SELF-DRIVING CAR NANODEGREE

PROJECT 4
ADVANCED LANE FINDING

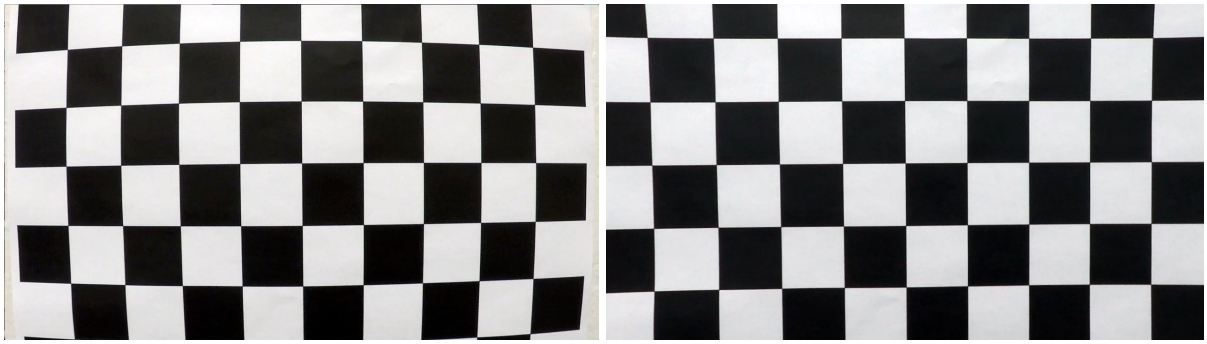
The goals / steps of this project are the following:

- 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.

Camera Calibration

The images provided in the “camera_cal” folder was used for calibration. OpenCV’s findChessBoardCorners function was used on all the images to find the points and obtain camera parameters such as rotation and translation vectors, camera matrix.

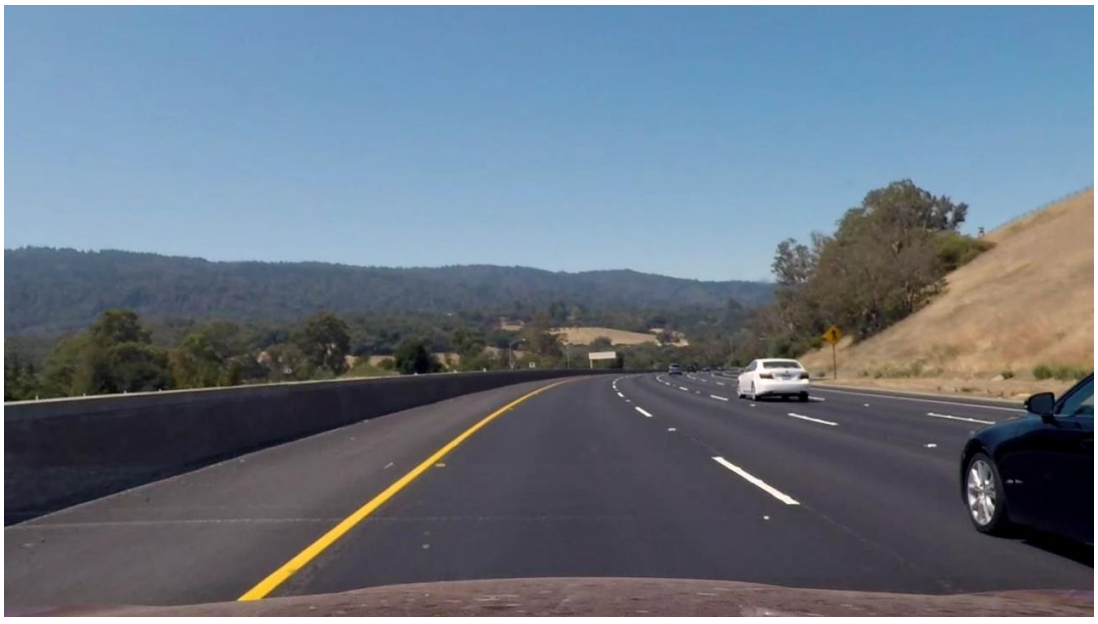
In the figures below **original** image (distorted) is displayed on the **left** and **undistorted** image is displayed on the **right**.



Here as seen in the code in the first two cells of the notebook, I used objpoints array to represent the world coordinates of the corners on the chessboard. imgpoints represent the pixel coordinates of the detected corners. These points were then fed to the cv2.calibrateCamera() function to obtain camera parameters. I used cv2.undistort() function to obtain the image on the right using the camera parameters obtained in the previous step.

Pipeline

Step 1: Obtain undistorted frames. Below is an example of an undistorted image



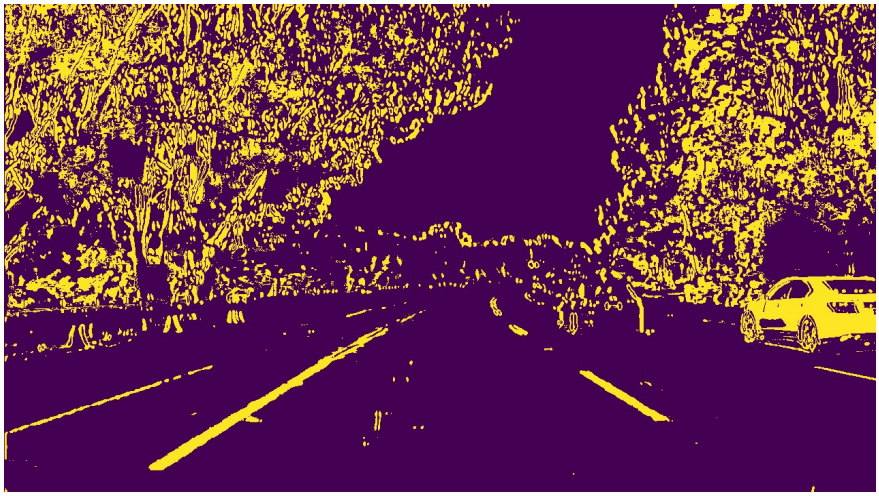
The above undistorted image was obtained using the camera parameters obtained during the camera calibration. cv2.undistort() function was used for the purpose.

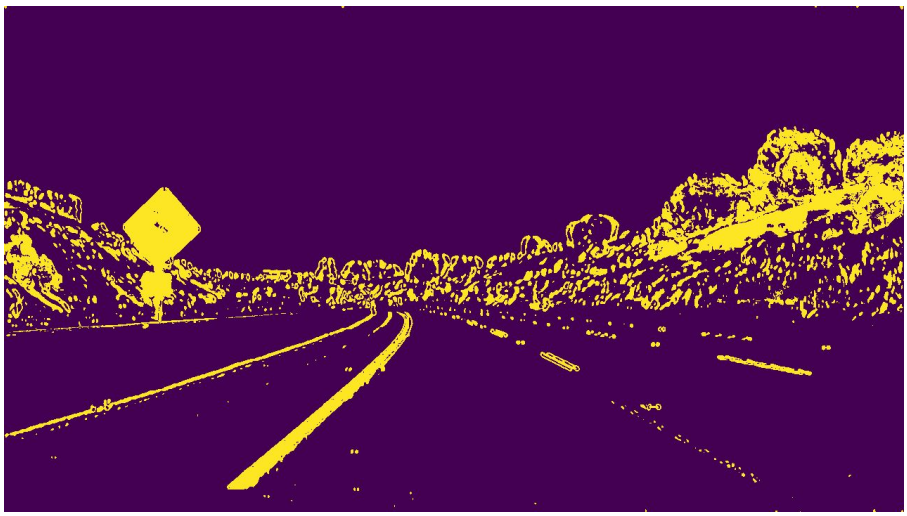
Step 2: Apply various threshold parameters to obtain images with only lane lines visible.

For this purpose I have used two methods namely sobel operator and thresholding in HSL color space. For sobel operator, I first convert the image to grayscale. Then I use cv2.Sobel() function with kernel size 15. This kernel size was chosen based on trial and error method where I found that this kernel size worked well for my images. Then the resultant image is subjected to threshold specified, to obtain lane lines as clearly as possible.

For the next part I convert my image to HSL color space using cv2.cvtColor() function. Then I have defined lower and upper thresholds for all three channels Hue, saturation and lightness. After various trials, I settled on the following values lower = np.uint8([0, 200,0]) upper = np.uint8([255, 255, 255]) to obtain the white lines from the image. I use cv2.inRange() function to obtain the mask which returns values only in the specified threshold. To obtain yellow lines in the image I use following thresholds lower = np.uint8([10, 0, 100]) , upper = np.uint8([40, 255, 255]). Since both lines are required I combine these two masks to obtain my final mask.

There were other methods which were explored such as using the HSV colorspace threshold, magnitude gradient thresholding. But these did not provide excellent results and thus I settled on combining the above methods to obtain a good mask.



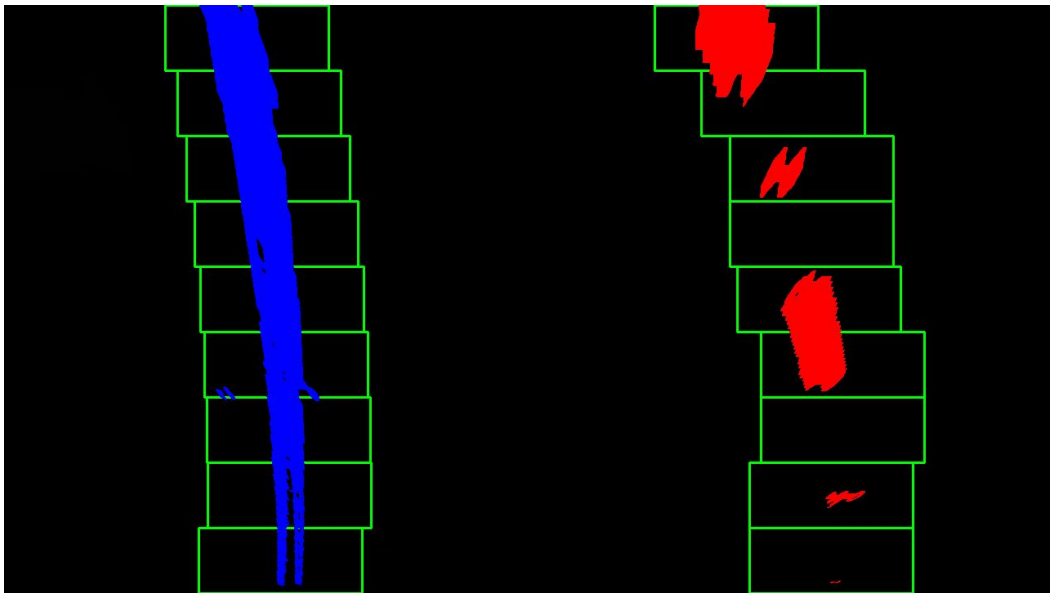


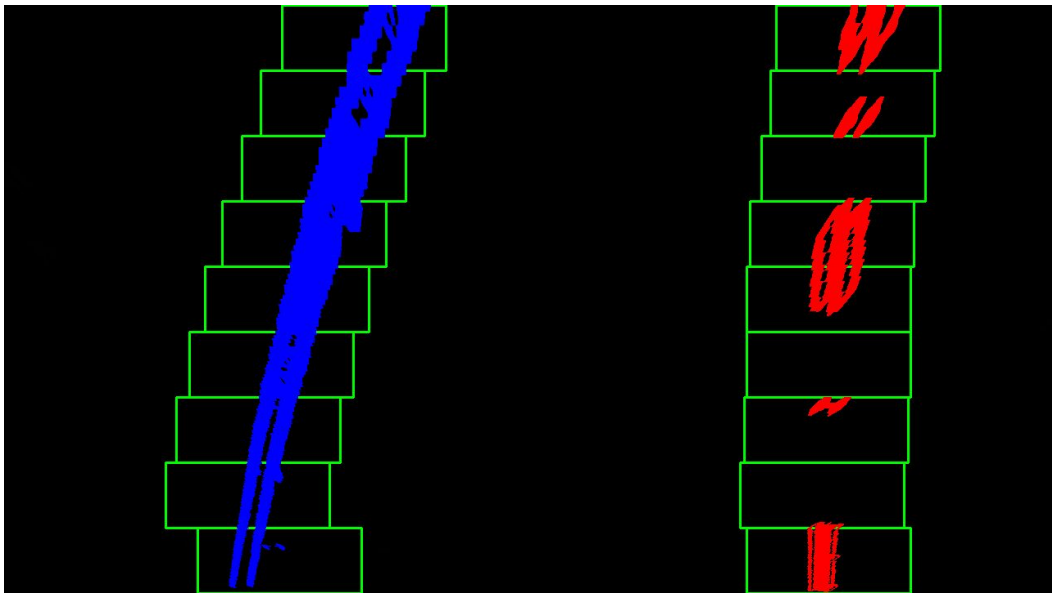
Step 3: Perform perspective transform to obtain bird's eye view of the lane lines.

Here I define a region of interest where lane lines are present. Using this as the source points and defining the destination points which map it to a bird's eye view I obtain the perspective transform. I use cv2.getPerspectiveTransform() function for the purpose. Using the resultant matrix I have obtained perspective matrix of all the frames. Few examples are shown below. cv2.warpPerspective() helps with this.

Step 4: Use the perspective transformed images to find lane lines.

I first obtain the histogram of the image to detect the possible position of the lane lines. The peaks in the histogram will help in identifying the lane lines. Number of windows is set to 9 and the height of the window to height of the image. Margin is set to 100 and minimum pixels is set to 50 to obtain reasonable approximation of the lane lines. Once obtained rectangles are drawn at each detection. Few examples are shown below





Here I apply this for only first few frames until I find a polynomial to fit the lane line curve. One different approach I have taken here is, once I find the points that indicate lane lines I store them in an array. I store the points only for 10 frames. So when I process new frames, I find the set of points for that frame and it to the list of points from previous frames and take the average. I set this average as the set of points for the new frame. This helped me improve the performance incredibly. Since I do this only for 10 frames the average that I obtain is more stable than just obtaining the points for one frame and using them. This has made the pipeline more robust.

Step 5: Calculate radius of curvature and position of the car in the lane.

Obtaining the radius of curvature is shown in the jupyter notebook cell titled “AFTER FINDING THE LANE LINES”. Here first we obtain the curvatures of the left and right lane lines and then average it to obtain the radius of curvature. Since these calculations are done in pixel space, an approximate conversion method is used to convert them to metres.

On the y-axis this conversion is 30/720 metres per pixel. On the x-axis this conversion is 3.7/700 metres per pixel.

Since we are assuming that the camera is mounted on the center of the car, the position is calculated as shown in the next block of code below the radius of curvature calculation.

Few examples of the final image with lane lines, radius of curvature and position of the car is as shown below


Radius: 1173.14779204 545.150461692
Position of the car: 0.0169003296759m from the center


Radius: 1666.01879938 784.249973781
Position of the car: 0.0273814417763m from the center
Farm
Hill Blvd