Udacity Self-Driving Car Nanodegree
Term One: **Computer Vision** & **Deep Learning**
Project Four: **Advanced Lane Detection**

Nishan Srishankar
Worcester Polytechnic Institute

2017

1 Introduction

This project deals with creating a more robust pipeline for lane detection. One of the fundamental aspects in implementing a self-driving car is perception and deals with how a car would identify aspects of interest that are necessary for its functioning or could impede its performance. This could include traffic signs, signals, neighboring vehicles, lane lines, and even chaotically behaving humans or cyclists.

The algorithm implemented for Project One utilized OpenCV Color-Palette selection, Canny Edge Detection, Region-of-Interest selection, and Houghline feature extraction/ transformation. It was found that this pipeline wasn't very stable during steep curves, sections of the road with a color-gradient, or during night-time driving.

2 Methodology & Implementation

This project is subdivided into seven steps which are explained in detail below:

2.1 Camera Calibration

To account for radial and tangential distortion caused by convex lenses in cameras, images are calibrated using standard chessboard images with tiles arrayed in a 9-column, 6-row pattern. Built-in OpenCV functions such as *findChessboardCorners* and *drawChessBoardCorners* are implemented on sample grayscale chessboard images which outputs successful returns and detected corners. It was found that corners in three chessboard images weren't detected.

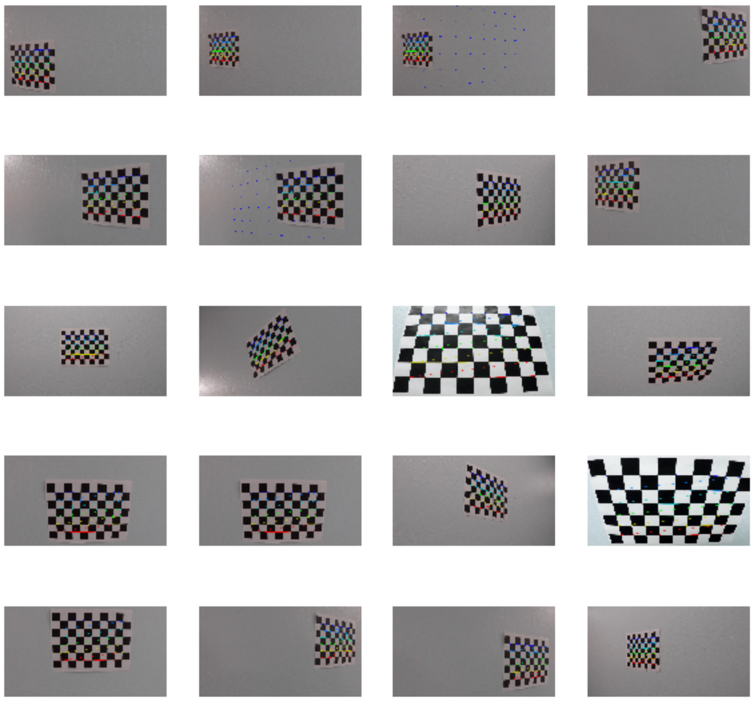


Figure 1: Chessboard Calibration by finding corners

2.2 Distortion Correction

The existing *calibratecamera* function was used to take in corners saved in an image-array ”detected” using a camera, and map these to a 3D object array obtained using Numpy’s *mgrid* since the pattern repeats regularly and returns an undistortion matrix and coefficients. The matrix and coefficients are saved in a pickle file to prevent the need for recalculation. This is used in *undistort* to obtain a corrected image, which can be seen in Figure 2 on chessboard images without detected corners and Figure 3 on a sample driving image. There is noticeable correction in the chessboard image, and slight correction in the driving image (recognizable by looking at the car’s hood).

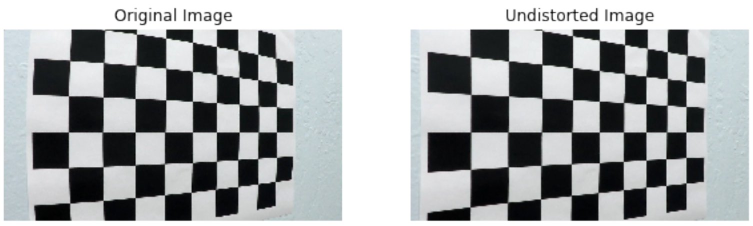


Figure 2: Undistortion on chessboard images

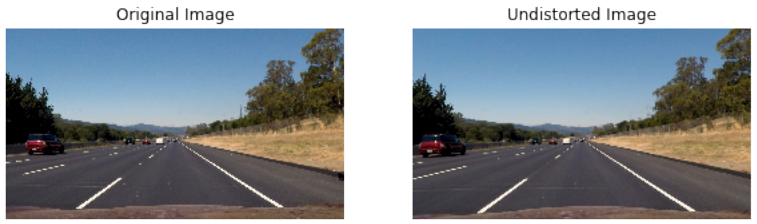


Figure 3: Undistortion on a real-world driving image

2.3 Color- and Edge- Thresholding

The next implementations of the pipeline are performed on undistorted images obtained from the previous steps. One of the main focus points of this project was color & edge thresholding. For color thresholding, the images are converted into HLS and HSV colorspace. A combination of HLS and HSV with differing high and low thresholds are used to detect yellow and white lane lines individually. The Luminosity-channel and Saturation-channels are utilized to create a binary image. The high and low channels for color thresholding are manually tuned.

While color thresholding is performed, simultaneous edge-thresholding is done as seen in Figure 4. The Sobel filter (in Canny-edge detection) was blindly used in Project One. Here, multiple Sobel filters are used to detect gradients. First, absolute Sobel gradients are calculated in the x- and y- directions, following which the magnitude is found. Finally, a directional gradient is implemented. These are done individually (with different minimum and maximum thresholds), and combined to form an edge thresholding image. Directional and magnitude sobel-gradients is one of the new features implemented here to find gradients (or "lines") that meet certain magnitude **and** angle requirements (as some lanes do). Green regions in Figure 4 represent pixels that satisfy edge-thresholding, and blue regions represent pixels that satisfy color-thresholding.

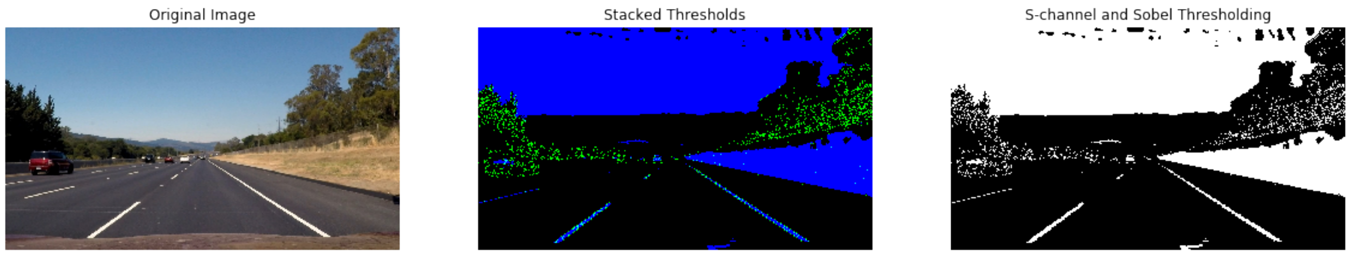


Figure 4: Color and Edge Thresholding Visualization

2.4 Warp Perspective

The second focus point of this project is warping perspectives to obtain a birds-eye view of the lane ahead. This is done by choosing a trapzezoidal source region of interest (with four points), and mapping it to a rectangular region of interest as seen in Figure 5 using *getPerspectiveTransform* and *warpPerspective*. The output of this transformation can be seen in Figure 6.

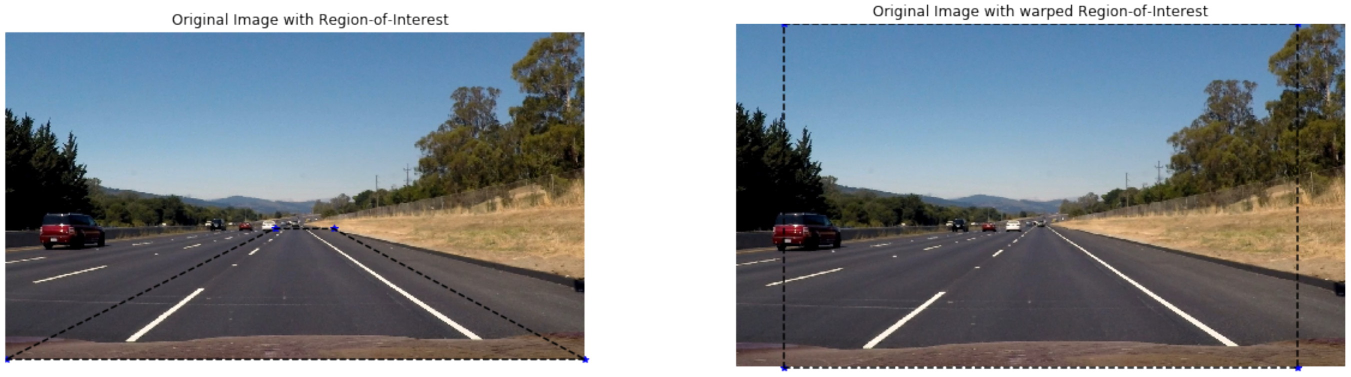

Original Image with Region-of-Interest
Original Image with warped Region-of-Interest

Figure 5: Birds-eye perspective transformation

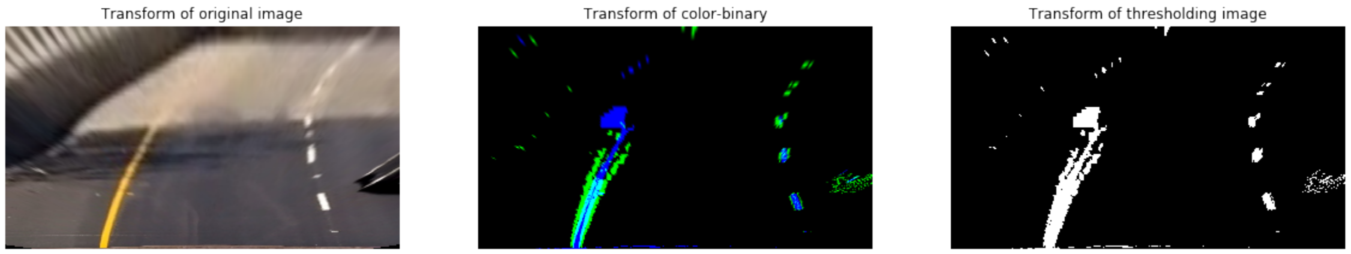

Transform of original image
Transform of color-binary
Transform of thresholding image

Figure 6: Warped view of a binary image

2.5 Line Fitting

A much more complicated algorithm is utilized to detect and fit lane lines. For the algorithm to know where to start searching for lane lines, a histogram of the bottom-quarter of the binary image is obtained and seen in Figure 7. The left maximum which represents the starting point of the left lane is found between the left-quaterpoint and midpoint of the lane. The right maximum which represents the starting point of the right lane is found between the midpoint and right-quarterpoint of the lane. The initial search is constrained such that the starting point isn't confused by incorrect detection of neighboring lane lines or traffic on the road for instance.

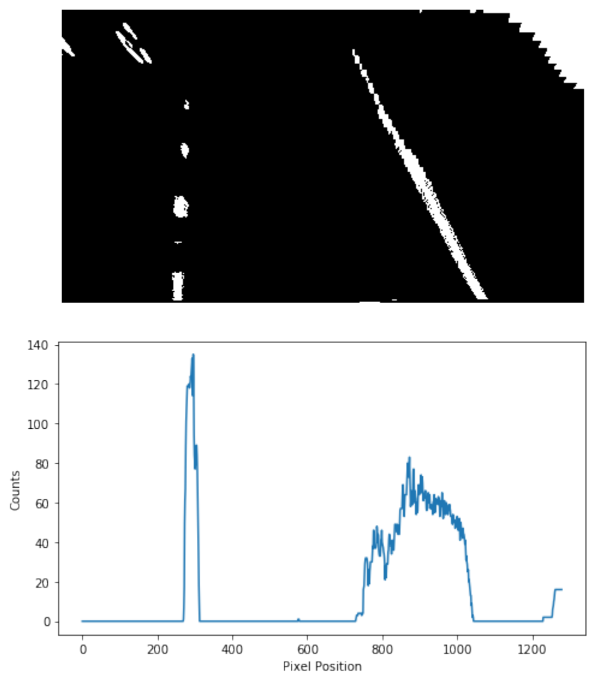


Figure 7: Histogram of binary image

Pixels representing lanes are found using a sliding-window algorithm (Figure 8). If there is a non-zero detection within the margins of an existing window, the center of the window is moved to that new position. This is repeated for the number of windows chosen by the user. However, if a previous match is found for both the left and right lanes, a modified version of sliding window is used for a small region around the fitted line to find the next polynomial (as seen in Figure 9).

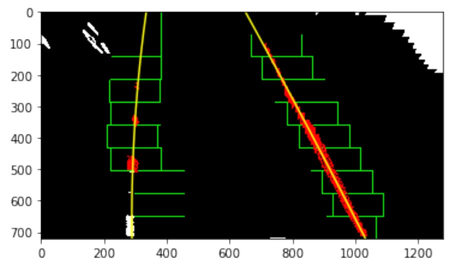


Figure 8: Sliding Window Fit

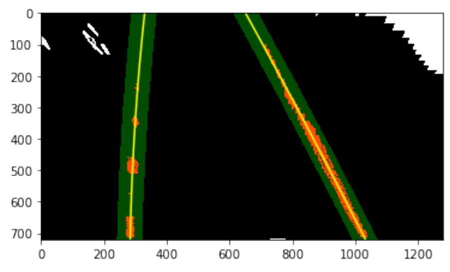


Figure 9: Sliding Window Fit with previous search

The pixels detected using the Sliding window search is fit using a second-order polynomial curve using *polyfit*.

2.6 Curvature and Offset Analysis

Since the lane lines are fit using second-order polynomials of the form shown in Equation 1, the curvature can be calculated by using Equations 2, 3. In Equation 3, A and B are two of the coefficients obtained from the second-order polynomial polyfit.

$$f\left(y\right) = Ay^2 + By + C \tag{1}$$

$$R_{curve} = \frac{\left[1 + \left(f'\left(y\right)\right)^2\right]^{1.5}}{\left|f''\left(y\right)\right|} \tag{2}$$

$$R_{curve} = \frac{\left[1 + \left(2Ay + B\right)^2\right]^{1.5}}{\left|2A\right|} \tag{3}$$

In addition to calculating the radius of lane-curvature, the offset of the car from the center of the road, and approximate velocity can be calculated as below. Since the car camera is centered on the hood of the vehicle, the location of the car with respect to either lane edge can be found by subtracting the midpoint of the image with the average left- and right-base value (from the polynomial fit). The velocity of the car can be found by detecting how fast centroids of a lane are detected in a video stream (and knowing the dimensions of such a lane- approximately 10ft).

These calculations required the conversion from pixel-space to real-world-space. The ratio was ballparked from the transformed image and knowing US highway lane dimensions, and is a source of inaccuracy.

2.7 Overlay

The final step of the procedure was to remap the overlayed polynomial lane lines (and lane region of interest) onto a blank image. This is done by using *warpPerspective*, by using an inverse warp-matrix (as opposed to using a warp matrix in the Warp Perspective subsection). The final output is seen in Figures 10 and 11.

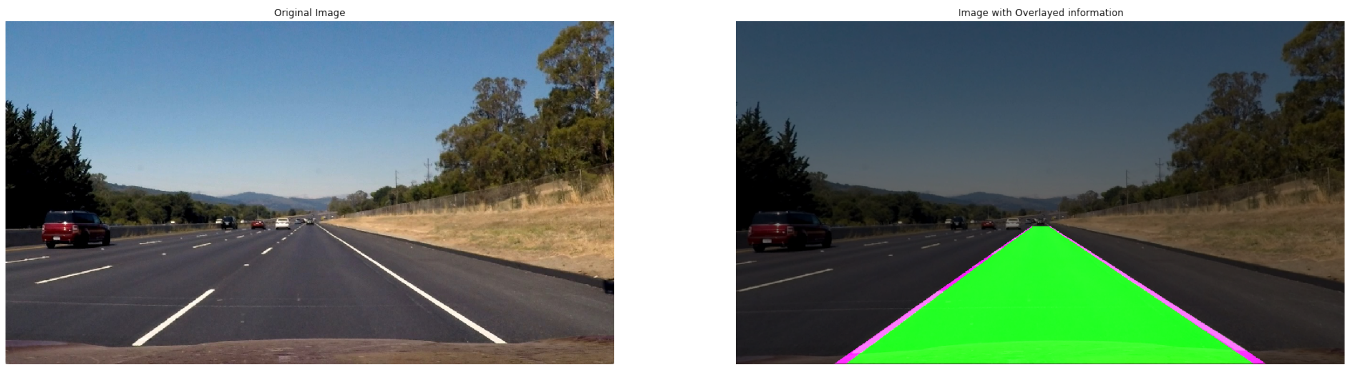
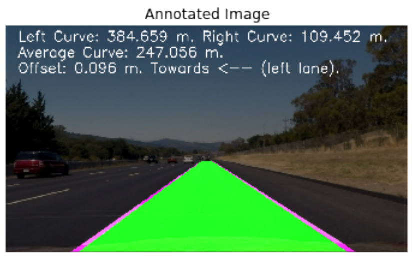


Figure 10: Overlaying polyfit lane lines and lane region-of-interest



```
Left Curve: 384.659 m. Right Curve: 109.452 m.
Average Curve: 247.056 m.
Offset: 0.096 m. Towards <-- (left lane).
```

Figure 11: Annotated original image

3 Results & Improvements

One of the modifications done to create a video pipeline (compared to action on simple images) was the implementation of a class to save useful information from the previous frame of a video, and to create sanity checks to ensure that large differences between road curvature and lane fit coefficients between subsequent images are ignored. Should there be a large difference, the sanity check fails, and hence data from the previous image is utilized for the next frame.

The output obtained using the pipeline for the main video and supplementary videos can be seen on Youtube.

As can be seen, this pipeline performs much better than the one implemented for Project One. However, there are still shortcomings such as sudden curvature changes, extremely faded lanes (or frames with extremely bright ambient light). The pipeline also assumes that the car sticks to one particular lane throughout. It should be noted that the pipeline here worked really well on the challenge video from Project One.

Sudden curvature changes are places where the road curves multiple times within the region of interest in concurrent frames. Another point of failure is if there is only one formal, detected lane line (such as on a dirt road), and hence the algorithm chooses random points to be taken as another lane resulting in an arbitrary polynomial and region of interest. Additionally, steep/inclined roads would result in incorrect perspective warping and hence cause failure in the pipeline. It can also be seen that there are small regions where polyfit results in wobbling lane lines (and overlayed regions of interest) despite no significant change of color or gradient. This can be due to roads that are not completely flat.

The algorithm can be improved by having a dynamic region of interest/perspective transforms, fitting a higher-order polynomial to lane lines or segment regions of interest based on distance from the car, and future lane-line prediction. This is important as initial calculations for color/edge thresholding, selection of a region of interest etc. were done on a single "perfect" image (with straight lane lines or slightly curved lane lines). Finetuning these values as well as sanity check upper-bounds could also improve the performance of this pipeline.