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.

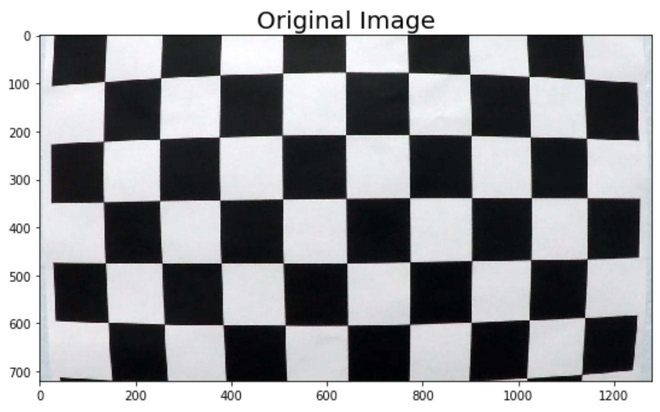
Camera Calibration

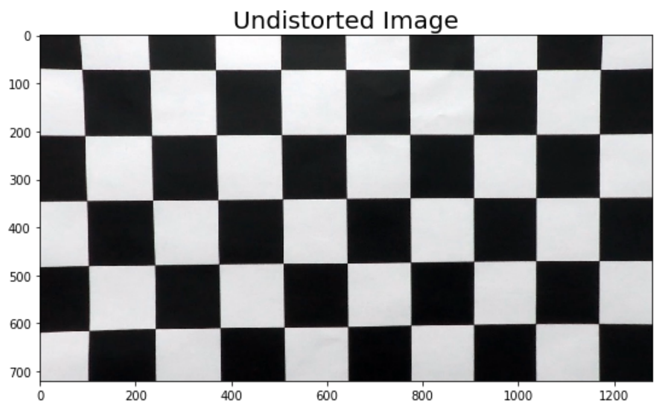
The code for this step is contained in the first code cell of the IPython notebook located in "./P4.ipynb"

The motivation here is that because the camera used in the car is a camera with a lens and not a pinhole, there will be some distortion of the image.

I start by preparing "object points" (objpoints), which are 3d points in real world space. The chess board has 9x6 squares, and I create a grid based on the corners of those squares. I use the OpenCV findChessboardCorners() function to find the "image points"(imgpoints), which are the 2d points of the image.

Based on the object and image points, I compute the camera calibration and distortion coefficients using the OpenCV calibrateCamera() function. I applied this distortion correction to the test image using the OpenCV undistort() function. here is the comparison of original and undistorted result:





Pipeline (single images)

For each road image, I applied the undistort() function from the previous Camera Calibration step.

1. Thresholding on both gradient and saturation channels

This step is shown in the P4.ipynb notebook in the gradientThreshold() function

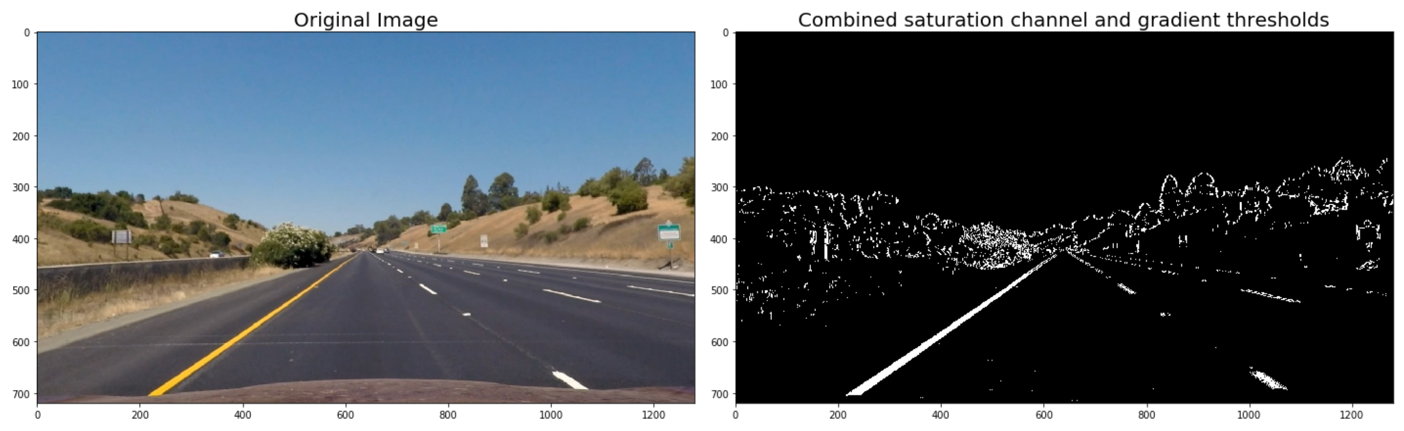
1. Thresholding on the saturation channel

The image is converted to the HLS color space. The saturation channel showed the best recognition of the lane lines. A threshold of 180 to 255 is used to filter the image. All pixels with values outside this range are not recognized.

2. Thresholding on the gradient

The image is converted to grayscale. The sobel function is applied in the x direction. This is scaled to the range of pixel values. All values between the threshold of 30 to 100 are recognized as possible lanes.

3. The two results above are combined into a single binary image. The comparison of the original and binary results are shown below:



2. Perspective Transform

This step is shown in the P4.ipynb notebook in the perspectiveTransform() function

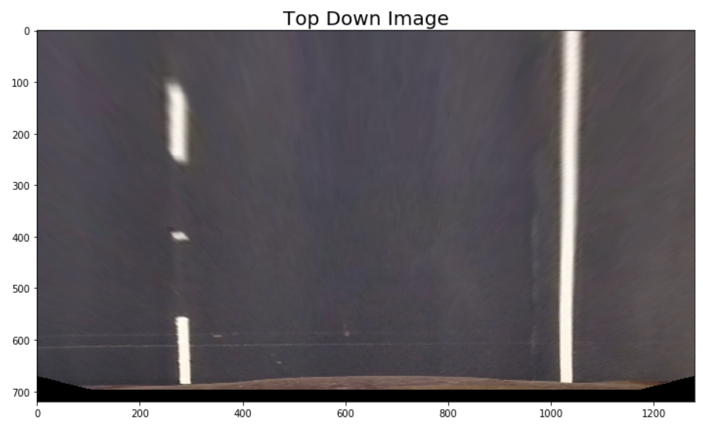
1. Source and destination points

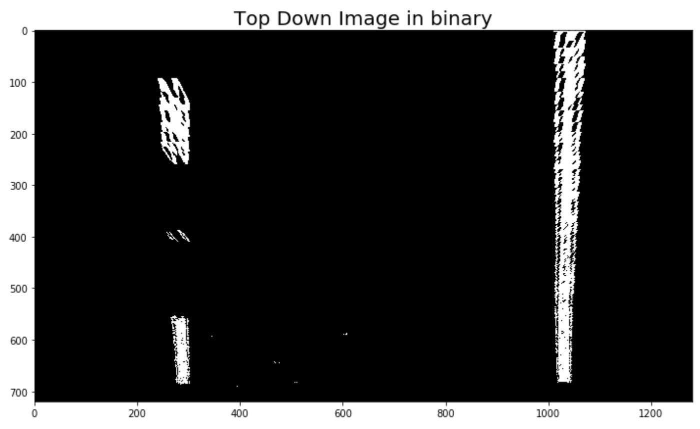
The 4 source points for the transformation were picked manually using the straight road image: test_images/straight_lines1.jpg. The destination points were then mapped evenly by fixing the bottom points and adjusting the top points to form a square. Here are the points that were used:

Source	Destination
560, 475	275, 100
725, 475	1025, 100
275, 670	275, 670
1025, 670	1025, 670

2. Perspective transform step

I used the OpenCV getPerspectiveTransform() function to get a transform matrix, M. This transform matrix, binary, original image were inputs into the OpenCV warpPerspective() function. The original image is used just to confirm the transformation. Here is the transformation of the straight road, we can see that the lines are roughly parallel:





3. Identification of lane pixels and fitting a line through them.

This step is shown in the P4.ipynb notebook in the detectLanes() function

1. Histogram

The rows of the binary image are summed and used to create a histogram of the image. There will be two large peaks in the histogram of the image. The left peak will correspond to the left lane. The right peak will correspond to the right lane. These are used as starting points in the sliding window detection of the lane pixels.

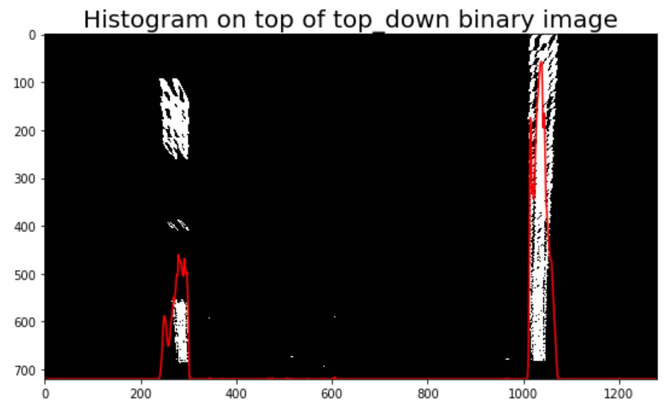
2. Sliding windows

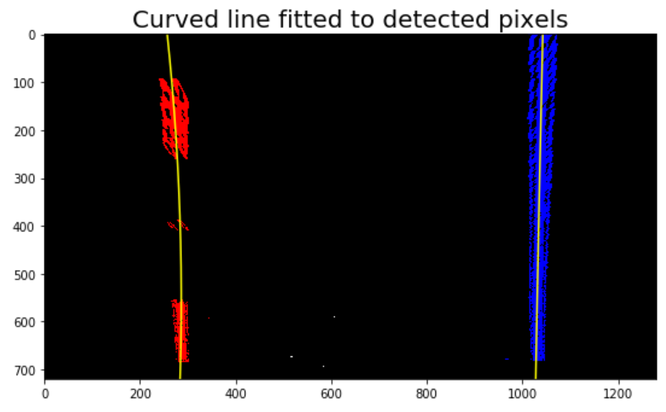
The image is cut into 8 windows for detection. Left and right lanes are detected separately. Starting at the starting points picked previously, the non-zero pixels are selected as lane pixels with margin of +/- 100 pixels. The indexes of the lane pixels are added to lists (good_left_inds and good_right_inds). The starting point is updated to an average if more than 10 pixels are found. All pixels found are aggregated for the next step.

3. Polynomial fitting

The Numpy polyfit() function is used to fit a 2nd order polynomial through the detected right and left lane pixels.

The histogram result, detected lane pixels, and the fitted lines are shown here:





4. Curvature and offset from center

This step is shown in the P4.ipynb notebook in the getCurvature() and getCenterOffset() functions

1. Curvature

The right and left curvature is determined from the polynomial using the Radius of Curvature equation (http://www.intmath.com/applications-differentiation/8-radius-curvature.php). The equation is evaluated at the bottom most points (or closest to the car). This curvature is in terms of pixels. The scaling factor to meters is given by the lecture as 30 meters per pixel in the y direction and 3.7 meters per 700 pixels in the x dimension. The curvature values are then displayed in meters. A center curvature is estimated as the average of the left and right curvature values.

2. Offset from Center

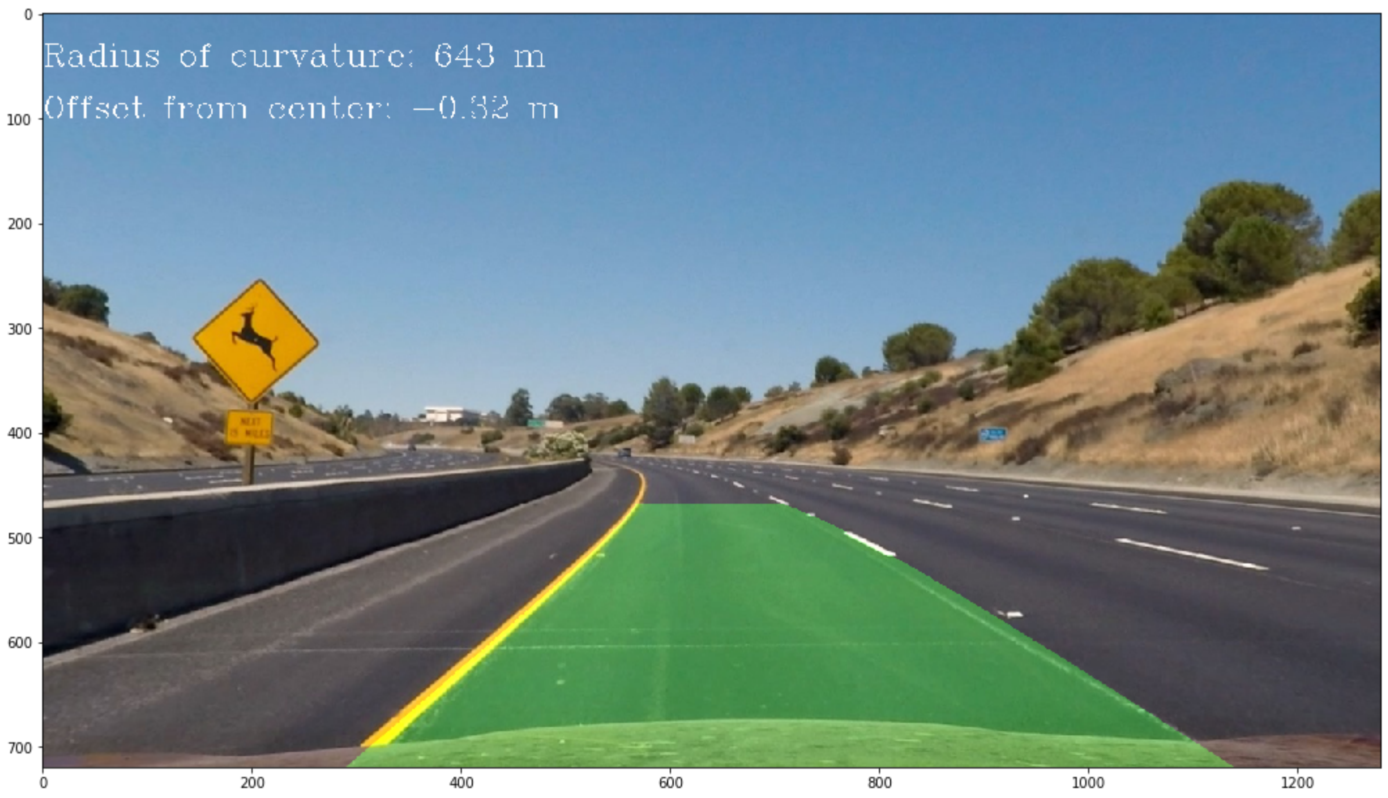
There is an assumption that the camera is mounted at the center point of the car. I compute the offset to the center based on the difference in the distance between the two lines at the bottom of the image and the center of the image. This is displayed in the final picture in terms of meters.

5. Visualization of the detected lane back onto the original image and perspective

This step is shown in the P4.ipynb notebook in the getLane() function

1. Drawing and warping the perspective back to the original view

A polygon is generated using the top down view and bounded by the lines detected previously. The transformation done previously to generate the top-down view is undone for the polygon. This means that the original source and destination points are reversed as inputs into the OpenCV warpPerspective() function. This image is then combined with the original road image. Here's an example picture of the output of this step:



6. Bounds checking

The previous coefficients detected for both left and right lines are stored. If the difference of the derivative of the left and right polynomials is greater than 0.60, then the previous polynomial is used instead. The out of range polynomial is discarded. The rationale for this is that the two lines are expected to change at roughly the same rate. This is done in the checkCurves() function.

7. Processing video

The input video, project_video.mp4, is processed frame by frame using the process_image() function. The output video is below:

The output video can be found at https://youtu.be/DVG9e6Tgs9o (https://youtu.be/DVG9e6Tgs9o) or at output_images/output.mp4

Discussion

There's a small amount of wobble on the right line because of the broken white line. The detected right line will stay in the lane boundaries. This could be improved by using a previous frame's detected pixels and having it weigh into the result.

I also tried to filter based on curvature. Very large curvatures indicate straight lines, which means that they can't be filtered out. The difference in the previous and current curvatures was attempted. This was not used in the final algorithm as it didn't improve the algorithm significantly.

The lane detection may fail if there are multiple vertical lines that can fool the algorithm into thinking there are false lane boundaries. This could be compensated by more graphic and polynomial filtering.

The algorithm may fail with significant curvy roads and also other moving vehicles with light colors. Both can be will need more computer vision techniques to deal with. If other vehicles can be detected, then they can be omitted. This would need machine learning techniques.

The algorithm takes about 2 mins 30 seconds to run on my computer for a 50 second clip. I would improve the performance so that we can run this in real-time. This may include not having to use sliding windows for every frame. Most of the lane detections would be based on the previous polynomial coefficients and a few of the detections would use sliding windows.