# Advanced Lane Line Finding

December 18, 2018

The goals / steps of this project are the following:

- 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.

## 1 Camera Calibration

The camera calibration was done at the beginning of the notebook. I followed and used the code from given examples and output the results to folder **/camera_cal_outputs**. Here are a couple examples:

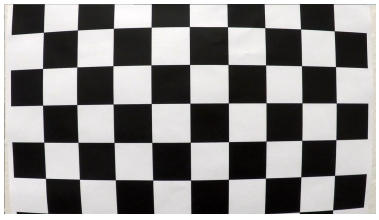

Figure 1: original image

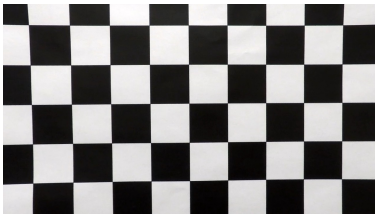

Figure 2: undistorted image

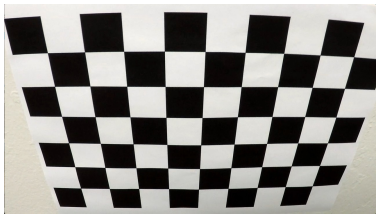

Figure 3: original image

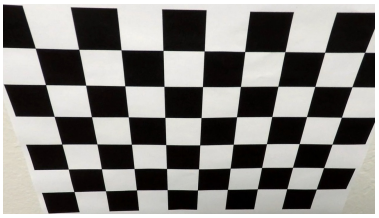

Figure 4: undistorted image

# 2 Pipeline (single images)

## 2.1 An example of a distortion-corrected image

The function **pipeline(img)** includes all steps for image processing. The first step was to undistort the images. The actual undistorted images are under folder **/output_images**, named **undist_*.jpg**.

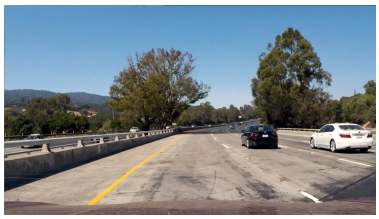

Figure 5: original image

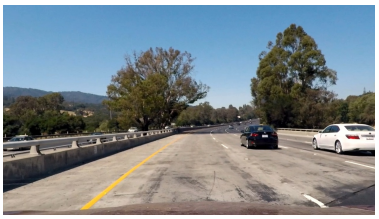

Figure 6: undistorted image

The difference is not obvious, but we can still see that the undistorted image was stretched a little.

## 2.2 Thresholded Binary Image

Binary thresholding was done in the function **threshold_bin_image(img)**. In this function I applied sobel x thresholding and S channel thresholding, then combined the two binary images to get the final image. Better results is possible with threshold parameters tuning and other color channels. Here are example outputs of my code.

Figure 7: binary image of test1

Figure 8: binary image of test2

The lane lines are pretty clear in the images.

## 2.3 Perspective Transformation

The function **perspective_transform_image(img)** did this. I picked the source points to be (570,460),(700,460),(260,680),(1050,680) and the destination points (450,0),(830,0),(450,img.shape[0]),(830,img.shape[0]), then the function called **cv2.getPerspectiveTransform(src, dst)** to get the transformation matrix and applied it to the input image.

Figure 9: perspective transformed image of test2

Figure 10: perspective transformed image of test3

The lane lines are clear and parallel in the images.

## 2.4 Finding Lane Pixels and Fitting Polynomial

The function **find_lane_pixels(binary_warped)** finds pixels that belong to the left lane and the right lane using sliding window method. Then the function **fit_polynomial(binary_warped)** calls it to fit a polynomial for each lane. Below is an example image with polynomial fit.

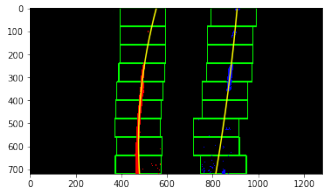

Figure 11: image with sliding windows and polynomial fit

If we already have a polynomial fit for each lane, then instead of calling **fit_polynomial(binary_warped)**, the function **search_around_poly(binary_warped, left_fit, right_fit)** is called. Instead of finding lane pixels from scratch, it searches for a fit within a margin of the previous fit.

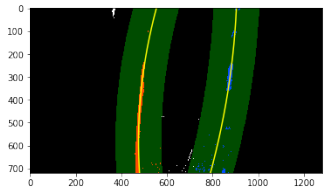

Figure 12: Polynomial fit from previous

## 2.5 Radius of Curvature and the Position of the Vehicle

Before calculating the radius of curvature, my code checked if the polynomial fits were valid - two lanes should be approximately parallel - and updated the lane objects accordingly. I kept 5 most recent valid fits. If the fits were valid, then the function **measure_carvature_real(binary_warped, left_fit, right_fit)** calculated the radius of curvature and the position of the vehicle in meters. If these values were valid - they should be relatively similar to the previous ones - we stored them to the lane objects. Finally the function **print_carvatureAndOffset(img, left_carvature, right_carvature, offset)** output these values to the original image.


Figure 13: test1


Figure 14: test2

# 3 Pipeline (video)

My video result is **project_video_output.mp4** under the parent directory. I think my code worked well on most of the frames, but there were wobbly lines, especially around 21s and 40s of the video, where the color of the pavement changed. I think the problem is with the binary thresholding part.