SELF-DRIVING CAR ENGINEER NANODEGREE

PROJECT #2

ADVANCED LANE LINE FINDING

SUBMITTED BY

EESHAN VIJAY DEOSTHALE

Project Goals

This project of Finding Lane Lines on Road has following goals:

1. Create a pipeline to identify lane lines on road given a camera image
2. This pipeline should use camera calibration and image filtering techniques to identify lane line pixels
3. Detect lane lines and fit polynomials for left and right lane lines
4. Determine curvature of the lane
5. Warp the detected lane lines back to original image
6. Apply this pipeline to a video stream
7. Output visual display of lane boundaries and numerical estimation of lane curvatures
8. Describe the pipeline and reflect on the process in written form

Structure of the Pipeline

The code for this project is located at

CarND-Advanced-Lane-Lines-Eeshan\examples\P2.ipynb

Python programming language was used to develop this pipeline. The pipeline to identify lane lines consisted of following steps.

1. Compute the camera calibration matrix and distortion coefficients given a set of Chessboard images.
2. Load an image of road in the form of a numpy array.
3. Apply distortion correction to undistort the image.
4. Compute the perspective transformation matrix by observing the lane lines in the test image to get a birds-eye view.
5. Use color transforms, gradient thresholds etc. to create a filtered/thresholded binary image.
6. Identify a region of interest on the image for lane line detection and mask the rest of the image.
7. Identify lane line pixels for left and right lane lines to find lane boundaries.
8. Determine the lane curvature and vehicle position with respect to the center of the lane.
9. Check if enough pixels were detected to identify lane lines properly and if the lane lines are parallel etc.
10. If the detection was perfect, use a lookahead frame count to search around previous detections in subsequent frames to save time.
11. Discard the frame and use previous detection if the new detection is not perfect.
12. Output visual display of lane boundaries and numerical estimation of lane curvatures.

Camera Calibration

The first part of the code is to compute a camera calibration matrix. The code for this step is contained in the first code cell of the IPython notebook located in "./examples/P2.ipynb".

First, an array for object points (*objp*) is created to represent an ideal 9*6 Chessboard. These are 3D points represented by 'x,y,z' coordinates, z-coordinate being 0.

Now for each image in the camera calibration directory, if the chessboard corners are detected by the *cv2.findChessboardCorners(gray, (9,6), None)* function, '*objectpoints*' are appended by a copy of '*objp* and the *imagepoints* are appended with the detected corners.

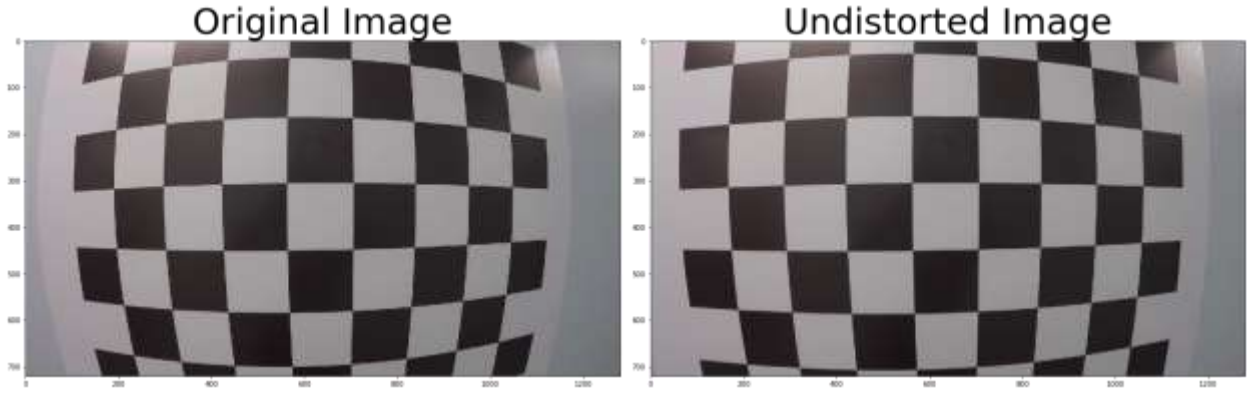
ret, corners = cv2.findChessboardCorners(gray, (9,6), None)

Now, these image-points and object-points are used to compute a camera calibration matrix as

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size, None, None)

The next code cell tests the camera calibration by applying distortion correction to a test image

undistorted = cv2.undistort(test_img, mtx, dist, None, mtx)



Perspective Transform

Next step is to compute a perspective transform matrix.

Code Cell 3 provides a code to compute a perspective transform matrix from a test image. The source and destination points are hardcoded as the mounting position of the camera is fixed and the region of interest to identify lane lines is only a few meters ahead of the car.

 # Identifying points on the image for perspective transform

 # source is a trapezoidal shape

src = np.float32([[300, 680], [1100, 680], [875,550], [460,550]])

 # destination is a rectangular shape

dest = np.float32([[300, 680], [1100, 680], [1100,550], [300,550]])

The perspective transformation matrix is then calculated as

M = cv2.getPerspectiveTransform(src, dest)

Minv = cv2.getPerspectiveTransform(dest, src)

warped = cv2.warpPerspective(test_undist, M, img_size)

This code is then applied to a test a test image for visualization

Pipeline for single images

First, functions are defined to transform, mask, filter images as these functions will be called for each frame in the video feed. The functions that are defined are as follows

a. *apply_transform(img, M, img_size)*
b. *abs_sobel_thresh(img, orient='x', sobel_kernel=3, thresh=(0,255))'*
c. *s_channel_thresh(img, thresh=(0,255))*
d. *mask_image(img)*
e. *find_lane_pixels(binary_warped)*
f. *fit_polynomial(binary_warped, leftx, lefty, rightx, righty, out_img)*
g. *fit_poly(img_shape, leftx, lefty, rightx, righty)*
h. *search_around_poly(binary_warped, left_fit, right_fit)*
i. *measure_curvature_real(binary_warped, leftx, lefty, rightx, righty, out_img)*
j. *draw_lane_lines(warped, left_fit, right_fit, Minv, undist)*
k. *x_dist(ploty, left_fit, right_fit)*

The pipeline is as follows

1. Load and undistort an image

The test image from "./test_images/" is used to test the pipeline on a single image

img = mpimg.imread('../test_images/test2.jpg')

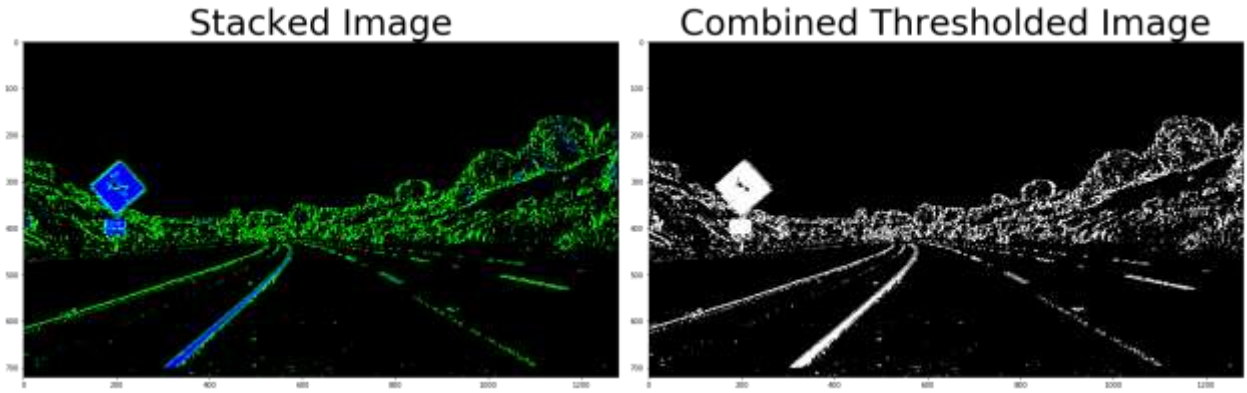
undist = cv2.undistort(img, mtx, dist, None, mtx)

The original and undistorted image are then visualized

2. Using color and gradient thresholds to create a thresholded binary image

Image transformed in HLS color space is chosen to extract pixels of interest to identify the lane lines. A combination of s-channel thresholds, and absolute 'Sobel' threshold in x-direction are used to create a binary image.

```
sobel_binary = abs_sobel_thresh(undist, 'x', 5, (20,90))
s_binary = s_channel_thresh(undist, (150,255))
combined_binary = np.zeros_like(s_binary)
combined_binary[(s_binary==1) | (sobel_binary==1)] = 1
color_binary = np.dstack(( np.zeros_like(s_binary), sobel_binary, s_binary)) * 255
```



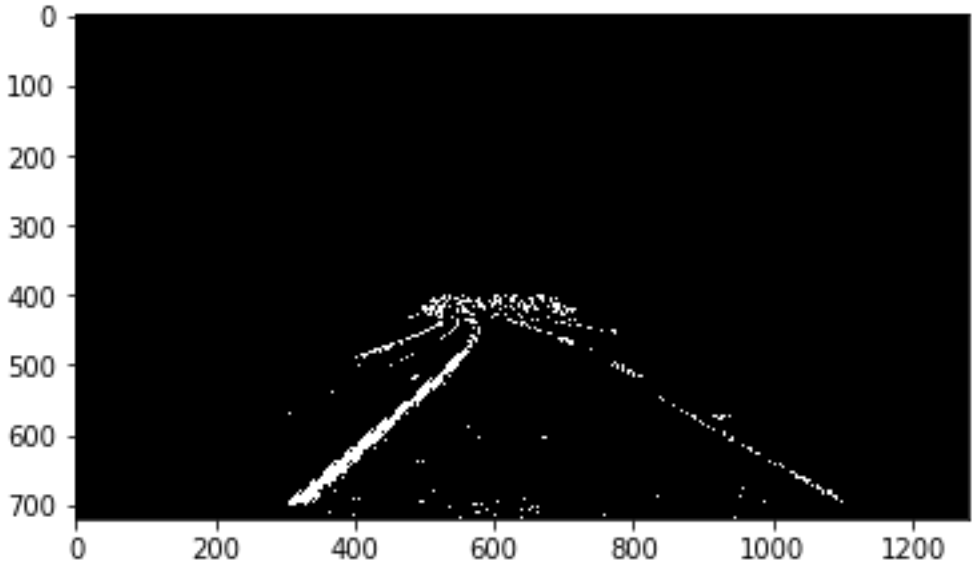
3. Masking the image

Since it is known that the camera is mounted on a fixed location on the car, the lane lines will always appear within a specific region in the image. This information is used to mask the image to only keep the region of interest which will contain the lane lines. The masking polygon is hardcoded.

```
vertices = np.array([[(100,imshape[0]),(520, 400), (680, 400), (imshape[1],imshape[0])]],
dtype=np.int32)
```

masked = mask_image(combined_binary)

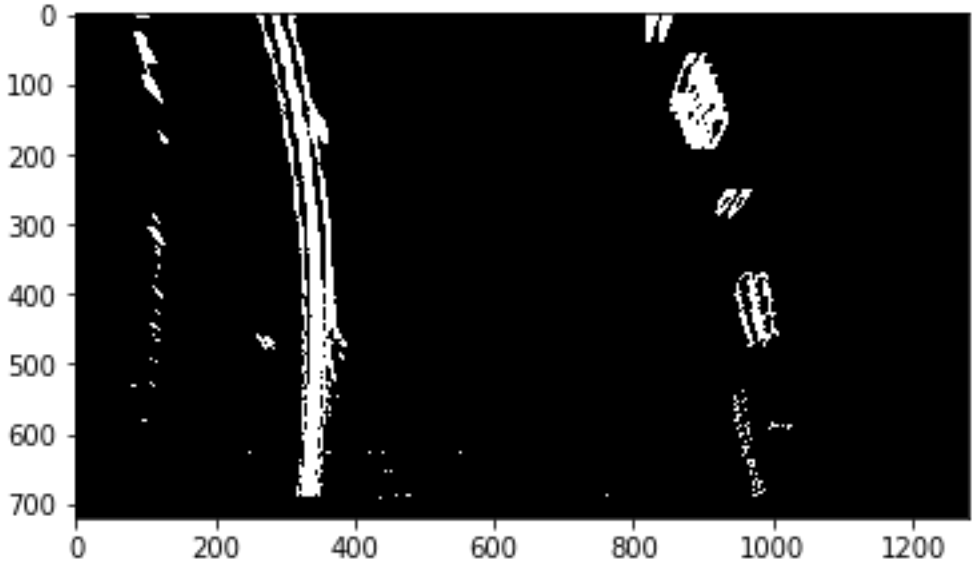
The masked binary image is as follows



Masked image is used henceforth in the pipeline to detect lane lines.

Perspective transform is then applied to this image.

warped = apply_transform(masked, M, img_size)



4. Identifying lane line pixels and fitting them into a polynomial

find_lane_pixels(binary_warped) function is used to identify left and right lane pixels.

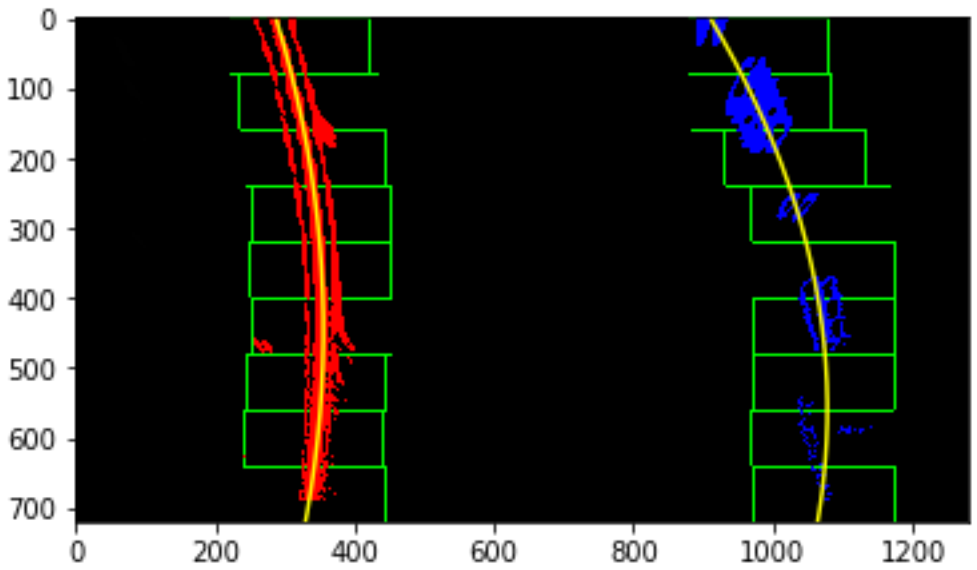
Algorithm described in course lectures is used to identify lane line pixels. Some modifications were made to shorten the length of the code. A brief description of the algorithm is given in the next paragraph.

The first step is to detect the start of the left and right lane line at the base of the image. For this, am intensity histogram of the bottom half of the image is taken to identify peaks in the left half and the right half of the image. A fixed number of pixels from both left and right border are ignored while identifying intensity peaks to ignore any noise at the image border that may come due to any grass or similar things if the vehicle is travelling in the corner lanes.

Once the base positions of lane lines are identified, the image is divided in 9 vertical windows to detect pixels. Each window is defined by its width and height. For each window, white pixels are identified and counted. If the number of detected pixels is greater than a certain predefined number, the base x-position for the next window is updated by taking a mean of x-positions of the detected pixels. In case not enough pixels are detected, base position for next window is kept same as the previous one and misdetection count is increased. Positions for all detected pixels are stored for both left and right lines.

If number of mis-detected windows is greater than certain fraction of total number of windows a Boolean value of 1 is returned to warn to discard the frame.

Once all the pixels are obtained for left and right lane lines, *fit_polynomial(binary_warped, leftx, lefty, rightx, righty, out_img)* function is used to fit second order polynomials and return the coefficients.



5. Calculate lane curvature and distance of the car from the center of the lane

Given the equation of the polynomial,

$$f(y) = Ay^2 + By + C$$

Lane curvature can be found out as

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

This formula gives curvature in pixels which can be converted to meters by using appropriate conversions.

ym_per_pix = 30/720 # meters per pixel in y dimension

xm_per_pix = 3.7/700 # meters per pixel in x dimension

To calculate lane offset,

The pixel length difference between the center of the image and the center of the lane is taken and converted to meters. Center of the image represents the location of the car assuming that the camera is mounted at the center of the vehicle width.

6. Displaying lane boundaries along with lane curvature and vehicle position

result = draw_lane_lines(warped, left_fit, right_fit, Minv, undist)

texted_image =cv2.putText(img=np.copy(result), text="Radius of Curvature is "+str(round(mean_curverad,2))+"m", org=(0,50),fontFace=3, fontScale=1, color=(255,255,255), thickness=2)

if center_dist>=0:

texted_image2 =cv2.putText(img=np.copy(texted_image), text="Vehicle is "+str(round(np.absolute(center_dist),2))+"m left of Center", org=(0,100),fontFace=3, fontScale=1, color=(255,255,255), thickness=2)

else:

texted_image2 =cv2.putText(img=np.copy(texted_image), text="Vehicle is "+str(round(np.absolute(center_dist),2))+"m right of Center", org=(0,100),fontFace=3, fontScale=1, color=(255,255,255), thickness=2)

plt.imshow(texted_image2)

7. Applying pipeline to a stream of images

A 'Line' class was created to store important variables for left and right lane lines and also to store some historical information as the video stream was processed. A reset variable is created to figure out if the search for the pixels needs to be done from scratch.

If reset is 0, then the lane line pixels are only searched in a region around the previously fitted polynomials to save time. This process is repeated for a fixed number of frames (lookahead count) before changing the reset value to 1.

The function *search_around_poly(binary_warped, left_fit, right_fit)* is used if reset is 0.

The pipeline is applied to each frame in the function *process_image(img1)*.

The final output video is located at "./examples/project_video_out_final.mp4"

Potential Shortcomings

1. One shortcoming is that the algorithm appears to be sensitive to road or lane line topologies.
2. The hardcoded values while calculating perspective transforms or while applying thresholds to images start not working perfectly for different lighting conditions and road topologies.
3. This pipeline did not work fluently with the Challenge video.

Potential Improvements

1. The whole pipeline needs to be slightly more robust in terms road topologies.

2. There needs to be some function to adjust gradient and color thresholds based on the road topology and lighting conditions in the image.
3. The pipeline should be able to handle certain special cases like challenge video.

This may require more time and effort and I will try to work on it as the course progresses.