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.

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

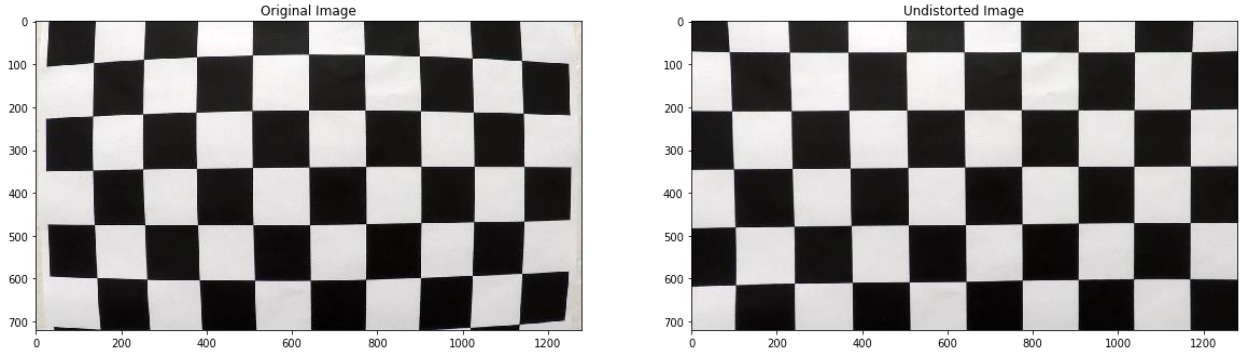
Much of the code was borrowed from the Advance Lane Lines lesson so much of the code looks similar to what is found in the lesson. Much of the difference is in how I used the Line() Class in keeping a history of lane characteristics and then filtering the results of each lane in order to achieve a smooth lane identification for the project.mp4 video. The output of this code was able to achieve normal.mp4 which shows does the lane overlay, radius of curvature identification, and lane center position on the video. This pipeline was not able to process the two challenge videos as tuning of the binary image lane identification could not result in proper identification of the lanes

Camera Calibration

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

The code for this step is contained in the 2nd and 3rd code cells of the IPython notebook located in CarND-Advanced_Lane_Lines.ipynb

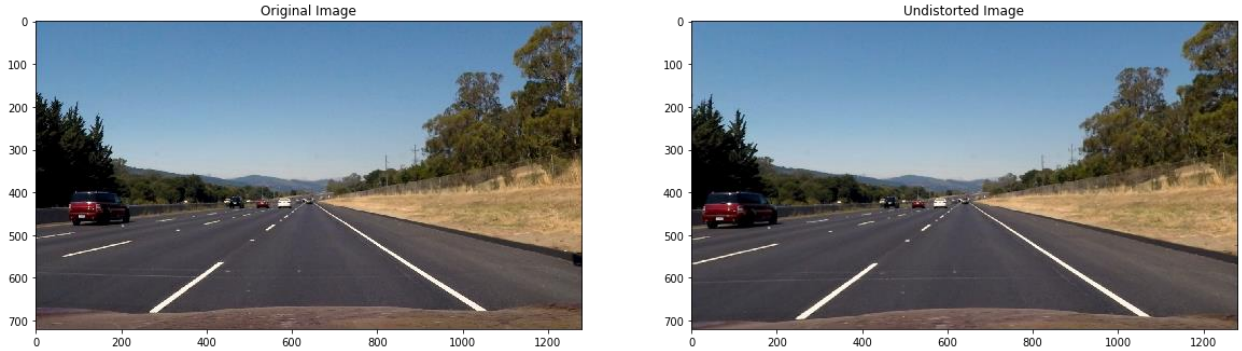
In order to undistort the images, I created a function ‘undistort’ that takes in an image, a camera matrix, and distortion matrix and uses the OpenCV function ‘undistort’ to return an undistorted image. In order to get the camera and distortion matrices, I used OpenCV’s calibrateCamera function to compute them. Here is the result of undistorting a calibration image



The calibrateCamera function takes in object points and image points. Object points are defined as a fixed grid whereas the image points are picked up from several calibration images. In this case there were 20 ([UPDATE] n =10 frames for faster response in adjusting lane marker) images used to pick up image points to do the calibration

Pipeline (single images)

Provide an example of a distortion-corrected image.



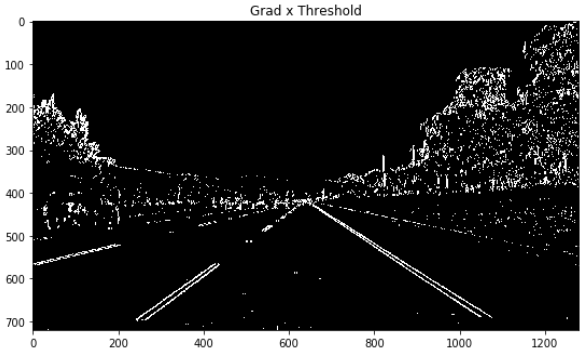
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like the pair below:

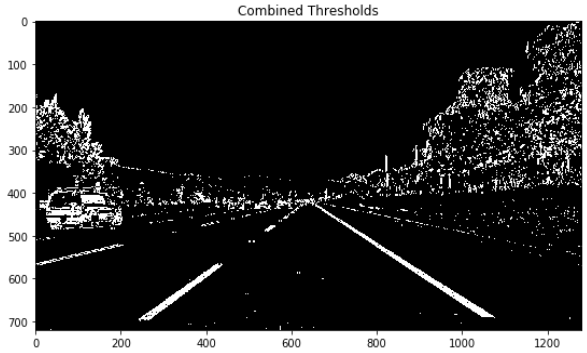
I created a rough pipeline in Cell 14 of the jupyter notebook in order to test the functions that I created to process the images. I basically used the calibration result from the calibration images and applied that result to one of the straight lane images to understand how well the functions are working

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.

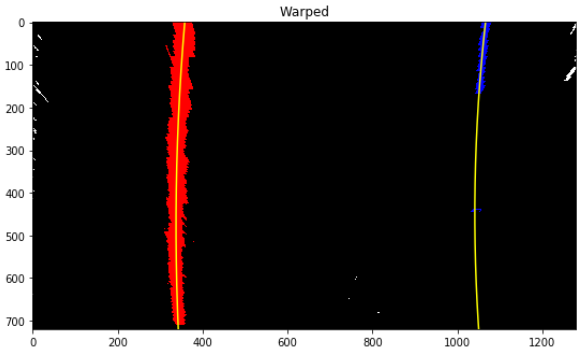
I defined several functions to perform gradients, gradient magnitude and direction, and color transformations. These functions are defined in cells 6 through 9 of the notebook. An example of the x-direction gradient binary image is shown on the left below. Ultimately, for the project video I settled on using just the result of an x-direction gradient and an s-layer from an hls color transform to get the combined image shown on the right below. I started trying the use other combinations to try the

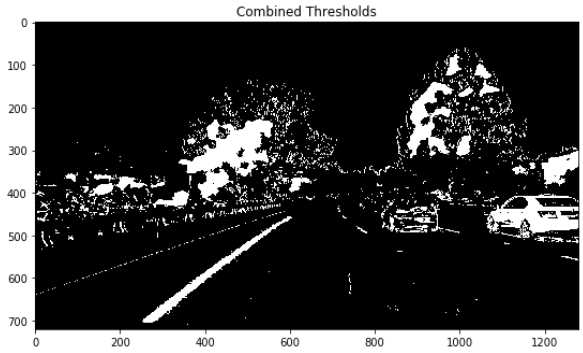
challenge video as well, but at the time of this write-up have not yet succeeded in finding a good combination.





[UPDATE]: Per the last reviewer's suggestion, I also added a LAB transform and added binary images based on this transform to the criteria for creating a thresholded binary image. The following is the result

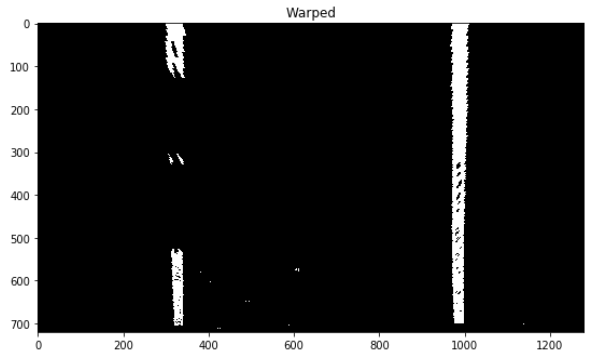


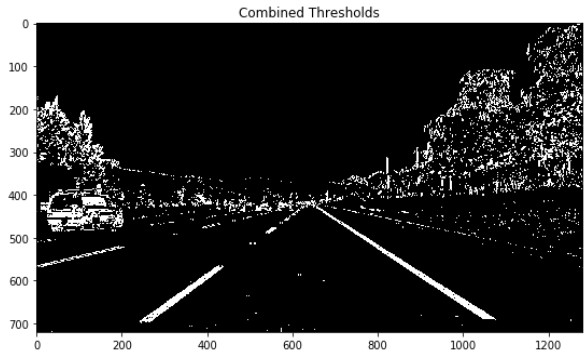


Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

I created a function called unwarp to do a perspective transform in the 10th cell of the notebook. In this function I transformed a trapezoidal area to a rectangle by starting from the centerline of the image and then defining the boundaries symmetrically at the bottom of the image up to a point close to where the lanes vanish into the horizon. I used the straight_lines images to adjust the top part of the lane until I got an image that had parallel lines. The top of the lane marker was also adjusted to get as much of the lane as possible to do the line fit. Finally, I also 'squished' the image in the x direction so that I could account for the bend in the lines. An example of the perspective transform is shown below. Note that the bottom of the lane lines in the warped image (300,720) and (1000,720) don't match the binary image.

The image on the left is the lane transformed to a rectangle, while the image on the right is the source image. Although hard coding values for the transformation was convenient, it is not well suited for videos such as the more challenging video in which the lane bends aggressively, and the lane does not extend into the image as far as the project video





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

There are two functions that I created in the code to identify lane-line pixels are in cells 11 and 13. Cell 11 is a function called poly_lanes_search and cell 13 contains poly_lanes. Cell 11 was used for the rough pipeline (cell 14) to check individual test images while the pipeline for checking test images and for creating the video is in cell 16 and uses the poly_lanes function.

I'll now describe the poly_lanes function since it contains the overall approach for achieving lane identification in the video output.

This function takes in a binary image of the lane that has been warped using the perspective transformation from the description above. It then make a judgment of using a sliding windows search to identifying lanes or using a window around previously detected lanes based on how many averaged windows (20 ([UPDATE] n =10 frames for faster response in adjusting lane marker)) have been found. If less than 20 ([UPDATE] n =10 frames for faster response in adjusting lane marker), use sliding windows, otherwise use the lane results.

The result is identifying the x and y position of each of the left and right lanes. These positions are then used in two second order polynomial fits (one for the pixels and one for the pixels converted to meters) to identify the lane positions. The fit using the pixels is used for lane identification in subsequent video frames while the one converted to meters is used for calculating the actual radius of curvature.

Then finally, if the lane characteristics criteria had been met (average distance between lanes and curvature over a minimum threshold) the lane fit coefficients are passed to the leftLine and rightLine instances of the Line() Class.

[UPDATE]: I also added a new lane curvature similary function, curve_similarity, in Cell 11. This function returns the average difference in curvature and the average ratio of curvature for each pixel in the y-direction. As a criteria, I settled on limiting the average ratio to between 0.1 and 10 for left line average curvature / right line average curvature

The trick here was to use the averaged coefficients of successful line detections to do the search for the next video frame. The following code snippet shows how this was done:

#get the average coefficients

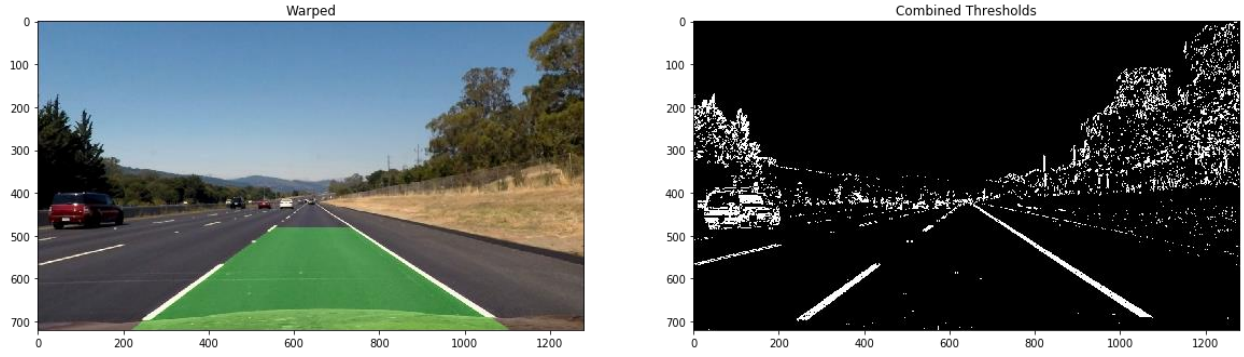
```
left_fit_avg = leftLine.best_fit.tolist()
right_fit_avg = rightLine.best_fit.tolist()

#use the avg coefficients to do the search
left_lane_inds = ((nonzerox > (left_fit_avg[0]*(nonzeroy**2) + left_fit_avg[1]*nonzeroy +
left_fit_avg[2] - margin)) & (nonzerox < (left_fit_avg[0]*(nonzeroy**2) +
left_fit_avg[1]*nonzeroy + left_fit_avg[2] + margin)))

right_lane_inds = ((nonzerox > (right_fit_avg[0]*(nonzeroy**2) + right_fit_avg[1]*nonzeroy +
right_fit_avg[2] - margin)) & (nonzerox < (right_fit_avg[0]*(nonzeroy**2) +
right_fit_avg[1]*nonzeroy + right_fit_avg[2] + margin)))
```

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.

I calculated the radius of curvature and the position Cell 16 of the notebook. For the radius of curvature, I took the average of the polynomial fit coefficients (including the coefficients of the current frame to fit a line onto the image. The reason I did this is that even if the current frame could not detect and compute a set of appropriate coefficients, we could still fit a curve using the last n = 20 frames ([UPDATE] n =10 frames for faster response in adjusting lane marker). This assumes that the car's position doesn't change relative to the lane very quickly and the radius of curvature changes slowly at the speeds travelled in the video. Therefore, although there will be a time constant associated with the filtering, it will result in smooth lane identification and can reject misdetections.

The second order polynomial coefficients are computed using the numpy polyfit function. The values are averaged by passing the coefficients to the instance of the Line() class, and using a method called x_fitted and x_fitted_m to do the averaging. The _m designation means that the data have been converted to meters

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



This step is also in the process_image() function in Cell 16. Here the green area is defined with the fillPoly function. Here, again, in order to create the lines, the averaged values of the polynomial fits are used to take advantage of the averaging filter defined in the Line().add_xfitted method

Pipeline (video)

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!).

Here's the link to my video:

https://github.com/avernethy/CarND-Advanced-Lane-Lines/tree/feature_hwk/test_videos_output/normal.mp4

Discussion

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?

Although I was able to make the video work on the project video, I was not able to complete the two challenge videos. There are several places where this project could be improved:

1. Binary image – I need to do more experimentation with different color spaces, threshold values and other combinations. One of the ideas I had was to try to find a) yellow lines b) white lines and combine the result. I think aggressive masking would also help, though I wonder about the performance of this type of masking for lane change
2. Lane criteria – The lane criteria I used is very simple and does not have an allowance to be able to do a sliding windows search after the first 20 ([UPDATE] n =10 frames for faster response in adjusting lane marker) frames have been identified. I would have liked to decrease the averaged frames back to zero for each frame that was not detected, then do a sliding windows search once I decrease the averaged frame to a threshold value
3. Python structure. Although I was able to implement the class somewhat successfully, I don't feel that I was able to utilize the methods to the full extent as some of the calculations within the process image could have been methods and there are some places in the code where there is repeated logic add_xfitted and add_xfitted_m are a good example. Also, I used the scoping rules to get away with not having to declare a variable globally, but need to understand classes better before improving
4. Finally, the code is not robust enough to handle sharp curves, disappearing lines and washed out lanes. There is still parts of the normal.mp4 vid when the calculation for radius of curvature 'freaks out'. There averaging helps but I feel that there is more work to be done a) detection and b) poor lane image rejection