Advanced Lane Finding write up

1) Calibrating the Camera

   Calibrating the camera is incredibly important. You can find the code in the AdvancedLines.ipynb notebook file in cell 6. I started by preparing object points which are the (x,y,z) coordinates of the chessboard corners in the world. I assume that 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 (x,y) pixel position of each of the corners in the image plane with each successful chessboard detection.

   I then used the output 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 obtain the following result.

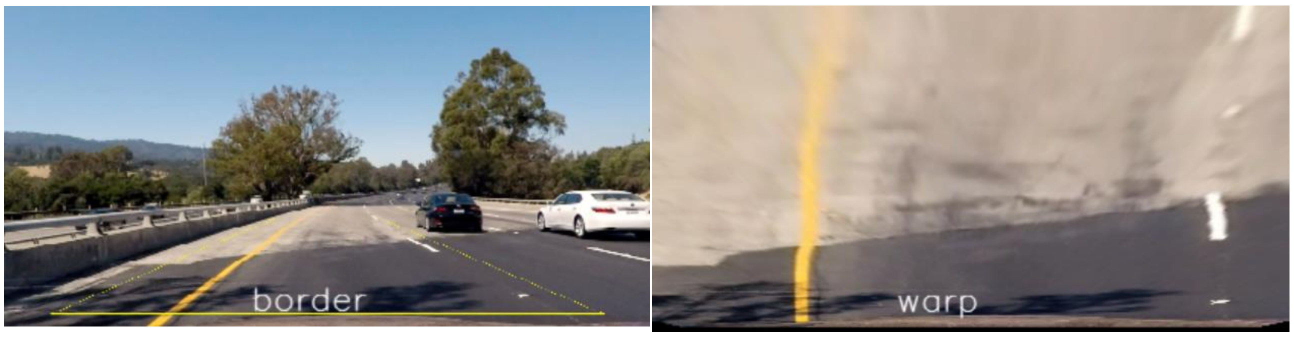
2) Pipeline (single image)

   a. The first step in my pipeline is to undistort the original image using cv2.undistort() and the objpoints and imgpoints from the previous step

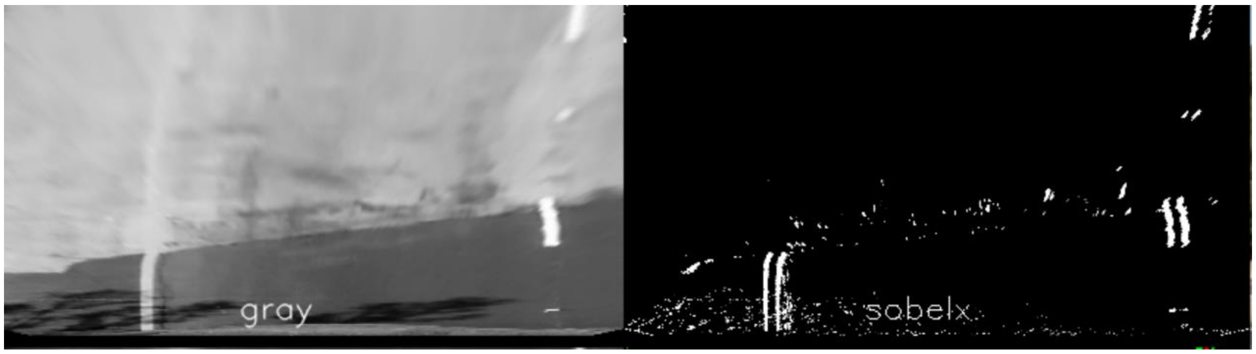
   As shown, there difference between the original and undistorted views is slight. The images on the edges are more stretched out such as the car on the right and the hood of the car.

   b. The next step was to create a warp transform of the image. This allowed me to view the road with a bird's eye view perspective. The area that I targeted is highlighted in yellow in the 'border' image.
   I hardcoded the source and destination points:

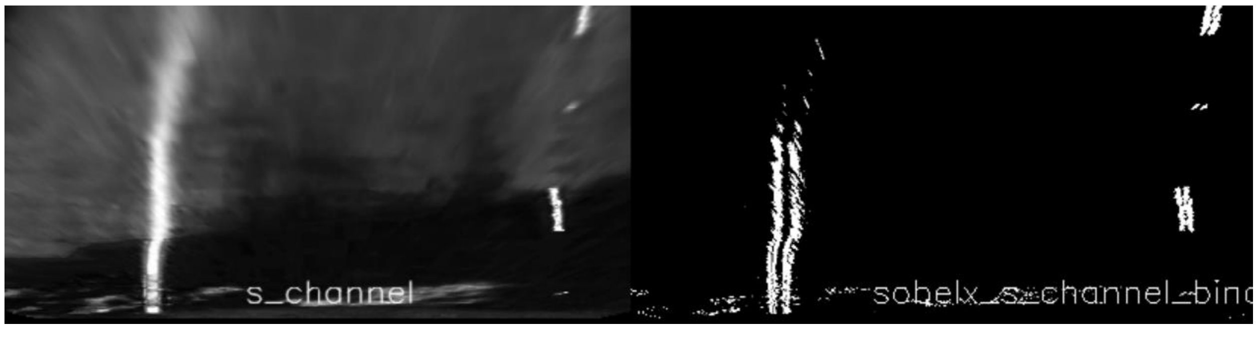
Source	Destination
535, 465	50, 50
745, 465	1230, 50
95, 670	50, 670
1185, 670	1230, 670



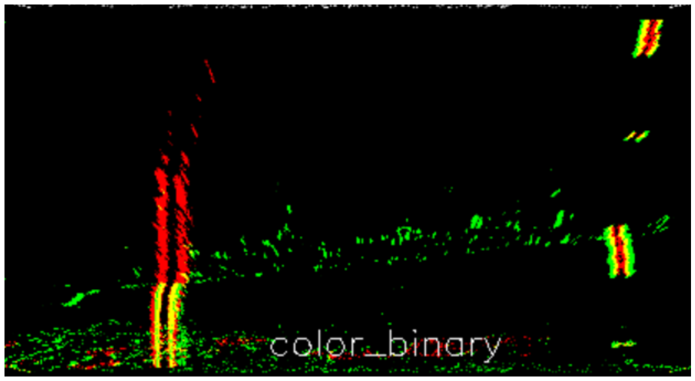
c. The next step I did was to create a copy warped image. On one of the copies, I converted it to gray scale and used cv2.Sobel on the x direction. The sobel function requires that the input only have 1 color dimension, so we convert it to gray scale beforehand. I then created a binary version of the image by saying any pixel with an amplitude above s_th_min but less than s_th_max will equal 1, while the rest is zero. As you can see, the sobelx on the gray scale image doesn't pick up the yellow line on the white road very well.



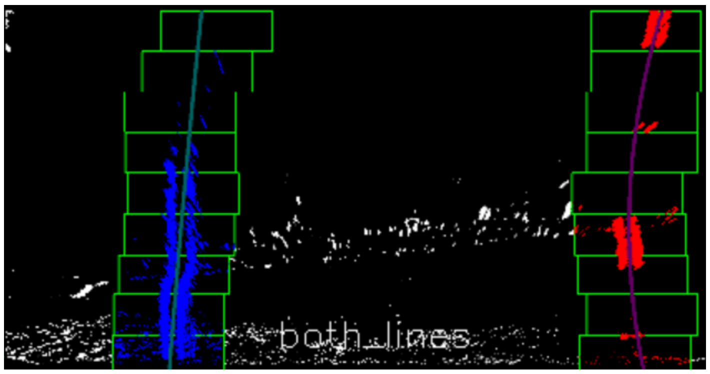
On the other copy, I converted the color to HLS. This is because the saturation channel helps us differentiate the line when the road and the lines look similar. I also used sobel in the x direction to prevent the s_channel from going crazy when we encounter shadows.



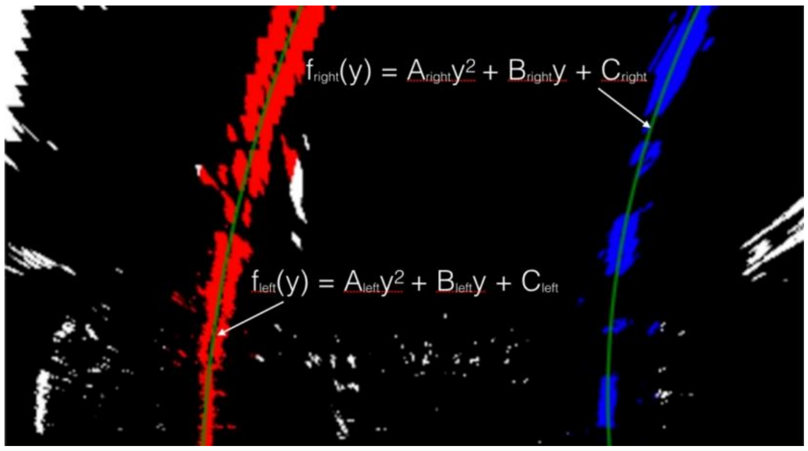
d. Combining the two binary channels gives us the following image (Green is from the Sobel'd gray channel and red is from the Sobel'd saturation channel.)



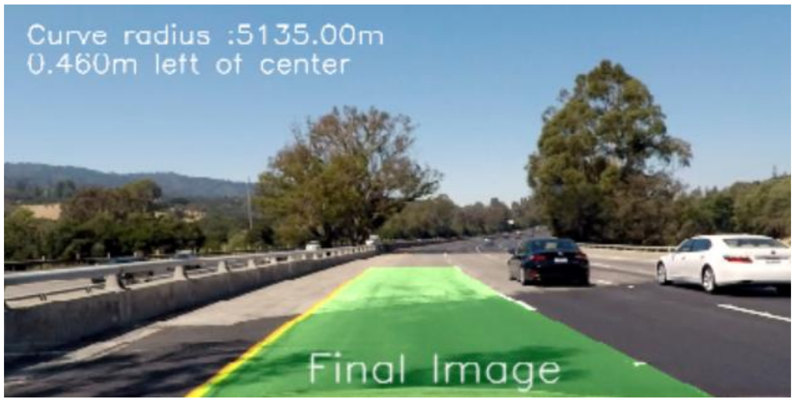
e. I then used the sliding window method to detect the lines.



I then used np.polyfit to find the second order polynomial coefficients of the line. The image below explains the curve equations.


$f_{right}(y) = A_{right}y^2 + B_{right}y + C_{right}$
$f_{left}(y) = A_{left}y^2 + B_{left}y + C_{left}$

f. Finally, I used the two lines and created a border using cv2.fillPoly and unwarped the image back to the calibrated version.

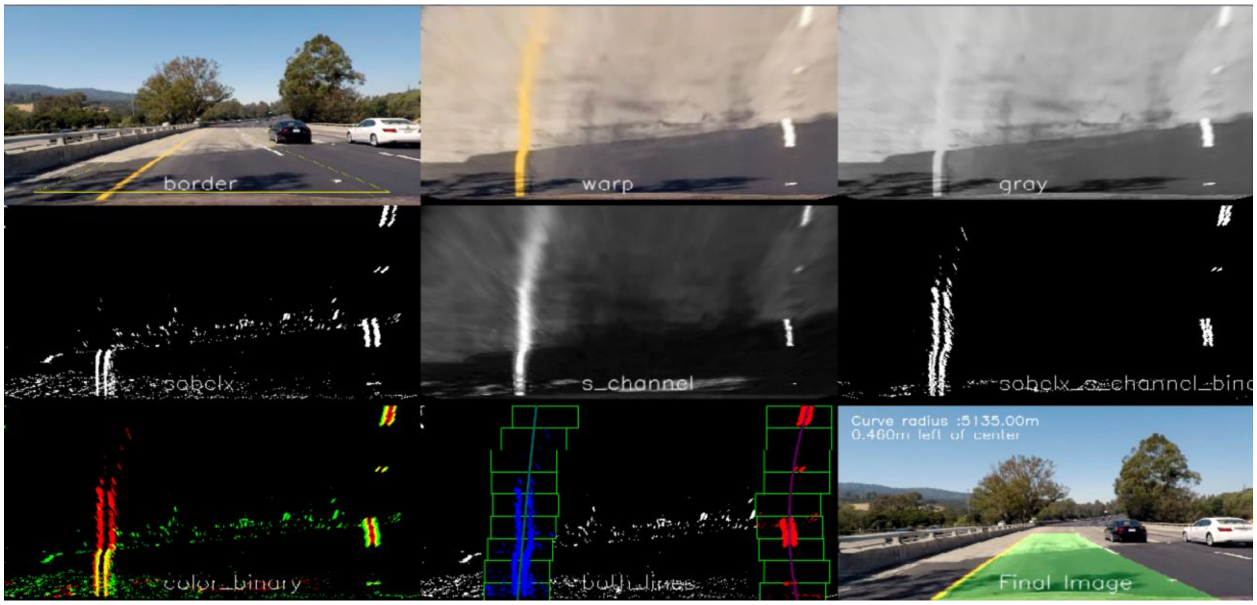


g. To calculate the radius and distance from center, I used the functions findRadius and findDistFromCenter. The function findRadius is a function in the Line object. We first use the polynomial coefficients for the line (after they've been translated into world space coordinates) and then solve for the equation:
Radius = (1 + (2*ws_cfft[0]*y*ym_per_pix+ws_cfft[1])^2)^1.5/abs(2*ws_cfft[0])

Where ws_cfft are the polynomial coefficients translated into world space coordinates, y is the point of evaluation (720 pixels), ym_per_pix is the translating factor that determines how many meters are in a pixel in the y direction.

To find distance from center, we took the x intercepts of the two lanes and averaged them together and named it our 'laneCenter'. We then took the difference between that point and the center of the image (the center of the image represents the car's position). We then translated the difference into meters by multiplying it by xm_per_pix which represents how many meters are in a pixel in the x direction.

h. To help debug my pipeline, I created a mesh of all of the images and put them on one frame. By visualizing every step, I was able to see where my code could have potentially done something incorrectly and was able to fix any outstanding issues.



3) Pipeline (video)
    a. The pipeline for a video is very similar to that for a single image. We used the processImage file to create a series of 'mesh' frames to create the video. That way, I can see where the weak points in my pipeline were.
    b. Another key difference is using the 'ezFind' function to determine the lines. Instead of using the 'slidingWindows' method on every frame, we can use 'ezFind' if we found a good line in the past n frames. Instead of starting from scratch, we start from our last position (lines shouldn't change too much between frames). This is a much faster way of finding lines and also helps us from detecting an incorrect line in case a particular frame is noisy.
4) Discussion
    a. Problems
        i. I noticed that the Sobel outputs from the gray and Saturation channels can produce noise. Since my threshold is set so low, it picks up these noise measurements and adds them to my final binary data. A solution to this is to increase my threshold, but sometimes the noise has a higher amplitude than my actual line. When this occurs, my line is measured incorrectly.
        ii. Another issue my algorithm has is when there is a sharp turn. This is because I am using the Sobel function in the x direction. This might be improved if I added the Sobel function in the y direction.
        iii. Finally, my pipeline has trouble if there is another line within my mask such as a split in the road or a dark shadow. When this occurs, my algorithm will pick up points from both the main line and the extra line and will combine the two, creating a bizarre line that takes a while to reset (this is why my pipeline fails the challenge video)
    b. Improvements
        i. To make it more robust, I could blur the images before sending it through the pipeline. This will reduce the amount of noise that is created by shadows from trees or differing types of pavement.

ii. Another way to improve it will be to ‘AND’ two or more filters together. This will drastically reduce the noise, however, it will also reduce the amount of points we get which may cause us to extrapolate an incorrect line.

iii. Finally, I could get a more accurate result if I changed the way I create my binary arrays. The way it currently works checks if a certain pixel’s amplitude is above a minimum threshold. If it is, then we set it to ‘1’, otherwise it’s set to ‘0’. Instead of doing it this way, I can add another layer to this selection by multiplying it by a random factor. If the pixel value is low, then the probability will also be low. This way, the pixels with amplitude closest to 255 will always appear, but those barely passing the threshold will randomly disappear.