Advanced Lane Finding Project

The goals/steps or requirements of this project are the following:

- Compute the camera calibration matrix and distortion coefficients from the given chessboard images
- Undistort the raw images using the above computed coefficients
- Use combined (color, gradient, magnitude and directional) binary thresholding to create binary image
- Apply perspective transform on the binary image to get bird's 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 of the image
- Warp the detected lane boundaries back onto the original image
- Output the display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Project Statement

This is the extension of the lane finding project (first project of the first term of Udacity SDC class). It is rather difficult to highlight (or the class called it lane boundary) drawing of the vehicle driving path. In real world, we have to calibrate vehicle camera of the cars. Since there are different kinds of camera for different OEM car manufacturers, we will not have to be able to take the chessboard images like this class is giving to us already.

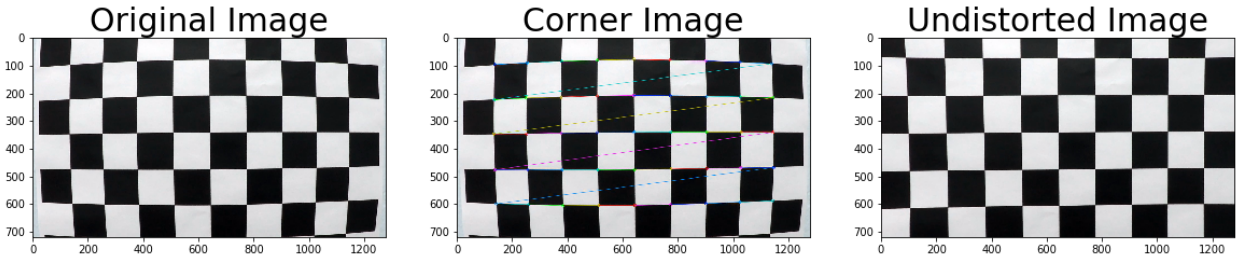
Camera Calibration

The first problem I have was the chessboard images are not quite capture 9*6 size so I was having difficulties to calibrate using the one size fit all size of the chessboard. I need to go open up those images manually to count what are the sizes of the chessboard box and code it manually to handle those sizes.

At first I initialize 2d points (image points) and 3d points (object points) to hold then use opencv findChessboardCorners function to get the corners. The object points are real world coordinates (x, y, z) and those are the same for each calibration. If I find the corners, depends on the chessboard box size, I append object points which are based on the size of the chessboard box and I append "corners" found to 2d image points.

After I computed coefficients using calibrateCamera() method, I save calibration matrix, distortion coefficient and image size in pickle file for using in the pipeline later. Pickle file is saved in the "camera_cal" folder of this project as "dist_pickel.p"

I then apply the matrix and coefficient to the cv2.undistort() function to undistort the images. The output images are stored in output_images folder of this project under: corners, undistorted, test_images, writeup_images folders. Here is what I obtained to visualize for this step.



Code Location:

I have two test code files: generate_output_images_test_codes.ipynb for generating images to use in this report and adjusting the parameter to visualize images and test_main_codes.ipynb for testing whether my final code combining together works or not.

This part of steps, you can find at the first cell of generate_output_images_test_codes.ipynb as well as at the first cell of test_main_codes.ipynb.

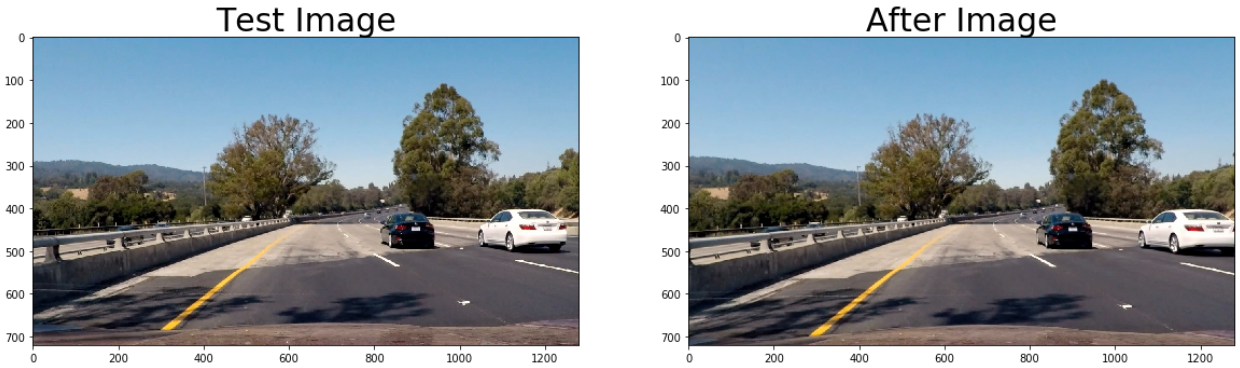
Cell 2, 3, and 4 of generate_output_images_test_codes.ipynb file has undistort function and generating all the corners, undistorted images and visualizing the sample image codes.

I place all the helper functions under p4Utilities.py and there is a main file called advance_lane_finding.py to run to generate output videos.

Pipeline (single image)

Then using the above computed calibrated matrix and distortion coefficient to apply on the test images from “test_images” folder in this project.

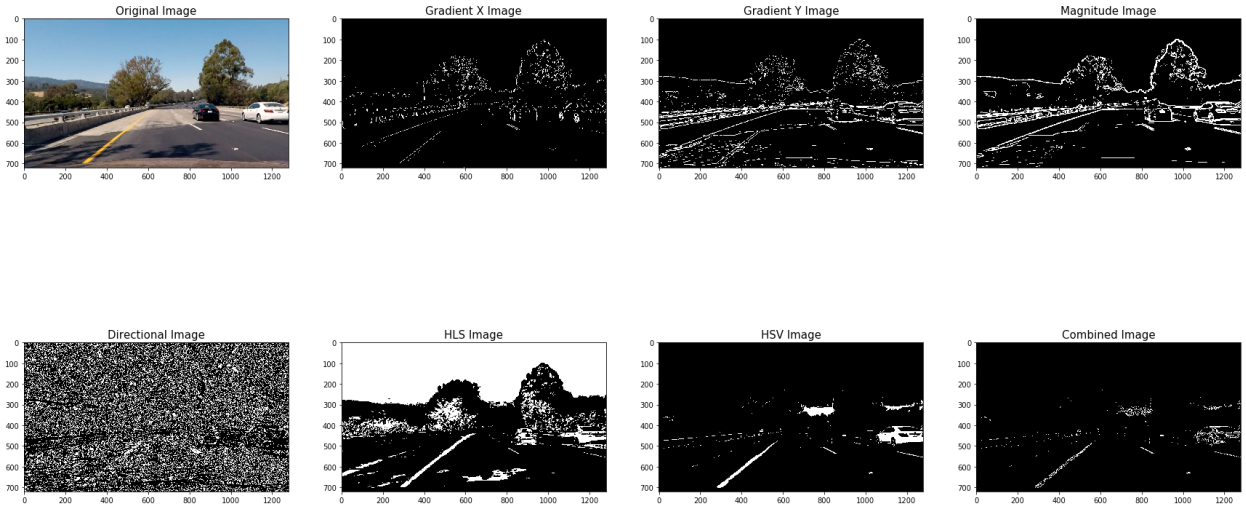
At first I undistort the image as you can see. The code to generate this sample output image is at fifth and sixth cell of the generate_output_images_test_codes.ipynb file.



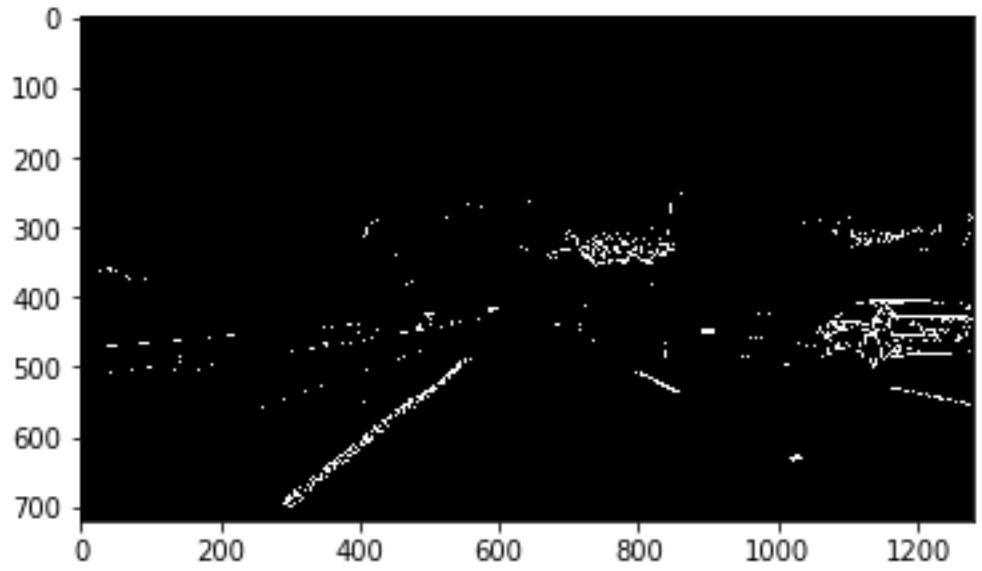
I then use the color, gradient (x, y orientation, magnitude, directional) threshold with different combinations to test to see which binary is the best output. I then decide to use which combination by visualizing on the notebook. Here I place the final combination output that I think it is best in my opinion.

The code to generate is at the cell 7 which defines all the thresholding functions and combined_threshold() function to be used in my final utility file, p4Utilities.py. It was one of the time consuming part of this project which I need to adjust threshold parameters.

Here you are the individual binary images and combined binary images that I used in my final code. This code is in cell 8 of the generate_output_images_test_codes.ipynb file.

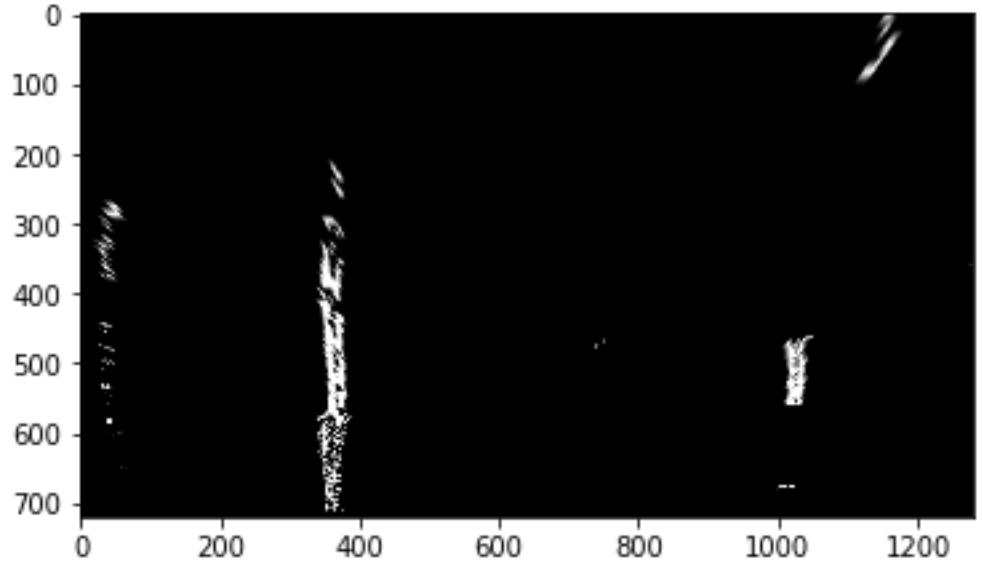


The code that testing the combined function which I use in my final file advanced_lane_finding.py is at cell 9 of the generate_output_images_test_codes.ipynb file. Here I obtain the combined binary image:

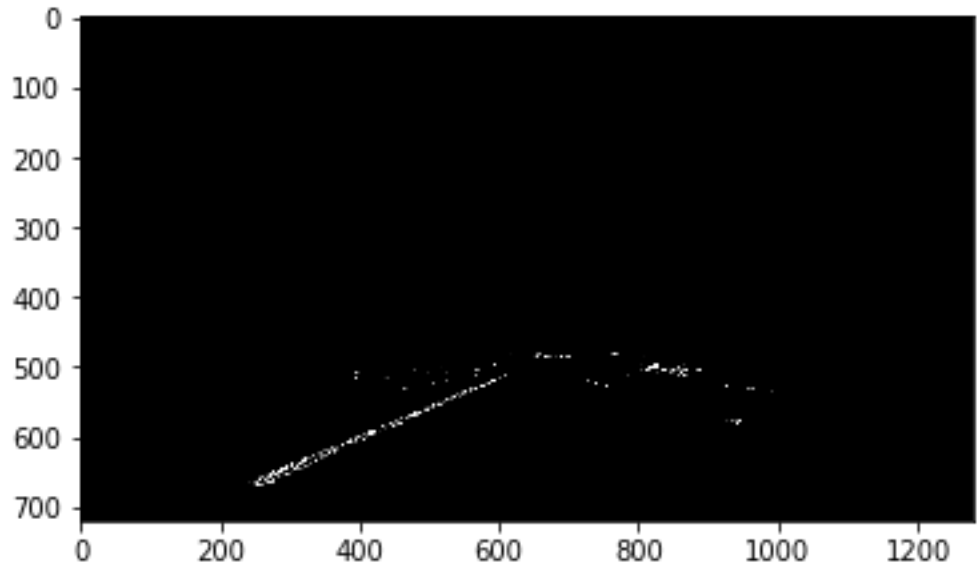


Next, I transform the perspective to be bird's eye view by using cv2.getPerspectiveTransofrm and cv2.warpPerspective. I manually set the source and destination four points each to transform the perspective. I wrap those functions in two functions (warp_image() and unwarp_image()) to return perspective transform and warped image and un warped image with inverse transform respectively. Those are defined in the cell 10 of generate_output_images_test_codes.ipynb file and p4Utilities.py file. I then visualize both warp and un warped image in notebook.

Warped

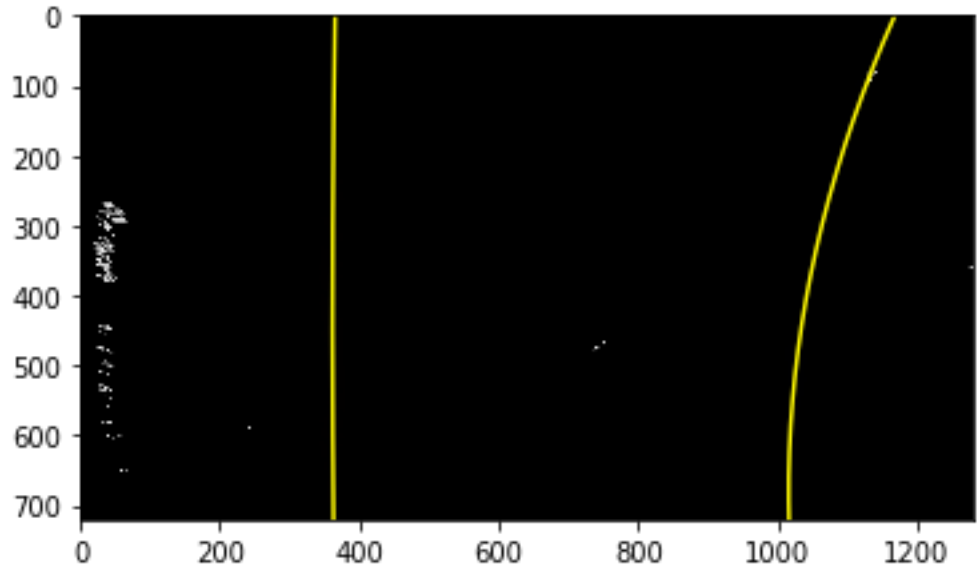


Un warped

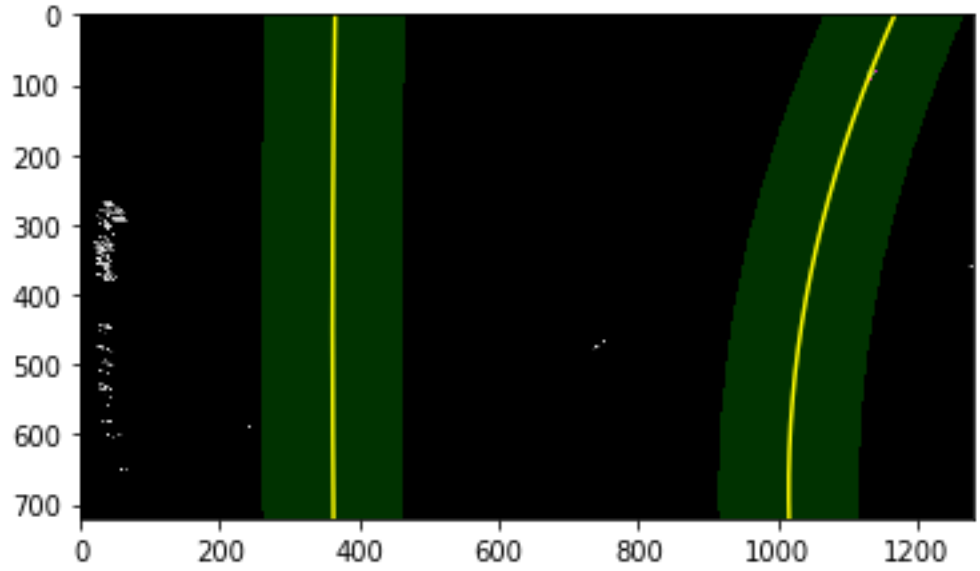


Lane Line Fitting

Next step in this pipe line is fitting the polynomial. I use histogram to fit left and right lane and polyfit sliding window methods support codes provided from Udacity' lesson. I created a function calls fit_polynomial() that takes the warped image and returns the left fit and right fit using second order polynomial. First take the peak of left and right lines moving with sliding windows moving along the left and right pixel positions and polyfit with second order polynomial function from numpy to generate left fit and right fit. Then visualize it in notebook as follow:



The code to fit the lines is defined in fit_polynomial() in generate_output_images_test_codes.ipynb file and p4Utilities.py file.



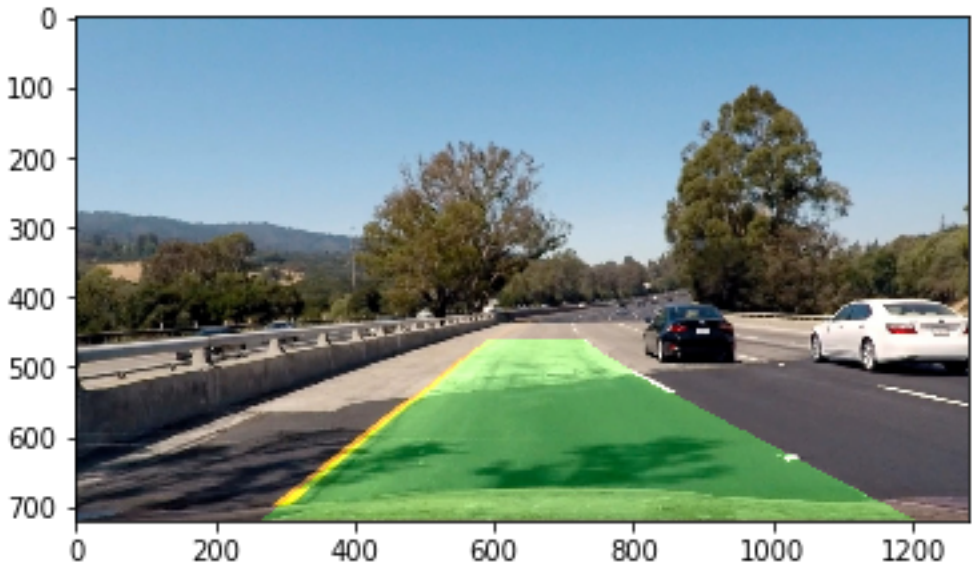
Calculating Curvature Radius and Estimating Vehicle Location

The code for this portion is implemented in cell 18 and 19 of the generate_output_images_test_codes.ipynb file. Since I don't want to calculate and print those values out from my final pipeline code, I did not implement in my pipeline function or my fit_polynomial() function. Just only for visualization.

Drawing Back to Original Image

After this step, I used the example from Udacity lecture to draw the polygon back on the original image. Then use the inverse perspective transform computed from unwarp_image() function earlier to get unwarped image using cv2.warpPerspective() and cv2.addWeighted() to draw polygon back on the original image. The code of this part is implemented in the draw_polygon() function in generate_output_images_test_codes.ipynb file and p4Utilities.py file.

The final image of this pipe line is as follow:



Code I used to draw back the polygon is wrapped in the function call draw_polygon() and it is at the second to last cell of the generate_output_images_test_codes.ipynb file and also defined in p4Utilities.py file.

Pipeline Video

The link to the pipeline video of this project is:

https://youtu.be/2UPecXXin68

Discussion

In general, this implementation that I have done is not robust. I did not consider the following:

I assume the color of the lanes, condition of the roads, quality of the images, and etc. to be as good as the images from test images or project_video.mp4 frame images. I am pretty sure that the implementation will have problems on some conditions like severe weather conditions, or fading lane marker or no lane marker roads or some other situations.

Another important thing is that the lane marking is not real time and if I have to process real time in embedded system with what I implemented, I am sure I cannot even keep up with the car speed that is in any parking lot like 5 mph. Something to improve and think about. After all, driving car is real time.

Another main thing is that I used chessboard images has been taken before by the camera which is used to drive. In real life those cameras are installed in the car and they were calibrated manually according to the OEM spec. It is not robust. I would like to implement camera calibration to be dynamic. Drive the car which equipped with camera for one minute to five minute and it picks up the matrix and distortion coefficients that we can use for real time processing. May be I can train with neural net. Install model in the car head unit, then live feed comes in and got the coefficient in minutes for further use.

The problems that I had during this project were:
Adjusting the parameter of the threshold values. What gradient to use, what color to use, and how to combine those to be usable effectively in this project. That was time consuming. Lack of experience in compute vision area might be my main problem. Reading in with moviepy, convert the color with opencv cost me some time as well. I have to look for what color space the moviepy read in to figure out which color conversion to use.

Another problem I have was, fitting the lane line, I was struggling with that when I started to work on this project but I went back and read the lectures and I found there is a sample utility code posted on. Part of this struggle I think is my lack of python and numpy skills. That cost me some times also.

In general, I learn more and more go through the lecture again and again. It is generally good that I can go back and learn the class lectures any time I want. That's the best thing about online learning.

Much more to work on for this project to make it to be robust, real time, dynamic camera calibration and performance improvement.