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

Image distortion occurs when a camera looks at 3D objects in the real world and transforms them into a 2D image; this transformation isn't perfect. Distortion actually changes what the shape and size of these 3D objects appear to be. So, the first step in analyzing camera images, is to undo this distortion so that you can get correct and useful information out of them.For this first we need to compute the camera calibration matrix and distortion coefficients from given set of chessboard images.

In this step, I used the OpenCV functions findChessboardCorners() and drawChessboardCorners() to identify the locations of corners on a series of pictures of a chessboard taken from different angles. The code for this step is contained in the IPython notebook located in "camera_calibration.ipynb".

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. Then I saved objpoints and imgpoints in camera_cal/calibration_pickle.p

Pipeline

1. Distortion-corrected image.

First I read saved camera calibration matrix and distortion coefficients from pickle file

I applied the distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

2. create a binary image.

I used a combination of color and gradient thresholds to generate a binary image.

The lane lines are more or less vertical so i used 2 kinds of gradient thresholds, along the X axis using sobel function and directional gradient with thresholds of 30 and 90 degrees.

Just using above threshold white lines were detected properly but yellow lines were not. SO for detecting yellow lines I used R & G channel thresholds and S channel threshold. Shadows in image was sometime mistaken as lane lines so to avoid that I used L channel threshold

Here's an example of my output for this step.

3. A perspective transform

The code for my perspective transform includes a function called `get_warped_image()`. The ` get_warped_image ()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I manually examined a sample image to extract the source points. Destination points are chosen such that straight lanes appear more or less parallel in the transformed image.

This resulted in the following source and destination points:

Source : bottom_left = [384,675]

bottom_right = [896, 675]

top_left = [581, 477]

top_right = [699, 477]

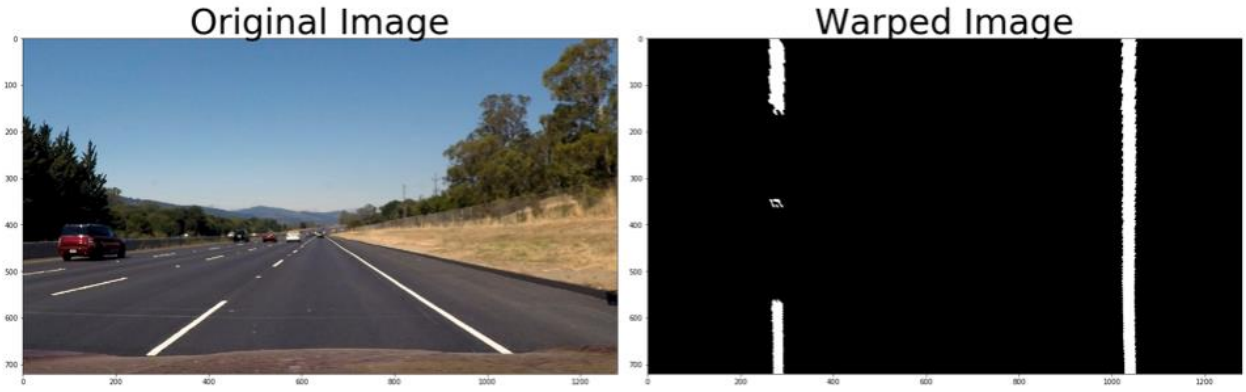
Destination : bottom_left = [384,720]

bottom_right = [896, 720]

top_left = [384, 0]

top_right = [896, 0]

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



4. Identify lane-line pixels and fit their positions with a polynomial

Histogram

The peaks in the histogram tell us about the likely position of the lanes in the image. So I take a histogram along all the columns in the lower half of the image.

Sliding window search

I then perform a sliding window search, starting with the base likely positions of the 2 lanes, calculated from the histogram. I have used 10 windows of width 100 pixels. The x & y coordinates of non zeros pixels are found, a polynomial is fit for these coordinates and the lane lines are drawn. This is implemented as find_lane_pixels() in the code.

The sliding window method is applied to the first frame only. After that we expect the next frame to have a very similar shape, so we can simply search within the last windows and adjust the "centroids" as necessary. In this section we search around a margin of 50 pixels of the previously detected lane lines. This is implemented as search_around_poly () in the code.

Figuring out bad frames

There will be some frames where no lanes will be detected or the lanes might not make sense. We determine the bad frames if any of the following conditions are met:

- No pixels were detected using the sliding window search or search around the previously detected line.
- The average gap between the lanes is less than 0.7 times or greater than 1.3 times the globally maintained moving average of the lane gap.

Averaging lanes

The lane for each frame is a simple average of 12 previously computed lanes. This is done in the get_averaged_line() in the .

bad frame

If bad frames are detected perform a sliding window search again (this is done in the find_lane_pixels () method in the code). If this still results in a bad frame then we fall back to the previous well detected frame.

Finding radius of curvature and offset

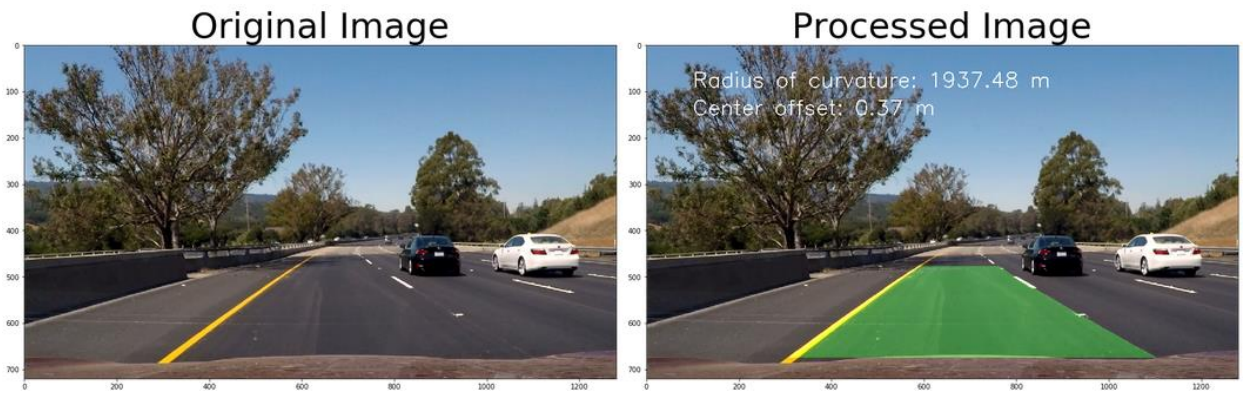
The radius of curvature is computed according to the formula and method described in the classroom material. Since we perform the polynomial fit in pixels and whereas the curvature has to be calculated in real world meters, we have to use a pixel to meter transformation and recompute the fit again.

I did this in measure_radius_of_curvature() in code.

The mean of the lane pixels closest to the car gives us the center of the lane. The center of the image gives us the position of the car. The difference between the 2 is the offset from the center.

5. Final result

I put all the above steps in a pipeline method in my code. After executing the pipe line the final result are as shown below



Pipeline (video)

Project video

Here's a link to my project video result

https://youtu.be/_JME_R6wmXE

Challenge video

Here's a link to my challenge video result

https://youtu.be/c69zAN7NcLY

Issues and Challenges

- I had to experiment a lot with gradient and color channel thresholding. Especially for non-well maintained or under construction roads, shadows cast by dividers, and tracks with very sharp curves. The lanes lines in the challenge and harder challenge videos were extremely difficult to detect as they were either too bright or too dull. For that I tried R & G channel thresholding and L channel thresholding
- The challenge video has a section where the car goes underneath a tunnel and no lanes are detected. To tackle this I had to resort to averaging over the previous well detected frames

Points of failure & Areas of Improvement

The pipeline fails for the harder challenge video. This video has sharper turns and at very short intervals. I think what I could improve is:

- Take a better perspective transform: Come up with the way to define the transformation trapezoid and come up with the appropriate source points for the perspective transform.
- I would have to start looking in implementing better solution for the edge or corner detection for sharper turns and the length of a lane is shorter as well for sharp change in light intensity