ENPM 673 Project 2

Daniel Sahu
Brenda Scheufele

8 March 2020

1 Problem #1

Problem #1 involves pre-processing a given video to prepare it for further computer vision tasks. The video is taken at night with poor lighting conditions and jittery perspective. Given the video quality we chose to focus on enhancing lane and obstacle (other car) information at the cost of color and background quality.

The video output of this algorithm can be found in this Google drive.

The following algorithms were employed:

1. Gamma Correction. This applies a nonlinear scale to our pixel intensities to enhance the darker regions.
2. Adaptive Histogram Equalization using the CLAHE algorithm. Normal Histogram Equalization is a poor choice for this image, since the blanket scaling of the Cumulative Distribution Function (CDF) doesn't handle the fact that we have very bright and very dark high interest regions. To account for that we used the CLAHE algorithm, which essentially applies a Histogram Equalization to each subsection of the image (while filtering out outlying regions).
3. Gaussian Blurring. This helps reduce the amount of noise in the image.

Although the image arguably looks worse at the end of this process, it does have enhanced lanes and features relevant to machine vision applications.

2 Problem #2

Problem #2 involves detecting and tracking lanes within a given data set. This information is used to predict upcoming turns in the road. This entire process can be broken up into three distinct processes, each of which we elaborate on below:

1. Per-Frame Lane Detection
2. Lane Tracking
3. Turn Prediction

Note that data set #1 is given as a series of .png images, which we pre-process into an .mp4 video to match the format of data set #2. The video output for both data sets can be found in this Google drive.

2.1 Per-Frame Lane Detection:

The Per-Frame Detection stage is our best attempt at extracting "lane-like" objects from an individual frame in isolation from the rest of the system. The only heuristic information we apply is that lanes will generally be in the lower half of the image. We use this to apply a Region of Interest (ROI) and ignore the upper half of the image.

For each frame we perform the following operations (see the accompanying Figures).

1. Example input frame given in Figure 1.
2. First we correct for camera lens distortion, using the given values of the K matrix and the distortion matrix D in the OpenCV function 'undistort'. This results in Figure 2.
3. Next, we convert the image to grayscale. This will be used in conjunction with the HSV image to pull out yellow and white lines. Grayscale is shown in Figure 3.
4. To ensure we get all the yellow as well as the white lines, we convert to HSV, shown in Figure 4. We then use a filtered range for yellow and white. The yellow range was found using colorpicker.py which provides a way to adjust the Hue, Saturation, and Value channels independently on the image to get the best range for each. The result of using HSV to mask everything except yellow and white is shown in Figure 5.

5. Next, we filter the image using a Gaussian Blur algorithm with a kernel of (5,5). The result is shown in Figure 6.

6. Using Canny for edge detection, we get the image in Figure 7.

7. We finish the pre-processing by selecting the region of interest, which is everything below the horizon line. This gives us Figure 8.

8. Then we use the Hough lines algorithm to determine all lines in the given image. See Figure 9.

9. Finally, we overlay the selected lines onto the image, Figure 10.

2.2 Lane Tracking

Although the Per-Frame Lane Detection works fairly well on straight roads with clear line of sight to both lanes, it begins to degrade significantly in the presence of noise and curved lanes. We account for this and other inconsistencies via a Lane Tracking algorithm which smooths out and retains lane information over the course of the entire video.

The tracker is seeded with an initial (manual) specification of what the "right" lane is for each video. This heuristic allows us to filter out given frame candidates with egregious slopes. Then, for each frame, the tracker is updated with all detected Hough Lines and used to predict the 2nd degree polynomial which fits it best.

The rough process (pseudocode) for a single frame is as follows:

1. Update:
   (a) Find all Hough lines that have slopes within our confidence interval (95%). If none are found, use the next closest.
   (b) Recalculate confidence interval using matches from the last N frames (5).
2. Predict:
   (a) Fit a 1st degree polynomial (line) to each match from the last N frames (5).
   (b) Build up a point cloud by sampling each line along its domain.
   (c) Fit a 2nd degree polynomial to the point cloud. This is our predicted lane.

This tracker was found to have fairly robust performance (the same parameters are used in both datasets). Potential improvements (which would help with Turn Prediction) are:

1. Fit the 2nd degree polynomial to the points which the Hough Lines were fit to. This would retain curve shape much better.
2. Limit the domain of the 2nd degree polynomial. Currently it is the superset of all matched lines.

2.3 Turn Prediction

With a robust prediction of lines over the course of the video we were able to implement a rough Turn Predictor. This operates by comparing the intersection of both lanes to their theoretical intersection if they were both straight lines. If the non-linear intersection is to the left of the linear intersection we assume the curve is veering to the left, etc. A small deadband of pixels was chosen to account for noise.

Unfortunately this approach suffers from the fact that our 2nd degree polynomials are very close to lines. Were that to be improved this approach would be more robust.

3 Results

The output of Part #1 and #2 can be found at this Google drive.

Overall we found Problem #1 very difficult to enhance. Although the approach we used would improve the quality of lane detection (and possible car detection), it clearly degrades the image quality significantly and introduces a large number of noisy artifacts.

For Problem #2 we are fairly happy with the lane detection and tracking. The lane tracking could stand to be improved, which would further improve its intended applications (like Turn Prediction). Despite this drawback it was found to work well over the wide range of frame inputs.

4 Figures

Figure 1: Input Frame

Figure 2: Rectified Image

Figure 3: Grayscale Image

Figure 4: HSV Image

Figure 5: After HSV Masking for yellow and white

Figure 6: Gaussian Blur

Figure 7: Detected Edges

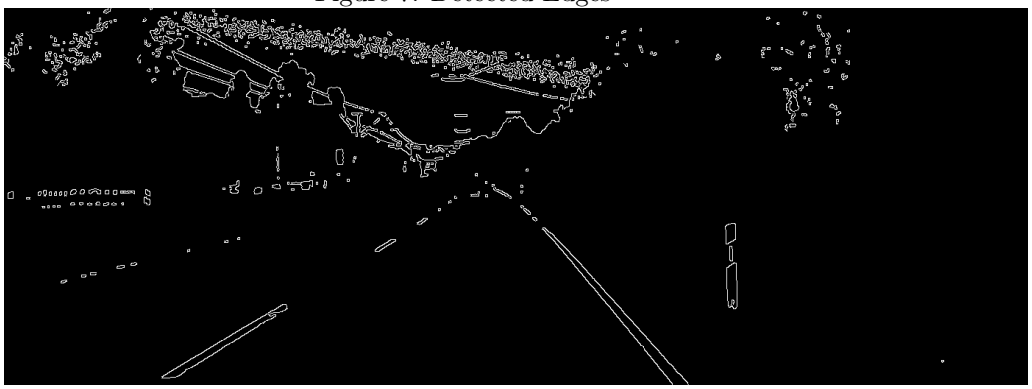


Figure 8: Region of Interest

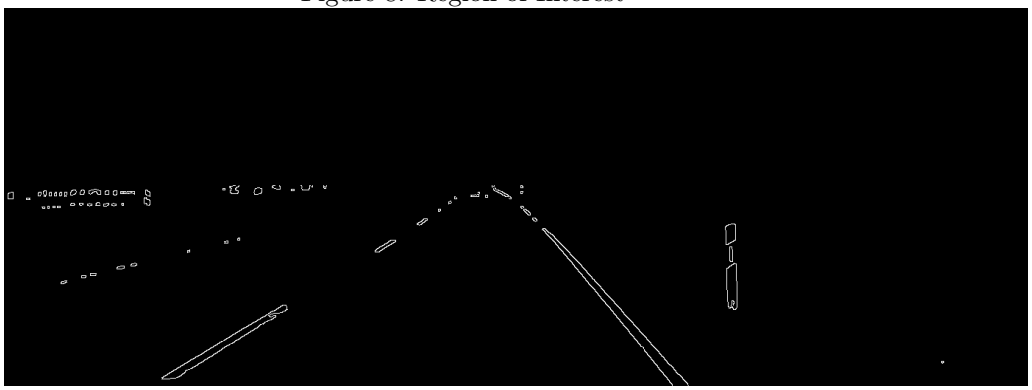


Figure 9: Hough Lines

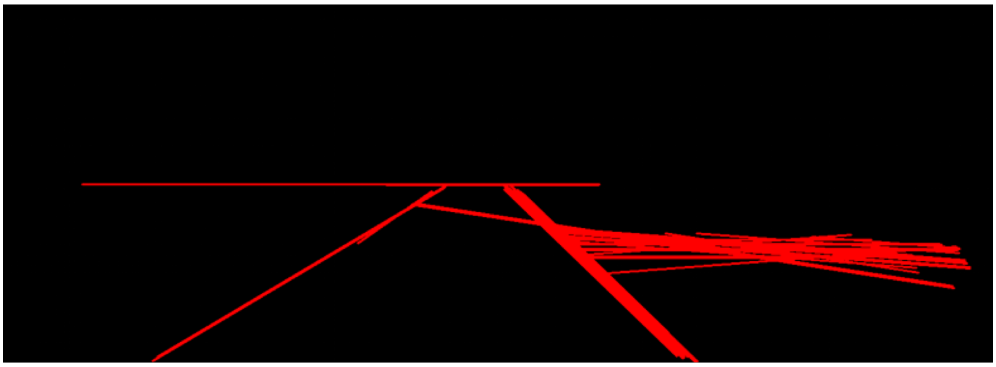


Figure 10: Lane Lines superimposed on image

5 Theoretical Overview

5.1 Hough Line

The following explanation on the Hough line technique came from the lecture and the OpenCV tutorial. Lines can be described in many ways, the most common are

$$y = mx + b \tag{1}$$

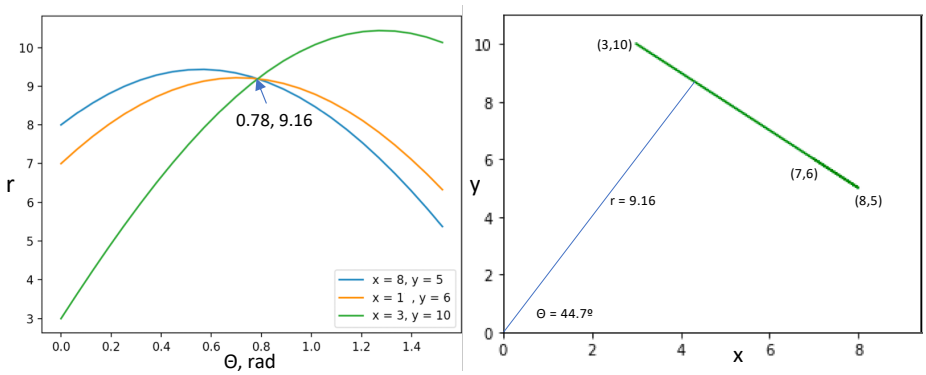
or the polar coordinate form

$$r = x * cos(\theta) + y * sin(\theta). \tag{2}$$

In the polar form, r is the perpendicular distance to the line from the origin, and θ is the angle from the x axis to r. For a point (x, y) there are an infinite set of points (r, θ) that follow a sine curve describing the infinite lines that (x, y) can be on. Every pair (x, y) have a sine curve of possible (r, θ). For two (x, y) points, the point where their sine curves intersect is the (r, θ) that describes a line that contains both points. An example is shown for the points$(x, y) = (8, 5), (7, 6), (3, 10)$, plotted below. The Hough technique calculates the curves for all (r, θ) that go through each point (x, y), and then finds the intersections of all the curves. If enough curves intersect, then that (r, θ) combination is determined to be the equation of a line, consisting of all the points which generated the intersecting curves.

The algorithm we used is the "HoughLinesP" which is the probabilistic Hough transform. Basically, the (r, θ) space is divided into accumulator bins, and every point is run through and placed into a bin. The bins with the most points are candidate lines, and, depending on parameters chosen, the highest are selected as 'detected' lines. The transform outputs pairs of points that are the endpoints of these detected lines. The first input to this function must be a binary image. The next input is a vector, *lines*, to store the output points, and then ρ, the resolution parameter, in pixels (we use 1). The next parameters are θ, the resolution of the parameter θ in radians (we use 20 deg), *threshold*, the minimum number of intersections to 'detect' a line (we used 30), and the *minLinLength*, the minimum number of points in a detected line (we use 100), and lastly, *maxLineGap*, the maximum gap between two points to be considered the same line (we used 250).

Figure 11: Hough line algorithm: left graph is sine curves for the given points; right graph is the resultant line that all points lie on



5.2 Homography

We tried this method to do turn prediction. First we manually picked 4 points on the first frame. Then we used the frame size to pick the point for the destination frame. Using the "findHomography" function, we transformed the image as shown in figure below. From this we could calculate the midpoint of the lines. Then we tried to calculate the subsequent distances from the midpoint; if the left lane got farther from the midpoint, it was a right turn and vice versa. We didn't have a lot of success with this method, and so adopted the method described above.

Figure 12: Homography transform