Advanced Lane Finding Project

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.

Rubric Points:

Note: all code references are to adv_lanefinding.py, this writeup will only reference line numbers

Writeup

The following pdf should explain how each rubric item was addressed.

Camera Calibration

Function calibrate_camera (lines 8-37) takes in a list of image locations and outputs the camera matrix and distortion coefficients necessary to undistort images taken by that hardware.

The chessboard is a known pattern by cv2 functions, so that is what was used to calibrate. The cv2 function findChessboardCorners was used on each image after changing to grayscale, and the corners were associated to generic object points in the real world. These pairs were then passed into the cv2 function calibrateCamera, and the resulting output could be later used by the cv2 function undistort to correct any skew in the images. Below is an example of undistortion using calibration3.jpg:

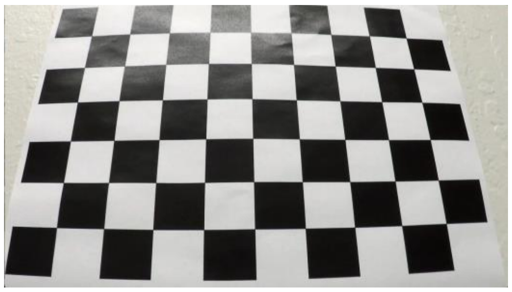


Figure 1 Raw Chessboard Image

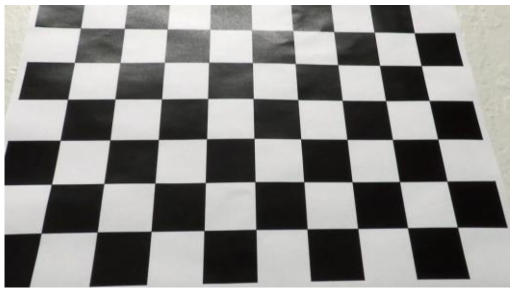


Figure 2 Undistorted Chessboard Image

Pipeline (single images)

1. Distortion-corrected image

Using the matrix and distortion coefficient results from camera calibration, the test images were undistorted using the cv2 function undistort. (line 256) Here is an example:

Figure 3 Raw

Figure 4 Undistorted

2. Creating a thresholded binary image using color and gradients

The function color_gradient_thresh (lines 40-73) was called for the next step of the pipeline. This function takes an undistorted image as input, as well as thresholds for the S, H, and R color channels and the X gradient. Through some manual testing, the default values were set to highlight the lane lines in the test images. For the color channels, each were isolated and compared to the thresholds given. The array created is a 1 if it is within the values, and 0 otherwise. The same approach was taken for gradients. Then, the pairwise combination of the color masks along with the results of the gradient mask resulted in a decent representation of the lane lines. Below is an example.

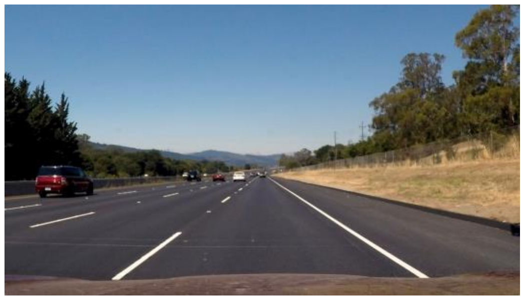


Figure 5 Raw

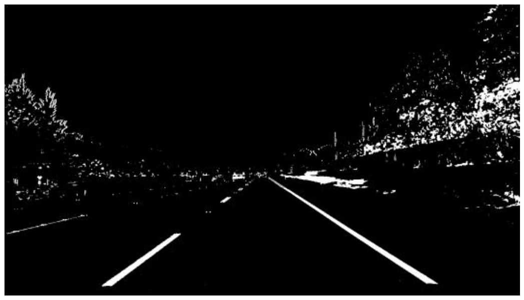


Figure 6 Binary Image

3. Perspective transform

The warp function (lines 75-89) performs the perspective transform on the image. The input is an image and the output is a warped image based on the hardcoded points in the function. These points were selected by hand under the assumption that the lanes would fall in about the same trapezoid. The points chosen were fed into the cv2 function getPerspectiveTransform to get the transformation matrix, and that is then fed into warpPerspective to get the warped image. Below are two examples of images that were warped by the pipeline.

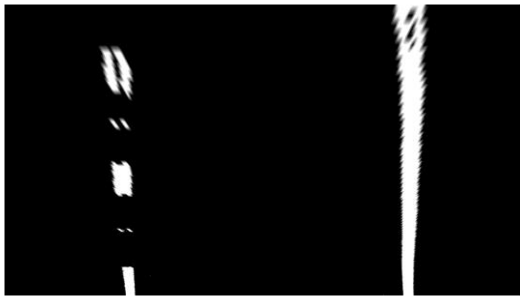


Figure 7 Perspective Transform of Fig 6

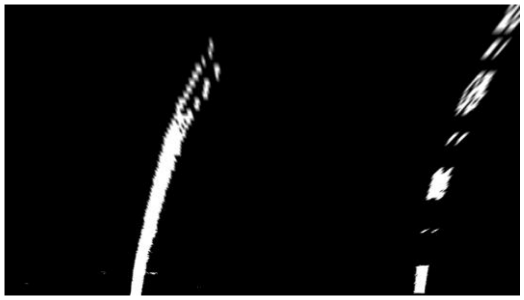


Figure 8 Perspective Transform of Curved Lane

4. Fitting a polynomial to the lane lines

The hefty function get_polyfit (lines 108-173) takes in the warped binary image and outputs the best fit left and right lines in both pixels and meters. The approach taken was a very straightforward sliding window as described in the lessons.

The image is first split up into 9 rows. The goal of the sliding window is to start from the bottom and center over where the most pixels exist. Then moving up one row, within a given margin, the box will recenter itself around the location of the most pixels. It does this until it reaches the top. The initial x location for the first windows are determined by the highest point of the histogram of the bottom half of the image.

When these sliding window is complete, the collected points are fed into numpy's polyfit function with a degree of 2. This will return the best fitting quadratic for the points given. Below are two plots of the best fit lines in yellow, along with the 9 sliding boxes in green and the left and right non zero points.

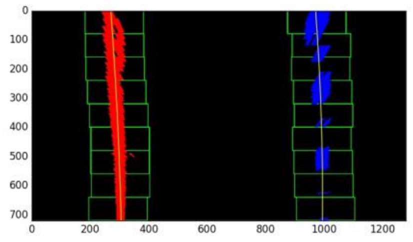


Figure 9 Polyfit for straight lanes

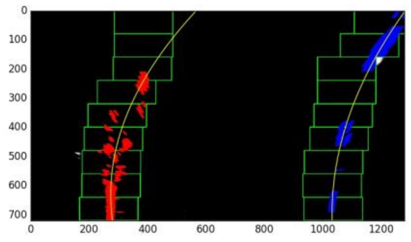


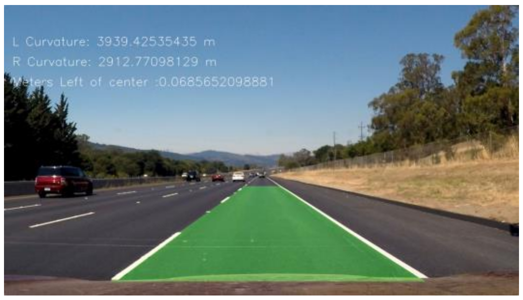
Figure 10 Polyfit for curved lanes

5. Curvature and vehicle position

The polyfit function above returns the best fit lines in both pixel units and meters. The functions get_xm_per_pix and get_ym_per_pix (lines 205-211) are used for pixel to meters conversion. The function get_curvature_meters (lines 213-225) takes in the meter-unit fits and calculates the curvature of the lanes. Then, the functions get_lane_center and get_center_offset_meters (lines 222-228) find the midpoint of the detected lane, and how far that is from the center of the image. It is then converted into meters for display on the final image.

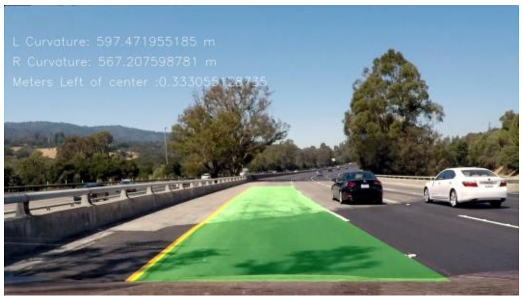
6. Overlaying lane area

The final step in the pipline is to draw the lane. Draw_lane (lines 230-247) uses the pixel-unit fit lines to determine border points for a polygon. This polygon represents the area of the detected lane. It is fed into the function unwarp (lines 91-105) to perform the reverse perspective transform. Additionally in the pipeline, the curvature and vehicle position text is overlaid (lines 270-276). Final image examples:



```
L Curvature: 3939.42535435 m
R Curvature: 2912.77098129 m
Meters Left of center :0.0685652098881
```

Figure 11 Lane area for straight lanes



```
L Curvature: 597.471955185 m
R Curvature: 567.207598781 m
Meters Left of center :0.333055128735
```

Figure 12 Lane area for curved lanes

Pipeline (video)

The video is located in the same github repo that this pdf is in, titled “project_video_output.mp4”. The pipeline used to create that was the same approach as was used for the test images (lines 279-304).

Discussion

Things that can be improved:

- Perspective transform points are hardcoded and hand chosen. There should be a way to do this automatically, but that would almost require that something like lane lines already be detected.
- The color thresholds are good enough for these test images, and the project video, but it fails to detect lines on the challenge videos. Some more investigation is needed in what combination of channels and gradients can better detect more general lane lines.
- The project video turned out surprisingly smooth without needing to add additional functionality on top of the test image set. Also, the sliding window approach was used for every frame, which may have slowed down processing a bit. The video would take about a minute to create.
- Curvature for straight lines is a very high number. It may be beneficial to find the max realistic curvature of a lane line, and hide the result if it’s over that number.