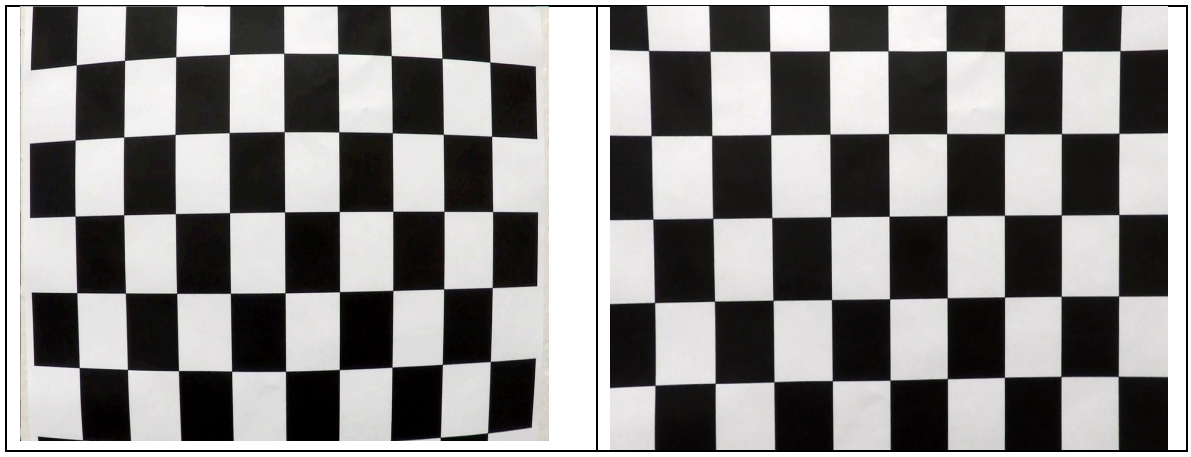
Advanced Lane Finding
Project 2

Sarbjeet Singh
January 12, 2019

Camera Calibration

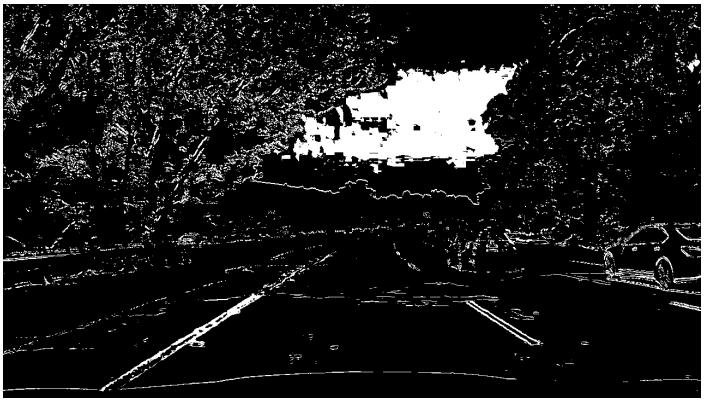
To calibrate the camera, nineteen images of a chessboard in various positions with respect to the camera are used. The images are iterated through the OpenCV function findChessboardCorners. If the function is able to find the corners within the chessboard, the object points (3D points which represent the ideal position of the corners) and the position of the detected corners are added to two separate lists, made up points from all images with detected points. This list of object points and the corners in each image is than used to undistort any image by computing the camera matrix and distortion coefficients using the OpenCV function calibrateCamera. The returned variables are passed to the undistort function within OpenCV which returns an undistorted image. Below is an example of a distorted image on the left and the undistorted image on the right. All of this is carried out in the first code block of the jupyter notebook.



The Pipeline

The first few steps in the image pipeline are to process the images through various steps to reach an image where the lanes lines can easily be found. The first step is to remove distortion and this is done with the same process described in the camera calibration step. An example of a distorted image (right) and undistorted image (left) is shown below. In code the function 'undistort' uses the variables found in the calibration step to return an undistorted image.

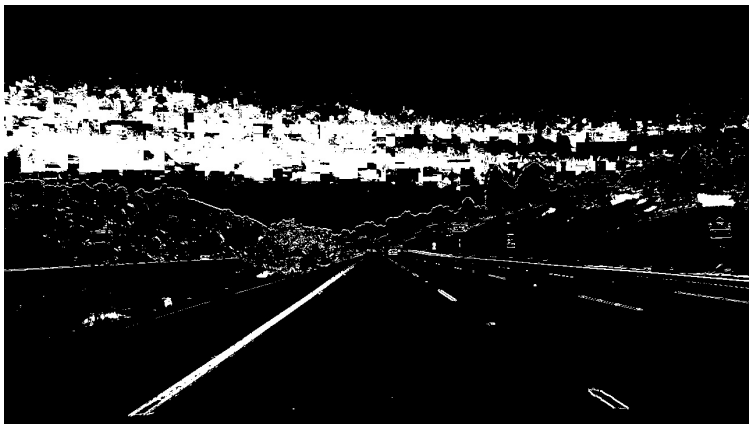
The next step in the code is to generate a binary image, which retains the lane lines and removes as much of everything else as possible. This is done in the function called 'threshold' which is passed an image. A combination of thresh holding methods are used. To develop which thresholds are most effective, I used each method (Sobel, Color Channels, Magnitude) on the set of 6 images and identified the advantages and disadvantages for each one by visually inspecting the results. This allowed me to determine which methods could be multiplied together and which could be added to achieve a final image that is highly effective for the lane finding algorithm to process on. The image shown below is achieved by finding the following binaries: L_Channel, S_Channel, SobelX, and SobelY. These binaries are than combined as follows: [((s_binary == 1)&(l_binary == 1))|((sy_binary == 1)&(sx_binary == 1))|((l_binary == 1)&(sy_binary == 1))]
The results for a few images is shown below:



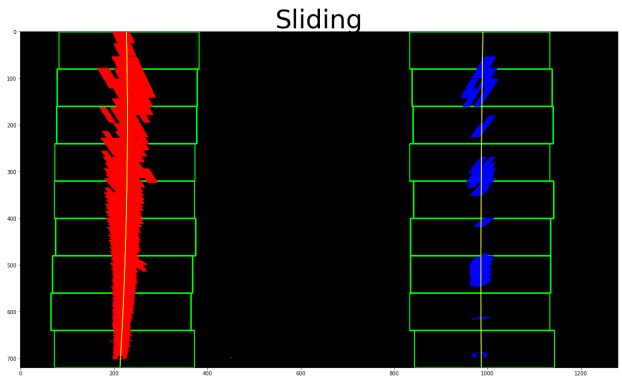
The next step in code is to perform a perspective transform and is done using the function called 'perspective transform'. This requires determining four source vertices and four destination vertices. The source vertices represent the points on the original image that I want to shift to the destination positions on the transformed image. To determine my source vertices, I utilized the road image where the lines are straight and transformed them to my destination points (which are easier to determine, in this case they form a rectangle). This process was trial and error, where I modified the source vertices until I achieved an output image that had the lines showing up as straight. These vertices are hardcoded and shown below:

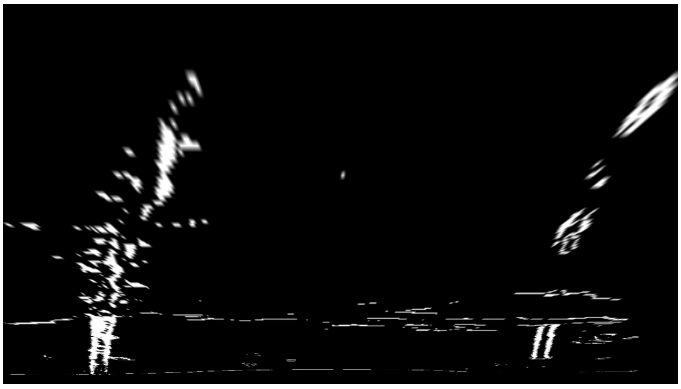
src = np.float32([[220,700],[595,450],[686,450],[1090,700]])
dst = np.float32([[200,700],[220,100],[1000,100],[1000,700]])

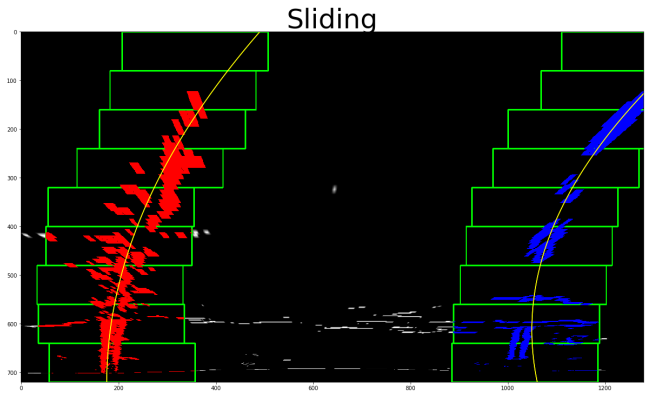
The vertices are passed to the OpenCV function getPerspectiveTransform, which returns a transform matrix that is passed to another function called warpPerspective, along with the image. The result is a "bird's eye view" image, examples of which are shown below.



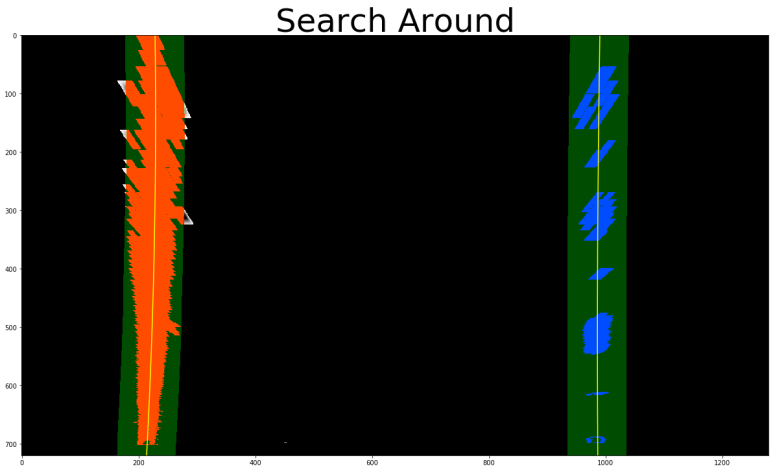
The perspective-transformed image of the lane can be used to now find lane lines. I utilized two methods to do this. The first method involves determining where the lines are and than moving up the image at set intervals, charting out the pixels that make up the lane line. To find the starting points, a histogram of the activated pixels along the y-axis can be generated using the histogram function within numpy. The two largest peaks in the histogram represent the left and right lanes. Now two rectangles are generated (in my code 150px wide and the 80px high) as the starting of the left and right lane. Within the rectangle, all pixels are found and than the window is moved vertically by 80 pixels and the process repeated. At every shift up, the position of the window horizontally is checked and adjusted to the mean of the pixels found within it. This way it can track the line as it curves. Once all the lane pixels are found, a polynomial can be fit to pass through all the found pixels. This is done using the numpy function polyfit. I fit a second order polynomial to all lane lines so that curves could be fit. Below are images of the output from this method. In code, the function 'fit_polynomial' is used for the sliding window method. Within it, the 'find_lane_pixels' function which implements the sliding window algorithm and returns indices for the left and right pixels to 'fit_polynomial'.

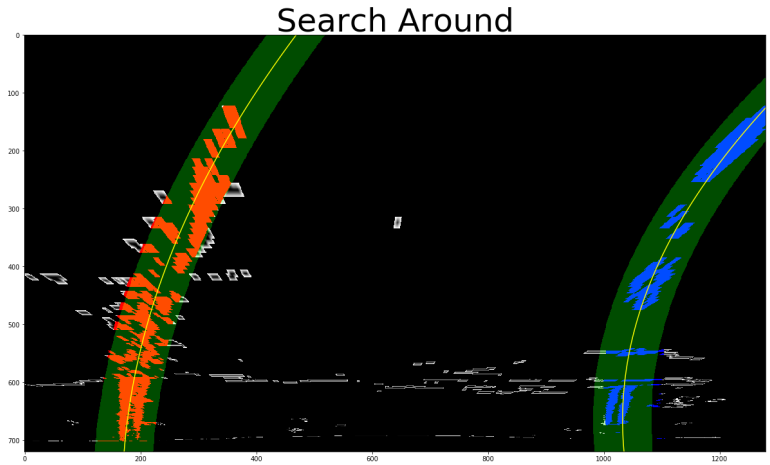






Another method to find lanes is the search around method. This method is only used once lanes lines have been found using the sliding window method. The search around method uses the polynomial found for the lane line and creates a window +/- 50px around the line. In code this is done by creating a shifted version of the polynomial location 50pix to the right and left and than searching for pixels within this region. The found pixels are than used to refit the polynomial. The function 'search_around_poly' implements this function is called by calling the 'fit_poly' function. Below are a couple examples of this method:





An additional step that is done in the “fit polynomial” functions of both methods described above, is finding the curvature of the polynomials. This is done by using the formula shown below:

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

Which is implemented in code as left_curverad = np.power(1+(2*left_fit_cr[0]*y_eval+left_fit_cr[1])**2,1.5)/(2*left_fit_cr[0])
right_curverad = np.power(1+(2*right_fit_cr[0]*y_eval+right_fit_cr[1])**2,1.5)/(2*right_fit_cr[0])

I did not solve for the absolute value of the denominator because if the curvature is positive or negative, it helps determine if the lanes lines are going left or right. This is useful in code when determine if both lanes lines make sense. If one lane line goes left and one right, you can reject the lines found.
Additionally, when finding the line curvature, the offset of the camera with respect to the lane center can be found. Under the assumption that center of the image should be the center of the lane, the offset is calculated by finding the point between the left and right lane and subtracting that from the image center.
The curvature and lane offset is converted from pixel values to meters using the conversion parameters:
ym_per_pix = 40/720 # meters per pixel in y dimension
xm_per_pix = 3.7/800 # meters per pixel in x dimension

The final result of all these processes is shown below. The function ‘measure_curvature_real’ is used in code to perform these steps.

The lanes lines that are found are warped back to the original perspective using the inverse transform matrix and projected onto the undistorted image. The curvature of the lanes lines is averaged and shown at the top of the image, and below it is the lane offset.

Discussion

There were numerous issues faced in implementing this project. The first issue was determining how to threshold the image to output a binary image, that could be used to easily determine the lane lines. The difficulty with this is that the different methods give inconsistent results around shadows and lane color changes. Thus determining which methods to use and how to combine them posed a challenge. This is also a point where the pipeline can fail. The current threshold function is tuned for the 280 highway with good lighting, as seen in the project video. Under different conditions however, such as more shadow, less visible lane lines, the current thresholds can fail and output lackluster results. To improve this, further investigation of computer vision techniques would have to be explored, or a more dynamic and complex method of changing threshold parameters on an image until lane lines are found would be needed.

Another issue faced was how determine what are good lines and what are bad lines in each image. There are numerous attributes of a line, and selecting which ones were effective to find good pair of lines without adding immense overhead with calculations posed a challenge. My chosen method involves checking if the lines have the same curvature and are curving in the same direction. The two lines average curvature is than compared to the previous average values and if it is within a certain percentage amount, the lines are deemed good.

One major failure that could occur in this image pipeline is that if the pipeline finds false positive lines at the beginning, the default values used to determine future good lines would become skewed. The bad lines can also skew the process that averages good lines over time and the pipeline could take a while to recover.