ENPM673 Project 2

Justin Albrecht and Brian Bock

Spring 2020

Table of Contents

1 Enhancing Video

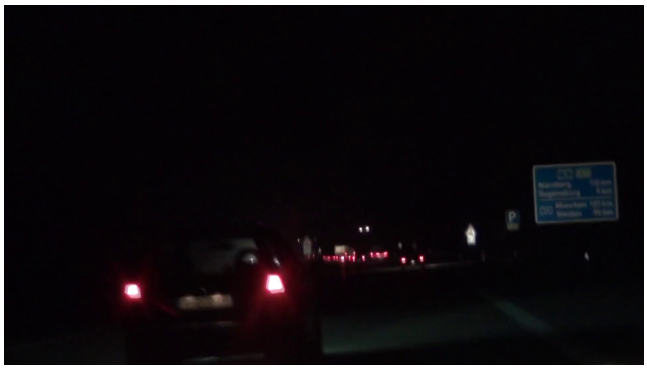
The task for this section was to improve the lightning and visibility in the provided video. The video is shot at night, which makes it dark. You can view the original video here: https://youtu.be/s23jnVK4rHs. Unlike the videos in Part 2 which use a rigidly mounted camera, the video in this part is shot with a hand controlled camera. Throughout the sequence, the camera moves and changes the view. To compound the difficulty, most of the video is both compressed and very out of focus. The lack of focus makes it nearly impossible to know if a smoothing kernel is over-applied - the video is blurry regardless. The compression shows up in interesting ways later; since most of the video is dark, large clusters of pixels are stored as the same color, and these clusters become readily apparent with a variety of image manipulations.

1.1 Brightness

Brightness (or pixel intensity) is simply a piece-wise addition to the image matrix. Since the original image is 8 bit, pixel values cannot exceed 255. Any value that does wraps around and is interpreted as a near 0 number. This produces a result that is visually interesting (if not useful) where the bright spots of the image show up black, but the bordering light rays appear as normal (Figure 1).

Figure 1: Over bright frame without 8 bit correction

To combat this issue, we use convert the image to 32 bit, add brightness, use `np.clip` to cap any values above 255 at 255, and then convert the image back to 8 bit for saving and displaying. On its own, brightness is not a great fix for this video. The sky and road both tend toward grey (and then white) as the brightness increases, without much improvement in visibility or image quality (Figure 2).
Here are two versions of the video with boosted brightness:
+25 - https://youtu.be/VFmRoxNZOys
+50 - https://youtu.be/LHDYsPBT0Z8



(a) Original frame

(b) Brightness +25

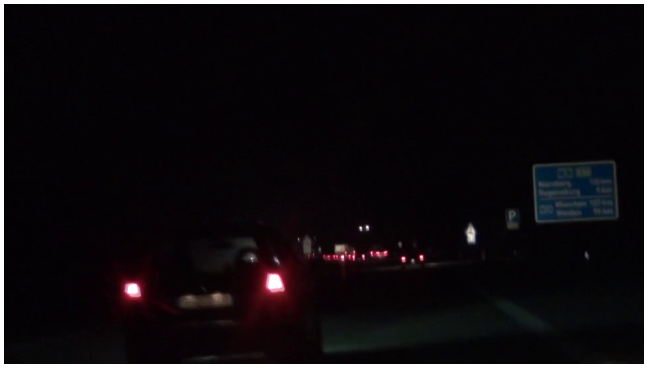
(c) Brightness +50

(d) Brightness +100

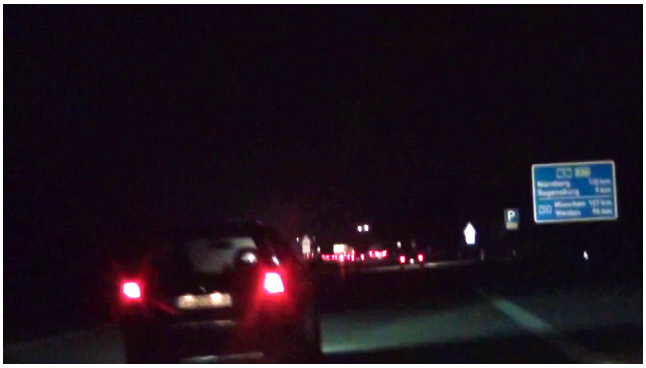
Figure 2: Frames with different levels of contrast

1.2 Contrast

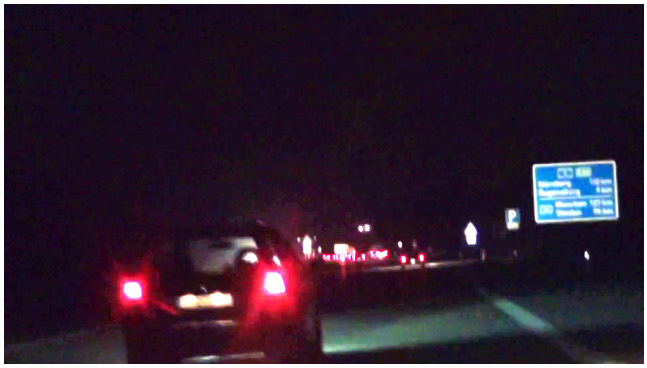
Contrast is a scalar piece-wise multiplication with the image matrix. It is susceptible to the same 8 bit wrap around issue as the brightness, and is fixed with the same approach. Increased contrast offers some small incremental improvements in the video, which you can see here (Figure 3):
*2 - https://youtu.be/Ob4xFdAFl68
*3 - https://youtu.be/I_JWFZJkPkM
*4 - https://youtu.be/2xkwpyO-ajc
*5 - https://youtu.be/IyASQW2F_Os



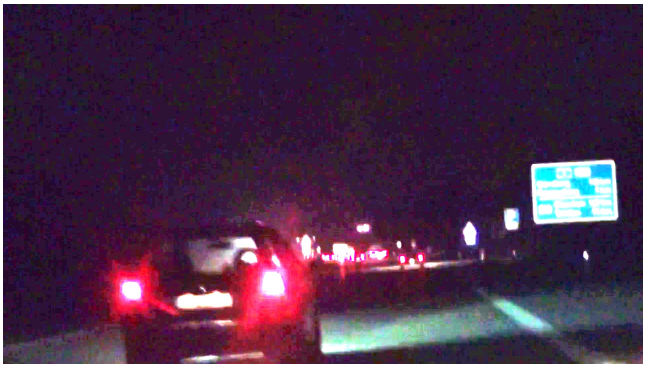
(a) Original frame



(b) Contrast *3



(c) Contrast *5



(d) Contrast *9

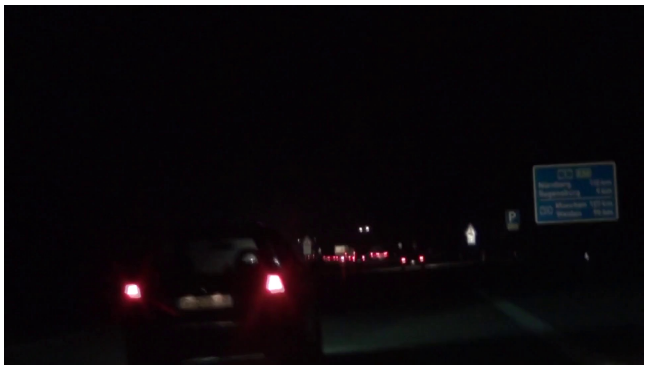
Figure 3: Frames with different levels of contrast

At higher contrast levels, we start getting a noisier, redder sky, and washed out road signs (Figure 3d). Car headlights also become excessively bright (Figure 4). This would be detrimental to the performance of an autonomous reliant on this video for navigation, as this headlight sun obstructs a significant portion of the field of view and misrepresents the size of the offending car.

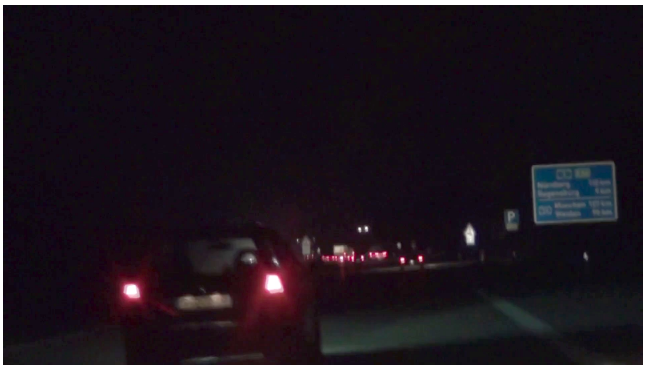
Figure 4: Over-contrasted (*9) frame with overbright car headlights

1.3 Gamma Correction

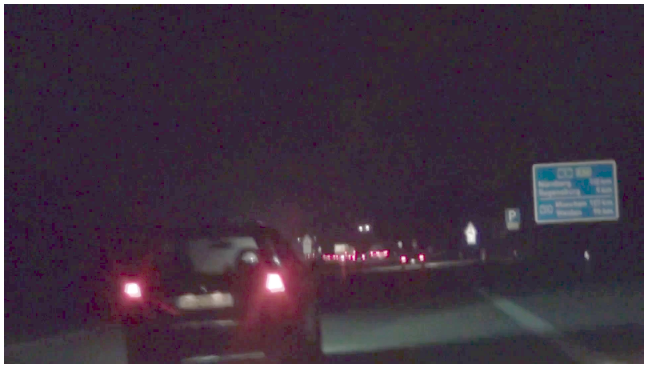
We next explore gamma correction based on the work done by Adrian Rosebrock [1].
Gamma=2 - https://youtu.be/zg0S08nHYJg



(a) Original frame



(b) Gamma=1.5



(c) Gamma=2.5

(d) Gamma=5

Figure 5: Frames with different gamma correction

1.4 Histogram Equalization

We attempted histogram equalization on the image. Histogram equalization on an RGB/BGR image would distort the colors, which is problematic for an autonomous car (which needs to know colors for traffic lights, brake lights, road signs, etc.). We convert the image to YUV color space, which has a dedicated channel for illumination (Y). We equalize the histogram of the Y channel and then convert the image back to BGR space. [2][3] The result looks like an old TV (Figure 6): https://youtu.be/53uUhmN6sZA

Figure 6: Frame with histogram equalization in the Y (YUV) channel

We can also try converting the image to the HSV color space, which similarly has a dedicated channel for intensity (V). The result has largely purple skewed color and is still a noisy mess (Figure 7): https://youtu.be/FYoVCjUv1X0

Figure 7: Frame with histogram equalization in the V (HSV) channel

2 Lane Detection

The goal of this assignment is to detect the lane lines from a dash cam of a car driving on a highway. We need to detect where the lane lines are in the image and then overlay a virtual lane over top the original video. There are two sets of videos, the first of which is mostly on a straight road, while the second video the car makes gradual turns. In the first data set, the video to analyze is comprised of 302 individual frames saved as images. The second data set is a 16 second video. With the exception of reading the video in, the steps to detect the lane lines in both of these data sets are very similar. In both cases, we treat the video as a series of independent frames.

2.1 Overview of image processing pipeline

- Import the next frame
- Use homography to warp the image to create a top-down view of the road
- Apply HSV threshold to create a binary image of the lane lines
- Generate a histogram of the number of white pixels in each column of the binary hsv threshed image
- Find the peaks of the histogram that represent the lane lines
- Gather the points around each peak and create a line of best fit through those points
- Create a new blank image, the same dimensions as the top down image and draw the lane lines, the lane, and arrows
- Warp the image back into the original camera coordinates and overlay it with transparency onto the original frame
- Determine turning direction

2.2 Top Down Image

In order to generate a homography matrix that will allow us to view the road as if we were looking from above we need to pick eight points. Four from the original source image and four where those points will map onto the destination image which in this case is the top down image. The first four points we pick by selecting two points just in front of the car on the lane and another two far ahead of the car on the lane. Each set of points share the same y-position in the image. In order to ensure that the top down image contains both lanes even if the car shifts position in the lane, we map the points not to the edge of the destination but instead 20% in on each side. Figure 8 shows how the points map between the two images.

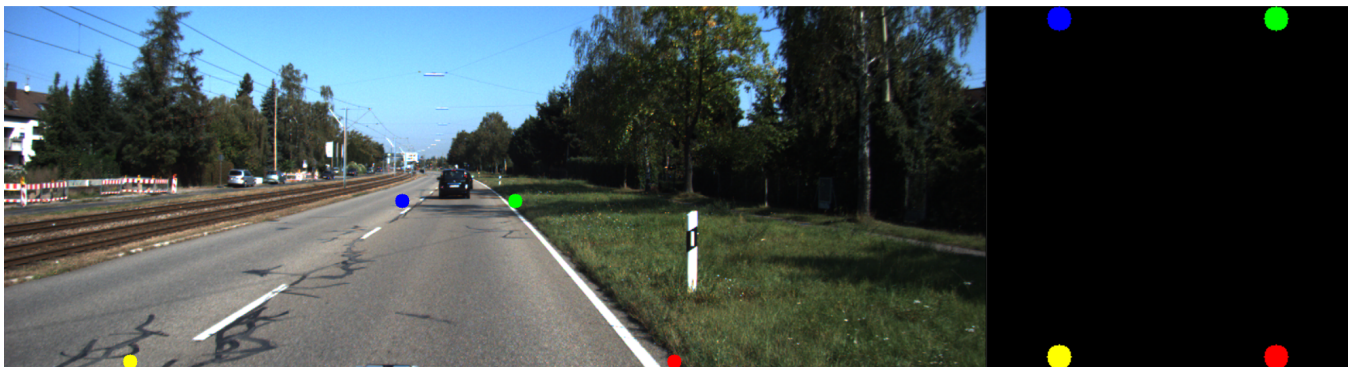


Figure 8: Point mapping between original and destination

Now that we have eight points we need to find the homography matrix that maps them. To do this we need follow the procedure as follows.

Generate an A matrix where our x_i, y_i are the points from the source image and the xp_i, yp_i are the points from the destination image.

$$A = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1 * xp_1 & y_1 * xp_1 & xp1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1 * yp_1 & y_1 * yp_1 & yp1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2 * xp_2 & y_2 * xp_2 & xp2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2 * yp_2 & y_2 * yp_2 & yp2 \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3 * xp_3 & y_3 * xp_3 & xp3 \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3 * yp_3 & y_3 * yp_3 & yp3 \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4 * xp_4 & y_4 * xp_4 & xp4 \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4 * yp_4 & y_4 * yp_4 & yp4 \end{bmatrix} \tag{2.1}$$

From this A matrix we can solve for the vector h, which is the solution to:

$$Ah = 0 \tag{2.2}$$

By reshaping our vector h, we can find the homography matrix:

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \tag{2.3}$$

To solve for the vector h, we can use Singular Value Decomposition or SVD. This will decompose our A matrix into three matrices, U, Σ, and V.

To perform the decomposition the following steps need to be followed:

1. Find eigenvectors (w_i) and eigenvalues (λ_i) for $A^T A$.
2. Sort λ_i from largest to smallest
3. Compose V as an $n \times n$ matrix such that the columns are w_i in the order of sorted λ_i
4. Create the list of singular values (σ_i) from the square root of the non-zero λ_i. $\sigma_i = \sqrt{\lambda_i}$
5. Compose Σ as an $m \times n$ matrix where the ith diagonal element is σ_i and all other elements are zero.
6. Use the relationship $Av_i = \sigma_i u_i$ to solve for the columns of U.

Note:
v_i is the columns of V
u_i is the columns of U

Using the steps above, we can decompose A into U, Σ, V.

After this decomposition we know that our h vector can be approximated by the column of V that corresponds with the small singular value in Σ. Since the singular values are ordered from largest to smallest, h can be approximated as the last column of V

$$h = V[:, -1] \tag{2.4}$$

We have written a function that computes this homography but for speed considerations we used the built-in opencv function `cv2.findhomography()`.

After we determine a homography matrix we can then warp the image from the normal camera coordinate system into the top down view. The resulting top down image is Figure 9.

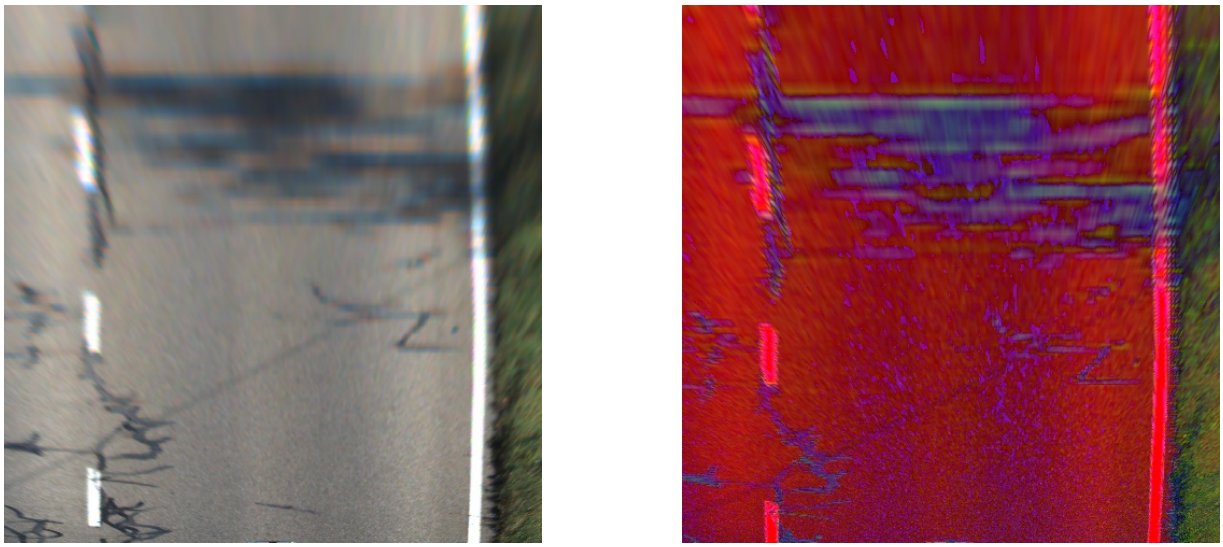
(a) Data set 1

(b) Data set 2

Figure 9: Top down images

2.3 HSV Threshing

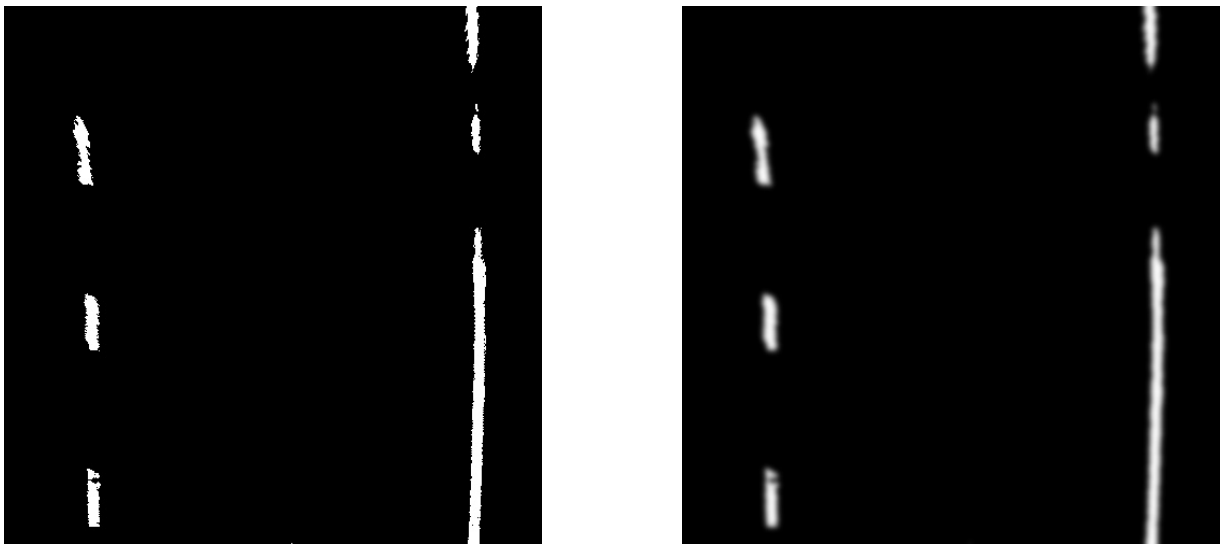
We converted each image from BGR space to HSV space (Figure 10b). The HSV color space is more robust and consistent under varied lighting conditions [4] [5] We experimentally determined the ideal HSV max and min threshold values that made just the lane lines clearly visible in the frame and eliminated most undesired image elements. We used these values to convert the HSV image into a binary image (Figure 11a). We then apply a Gaussian blur with a square kernel size of 15 to the binary image (Figure 11b). This is necessary to reduce noise that would otherwise make edge detection difficult.



(a) Original Warped Road Frame

(b) Frame in HSV color space

Figure 10: Example photo showing the HSV conversion

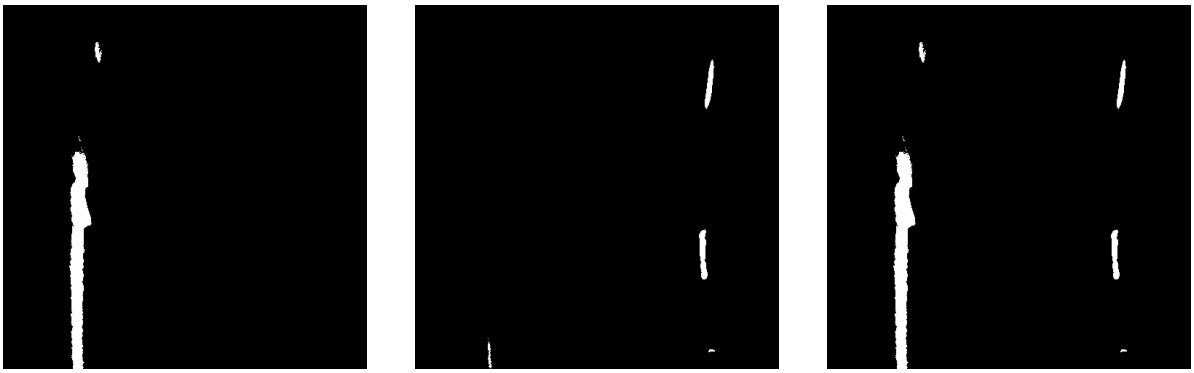


(a) Binary frame threshed from HSV

(b) Binary frame with Gaussian blur

Figure 11: Example photo showing the result of a binary threshing and Gaussian blur

Using a single threshold worked very well for the first data-set where both lane lines were white, but it started to fail for the second video where the left-lane was yellow. We tested with a variety of thresholds but we were never able to find a threshold which worked for the entirety of the video. In order to mitigate this issue we used two thresholds, one for white and one for yellow. Each of the threshold values were used to create a binary image. These binary images were then added together with a `cv2.bitwise_or`. This way even if one of the threshold catches part of the other lane it is not counted twice (Figure 12).



(a) Yellow Threshold

(b) White Threshold

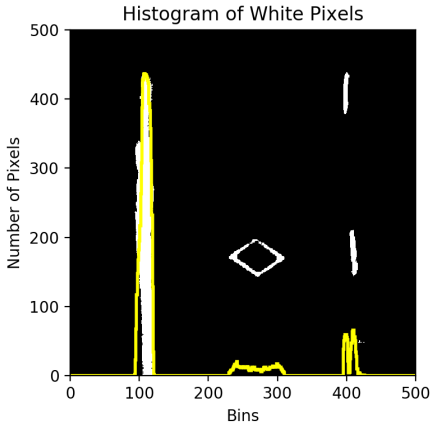
(c) Combination of the two

Figure 12: Results of using multiple threshold values

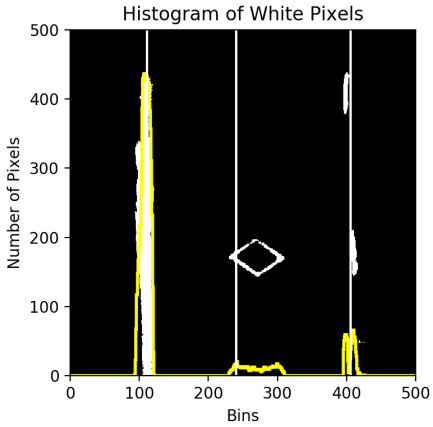
2.4 Histogram Peak Finding

Now that we have a threshed binary image, we need to determine what pixels are associated with which lane. To do this we found the peaks for a histogram of white pixels in the binary image for each pixel column in the image. The reasoning is that the lane lines from the top down should be fairly straight, therefore there should be peaks in the histogram where vertical lines are.

To find the peaks we considered multiple strategies but eventually decided to use a built in package within scipy named `find_peaks_cwt`. This function smooths the histogram then finds anywhere where there are peaks. This function also allows the user to specify the minimum distance between peaks, which allows for us to compensate for when there would be multiple peaks within the region of a lane line.



(a) Histogram for white pixels in threshed image



(b) Vertical Lines where peaks were detected

Once we have found all the peaks in the histogram we need to determine which peaks is associated with the lane lines. For the first frame we just use the peaks with the highest value. On subsequent frames we test to see if the peaks are within a reasonable threshold of the last frame. Doing this we are able to initialize two booleans, `found_left_lane` and `found_right_lane`. These booleans determine whether we try to generate a new lane line for the current image or just reuse the line from the previous frame. This is helpful for regions like under the bridge where we are unable to see the lane lines for a few frames. This also ensures that the lane lines are not too jittery. This approach does however require the first frame to be correct. Figure **??**

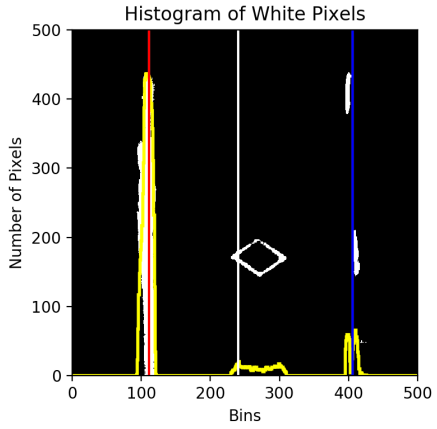


Figure 14: Red peak denotes left lane center, Blue peak denotes right lane center

2.5 Lane Pixel Candidate

After the histogram peak detection, we have found a point close to the center of each lane. We then want to gather all the points within some region around the peak and fit a line to the points. We gather points within ± 25 pixels of each peak location. These pixels are the lane candidates that we use for line fitting.

For simplicity we decided to use only fit a straight line to the points but the method would allow for a more complex curve to be fitted to the data without too much effort. We then use `polyfit` to find the lane line coefficients. Our peak finding already distinguishes which peak is which so we now have an equation for both the left and right lane that fits the points of the HSV threshed image. As mentioned previously if either of the booleans for peak detection are `False`, we just use the line coefficients from the previous frame.

2.6 Lane Overlay

From the left and right lane coefficients, we generate the points that bound the lane in the top down squared image. We create a new all black image with the same dimensions as the top-down squared image. Within this image, we

use the defined corners to draw a polygon which becomes our green lane overlay. Thick red lines are drawn from the top to bottom corners to bound the lane, becoming our lane lines. The next major step is to add the arrows to the center of our lane. We take the two top points and average them to get a midpoint between them. We repeat this to get a midpoint at the bottom, and then draw a line between these midpoints. This is the line that our arrows will be drawn on. We define the length and spacing of our arrows, and then use `cv2.arrowedLine` to generate the arrows. We now have a generated lane overlay with lane lines, a green road, and yellow arrows in the world frame (Figure 15).

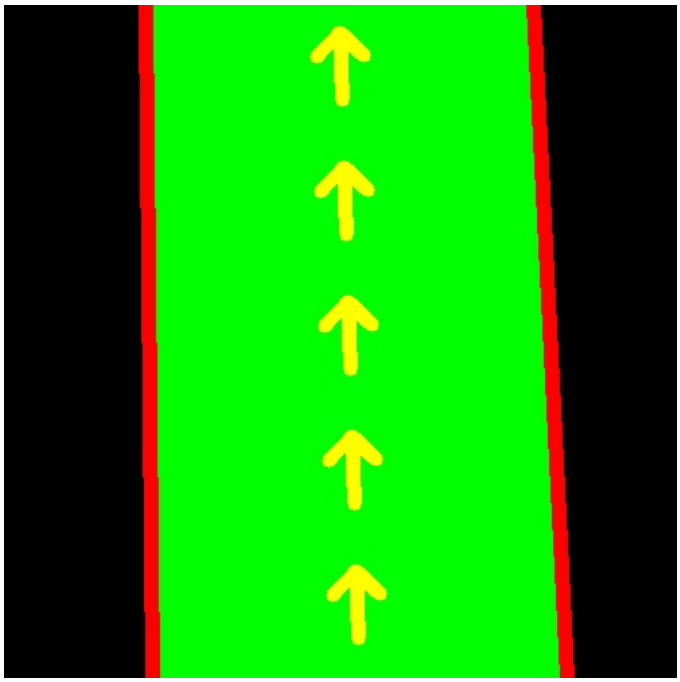
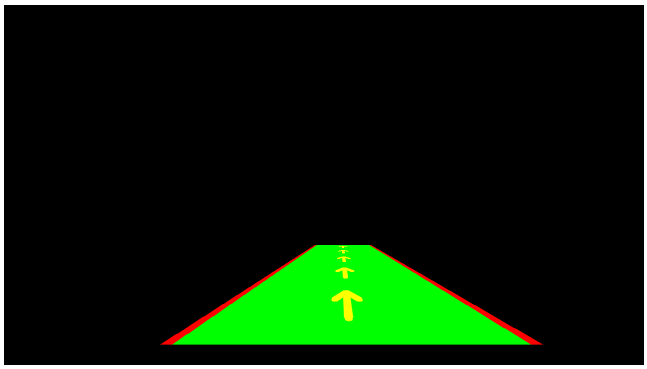


Figure 15: Overlay generated

Now we use the inverse Homgraphy to warp this overlay into the camera frame. The shape of this overlay is blacked out in a copy of the original frame, and then a bitwise OR combines the lane overlay with the road blacked frame.



(a) Image with just the warped road overlay

(b) Frame with road region blacked out

This new frame with the image and lane overlay is then recombined with the original frame (using `cv2.addWeighted` and alpha=0.5) to make the final image with a semi-transparent lane overlay (Figures 17, 18)

Figure 17: 1st Frame from Data Set 1 with the lane overlay shown

Figure 18: 1st Frame from Data Set 2 with the lane overlay shown

2.7 Turn Prediction

We take the slope of our lane midpoint line (used to draw the arrows) and compute it's slope.

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{2.5}$$

This slope has a bit of noise and changes a small amount between frames. We experimentally determined a reliable range of m which denote the car is turning right, driving straight, or turning left. Based on this information, we print the slope and car action on the frame, and show it to the screen (Figures 19a, 19b, 19c).

(a) Car turning left (b) Car driving straight (c) Car turning right

Figure 19: Turning Detection

There is no turning in the first data set, so this method is not implemented with that video.

2.8 Discussion on Hough Lines

Another possible way to solve the problem of lane detection would be to use Hough lines. Hough lines work by using a voting system. The process is as follows:

- Determine the edges of the image
- Initialize a Hough Matrix H with all values set to zero
- For each edge point in the image find many equations of a line, that passes through the point, in the form $d = xcos(\theta) + ysin(\theta)$.
  - To do this you should find a new d for each θ between 0° and 180°
  - Add 1 vote to that index (H[d,θ])
- Find the maximum in the H matrix. These indicies of the max are the d and θ that represents a line that fit the most edges in the image.
- For lane detection you should find the equation that fits the each of the lane lines as maximums within H, you will probably need to add some filtering to remove extraneous lines in the image.

2.9 Applicability to Other Videos

We were able to apply our solution from the first data set of this project to the second with minimal modification (only adjusted the HSV threshold and road bounding points). This increases our confidence in the applicability of our solution to other videos. The techniques used in this project should be work with minimal modification on any fixed camera driving on a road with the following conditions:
The road must be reasonably well lit. This approach would probably not work as well at night, unless the lane lines were illuminated or highly reflective. In any condition where the lane lines are clearly visible, this technique should work. The HSV thresholds might need to be adjusted to compensate for drastically different lighting conditions.
Using multiple thresholds, lane lines can be yellow or white. These are the most common colors for lane lines, so this should have wide applicability. In order to capture other lane line colors, you could add another HSV threshold, which is an easy task.
The road points would need to be reconfigured for any new camera configuration or vehicle, but if chosen well they should be applicable for any driving that car does (given the above conditions). Since the program analyzes histogram peaks and not position, it does not care about the lane width. The program could therefore not restricted to roads of a comparable lane width, and should handle roads with a variety of lane widths. Lanes are often wider on highways and parkways than they are on local roads. It is important to define the road region points on a wide lane, and to allow for some buffer width. If the points were defined on a narrow road, regions of a wider road (and the lane lines) would be lost.
Since this project is founded on lane line detection, it would be useless on roads without any lane lines.

2.10 Videos

2.10.1 Data Set 1

Lane Overlay - https://youtu.be/__XLG9Cgs0I

2.10.2 Data Set 2

Lane Overlay - https://youtu.be/NvaUFfDh_DU
Lane Overlay with turning - https://youtu.be/QNkY0j8-9CY
Pipeline Visualization - https://youtu.be/twkbwrYAsu4

3 Code

You can view all of the code related to this project at https://github.com/BrianBock/ENPM673_Project2/

4 References

[1] A. Rosebrock, "OpenCV Gamma Correction," Oct 2015. [Online]. Available: https://www.pyimagesearch.com/2015/10/05/opencv-gamma-correction/

[2] M. A. Jazaery, "OpenCV Python equalizeHist colored image," Jan 2016. [Online]. Available: https://stackoverflow.com/questions/31998428/opencv-python-equalizehist-colored-image

[3] L. Dashko, "How to Obtain a Single Channel Value Image from HSV Image in Opencv 2.1," Jan 2018. [Online]. Available: https://stackoverflow.com/questions/17185151/how-to-obtain-a-single-channel-value-image-from-hsv-image-in-opencv-2-1

[4] V. Gupta, "Color spaces in OpenCV (C++ / Python)," May 2017. [Online]. Available: https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/

[5] S. Mitchell, "ENPM809T 03 OpenCV Fundamentals," Feb 2020. [Online]. Available: https://myelms.umd.edu/courses/1277572/pages/03-perception?module_item_id=9793888