Project 2: Lane Detection

ENPM 673 – Perception for Autonomous Robots

March 13, 2019

Rene Jacques, Zachary Zimits

Pipeline/Homography

We began this project by choosing the frame shown in Figure 1. We selected the four points shown red in the image as our image points. We selected these points because we know the distances between them in the world coordinates. We assumed that the lane was 12 feet wide and that the distance between the white dashes are 30 feet. There exact location in the world frame was not necessary but to keep the same scaling factor we chose that the pixels coming across the lane would be 120 pixels apart and the points running the length of the lane would be 300 pixels apart. Using cv2.findHomography() function we calculated the homography matrix between the camera and world frame. Next we used cv2.warpPerspective to warp so that we were looking down on the road in world coordinates Figure 2. We adjusted the x and y offset of the world coordinates in the findHomography function until we ended up with a good field of view in our flattened image.

Figure 1: Static Image for Homography Calculation

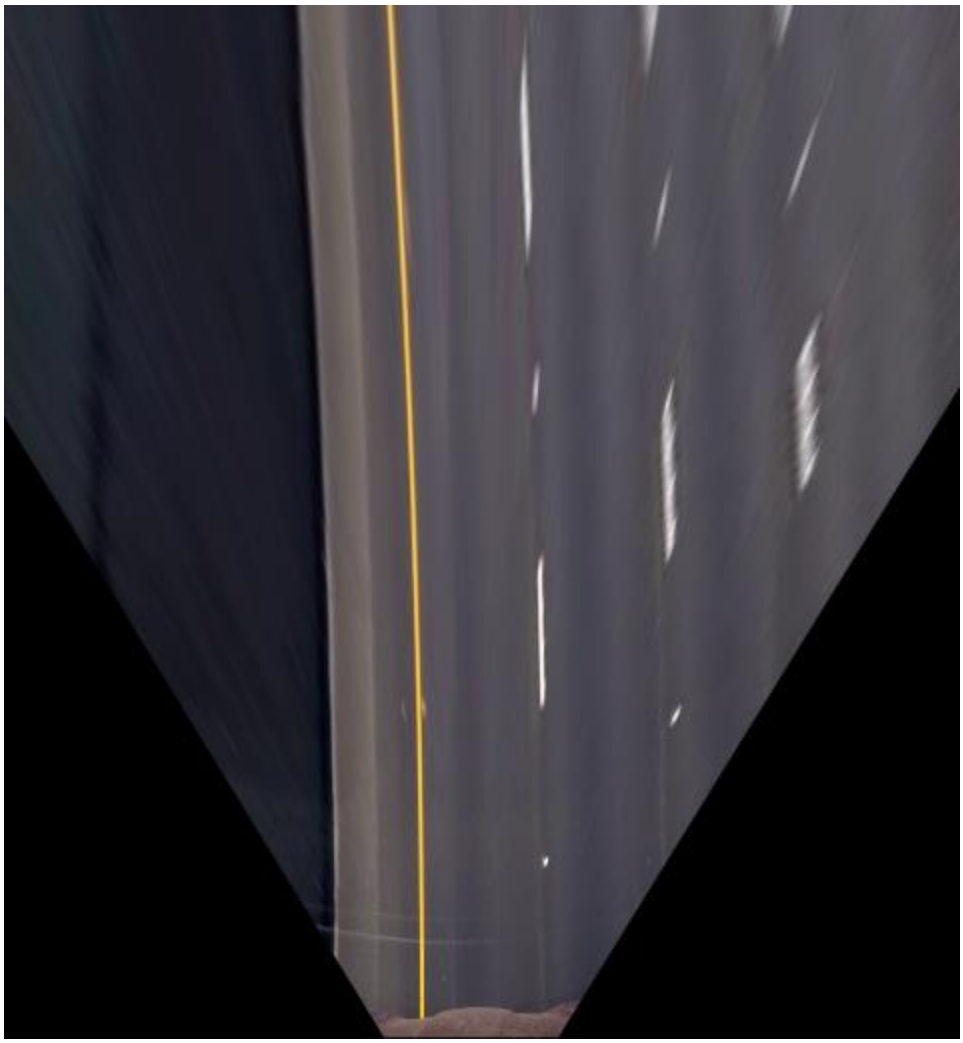

Figure 2: Image in World Coordinates

Color Segmentation

Before unwarping, the video frame must first be filtered to obtain only the yellow and white lane lines. In order to reduce extraneous data the top half of the frame is blacked out before color processing begins. Six color masks, three each for white and yellow pixels, are created using cv2.inrange() based on the original RGB image, the image converted to HLS, and the image converted the HSV. Then all three of the yellow masks and all three of the white masks are combined using cv2.bitwise_and(), producing two masks, one for white pixels and one for yellow pixels, which are then combined one final time using cv2.bitwise_or() to create the white yellow color mask. This final mask is combined with the original frame using cv2.bitwise_and to produce the color segmented image (Figure 3) which is then unwarped (Figure 4) using the homography calculated earlier to create the image that will be used to create the histogram.

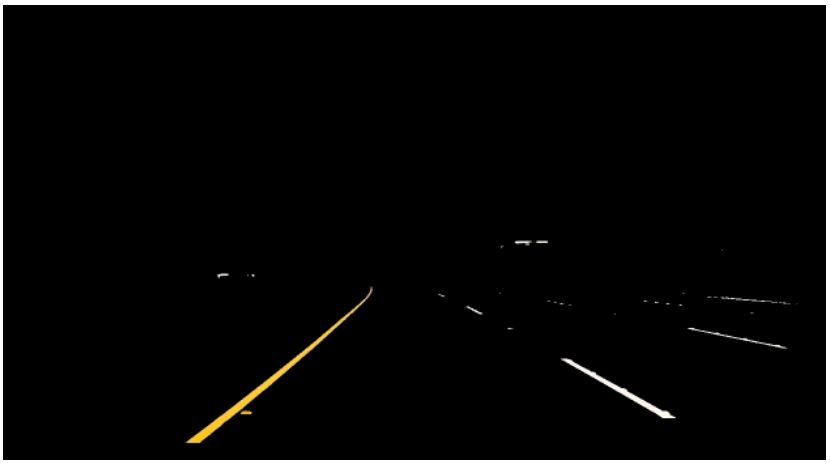

Figure 3: Color Processed Image

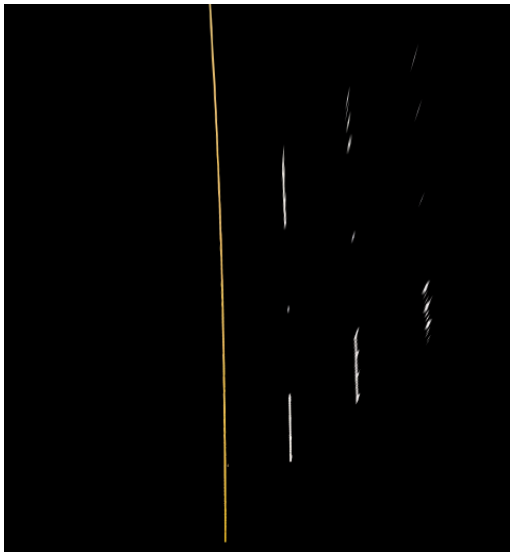

Figure 4: Color Processed World Frame

Histogram

To start the histogram the image is threshold so that the pixels are either 0 or 1. Next each column was summed to calculate the number of data pixels in each column. We graphed this data as an image so that it would update live with the video. To better understand the histogram data, we graphed it under the processed image so that we could see a direct correlation between the image and the histogram (see Figure 5).

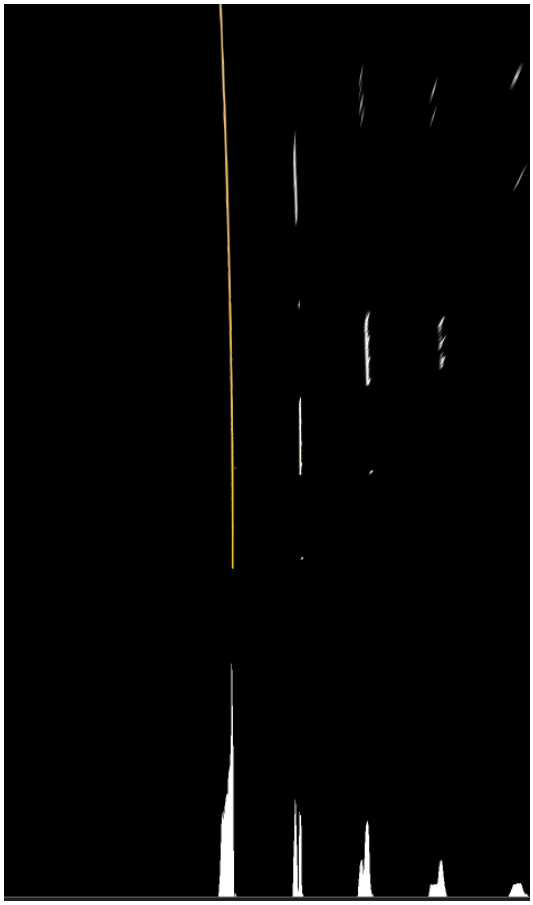

Figure 5: Histogram and World Frame

Identify Lanes

The next step was to process the histogram in order to determine what area of the unwarped image contained lane each lane line individually so that that a line could be fit to the pixels within that area alone. To do this we extracted the bottom row of the histogram image and parsed through it to locate the left and right sides of each peak. A visual representation of the extracted coordinates is shown in Figure 6.

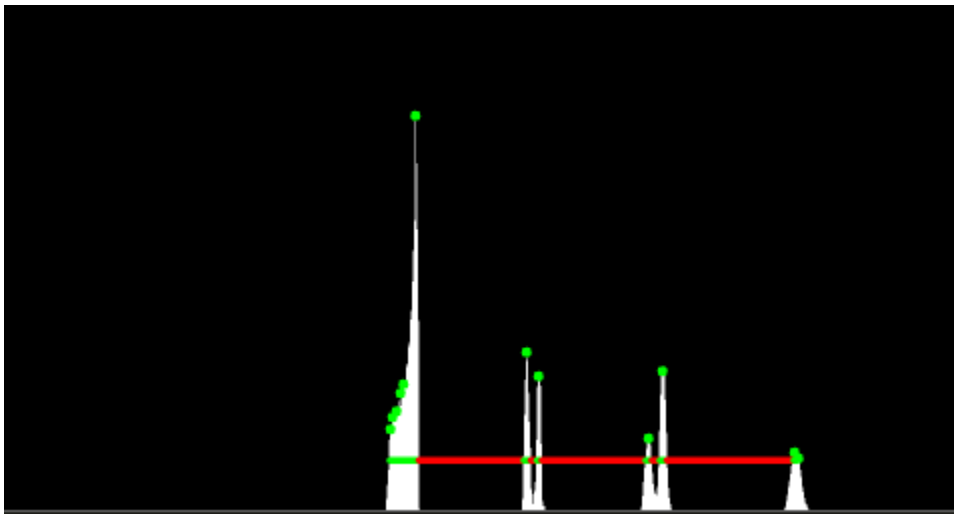

Figure 6: Histogram with Peaks (green circles) and Peak Edges (green and red lines between each edge)

The bounds of each lane are then determined by iterating through the list of peak edges and calculating the distances between each edge. If the distance is reasonable for a lane candidate both sides of the lane candidate are added to the output list of lane bounds. In order to capture the left side of the first lane line the edge that is in the index before the left bound of the first lane candidate (peak_edges[i-1]) is also added to the front of the output list. A visual representation of the lane bounds is shown in Figure 7. Where the cyan rectangles show the portions of the histogram that are lane markers and the blue rectangles show the area of the lanes.

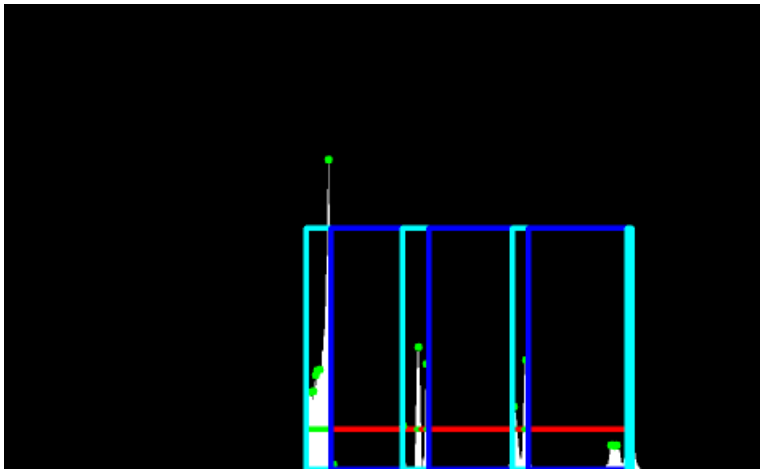

Figure 7: Histogram with Lane Boundaries

The lane bounds can now be used to define the area of the unwarped image that should be searched to locate each set of pixels that corresponds to a single lane line. This is done by retrieving each lane bound whose index divided by 2 has no remainder (i%2 == 0) and the lane bound in the index position in front of the first one (i+1). These two bounds will always refer to the left and right edge of a lane line, and also correspond to the left and right bounds of the area that needs to be searched to extract lane pixels for line fitting.

Lane Fitting

After each of the lane boundaries have been identified we compile a list of points that make us each of the boundaries we passed them into np.polyfit(). We got an equation for a line from that equation, but it tended to fit the line close to the top of the parabola so if we extended the line the lane would double back on itself. To fix this we flipped the x and y coordinates into np.polyfit(). This caused the function to return a parabola with a very wide apex since it could no longer double back on its self. We graphed these equations in the world frame by creating an array called *x_prime_vals* that is made up of a *np.linspace()* of the y axis. We plug these values into the polynomial equation calculated earlier to find *y_prime_vals*. We graphed these in to world frame using *cv2.polylines() using y_prime_vals* for the x coordinates and *x_prime_vals* for the y coordinates. The result can be seen in Figure 10.

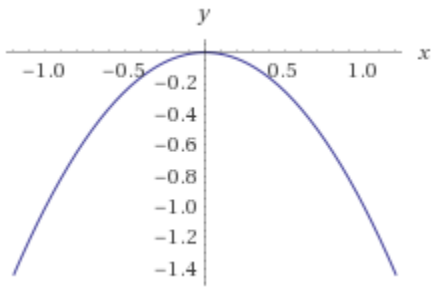

Figure 8: Standard y =-x^2

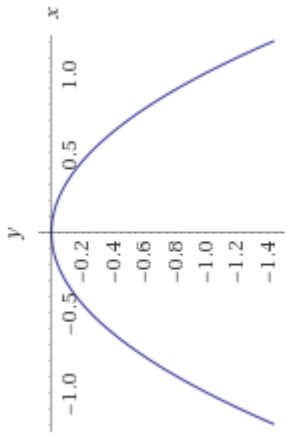

Figure 9: Rotate y=-x^2 no longer possible

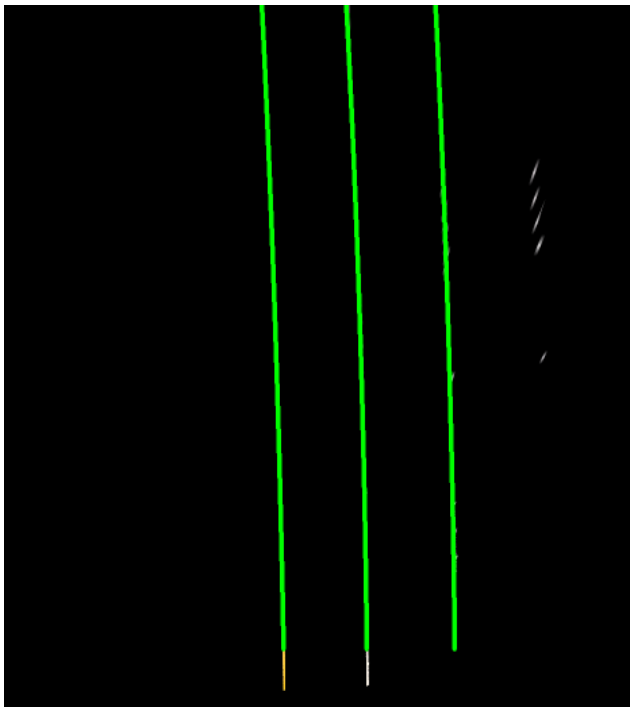

Figure 10: World Frame with Fit Lines

To visualize the lanes shown later we grouped lane boundaries that were next to each other and combined them in OpenCV contour format and used cv2.drawCountours() to draw them in world frame. We then used *cv2.warpPerspective()* with the inverse of the homography matrix to unwarp it back to the image frame and used a *cv2.addWeighted()* function to combine our lane drawing with the original image.

Calculations

Radius of Curvature

To calculate the radius of curvature for the road we used the equation:

$$R = \frac{\left(1 + \left(\frac{dy}{dx}\right)^2\right)^{\frac{3}{2}}}{\frac{d^2y}{dx^2}}$$

which calculates the radius of curvature at any given point x for the polynomial that we fit to the lane lines. To calculate the derivatives of our polynomial we created two functions, one for the first derivative and one for the second derivative that take as input a single x value. Then to calculate the full radius of curvature for one line we calculate the individual radius for each x coordinate of our polynomial and then use the average of all these radii as the final radius. The unit of the output radius was in pixels, so we multiplied the radius by a conversion factor (calculated using the pixel distance between two lane lines divided by the width of the lane e.g. 137px/3.7m) to get the radius in meters.

Vehicle Offset from Center

To calculate the offset of the vehicle from the center of the lane we use the fact that the camera should be centered on the vehicle. This means that the center of the image the camera sees should approximately correspond to the center of the vehicle. Therefore, we can simply calculate the distance from the right edge of the lane closest to the center of the image on the left from the center of the image, which should correspond to the distance of the vehicle from the center of the lane. This distance was in pixels, so we converted using the same conversion factor from above to get the distance in meters.

Turn Direction

Turn direction is simply calculated based on the sign of the radius of curvature. If the sign is negative then the vehicle is turning left, and if the sign is positive then the vehicle is turning right. If the radius of curvature is larger than 20 km then we say that the vehicle is moving straight.

Figure 11: Final Output Turning to the Left

Figure 12: Final Output Going Straight

Figure 13: Final Output Turning Right

Figure 14: Final Output challenge_video

Hough Lines

Before processing Hough Lines the image must be processed for edges so that the detected edges are in white and all other pixels are in white. With this image a list of the white pixel coordinates is created and an array of zeros is created $H[\theta, D]$. Going through the list of white pixels you plug the coordinates into the equation

$$D = xcos(\theta) + ysin(\theta)$$

And solve for D for every theta 0 to 180. After solving each D you find that container in H ($H[\theta_i, D_i]$) and increment the value stored in that element. After processing these values you look at which elements of H are above a certain value. You can then convert these thetas, and Ds, into a line that can be graphed in the image space.

Project Video vs Challenge Video

Due to light and saturation differences the color segmentation parameters we used to the project video did not produce the same results for the challenge video. After extensive tuning we were able to detect both yellow and white lines in the challenge video, but when using the same parameters in the project video we were unable to get reliable results. In order to fix this in the future we might be able to calculate some threshold value for each frame of the video and adjust our color segmentation parameters based on this threshold.

Despite choosing specific parameters to detect the yellow and white lines in the challenge video our lane line processing was not able to consistently identify the line pixels and fit lines to them. In order to fix this we processed a white pixel histogram and a yellow pixel histogram separately. This produced accurate line fitting using our original methods on the yellow pixel data. However, there was too much noise in the white color segmentation histogram for us to be able to extract reliable lane pixels and fit lines to them. The solution we found was to identify the highest peaks on the white histogram that was closest to the center of the image, which should always the lane lines for the lane the vehicle is currently in. Using this data we were able to calculate the line fit normally and obtain reliable lane detection for the challenge video.

Likelihood of Success

For inputs like the project_video with a large contrast between the lane markings and the road surface our method of calculating is going to have good results since the histogram has very distinct peaks that it can detect. In the challenge_video the lane markings are not as pronounced, and our color segmentation was not able to yield as strong of a histogram. With good data the histogram can accurately detect neighboring lanes and even a lane two over in the right circumstances. This is a powerful tool for switching lanes.

Figure 15: Final Output Multi-lane Detection