CarND Advanced Lane Finding Project Writeup

Tiffany Huang

June 21, 2018

1 Advanced Lane Finding Project

This project consists on the following steps/tasks:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

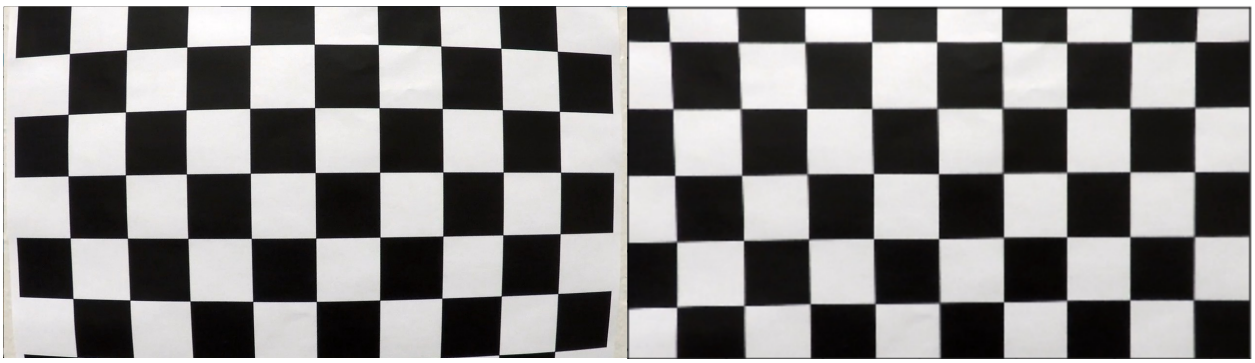
My project code can be found here: `https://github.com/tahuang/Advanced_Lane_Finding`. Next, I will consider each of the rubric points individually and describe how I addressed each point in my implementation.

2 Camera Calibration

I calculated the correct camera matrix and distortion coefficients using several calibration chessboard images and the OpenCV `calibrateCamera` function. The code can be found in section 1 of `lane_finding.ipynb`. The following is an example of a successfully undistorted calibration image:

3 Pipeline

The code to run the entire pipeline is in the last section of `lane_finding.ipynb` titled "Pipeline".



(a) Original calibration image (b) Undistorted calibration image

Figure 1: Example of distortion corrected calibration image

3.1 Distortion Correction

Distortion correction was done using the camera calibration parameters found in the previous step and the OpenCV `undistort` function. The code can be found in section 2 of `lane_finding.ipynb` and line 21 of the last cell. An example of a distortion corrected image is below. The undistorted image doesn't look much different because there was not too much distortion in the original image.

(a) Original image (b) Undistorted image

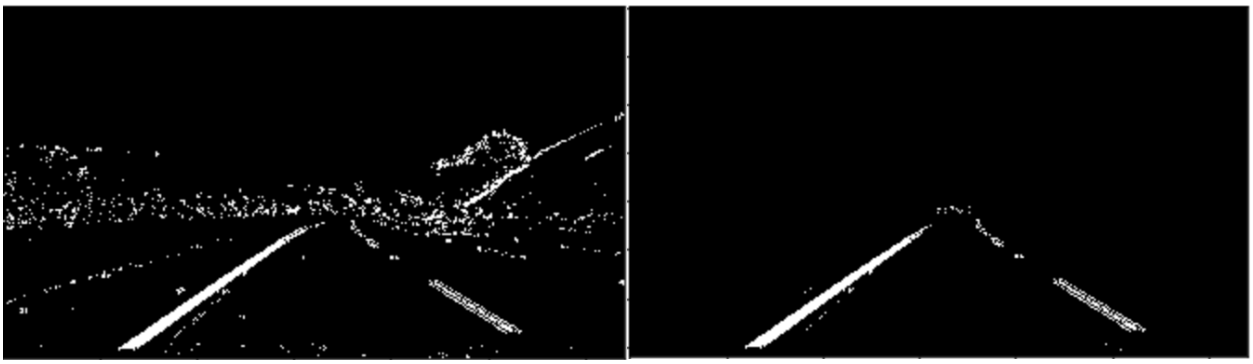
Figure 2: Example of distortion corrected image

3.2 Thresholded Binary Image

Next, I created a thresholded binary image by first Gaussian blurring the image to reduce noise. I then applied the $Sobel_x$ and $Sobel_y$ operators and a HLS color threshold to the image. Table 1 shows the thresholds I used for each filter. I then combined all of the binary images from each method to create a combined threshold. Figure 3 shows an example of the combined output. The code for the thresholding is on lines 26-33 of the last cell and in the cells under section 3 of `lane_finding.ipynb`. After thresholding, I also cropped a region of interest to focus on the important pixels and reduce noise and outliers (code on line 38 of last cell).

Operator	Kernel Size	Min Threshold	Max Threshold
Absolute Value of Sobel in X direction	15	40	90
Absolute Vallue of Sobel in Y direction	15	40	150
Magnitude of Sobel	15	20	150
Direction of Sobel	15	0.4	1.2
HLS Saturation	N/A	190	255

Table 1: Filter Thresholds



(a) Combined thresholded image (b) Region of interest image

Figure 3: Example of thresholded binary image

3.3 Perspective Transform

The next step was to apply a perspective transform on the image to transform the lane markings to a bird's eye view. This transform will help us to calculate the lane curvature. I used OpenCV's `warpPerspective` function and the code can be found in section 4, lines 4-7 of the second cell under the "Pipeline" section, and line 43 of the last cell. The following is an example of the perspective transformation:

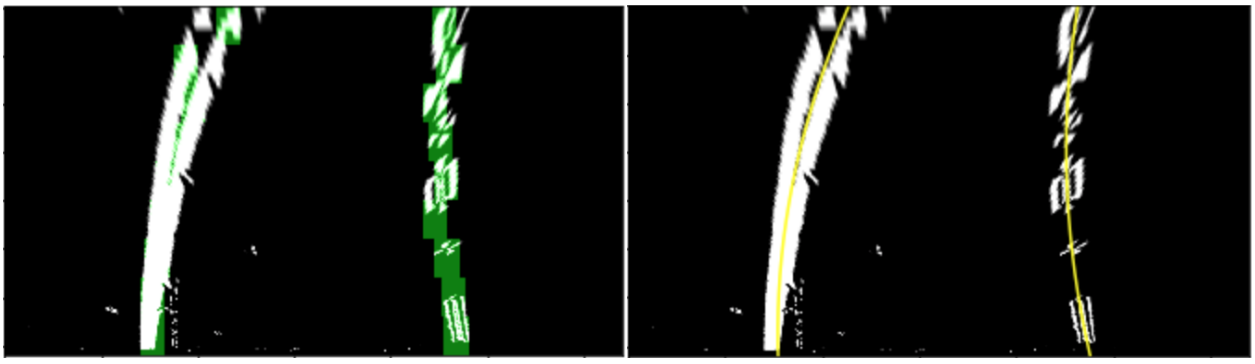
(a) Before perspective transform (b) After perspective transform

Figure 4: Example of perspective transform

3.4 Fitting a Polynomial to Lane Pixels

Once I have transformed the image to a bird's eye view, I detect the lane pixels in the image and fit a quadratic function to the pixels. For the first image, I used the convolution approach, where I sliced the image into 9 vertical layers and first found the maximum convolution signal with the a small window template to find the starting places for the left and right lane. Then, for each following layer, I search in a margin around the past left lane centroid and the past right centroid, find the maximum convolution signal, and mark that as the left lane centroid and right lane centroid. If no lane pixels are found in the layer, I just keep the past left and right centroids. Then, I search for nonzero pixels around the lane centroids and fit a quadratic function to the pixels using `np.polyfit`. The code for this is in section 5 of `lane_finding.ipynb` and lines 50-51 of the last cell.

For every following image, I simply do a margin search for nonzero pixels around the best polynomial fit so far for the left and right lanes. Then, I fit a quadratic function to the new pixels with `np.polyfit`. The code for this is in section 5 of `lane_finding.ipynb` and lines 52-53 of the last cell. Figure 5 shows an example of the lane centroid search result for the first image and an example of fitting a quadratic to the lane pixels.



(a) Convolution window search result (b) Polynomial fitting result

Figure 5: Example of lane pixel detection and polynomial fitting

3.5 Radius of Curvature and Offset from Center

Next, I calculate the radius of curvature by transforming the lane pixels into meters, refitting a polynomial to the lanes in meters, and calculating the radius as follows:

$$r = \frac{[1 + (\frac{dx}{dy})^2]^{\frac{3}{2}}}{|\frac{d^2x}{dy^2}|} \tag{1}$$

With a quadratic fit for the lanes, where $x = ay^2 + by + c$, Eq. 1 then becomes:

$$r = \frac{[1 + (2ay + b)^2]^{\frac{3}{2}}}{|2a|} \tag{2}$$

The code for radius curvature calculation is in section 6 of `lane_finding.ipynb` and line 56 of the last cell.

The offset of the car from the center of the lane was calculated by finding the halfway point between the first left lane pixel and the first right lane pixel, comparing this to the width of the image divided by 2, and then converting to meters. It is assumed that the camera is mounted at the center of the car. The code for finding the offset can be found in section 6 and line 57 of the last cell of `lane_finding.ipynb`. For example, for the image displayed in Fig 4b, the radius of curvature was 508.06 m for the left lane boundary and 599.81 m for the right lane boundary and the offset from center was -0.222 m.

3.6 Plotting the Lane Area

Finally, the last step was to warp the lane boundaries back onto the image. I do this by creating a polygon of the left and right points using OpenCV's `fillPoly` function. Then, I draw the polygon on a warped blank image and warp the polygon image back to the original image perspective with OpenCV's `warpPerspective` function and the inverse perspective matrix found earlier in the pipeline. I then combine this image with the original image using the OpenCV `addWeighted` function. An example with the lane fit plotted on the original image is shown below:

Figure 6: Warping lane boundaries back onto image

4 Output Video

My output video can be found in `lane_video.mp4`.

5 Discussion

My pipeline would fail in drastically different lighting conditions than the one given for this project since I use filters on image gradients and color saturation thresholding. Any sort of lighting that would make it difficult to distinguish the lane markings from the road such

as driving at nighttime or having part of the road heavily shaded would cause a failure. A possible solution to this would be to incorporate another sensor like Lidar that could pick up the lane marking reflectors on the road.

Another possible failure case is if the lane markings are very windy or strangely shaped. In these cases, my quadratic fit of the lane pixels would fail since the lane does not follow a quadratic model. I could use a higher degree of polynomial for the polynomial fit to accommodate potential shapes, but this may not be a good solution because higher degree polynomials may introduce more noise and fluctuation in the fit. Additionally, if the lanes don't have well-defined and nicely painted lane markings, my filters would not work very well and it would be difficult to extract the lane boundaries. Once again, another sensor that could detect lane reflectors or some other kind of reasoning such as following a vehicle in front of you would help with this issue.

Finally, another confusing case would be if the car changed lanes. This would cause some confusion in the lane boundary tracking and the car would need to take some time to reset and recover the new lane boundaries. A possible solution would be to have the car send a signal to the pipeline when it performs a lane change so that the pipeline can immediately reset.