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

Writeup / README

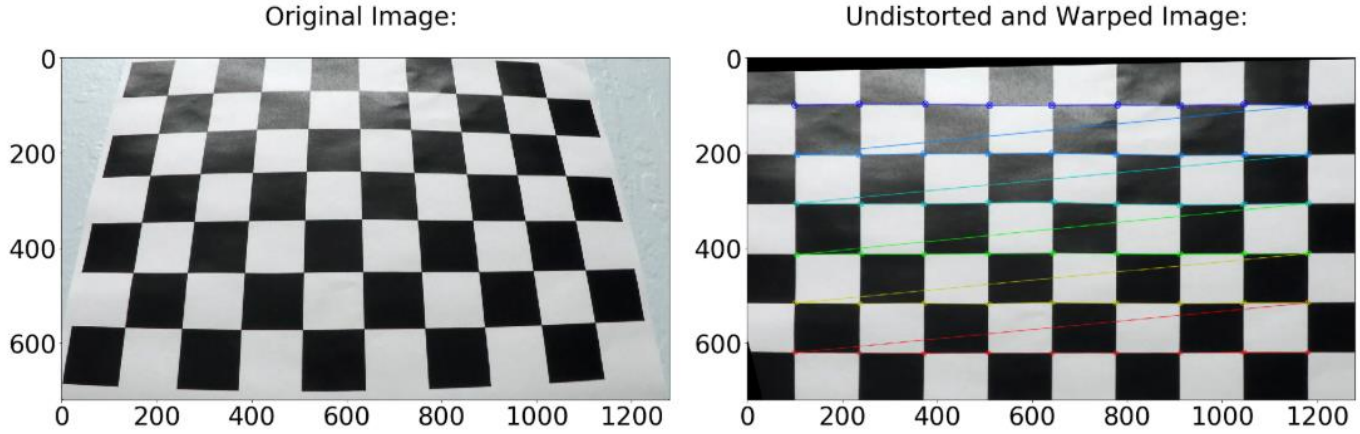
Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

At first "object points", which will be the (x, y, z) coordinates of the chessboard corners then the chessboard is fixed on the (x, y) plane at z=0 (depth), such that the object points are the same for each calibration image. Objp is just a replicated array of coordinates, and objpoints will be appended with a copy of it every time.It 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.

For above solution OpenCV functions findChessboardCorners and drawChessboardCorners to identify the locations of corners on a chessboard photos in camera_cal folder taken from different angles.

Then the output objpoints and imgpoints to compute the camera calibration and distortion coefficients using the cv2.calibrateCamera() function. I applied this distortion correction to the test image using the cv2.undistort() function and obtained this result:



Pipeline (single images)

1. Provide an example of a distortion-corrected image.

Applied a distortion correction to raw images placed in folder test_images.

Input : calculated camera calibration matrix and distortion coefficients to remove distortion from an image, and

Output : the undistorted
image.

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

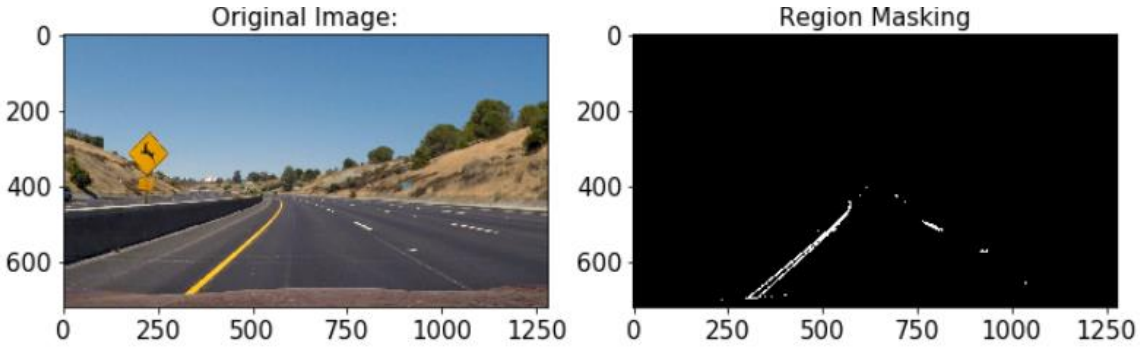
I used a combination of color and gradient thresholds to generate a binary image in P4_Advance_Lane_Finding.ipynb.

Converted the warped image to different color spaces and created binary thresholded images which highlight only lane lines and ignore everything else.

Following color channels were used:

1. 'S' Channel from HLS color space, with a minimum threshold = 180 & max threshold = 255
    - Good: Identifies the white and yellow lane lines,
    - Bad: Did not pick up 100% of the pixels in either one with the tendency to get distracted by shadows on the road.
2. 'L' Channel from LUV color space, with a min threshold = 225 & max threshold = 255,
    - Good: Picks up almost all the white lane lines, but
    - Bad: completely ignores the yellow lines.
3. 'B' channel from the LAB color space, with a min threshold = 155 & max threshold = 200,
    - Good : Identifies the yellow lines much better than S channel, but
    - Bad: Completely ignores the white lines. Created a combined binary threshold based on the above three mentioned binary thresholds.

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



3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

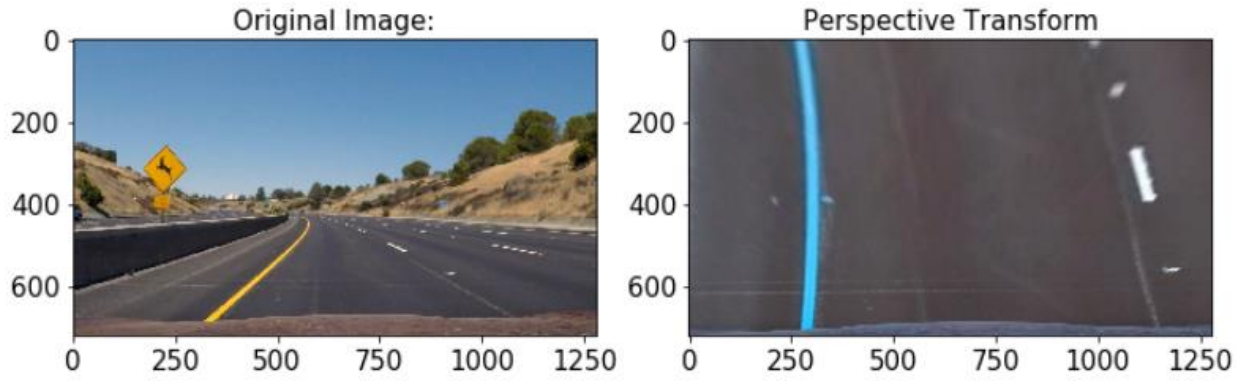
The code for my perspective transform is in P4_Advance_Lane_Finding.ipynb, which includes a function called transform_perspective() It takes as inputs an image (img) and hardcodes the source (src) and destination (dst) points.

It uses the CV2's getPerspectiveTransform() and warpPerspective() functiions.

I chose the hardcode the source and destination points in the following manner:

```
src = np.array([[(width*0.4, height*0.65),
                    (width*0.6, height*0.65),
                    (width, height),
                    (0, height)]],
                    dtype=np.float32)
dst = np.array([[0,0],
                [img.shape[1], 0],
                [img.shape[1], img.shape[0]],
                [0, img.shape[0]]],
                dtype = 'float32')
```

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.

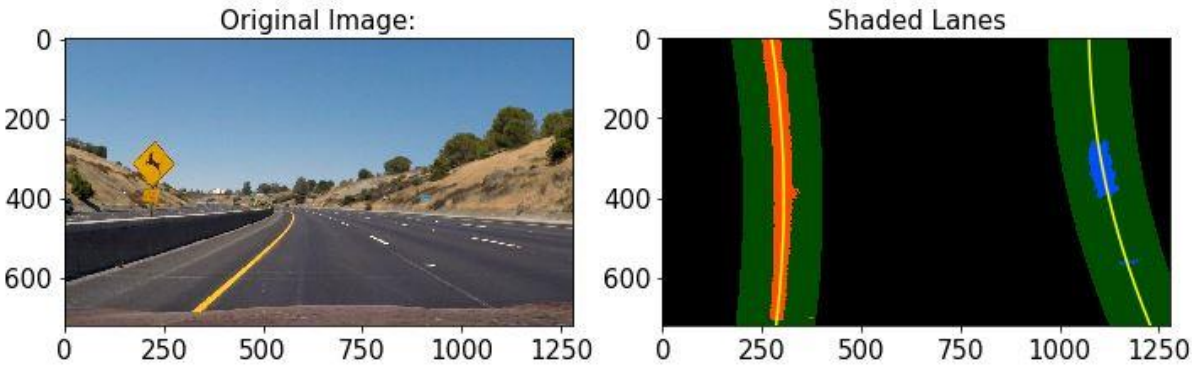


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The code snippet is P4_Advance_Lane_Finding.ipynb and function is named Fitting Shaded Lines().

Starting with the combined binary image to isolate only the pixels belonging to lane lines, I fit the polynomial to each lane line, as follows:

1. Identified peaks in a histogram of the image to determine location of lane lines.
2. Identified all non-zero pixels around histogram peaks using the numpy function numpy.nonzero().
3. Fitted polynomial to each lane using the numpy's fn. numpy.polyfit().



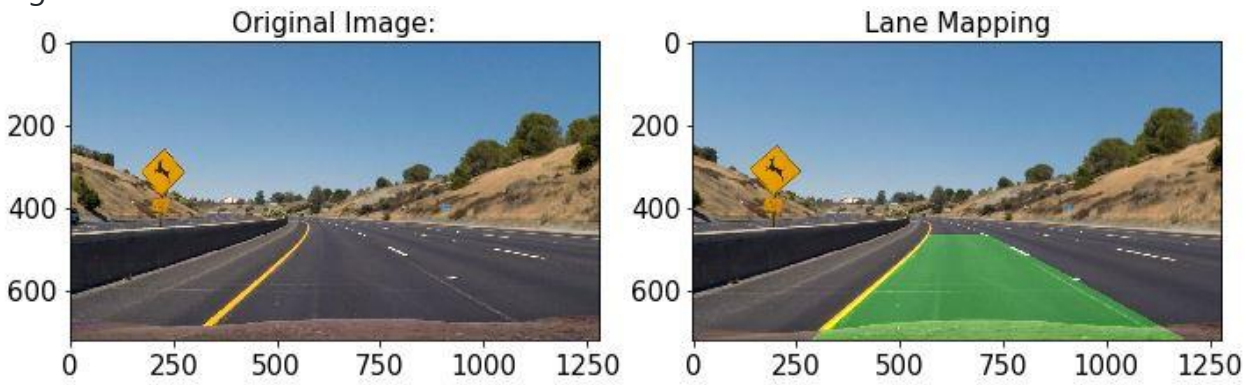
5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

The code is in Advanced Lane Finding.ipynb and function is named def roc_in_meters. here the snippet

```
def roc_in_meters(ploty, left_fit, right_fit, leftx, lefty,rightx,righty):


    lefty = np.array(lefty).astype(np.float32)
    leftx = np.array(leftx).astype(np.float32)
    righty = np.array(righty).astype(np.float32)
    rightx = np.array(rightx).astype(np.float32)


    xm_per_pix = 3.7/700 # meteres/pixel in x dimension  ##corrected
    ym_per_pix = 30.0/720 # meters/pixel in y dimension  ##corrected

    y_eval = np.max(ploty)


    left_fit_cr = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 2)
    right_fit_cr = np.polyfit(righty*ym_per_pix, rightx*xm_per_pix, 2)

    left_curverad = ((1 + (2*left_fit_cr[0]*np.max(lefty)*ym_per_pix + left_fit_cr[1])**2)**1.5) \
                             /np.absolute(2*left_fit_cr[0])
    right_curverad = ((1 + (2*right_fit_cr[0]*np.max(lefty)*ym_per_pix + right_fit_cr[1])**2)**1.5) \
                            / np.absolute(2*right_fit_cr[0])

    avg_curverad =  (left_curverad + right_curverad)/2

    #For Video:

    LEFT.radius_of_curvature = left_curverad
    RIGHT.radius_of_curvature = right_curverad
    AVG.radius_of_curvature = avg_curverad
    return left_curverad, right_curverad, avg_curverad

pipeline(op='Radius of Curvature in Meters')
```

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

Here is an example of my result on a test image:



Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

1. Output: project_video.mp4

2. Output:
challenge_video.mp4

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Pipeline that I have developed is doing a fairly good job in detecting the lane for given car by taking proper center of it. So far code is running good for ideal conditions where lane lines are well defined.

Above code shows decent result on challenge_result.mp4 video but there are few misses in lane lines specially near shadowed areas.

Code didn't do well on the harder_challenge_video.mp4. Errors coming were like not able to track the path at all.

I think possible way of making it more robust require lot of image processing job to remove noises and extract only those patterns or lane line which are required.