Advanced Lane Finding

Goals:

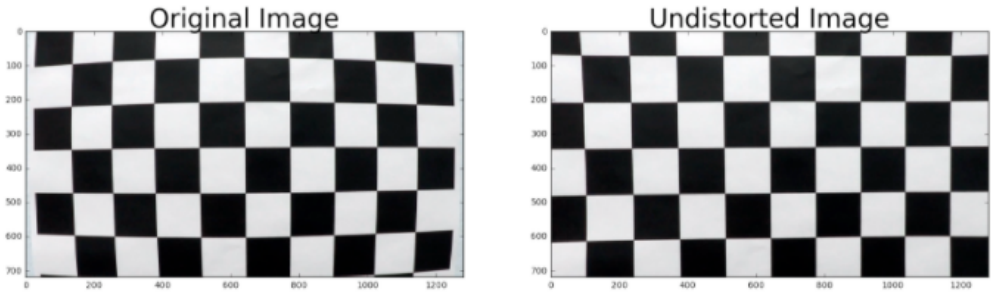
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Use color transforms, gradients, etc., to create a thresholded binary image.
- 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.

Methodology:

Step 1: Computing the camera calibration matrix using chessboard images

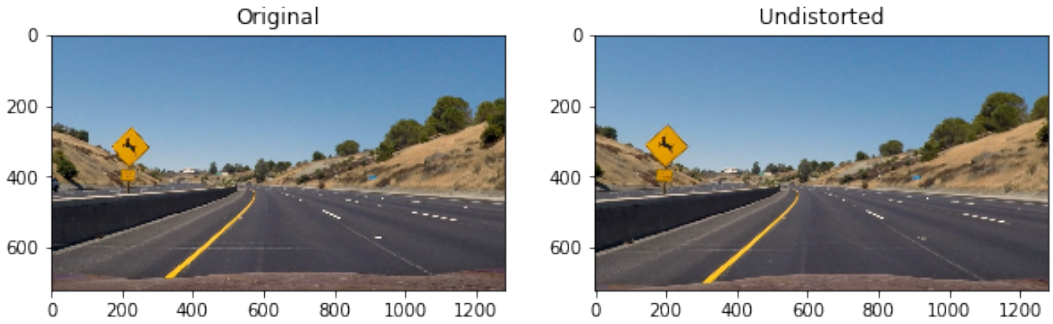
As explained in the lectures, I selected the "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, “objp” is just a replicated 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.

Then we can use the “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:



The code for this step can be found in the cell 2 of the attached IPython notebook.

Step 2: Applying distortion correction to raw images

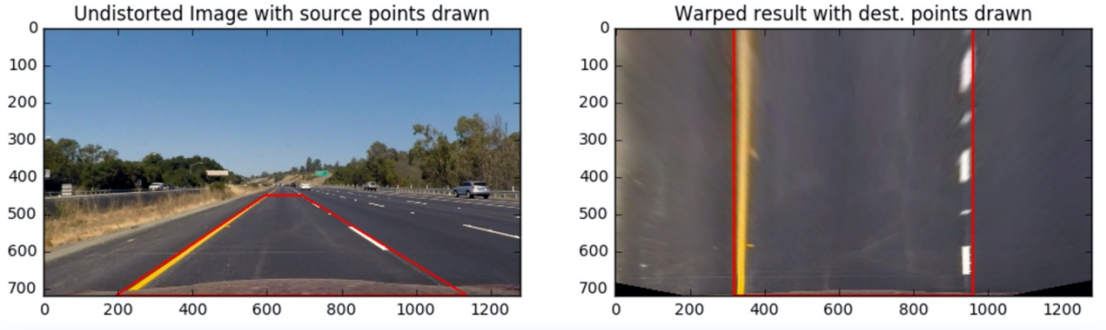


The code for this step can be found in the cell 3 of the attached IPython notebook.

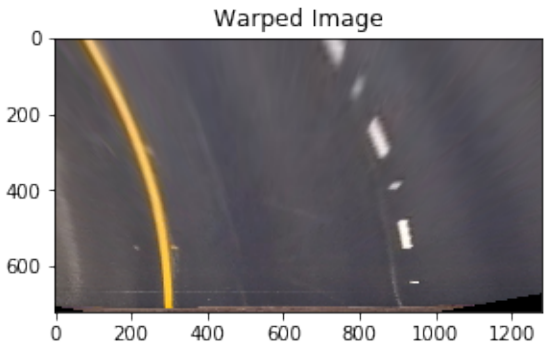
Step 3: Apply a perspective transform to rectify binary image ("birds-eye view)

I chose the source and the destination points by eyeballing the image plot. Then I obtained the Perspective transform matrix M using the cv2.getPerspectiveTransform. The warped image was then obtained using the cv2.warpPerspective.

Source	Destination
150,720	200 ,720
590,450	200 ,0
700,450	980 ,0
1250,720	980 ,720



Sample warped image



The code for this step can be found in the cell 4, 5 of the attached IPython notebook.

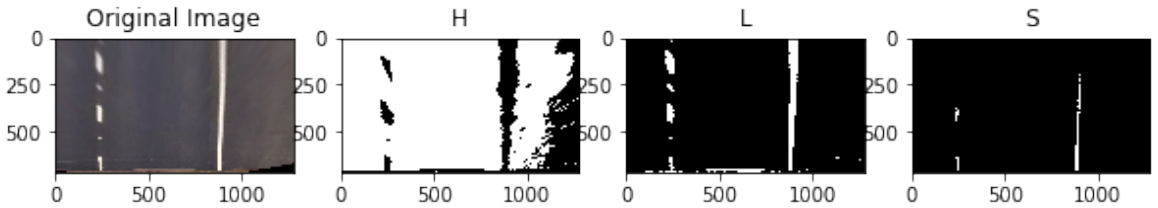
Step 4: Use color transforms, gradients, etc., to create a thresholded binary image

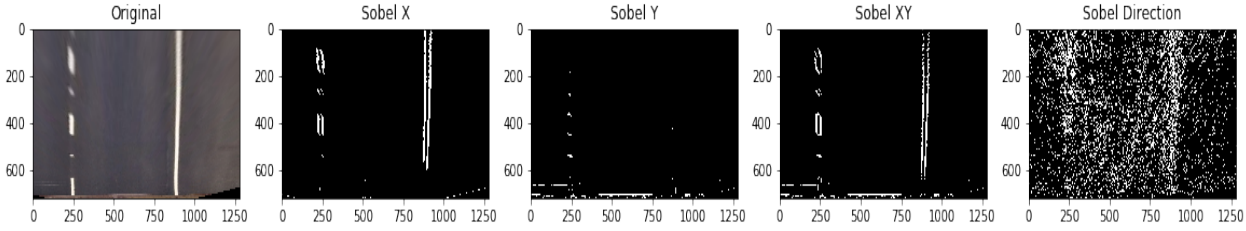
See code cell 7 onwards of the IPython notebook. I used “cv2.COLOR_BGR2HLS: color conversions

When we visualize individual components, we find that the S channel, and the L channel the above transformations respectively give a good indication of the lane lines.

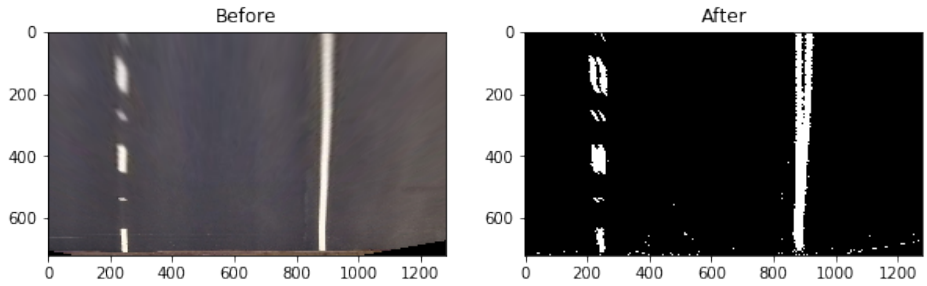
Also, I computed the sobel magnitude gradient in x, y and xy directions and also the direction gradient. From the images below, the best suited was the sobel x gradient.

Please see the individual components below,



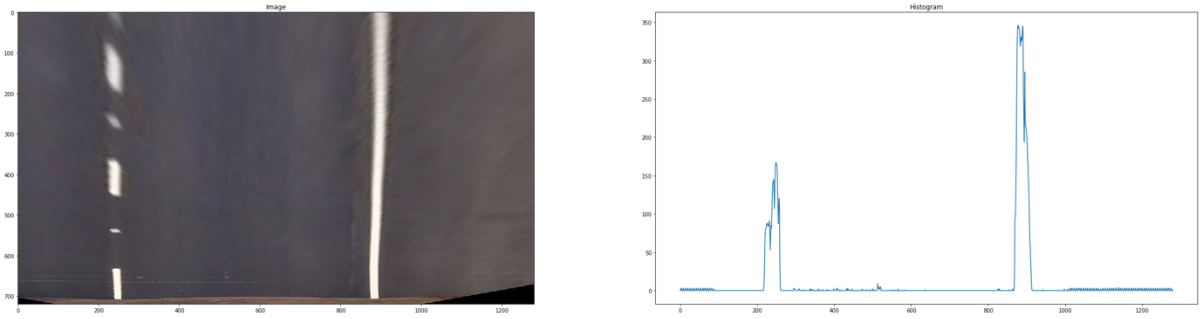


Combining the appropriate thresholds, I got the following result with was fairly good.



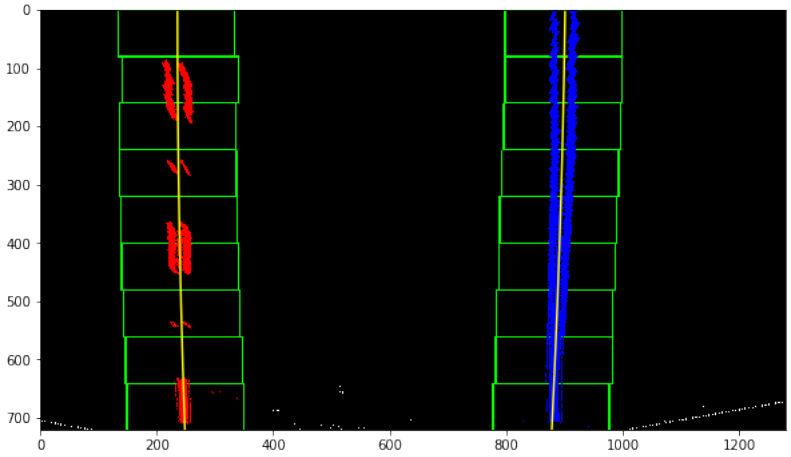
Step 5: Detect lane pixels and fit to find the lane boundary

From the below histogram of the warped image, we can see that the two peaks represent the left and right lane lines on the road. I used the sliding window method (Code cell 12) to detect the lane pixels.



To perform the sliding window search, the following steps were followed (cell 13-14),

- Find the peak of the left and right halves of the histogram for the left and right lane lines respectively.
- Identifying the position (x,y) of all pixels with non-zero values
- Identifying window boundaries in x and y
- Drawing the windows on the image
- Identifying the non zero pixels within the window and appending those pixels to corresponding lists for left and right lane lines.
- Using polyfit function to fit the identifies points in a second order polynomial.

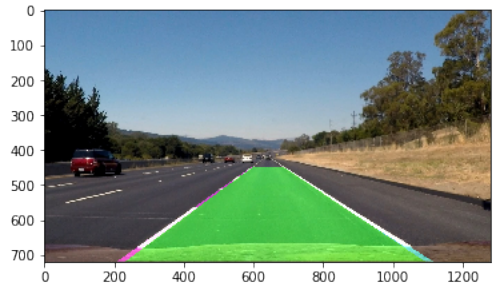


Step 6: Determine the curvature of the lane and vehicle position with respect to center

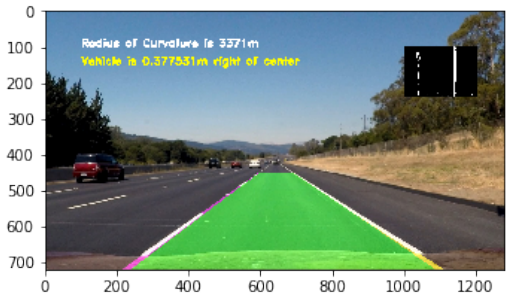
The code for this step can be found in the cell 18 of the attached IPython notebook

Step 7: Warp the detected lane boundaries back onto the original image

The code for this step can be found in the cell 16-17 of the attached IPython notebook



Step 8: Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position



The code for this step can be found in the cell 20 of the attached IPython notebook

Pipeline (Video)

My pipeline seemed to perform reasonably well on the entire project video. The output video can be seen in the files attached

Discussion

- My pipeline didn't seem to work really well on the challenge videos, there were some outliers
- If a polynomial fit was found to be robust in the previous frame, then rather than search the entire next frame for the lines, just a window around the previous detection could

be searched. This will improve speed and provide a more robust method for rejecting outliers.