(left_lanes_array) == 0:
    return np.array([])
if len(left_lanes_array) == 0:
    ### Using mean
    right_slope = df_right["slope"].mean()
    right_intercept = df_right["intercept"].mean()

    ### Using median # right_slope = df_right['slope'].median()
    # right_intercept = df_right['intercept'].median()
    right = np.array([right_slope, right_intercept])
```

```
        right_line = construct_the_lanes(img, right)
        return np.array([right_line])
    elif len(right_lanes_array) == 0:
        ### Using mean
        left_slope = df_left["slope"].mean()
        left_intercept = df_left["intercept"].mean()

        ### Using median # left_slope = df_left['slope'].median()
        # left_intercept = df_left['intercept'].median()
        left = np.array([left_slope, left_intercept])
        left_line = construct_the_lanes(img, left)
        return np.array([left_line])

    left_slope = df_left["slope"].mean()
    left_intercept = df_left["intercept"].mean()
    left = np.array([left_slope, left_intercept])

    right_slope = df_right["slope"].mean()
    right_intercept = df_right["intercept"].mean()
    right = np.array([right_slope, right_intercept])

    left_line = construct_the_lanes(img, left)
    right_line = construct_the_lanes(img, right)
    return np.array([left_line, right_line])


############################
# Used to test an image or a single video frame.
# Comment out from here till line 225 and "uncomment" lines 229 to 252
# img = cv2.cvtColor(cv2.imread('2.jpeg'), cv2.COLOR_BGR2RGB)
# image_frame = np.copy(img)
# smoothed_image = smooth(image_frame)
# gray_image = convert_to_grayscale(smoothed_image)
# objects = cascade.detectMultiScale(gray_image)
# for (x,y,w,h) in objects:
#     cv2.rectangle(image_frame,(x,y),(x+w,y+h),color,3)
#     cv2.putText(image_frame,"Traffic sign",(x,y-5),\
#     cv2.FONT_HERSHEY_COMPLEX_SMALL,0.7,(255, 255, 255),2)
#     roi_color = image_frame[y:y+h, x:x+w]
#     d = d_to_object.calculate(y+h, 9)
#     cv2.putText(image_frame,"Distance: {} cm".format(d),\
#     (x,y-25),cv2.FONT_HERSHEY_COMPLEX_SMALL,0.7, (255, 255, 255),2)
# edge_image = detect_edges(gray_image)
# ROI_image = ROI(edge_image)
# threshold_result = apply_threshold(ROI_image)
# detected_lines = find_lines(threshold_result)

# final_lines = data_analysis_of_lines(image_frame, detected_lines)
# line_image = display_lines(image_frame, final_lines)
# plt.imshow(line_image)
# plt.title('Hough Lines after Post-processing (using Mean)')
# plt.show()

# Use ctrl + / to comment out blocks of code

######################
# Create a capture variable for the video
def main():
```

```
windowname = "Detected Lanes"

# Create a capture variable for the video
video = cv2.VideoCapture(0)

filename = "Result.mp4"
codec = cv2.VideoWriter_fourcc(*"XVID")
framerate = video.get(cv2.CAP_PROP_FPS)
# write the video
VideoOutPut = cv2.VideoWriter(filename, codec, framerate, (640, 480))

if video.isOpened():
    ret, frame = video.read()
else:
    ret = False

image_frame_array = []
# Read each frame once the video starts
while True:
    # Create an empty variable for the boolean value and frame for each frame
    ret, image_frame = video.read()
    if image_frame is None:
        break
    image_frame = image_frame.astype("uint8")

    smoothed_image = smooth(image_frame)
    gray_image = convert_to_grayscale(smoothed_image)

    objects = cascade.detectMultiScale(gray_image)
    for (x, y, w, h) in objects:
        cv2.rectangle(image_frame, (x, y), (x + w, y + h), color, 3)
        cv2.putText(
            image_frame,
            "Traffic sign",
            (x, y - 5),
            cv2.FONT_HERSHEY_COMPLEX_SMALL,
            0.7,
            (255, 255, 255),
            2,
        )
        roi_color = image_frame[y : y + h, x : x + w]
        d = d_to_object.calculate(y + h, 9)
        cv2.putText(
            image_frame,
            "Distance: {} cm".format(d),
            (x, y - 25),
            cv2.FONT_HERSHEY_COMPLEX_SMALL,
            0.7,
            (255, 255, 255),
            2,
        )

    edge_image = detect_edges(gray_image)
    ROI_image = ROI(gray_image)
    threshold_result = apply_threshold(ROI_image)
    detected_lines = find_lines(threshold_result)

    try:
```

```
            final_lines = data_analysis_of_lines(image_frame, detected_lines)
        except TypeError:
            final_lines = None
        line_image = display_lines(image_frame, final_lines)
        # Obtain the size of one frame and store all frames in an array
        height, width = line_image.shape[:2]
        image_frame_array.append(line_image)
        VideoOutPut.write(line_image)
        cv2.imshow(windowname, line_image)
        # Wait 1ms between each frame
        # Use the q button to stop the video
        if cv2.waitKey(1) == ord("q"):
            break
    cv2.destroyAllWindows()
    VideoOutPut.release()
    video.release()


if __name__ == "__main__":
    main()
```