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

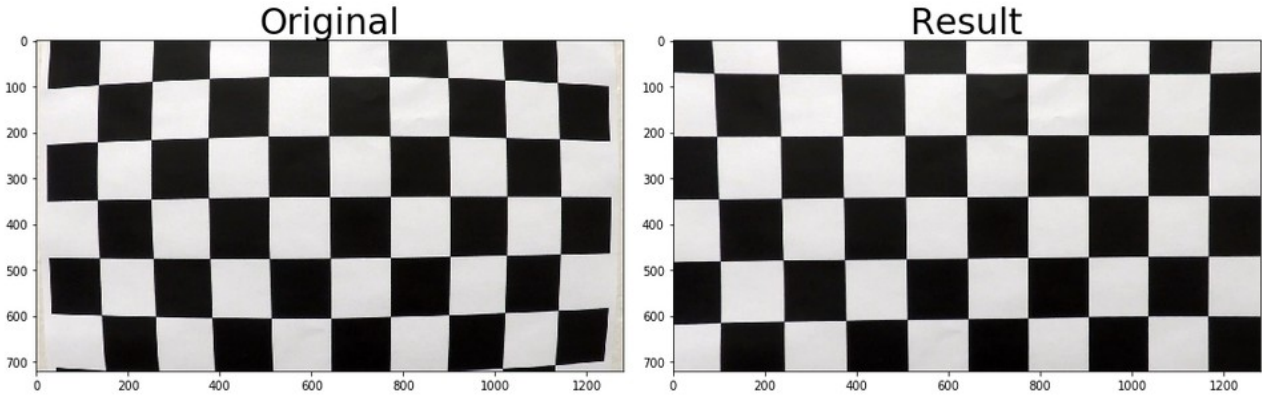
Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Camera Calibration

The code for this step is contained in lines `class CamParams()` of the file called ./examples/util.py.

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



I saved the results into `CamParams` class, and I also provided `save` and `load` function for future use avoiding repeated calibration.

```
self.mtx = mtx
self.dist = dist
self.rvecs = rvecs
self.tvecs = tvecs

def save(self, cam_num):
    ...

def load(self, cam_num):
    ...
```

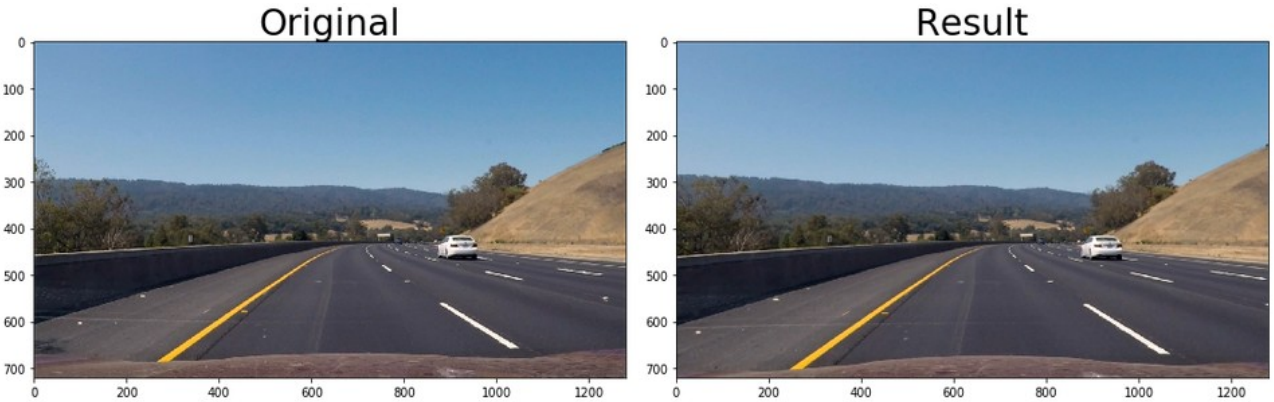
Pipeline (single images)

---

1. Camera Undistortion

In this step, get calibrated camera parameters into `cv2.undistort` to apply undistortion on an image.

Notice that this step has been writen into `util.LaneDetector.preProcess()` and will be automatically invoked before any other processing.



---

2. Perpective Transform

In order to performe a perspective transform, four source points of a quadrangle should be identified for computing transformation matrix `M` . To make sure the four source points are in a trapezoidal shape representing a rectangle, I have to find a pair of parallel lane lines as straight as possible in the image. I provided `persTrans.get_y_from_x()` and `persTrans.get_x_from_y()` to help get the coordinates of the four points in the image coordinate, that is to fine-tune the corrdinates of the four points choosed to obtain transformation matrix.

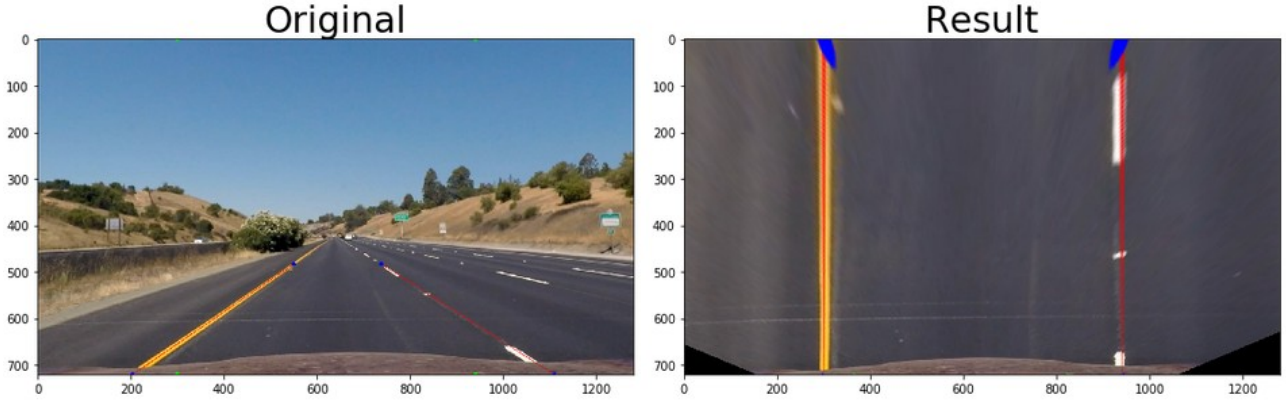
This resulted in the following source and destination points:

Source	Destination
204, 720	300, 720
1110, 720	940, 720
550, 483	300, 0
738, 483	940, 0

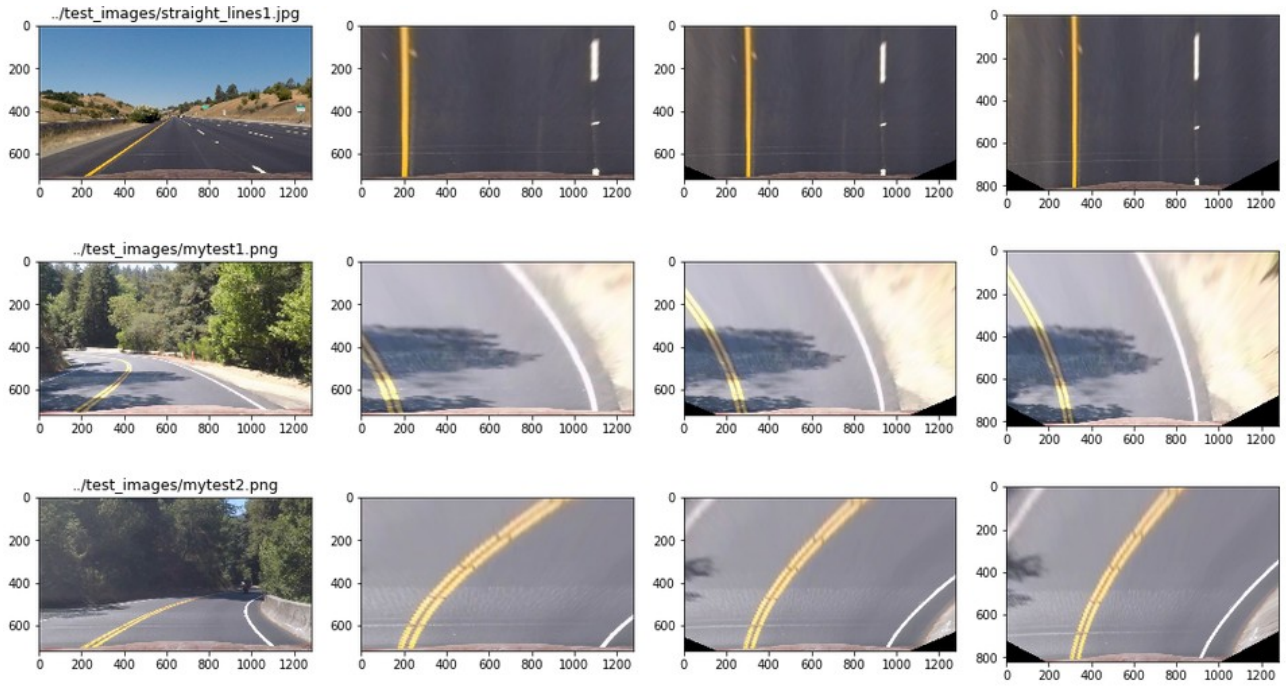
You can identify the code in `class PersTrans()` of ./examples/util.py.

```
# Compute perpective transform
pts = [np.float32([[204, 720], [1110, 720], [550, 483], [738, 483]]),
       np.float32([[300, 720], [940, 720], [300, 0], [940, 0]]),
       (720, 1280)]

self.pers_M = cv2.getPerspectiveTransform(pts[0], pts[1])
self.pers_M_inv = np.linalg.inv(self.pers_M)
self.pers_shape = pts[2]
```

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.



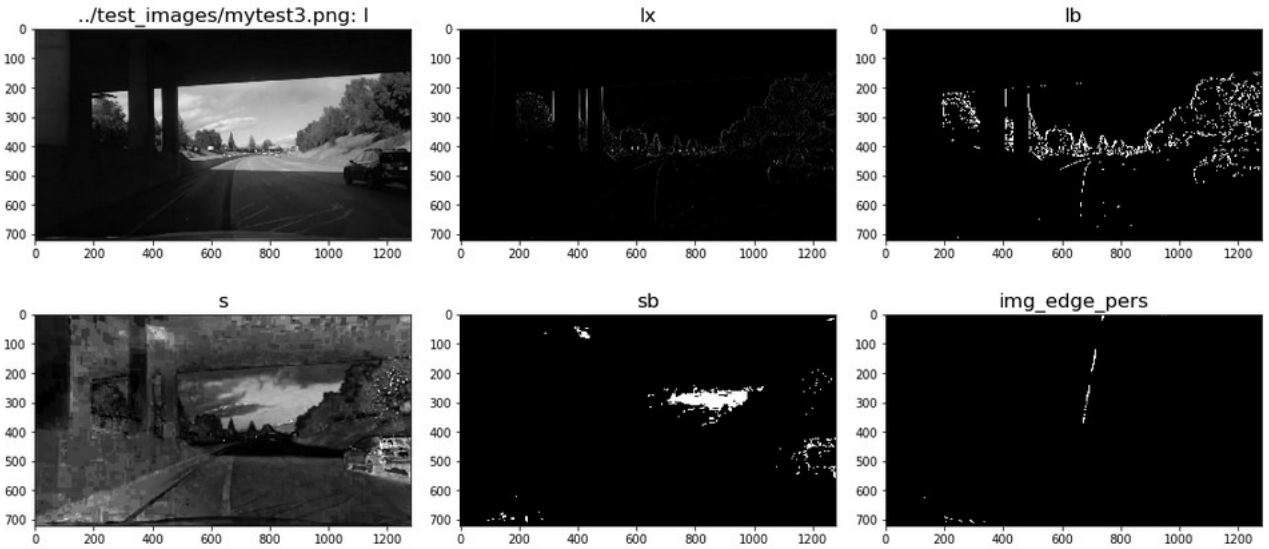
Here are more results demonstrate why I choosed the Source and Destination points metnioned above, which resulted in best result displayed in the 3rd column .



3. Edge Detection

I've tried the edge detection method discussed in the class, and also changed the order of edge detection and perspective tranform to find which order is better. The comparison shows that the order of which one going first has little effect on the final result. Overall， edge detection going first will results in fewer noise, so I did perspective transform after edge detection.

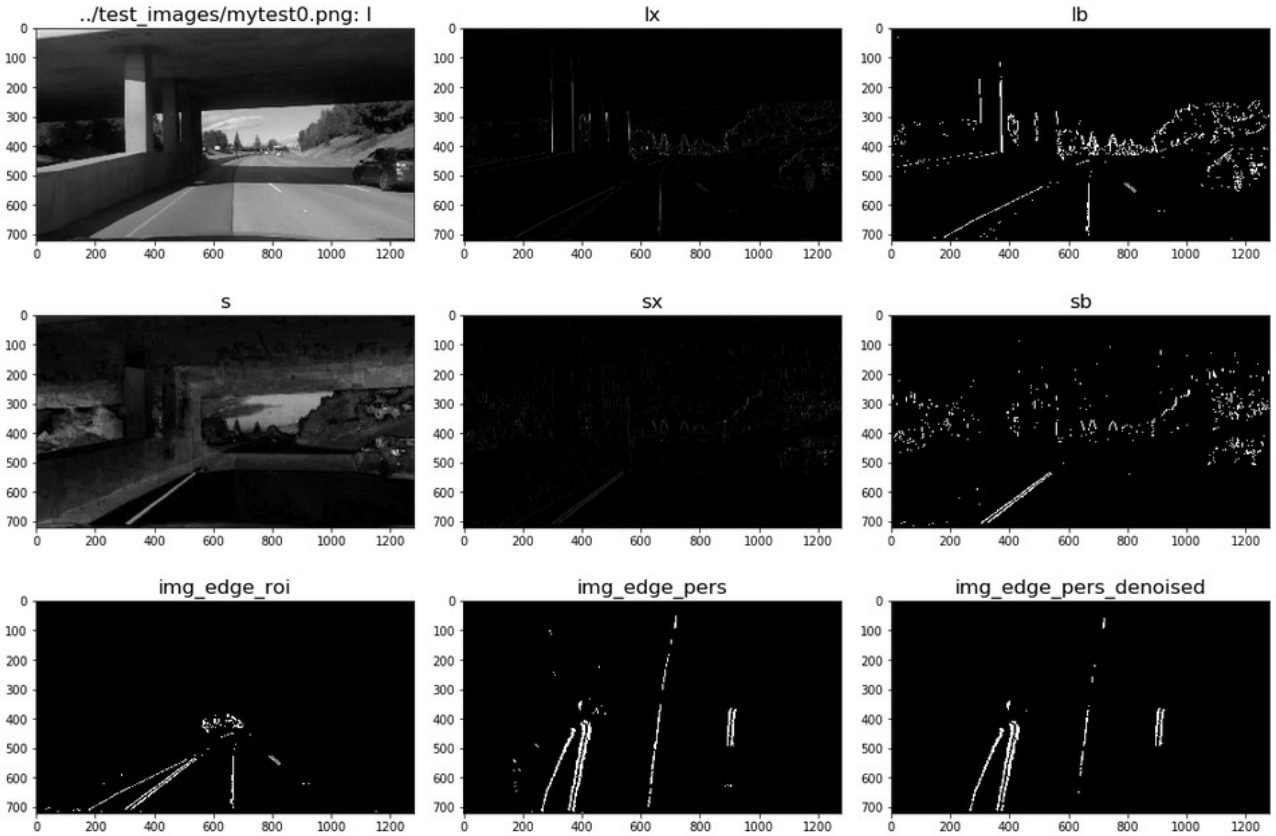
However, from the result shown after running below cell, we can see that the original color and gradient algorithm does not work well under shadow environment such as "myest0.png" and "mytest3.png".



To make the code can work well in dark environments, instead of thresholding s_channel I applied an sobel along horizontal direction on s_channel which improve a lot on edge detection.

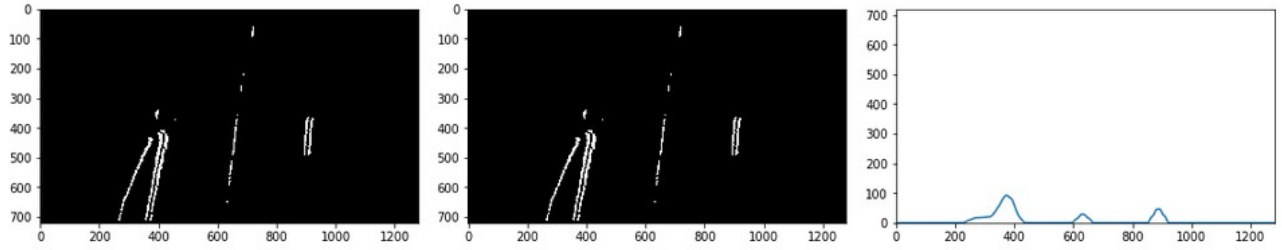
```
# Sobel x
sobelx_l = EdgeDetector.Sobel(l_channel, 1, 0)

# Threshold x gradient
sob_x_binary_l = EdgeDetector.BinaryThresholding(sobelx_l,
sob_x_thresh)

# Threshold color channel
# Sobel x on s_channel
sobelx_s = EdgeDetector.Sobel(s_channel, 1, 0)

# Threshold x gradient
sob_x_binary_s = EdgeDetector.BinaryThresholding(sobelx_s,
sob_x_thresh)
```

Here is an example of improved edge detection. You can find more through examples/exampy.ipynb.



4. Finding Lines Pixels and Fit Polynomial

There is no big change here with class solution, Histogram Peaks, but I take the edge width into account when I compute the histogram. Instead of summing up all pixels along each column, I sum up the pixels inside adjacent columns which I set the width to 40, see `DetEdgePeaks(binary_edges, edge_width=40)` . Thus, many small edges will be dropped as a lane line should have a certain width.



Now I have got two highest peaks from the histogram as a starting point for the determining where the lane lines are. Now I will use sliding windows moving upward in the image to determine where the lane lines go. Notice that in my solution the movement trend of sliding windows is used to predict the center of new sliding window, which is very important to track a curved lane line.

In order to save time and utilize the characteristics that lane lines appear at a close position on the two frames, a margin around the previous line position is the first target to search. However, there is the necessity for checking similarity and confidence of previous lane lines, and switch back to whole region resarch if fail to meet the requirement.

---

5. Measuring Curvature

I did Fit Polynomial using below equation. However, this calculated the radius of curvature based on pixel values, so the radius we are reporting is in pixel space, which is not the same as real world space.

$$x = A'y^2 + B'y + C$$

It might actually need to repeat this calculation after converting our x and y values to real world space, which will increase processing time. I am inspired by the hint on the class web page and deduced below equations.

$$m(x + o) = A(ny)^2 + B(ny) + C$$
$$mx = (An^2)y^2 + (Bn)y + (C - mo)$$
$$x = (An^2/m)y^2 + (Bn/m)y + (C/m - o)$$

> o means offset, caused by perpective transform

$$\begin{cases} An^2/m = A' \\ Bn/m = B' \\ C/m - o = C' \end{cases}$$

Finally I get the replaced solution.

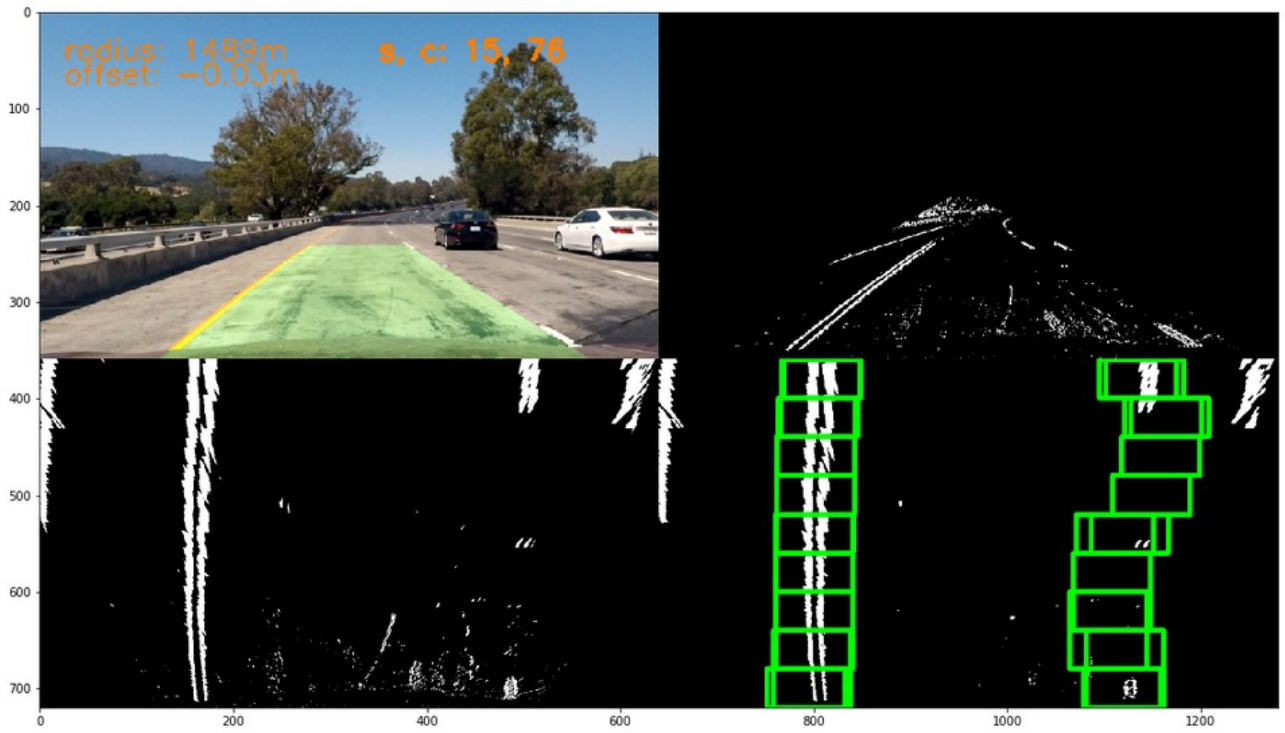
$$\begin{cases} A = A'm/n^2 \\ B = B'm/n \\ C = C'm + mo \end{cases}$$

Here is part of the implementation codes.

```
A = left_fit_cr[0]*xm_per_pix/(ym_per_pix*ym_per_pix)
B = left_fit_cr[1]*xm_per_pix/ym_per_pix
dy = 2*A*y_eval + B
ddy = 2*A

curverad = (1+dy**2)**1.5/np.abs(ddy)
```

6. Final Result

I implemented this step in the section of "Whole Processing of an Image or a video" which includes in "./examples/example.ipynb". Here is an example of my result on a test image:



Pipeline (video)

Here's a link to my video result, please see "./output_images/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?

The edge of the road and the shade of the tree can have a big impact on the results. In the case of fit polynomial, I only considered that the distance between the left and right

lane lines should not be too large or too small, and did not take advantage of their similar curve rates. In sliding windows, if you consider the similarity of the curvature of the two lines, you will avoid being disturbed by noise.

Add a filter to stabilize the position and curvature of the lane line.