Advanced Lane Detection for Self-Driving Cars

1 INTRODUCTION

This project revolves around using advanced computer vision techniques to accurately detect lanes for self-driving cars. The main libraries utilized are openCV for its powerful computer vision algorithms and python's numpy library for managing image arrays and data manipulation. The pipeline was built using three different classes to manage the different layers of information exchanged in the pipeline as well as to improve readability.

The steps taken to achieve the final result are outlined below.

1. Calibrate the camera to obtain the camera matrix and distortion coefficients.
2. Undistort images before processing begins.
3. Initialize two lane line objects to store lane data across the video processing.
4. Develop a mask of the image where edges were detected using thresholds on sobel gradients and color transforms which are stacked together.
5. Create a histogram of the image to find pixel across the image.
6. Use sliding window objects to detect left and right lane lines.
7. Grab the detected pixel indices and use it to fit a line, calculate radius curvature data and distance from center of lane.
8. Feed information into the lane overlay functions and display results.

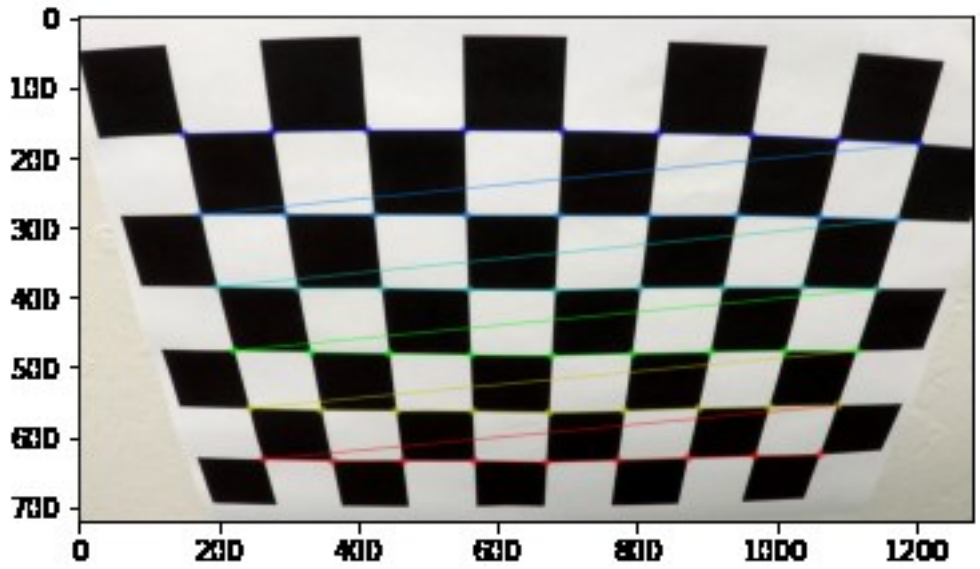
The code in the P2.IPython notebook is properly organized to match the above steps in terms of readability.

While the pipeline doesn't work well with the optional challenge videos, it runs well for the main project video. Ways in which it could be further improved to work on the challenge videos are outlined in the conclusion.

2 CAMERA CALIBRATION

The steps used to calibrate the camera match with what was taught in the lecture videos, and the code for it can be seen under the Camera class in the code. A camera object is created to store the camera matrix data and the distortion coefficients. Lists of successful and failed images are also saved for display purposes later.

Object and image corners are obtained by applying a grayscale transform to a series of black and white chessboard images. The function *cv2.findchessboardcorners* is used to find the intersection of the chessboard corners according to a pattern which are then stored in list arrays to be used with camera calibration later. They are also used with the function *cv2.drawchessboardcorners* to check the patterns are found correctly.



The corners are then fed into the openCV function *cv2.calibrateCamera* to get the matrix and distortion coefficients which are used in another function to remove radial distortion from images as they are being fed from the video or test images later on. An undistorted image is shown below:

3 Binary Image of Detected Edges

The code for this section can be seen under the “Gradient and Color Thresholds” section in the P2 file. This portion of the pipeline is responsible for creating a binary image by using a mix of color-space transforms and sobel edge detection.

Four different thresholds are applied to the different components in the following order: absolute value along the x axis, absolute value along the y axis, magnitude value of sobel, direction using the arctan function which are all final combined into a single mask where edges are likely to be found.

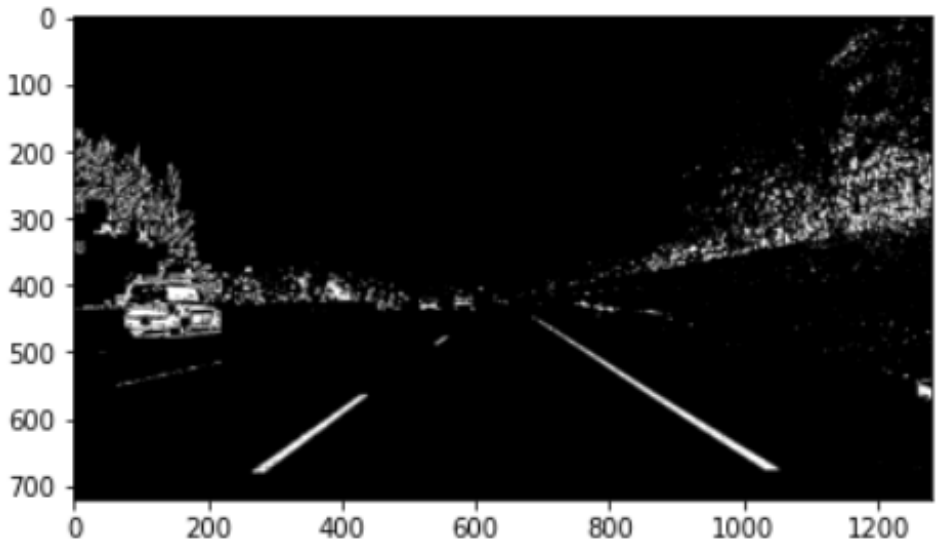
Sobel mask code:

```
def sobel_mask(img):
    '''
    applies three different components of sobel edge detection with individual threshold
    values for each.
    :param img: input image
    :return: mask of edge-detected image
    '''
    sobel_x = sobel_absolute_value_mask(img, axis='x', sobel_ksize=3, threshold=(20, 100))
    sobel_y = sobel_absolute_value_mask(img, axis='y', sobel_ksize=3, threshold=(20, 100))
    magnitude = sobel_magnitude_mask(img, sobel_ksize=3, threshold=(20, 100))
    direction = sobel_direction_mask(img, sobel_ksize=3, threshold=(0.7, 1.3))

    s_mask = np.zeros_like(img)
    s_mask[((sobel_x == 1) & (sobel_y == 1)) |
           ((magnitude == 1) & (direction == 1))] = 1

    return s_mask
```

Moreover, the image is transformed into the HLS color space where the contrast between the yellow and white lines with the road is higher than normal, and specifically in the S-channel. This is then combined with the sobel mask after a separate threshold is applied to get the final image.



4 PERSPECTIVE TRANSFORM

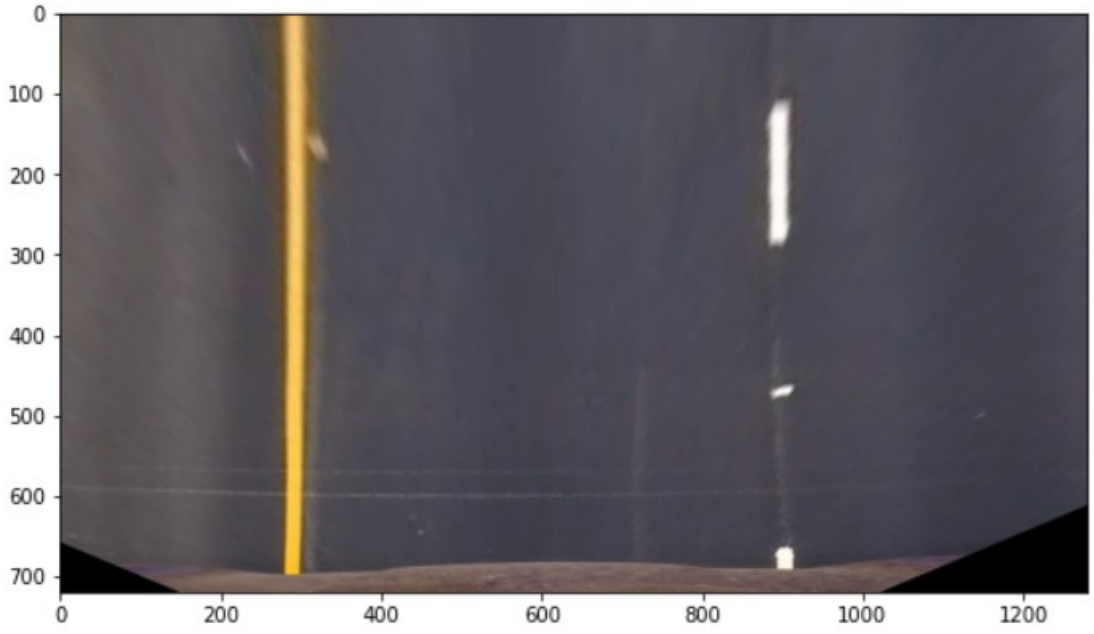
The code for this section is found under the perspective transform section in the P2 file. The main goal here is to produce a birds-eye view of the lane so that lane lines are vertical and can be summed through

a histogram. This is achieved by using the openCV functions *cv2.getPerspectiveTransform* and *cv2.warpperspective*.

The core of this portion of the pipeline does have a ripple effect on the overall capability of the program in the way which source points are chosen. The source points define what will be seen by the lane-finding segment later on so it was crucial to choose universal parameters or something very close to it. This is, obviously, difficult and suffers from a bias problem as the points I chose fit well for the main project video but suffer under different conditions such as the ones seen in the challenge videos. However, after numerous iterations I chose the following source and destination points to warp the perspective to and of which the result is seen below.

Points	Source	Destination
P1	556, 480	300, 0
P2	734, 480	900, 0
P3	220, 720	300, 720
P4	1105, 720	900, 720

Perspective transform result:



5 IDENTIFYING LANE LINE PIXELS AND FITTING THE LINE

This portion of the code is covered under several headlines in the P2 file, including: Sliding Window class, Lane class, Helper Functions and some portions of the pipeline.

To find the lane line pixels we start by using a histogram as mentioned before. This allows us to sum the white pixels in a warped binary image to find where the vertical lines of the lane sum up to find the likeliest spot where the lane might be.

For each lane, a sliding window is created at the point with maximum pixel intensity (or in case of a video feed, the window center will use past data to be more efficient). Several parameters are specified here including margin, which specifies the width of the sliding window, minpix which controls how many pixels must be detected before the window shifts its center position and the number of windows in a frame which directly control the height of each window.

A simple comparison of the pixel indices will net us the good pixels where lanes are likely to be located by the sliding windows by comparing the nonzero values of the binary image with the window's geographical position through simple comparators as shown below.

```
# check if indices are within the windows area, take only y-positions
good_inds = ((nonzero[0] >= self.y_high)
             & (nonzero[0] < self.y_low)
             & (nonzero[1] >= self.x_low)
             & (nonzero[1] < self.x_high)
             ).nonzero()[0]
```

The windows are then stored in each separate object created earlier for each lane. Once all the windows are created and the good indices data is returned to the lane, the function *generate_lane_points* and *fit_points* will attempt to create the y and x points of the line and fit it using a second degree line polynomial as shown in the lecture notes.

Further functions were created to enhance lane finding capabilities of the sliding windows and ensuring they do not get frozen in place in the event they land in a location in the image where pixels are not present in the binary image since their location is dependent on historical data.

Of note was the fact that window centers are not placed as per the last window created, but the first window in the previous frame instead (lane.windows[-nwindows] vs lane.windows[-1]). This gave me a more reliable result when tracking lane line particularly when shadows or other shapes present artifacts in the binary road image.

6 Radius of Curvature and Distance from Center

To calculate these two values, functions were written to extract the information from each separate lane line object as required in each frame.

The radius of curvature was calculated as mentioned in the notes by inserting the fit line data into the derivative and using the equations found in the lecture notes and which can be seen in the code snippet below. The radius was calculated for each lane line separately and then an average was taken as the final value.

```
def get_radius(self):
    '''
    find the radius of the curve by using differentiation of the polynomial
    :return: radius value (int)
    '''
    ym_per_pix = self.image_lane_length / self.meter_per_y_axis
    xm_per_pix = self.land_width / self.meter_per_x_axis

    points = self.generate_lane_points()
    x = points[:, 0]
    y = points[:, 1]

    fit_curve = np.polyfit(y * ym_per_pix, x * xm_per_pix, 2)
    first_derivative = 2 * fit_curve[0] * self.meter_per_y_axis * ym_per_pix + fit_curve[1]
    second_derivative = 2 * fit_curve[0]
    radius = int(((1 + (first_derivative ** 2) ** 1.5) / np.absolute(second_derivative)))
    return radius


def get_camera_center(self):
    '''
    find the distance from the camera to the lane line
    :return: distance value
    '''
    xm_per_pix = self.land_width / self.meter_per_x_axis

    points = self.generate_lane_points()
    x = points[np.max(points[:, 1])][0]
    from_center = np.absolute((self.img_xsize // 2 - x) * xm_per_pix)
    return from_center
```

This does seem to work well during curves but during a straight road the radius value becomes increasingly erratic. I did try to rectify this in several ways but I wasn't entirely sure what the right approach would be so I just stuck to what is shown in the lecture notes.

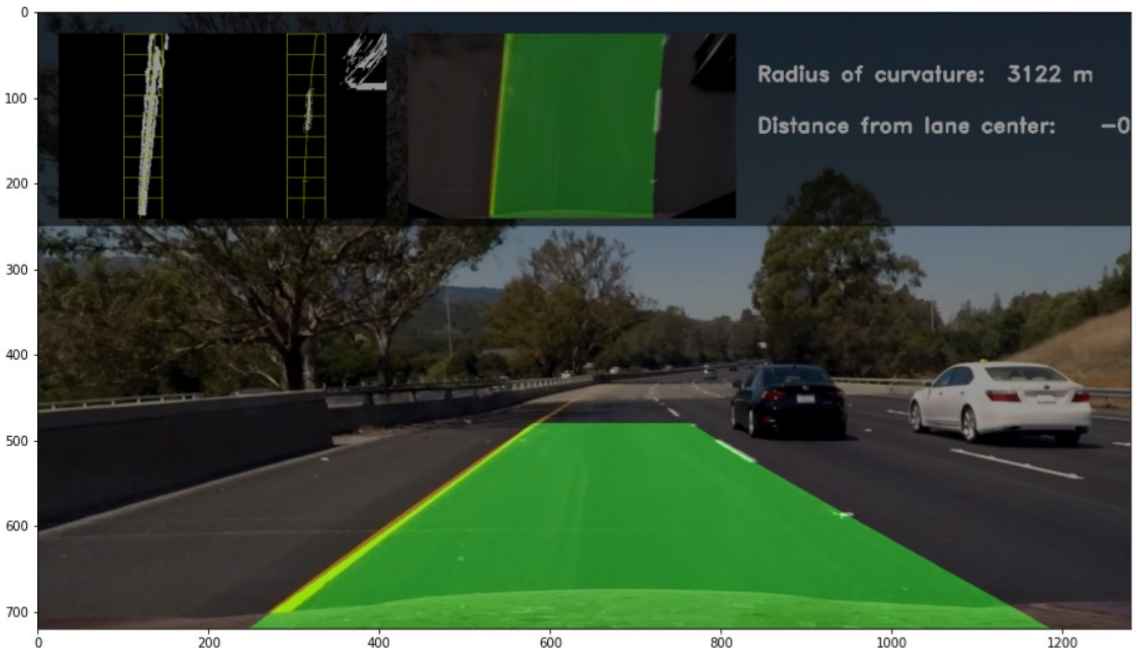
Finally, the get_camera_center function uses the latest line generated points to find the maximum x-axis point and use it to find the distance to the center from the lane line.

For both functions, fixed values were used to convert the pixel measurements to meters.

Finally, lane overlays were created to display the binary image, the warped lane perspective, the sliding windows, the radius and distance from center all at the top of the image. Please note this was inspired by github user ricardosllm's on his repo and I used the same method if displaying the final result as I think it looks very elegant, shows a lot of information and looks good on a github profile.

7 FINAL RESULT

The final result can be seen below as one of the example images.



8 CONCLUSION

The pipeline does work rather well on the project video and performs far better than the techniques learned in the first project of this course. However, the challenge videos make it apparent that it is not perfect in any way. Tree shadows and curbside lines in particular present difficulties towards the lane-finding capabilities of the pipelines which can be further improved upon.

One suggestion that I looked into early on was to minimize the warp perspective source points as much as possible to eliminate as much outside interference as possible whether from other vehicles or curbside lines. The optimal solution in this case would be to calculate the vanishing point of the road which would track it along curves and on straights but the several literature papers I've read on this subject used machine learning networks to produce their result which I think is out of the scope of this project.