****Advanced Lane Finding Project****

The goals / steps of this project are the following:

1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
2. Apply a distortion correction to raw images.
3. Use color transforms, gradients, etc., to create a thresholded binary image.
4. Apply a perspective transform to rectify binary image ("birds-eye view").
5. Detect lane pixels and fit to find the lane boundary.
6. Determine the curvature of the lane and vehicle position with respect to center.
7. Warp the detected lane boundaries back onto the original image.
8. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

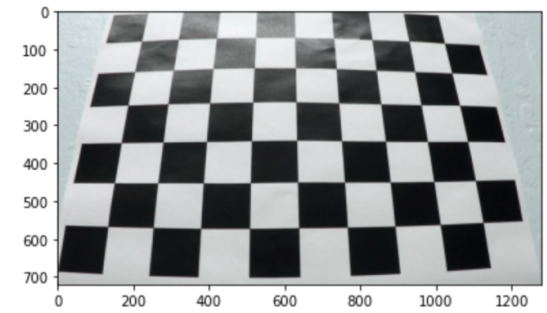
Writeup / README

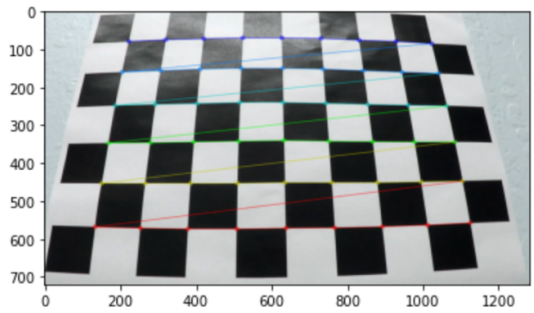
Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

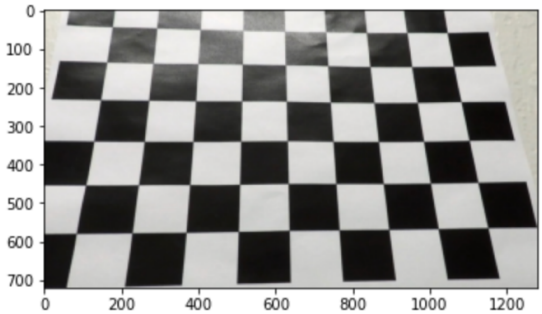
Camera Calibration

I began by reading in an image of a chessboard given to us. Counting the corners, I determined I needed 9 horizontal and 6 vertical corners detected. I created two arrays, *objectpoint*, that will be the 3D (x,y,z) points for real world plane and *imgpoint* that will be 2D (x,y) points for the image plane. These arrays will be appended every time the chessboard corners are found in a test image. The imgpoint will give me the pixel position for the corners.

Then I used cv2.calibrateCamera() function to obtain camera matrix, distortion coefficients, and also rvecs and tvecs. We use camera matrix and distortion coefficients with cv2.undistort() function to undistort the image.





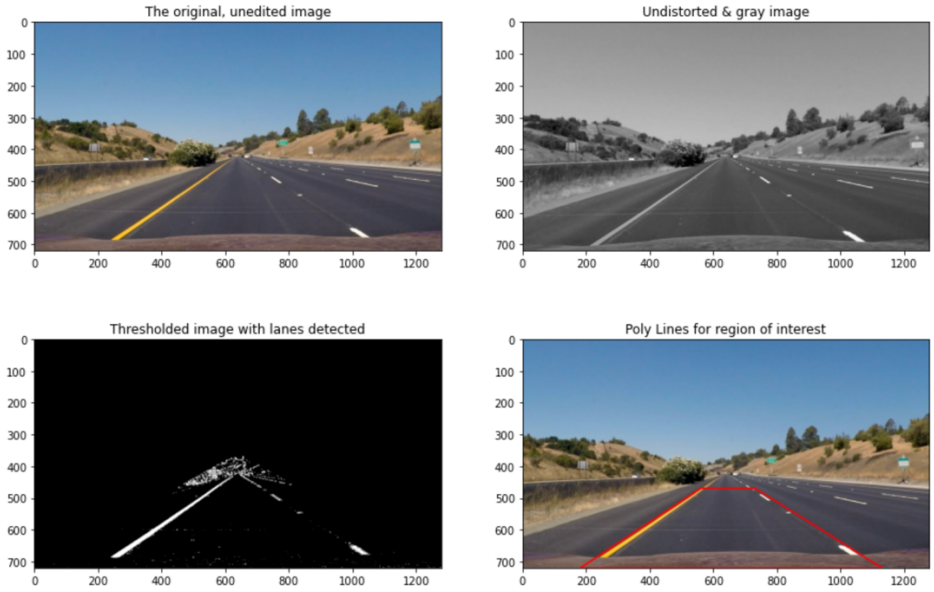


Color thresholding *(def sobel_process)*

Next, I imported a lane image and turned it to gray and then hls using cv2.cvtColor() functions. I divided the HLS image into H, L & S components, and took the derivative using only the L component using cv2.Sobel() function, in both x & y direction.

Once I had the thresholds that detected the lane lines well, I combined the gradients and created a mask.

Then using vertices, I created polylines for region interest for warping, which is the next step.



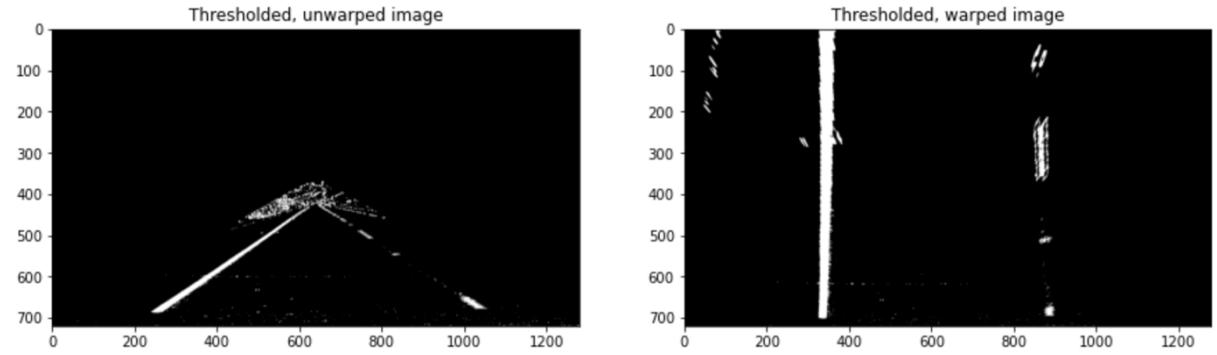
Perspective Transform *(def get_warped)*

For the perspective transform, I needed the source points and destination points. I go the following points:

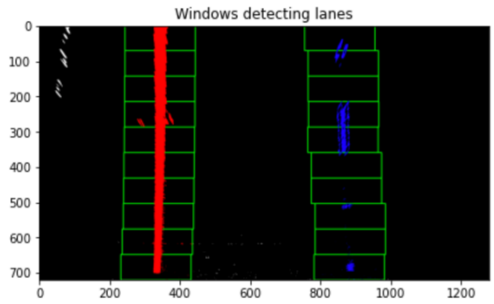
src = np.float32([180,720], [1130,720],[730,470],[560,470])

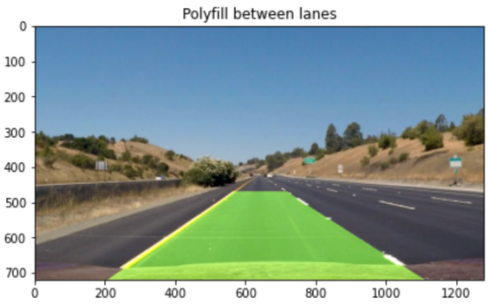
dst = dst = np.float32([[320,720],[900,720],[900,1],[320,1]])

Then I used the cv2.PerspectiveTransform() function to get the matrix of perspective transform points. We use this with cv2.warpPerspective() function to get the following results. We can also warp first and color grade later but I found this to be an easier approach.



Histogram & Sliding window *(def get_pts, def line_fit, def avg_line)*
To detect the lanes, we will use histogram peaks and then create a process to find lanes in the warped image using sliding windows. For this, we use draw the windows first and then stack them over the detected lanes.
Finally, we fill color inside detected lane points and lay it over the original image.





Radius curvature *(def avg_radius)*
I didn't do the curvature for a separate image but the video will have a constant radius and curvature points running through it as the frames change.

Pipeline (video)

The video is imported and then each frame is processed through process_img() which will go through all the steps above and return the frame to be written in a videofile. This videofile is in output_images folders, saves as project_video_test.mp4.

Discussion

There were several challenges I faced while working on this project such as getting lane points and thresholding shadows and light with different color spaces. I was able to solve these challenges for the most part but had to experiment with color thresholds a lot to find lane points during the tree shadow patch. Combining thresholds was also a little challenging at first. A simple approach of "or" function didn't work so I had tried implementing sxbinary with all the colorspaces and played around until I found the perfect combination.

Initially, I also had difficulty finding exact lane points because my thresholds weren't good enough to find them. It took me a while to realize this and had to review a lot of changes I made thinking the fault in code was due to lane detection function.

My pipeline worked well with the project video given but it would be difficult for it to detect lanes with rapid turns such as the ones in harder challenge video. This is because my source points create vertices at points that will not be out of lane limits for it. Additionally, the color thresholds will require adjusting along other challenges like the tree shadows we had in this one. I considered taking shorted length of lane for detection and fillpoly buy that concluded to

be more difficult at the light/shadow patches. If I tried to define the color spaces better, I might be able to solve this problem as well.