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

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

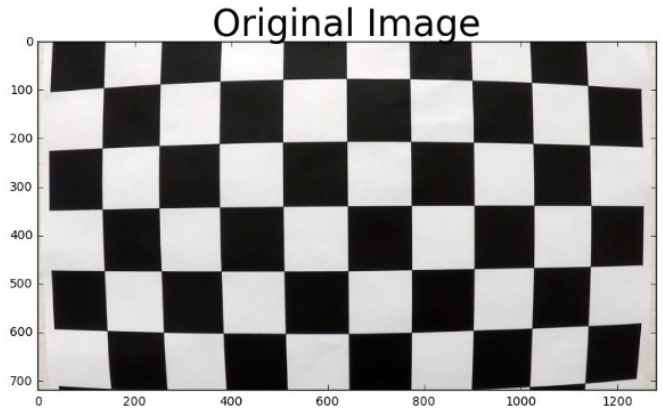
The project is implemented using IPython Jupyter notebook and the contents are below,

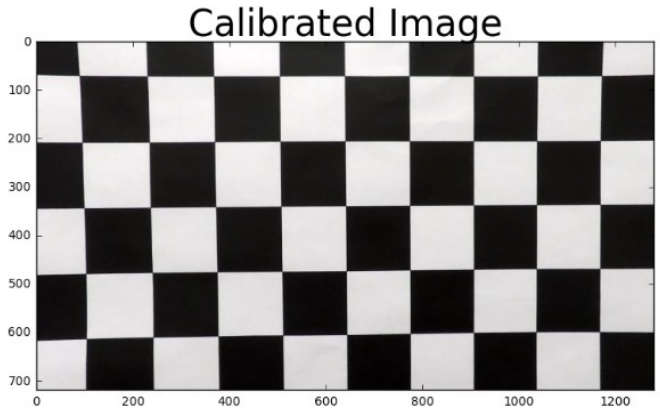
File Name	Description
Project4-solution.ipynb	IPython notebook containing solution code
Project4-solution.html	HTML snapshot of the final solution code
project_video_out.mp4	Solution Output video
Project4-solution-writeup.pdf	Project Write up

Camera Calibration

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

First step is to map object points (3D world points) to image points (2D image). This is done using the help of a standard checkerboard/chessboard board image. All the images in the folder “camera_cal” is read one by one and converted to “Grayscale” images. By using OpenCV’s findChessboardCorners(…) we can find 9x6 inner corners for images taken at different perspectives and viewpoint. Now using these corners as image points, and preset array of 9x6 corners as object points, we can use OpenCV’s calibrateCamera(…) function to return camera parameters such as camera matrix, distortion coefficients etc. The distortion coefficients, has a mapping of object points to image points which can be applied to original images to get an undistorted/calibrated image as shown below.

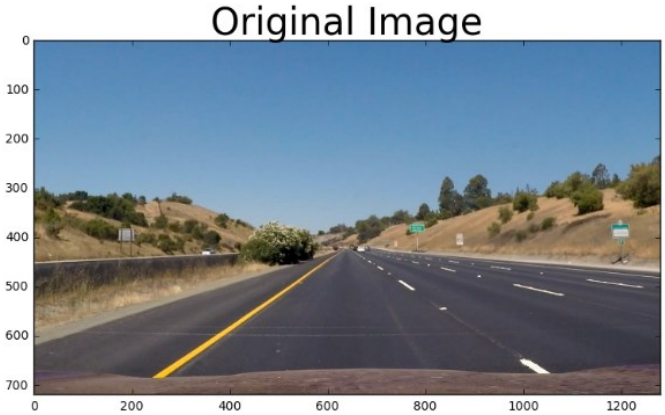


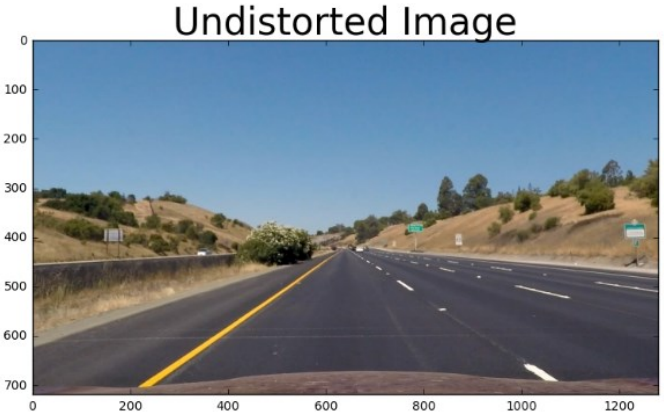


Pipeline (single images)

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

Once we obtain camera matrix and distortion coefficients, we can apply on the test images present in “test_images” folder. Camera distortion correction can be easily identified in the below image by noticing the hood of the car at the bottom of the image as shown below.





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.

In my solution, I've created a binary image after getting a "bird's eye-view" or a "top-down-view" of the lane. This saves significant compute time from applying these techniques on a full image. As we are more interested in how the lanes look when "Binarised" the other regions can be ignored.

Please refer to Cell: In [11] in the supplied IPython notebook and function

```
def getBinaryImage(img):
```

In my solution, I've used both RGB and HSL color spaces to get a good detection of lane points in the top-down-view. I use a 5x5 Sobel Gradient filter on the color image providing, Gradx, Grady, GradMag and GradDir outputs. By trial and error, and tying on multiple "test_images" the thresholds were heuristically selected. This result was combined with the result of saturation channel which is also thresholded to strongly identify, light-yellow and bright white lanes.

As seen in the below image, using Saturation channel in HLS space, gives a nice strong band of pixels for lanes. The Sobel gradient detections, provide good detections for the broken while lanes along the borders as seen.

Sometimes, when there is change in color of the road or when there are shadows on the road, pixels other than lane pixels also gets highlighted. This is a big threat to the histogram based lane searching method as the histogram will have low signal to noise ratio and starting points will be randomly skewed to the right, left or middle of the image.

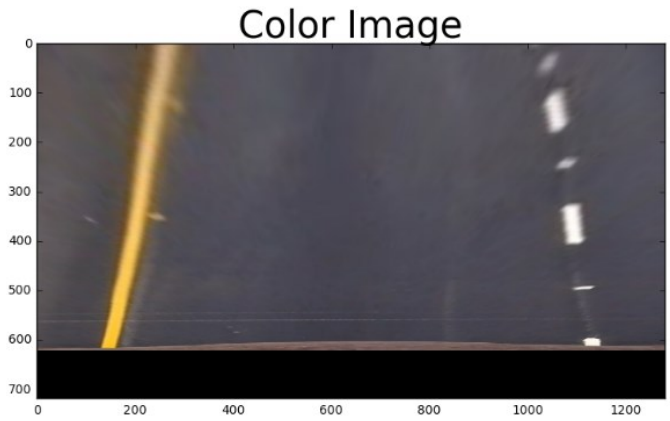
A host of helper functions and code with "Binarises" color image can be found in cells 10

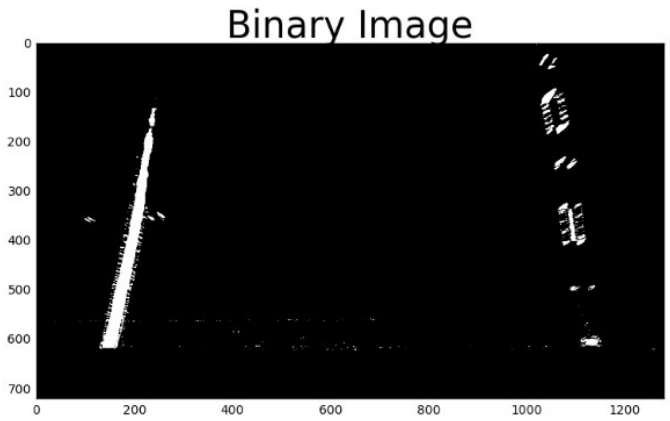
```
def abs_sobel_thresh(img, orient='x', sobel_kernel =3, thresh=(0, 255)):

def mag_thresh(img, sobel_kernel=3, mag_thresh=(0, 255)):

def dir_threshold(img, sobel_kernel=3, thresh=(0, np.pi/2)):

def hls_select(hls, thresh=(0, 255)):
```





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

The perspective transform is applied right after camera calibration and before “Binarising” the input. This reduces compute significantly as we are interested only on points on the lane and not the entire image with surroundings.

Firstly a region-of-interest is applied on a test image which encompass both left and right lanes snuggly. The ROI’s should also have a small margin on left and right of the image especially when the road bends towards right/left. This step is very crucial and painstaking as getting a right ROI fit on the lane is very crucial in the pipeline. Too much margin will include road boundaries or neighboring vehicles which may skew and affect the results. The source points and destination points along with the rendering are as below

Full View

```
[[  560.   460.]
 [  180.   710.]
 [ 1130.   710.]
 [  720.   460.]]
```

Birds-eye-view

```
[[  100.   100.]
 [  100.   620.]
 [ 1180.   620.]
 [ 1180.   100.]]
```

The corresponding code can be found in cells 6, 7, 8 and 9.

```
def warpImage(img, src, dst):
```

Using source and destination points and using OpenCV’s getPerspectiveTransform() function we can get a projection matrix M which can be used to perform a “Prespective-warp” of the source image. The result of warping can be observed as shown above

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

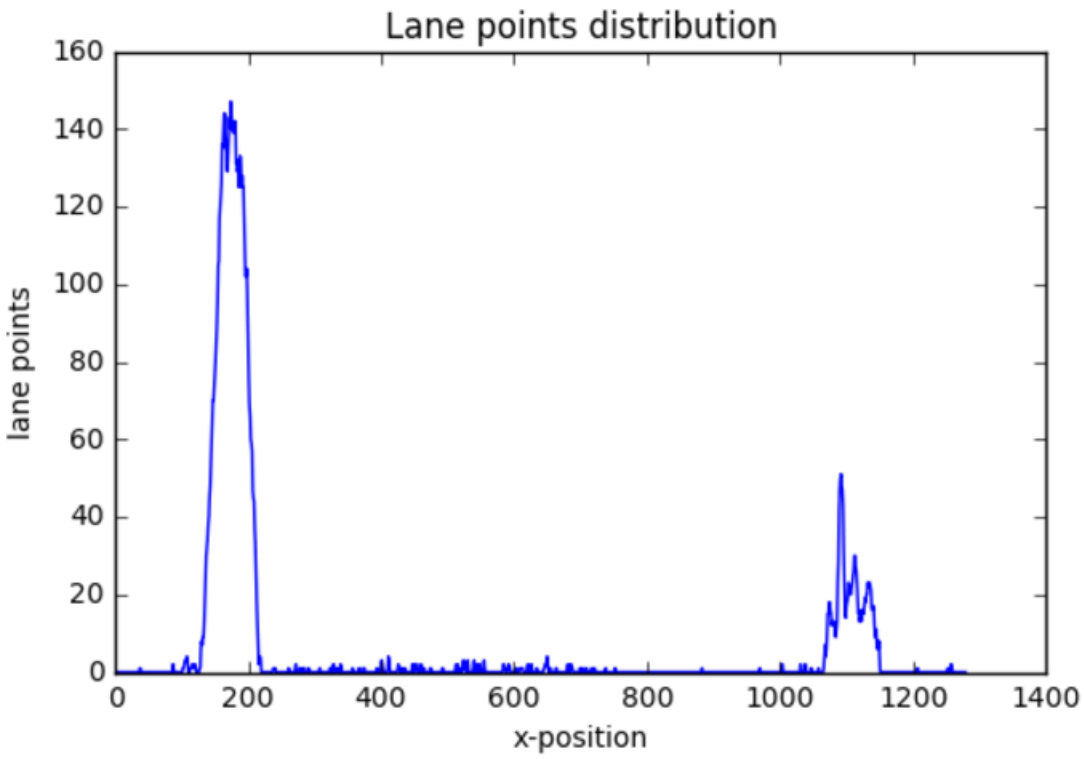
I used the method which was taught in the course to fit a polynomial. First step was to identify starting search position by taking a histogram of the binary image at half its height from the bottom as shown below. This image clearly gives to peaks which represent maximum number of pixels lying on the lanes.

Code is in cell 13 in the notebook.

Using this as a starting point, I used the sliding window technique, to search the lane from bottom of the image towards the top. The sliding window height was set to 80 pixels, so for an image height of 720pix, we can have a max of 9 windows. The width of each window (horizontal search range) was set at 100pix. An early exit condition of at least 50pix requirement in one window was also used. Taking cues from the histogram, the staring x position of left and right lanes was provided to the searching algorithm defined in cell 17

```
def searchNew(binary_warped, x_base, margin=100):
```

The function keeps track of (x,y) positions of the pixels constituting the lane, which is basically a non zero value in the binary image. Once the entire image is searched, it fits a polynomial and returns an curved line. The function searches one lane at a time, and will have to be separately called for right and left lanes.



For subsequent frames, we need to perform this exhaustive search but can used the lane line fits found in the previous frame and just search around that line. This is done in cell 14,

```
def checkWithPrev(binary_warped, prev_fit, margin = 100):
```

Once we get independent left and right lane fits, we fit both the lanes back on the binary image in cell 14 and 18

```
leftx_base, rightx_base = getNewBase(binary_warped)

#Search left line
left_fit = searchNew(binary_warped, leftx_base)
#Search right line
right_fit = searchNew(binary_warped, rightx_base)

left_fitx, right_fitx = fitLine(binary_warped, left_fit, right_fit)
```

To track lanes in multiple frames I used the Line() class to track various parameters of Left and Right lane separately defined in cell 21

```
# Define a class to receive the characteristics of each line detection
class Line():
    def __init__(self):
        # was the line detected in the last iteration?
        self.detected = False
        # x values of the last n fits of the line
        self.recent_xfitted = []
        #average x values of the fitted line over the last n iterations
        self.bestx = None
        #polynomial coefficients averaged over the last n iterations
        self.best_fit = None
        #polynomial coefficients for the most recent fit
        self.current_fit = np.empty((0, 3), dtype='float')
        #radius of curvature of the line in some units
        self.radius_of_curvature = None
        #distance in meters of vehicle center from the line
        self.line_base_pos = None
        #difference in fit coefficients between last and new fits
        self.diffs = np.array([0,0,0], dtype='float')
        #x values for detected line pixels
        self.allx = None
        #y values for detected line pixels
        self.ally = None
        #Average window size
        self.win_size = 0
```

A temporal window of 5 frames is used to track poly fits of left and right lanes, and calculate a running average of 5 frames which is assigned to self.best_fit. For the first frame, the lane_detect() function will use exhaustive search technique to detect lanes, but after that it will rely on searching around a neighborhood of average lane positions.

```
    #Search left line
    if(left_line.detected == False):
        left_fit = searchNew(binary_warped, leftx_base)
        left_line.detected = True
    else:
        left_fit = checkWithPrev(binary_warped, left_line.best_fit)
```

Maintain a running window of size 5 like a FIFO (First In First Out) where new detections are appended and old detections are deleted.

```
  if(left_line.win_size < 5):
        left_line.current_fit = np.append(left_line.current_fit, np.array([left_fit])
, axis=0)
        left_line.win_size += 1
    else:
        left_line.current_fit = np.delete(left_line.current_fit, (0), axis=0)
        left_line.current_fit = np.append(left_line.current_fit, np.array([left_fit])
, axis=0)
```

Finally the best fit of left and right lanes are just a mean of previous of new + 4 previous samples which is provided to fit the lines in the current frame.

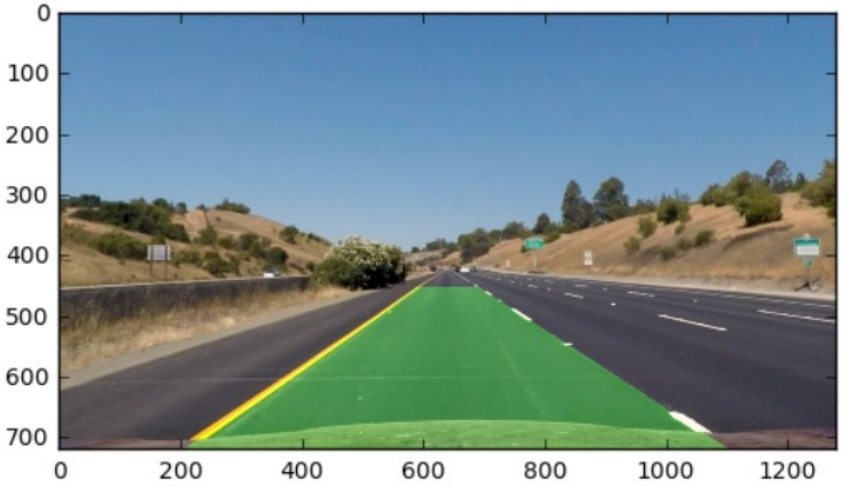
```
    left_line.best_fit = np.mean(left_line.current_fit, axis=0)
    right_line.best_fit = np.mean(right_line.current_fit, axis=0)

    left_fitx, right_fitx = fitLine(binary_warped, left_line.best_fit, right_line.bes
t_fit)
```

The lane lines are now re-projected back to color image by first computing a polygon between left and right planes, and re-projecting it back using inverse perspective transform matrix M_inv. Please refer cell 19

```
def reprojectLane(binary_warped, undist, left_fitx, right_fitx, M_inv):
```

The resulting image is an alpha blend of re-projected lane and color image as shown below,



The project video is read using skvideo module which was useful to read and write all the frames. Cells, 24, 25, 26

```
left_line = Line()
right_line = Line()

result = []
for i in range(len(inp_vid)):
    out = lane_detect(inp_vid[i], mtx, dist, src, dst, left_line, right_line)
    result.append(out)

print('Done')
result_arr = np.array(result)

plt.imshow(result_arr[0])`
```

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 radius of curvature of the road is calculated using the formula shown below.

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

In the case of the second order polynomial above, the first and second derivatives are:

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

So, our equation for radius of curvature becomes:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

The radius of curvature(RoC) and position of vehicle from center (Drift) is implemented in cell 22.

```
def radiusOfCurvature(binary_warped, undist, leftx, rightx, left_pt, right_pt):
```

In order to represent the RoC in terms of real world metrics (in meters), the pixels contributing to the lanes are scaled assuming 30m in depth along y_direction and 3.7m in width along x_direction as per US Gov Regulations. The lane height is set as image height and width is obtained based on leftx and rightx locations obtained using histogram.

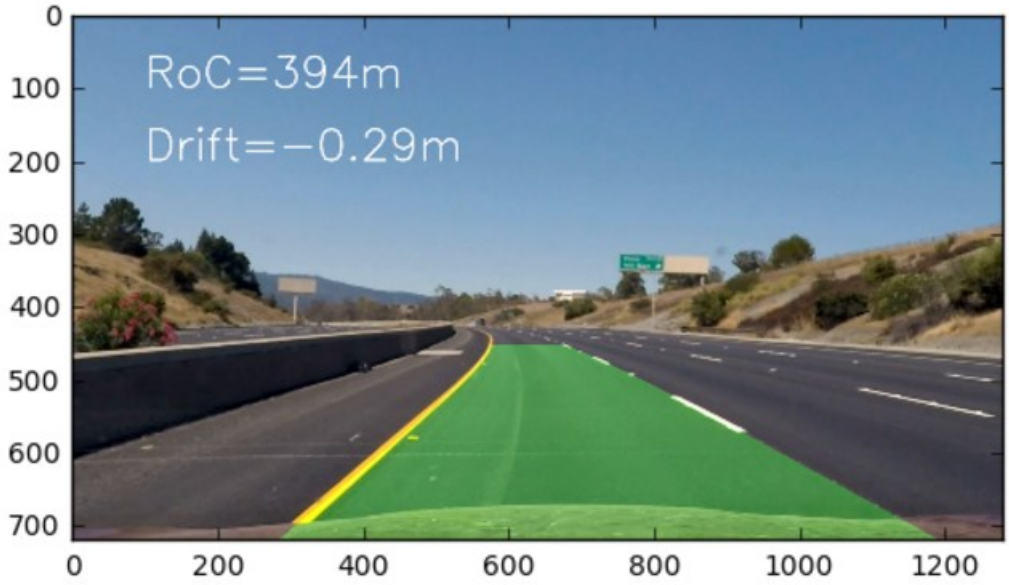
To compute drift, The center of detected lane is computed using leftx and rightx locations and subtracted from actual center of the image which is 1280/2 as shown below.

```
    roc = np.mean(np.array([left_curverad, right_curverad]))
    cal_center = np.mean(np.array([left_pt, right_pt]))
    drift = ((binary_warped.shape[1] / 2) - cal_center) * xm_per_pix
```

A negative value says that the vehicle is drifted towards left from center and a positive value says its drifted right.

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

A snapshot of the 1st frame with lanes re-projected onto the color image with values of RoC and Drift overlayed at the top left corner of the same 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!).

Please checkout the video **project_video_out.mp4** for the final result.

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?

Some of the problems I faced during the project are listed below.

1. Selection of ROI - This was a very painful process to start with as identifying points on a straight road and identifying points on a curved road is crucial. Several iterations of manual selection of points were done to accommodate enough information to get enough lane pixels in the 'Bird's-eye-view' image.

2. Creating Binary Image – Making sure only lane pixles are captured correctly and other deformations in the road, color change, road boundaries, perspective transform artifacts, car hood etc. are masked needed some fine tuning of kernel sizes and color thresholds. Using HLS space, especially Saturation (S channel) helped a lot in highlighting Yellow and White lanes but it also amplified, shadow regions and especially patches where the road color changed. This affected starting position of sliding window search but this was handled by keeping track of a running average of detected tracks.

3. Running Average – Maintaining running average helped smoothen the output especially in places when the road texture and color changed and also when tree shadows are casted on the road. Running average currently is just a history of detections in the last 5 frames but could be more robust by carefully selecting which detections to be kept in the running average window and which ones to remove.

4. Radius of Curvature – Providing the best_fit for last 5 frames to detect radius of curvature provides reasonable estimate of the road curvature but this is also highly affected in challenging places on the road. Maintaining a separate smoothing window for RoC and Drift will help eliminate spurious values and provide consistent output.

The pipeline is likely to fail in challenging positions of the road when there is change in texture, color, shadows on the road and also in places where lane markings are not so clear. Careful selection of "good" detected lanes will help make this solution more robust and applicable to other videos as well.