# Project 4 – Advanced Lane Finding

---

**Advanced Lane Finding Project**

The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images. (X)
- Apply a distortion correction to raw images. (X)
- Use color transforms, gradients, etc., to create a thresholded binary image. (X)
- Apply a perspective transform to rectify binary image ("birds-eye view"). (X)
- Detect lane pixels and fit to find the lane boundary. (X)
- Determine the curvature of the lane and vehicle position with respect to center. (X)
- Warp the detected lane boundaries back onto the original image. (X)
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position. (X)

## Rubric Points

### Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

---

### Writeup / README

**1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.**

This project was created to be able to detect lane lines that may not conform to a single type of scenario, particularly to detect lane lines that are obscured by bright light or shadows, and to detect curved lane lines

# Camera Calibration

### 1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is in the function 'camera_calibration' in the main.py file. The function accepts a list of chessboard images provided by Udacity, and uses parameters nx (number of corners in the horizontal direction), ny (number of corners in the vertical direction), and the image shape given as (720, 1280). The aim of this function is to obtain distortion coefficients a camera calibration matrix that will be used to undistort the images of the road obtained by the same camera later in the project. This is achieved by preparing 'object points', which are the (x, y, z) coordinates of the chessboard corners in the real world, and 'image points', which are points in 2D image space that the object points are to be mapped to. To achieve this, we use the OpenCV function findChessboardCorners, which returns the image points that correspond to a particular set of object points. The image points are appended to the imgpoints list for every calibration image. Finally, once all the object points and image points have been collected, the OpenCV calibrateCamera function is called, which returns the distortion coefficients, camera calibration matrix, along with rotation and translation vectors. For this project, however, we are using only the distortion coefficients and the camera calibration matrix.

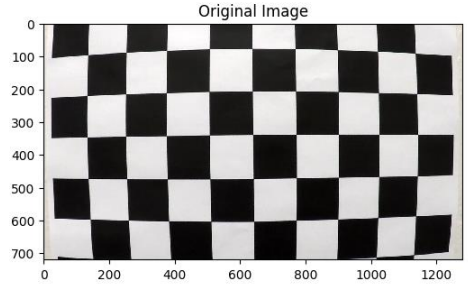


*Figure 1: Original Distorted Chessboard Image*

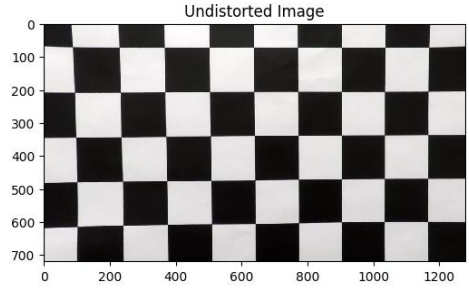


*Figure 2: Undistorted Chessboard Image*

## Pipeline (single images)

### 1. Provide an example of a distortion-corrected image.

Distortion correction was performed on each image as a part of the pipeline in the main program. The correction was performed using the OpenCV undistort function, which uses the calibration matrix and distortion coefficients obtained from the calibration step and returns the undistorted image.

*Figure 3: Distorted Image straight_lines1.jpg*

*Figure 4: Undistorted image straight_lines_1_undistorted.png*

**2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.**

This step was achieved by the use of the functions abs_sobel_thresh, mag_thresh, and dir_threshold. Each of these functions applies a different type of threshold, and allows for the selection of the colorspace. For this project, I chose the following technique, a combination of gradient thresholding in both the x and y directions, gradient magnitude thresholding, and gradient direction thresholding

```
ksize=3
gradx = abs_sobel_thresh(img, orient='x', sobel_kernel=ksize, thresh=(10, 100),
colorspace='HLS', channel=2)
grady = abs_sobel_thresh(img, orient='y', sobel_kernel=ksize, thresh=(10, 100),
colorspace='HLS', channel=2)
mag_binary = mag_thresh(img, sobel_kernel=ksize, mag_thresh=(10, 100),
colorspace='HLS', channel=2)
dir_binary = dir_threshold(img, sobel_kernel=ksize, thresh=(0.5, 1.6),
colorspace='HLS', channel=2)
combined = np.zeros_like(dir_binary)
combined[((gradx == 1) & (grady == 1)) | ((mag_binary == 1) & (dir_binary == 1))] = 1
```

The result looked like this:

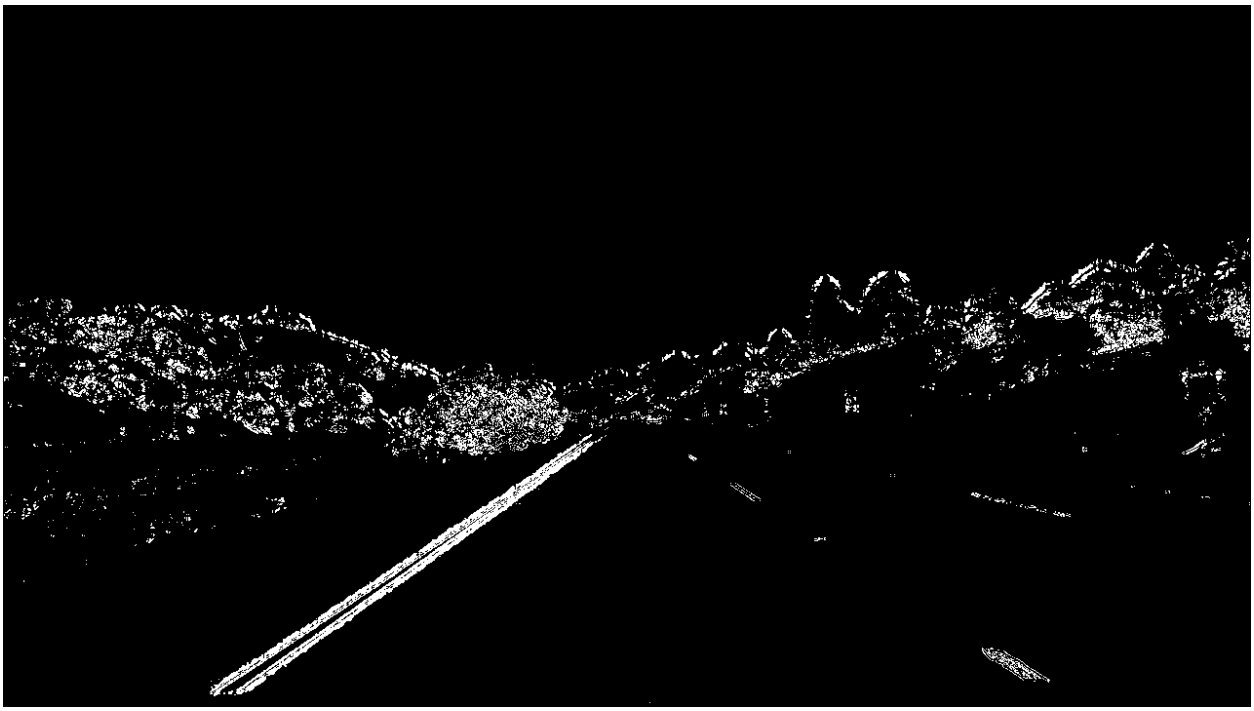

*Figure 5: Thresholded binary image straight_lines1_2_binary.png*

### 3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The perspective transform was used to transform the image as if the camera is looking vertically down on the road. In order to select the points, I used the information available publicly that each lane line is 10 feet long, and that the spacing between lane lines is 30 feet. Furthermore, we had information that the meters per pixel value in the y direction was 30/720 = 0.0417 meters per pixel. So, to represent a height of 720 pixels correctly in the warped image, the real length was 30 meters or 98.4252 feet. This meant that I had to incorporate at least 3 lane lines in the image (of course, this is just an approximation). Using this information, I obtained the following source and destination points.

| **Source** | **Destination** |
|---|---|
| [253, 686] | [253, 686] |
| [1041, 677] | [1041, 686] |
| [584, 457] | [253, 0] |
| [696, 457] | [1041, 0] |

The **warp** function in main.py uses the OpenCV function getPerspectiveTransform to obtain the transform matrix and the inverse transform matrix, and the warpPerspective function to warp the image using the computed matrix. The result looks like this:

*Figure 6: Warped binary image straight_lines1_3_warped.png*

As expected, in this case, the lines are straight in the warped image

### 4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

After the warping step, the next step is the use of the sliding window method to find lane lines. This is achieved using the sliding_window_init and the sliding_window_next functions in main.py. I use a setting of 15 windows, a margin of 100 pixels to the left and right of the lane lines, a threshold value of 50 pixels to decide whether to recent the subsequent window, a scale value of 30 meters per 720 pixels in the y direction and 3.7 meters per 700 pixels in the x direction (as suggested in the course content). This method turned out to be quite robust for all test images. In the same function I select the left lane or the right lane to be used for the purpose of radius of curvature, and also compute the position of the vehicle with respect to the center. Much of the code in this function was suggested in the course. The sliding window detection is initialized on only the first frame of the project_video.mp4 file. Subsequent frames use the information returned by the sliding_window_init function and use the sliding_window_next function instead, so that the windows do not have to be recomputed.

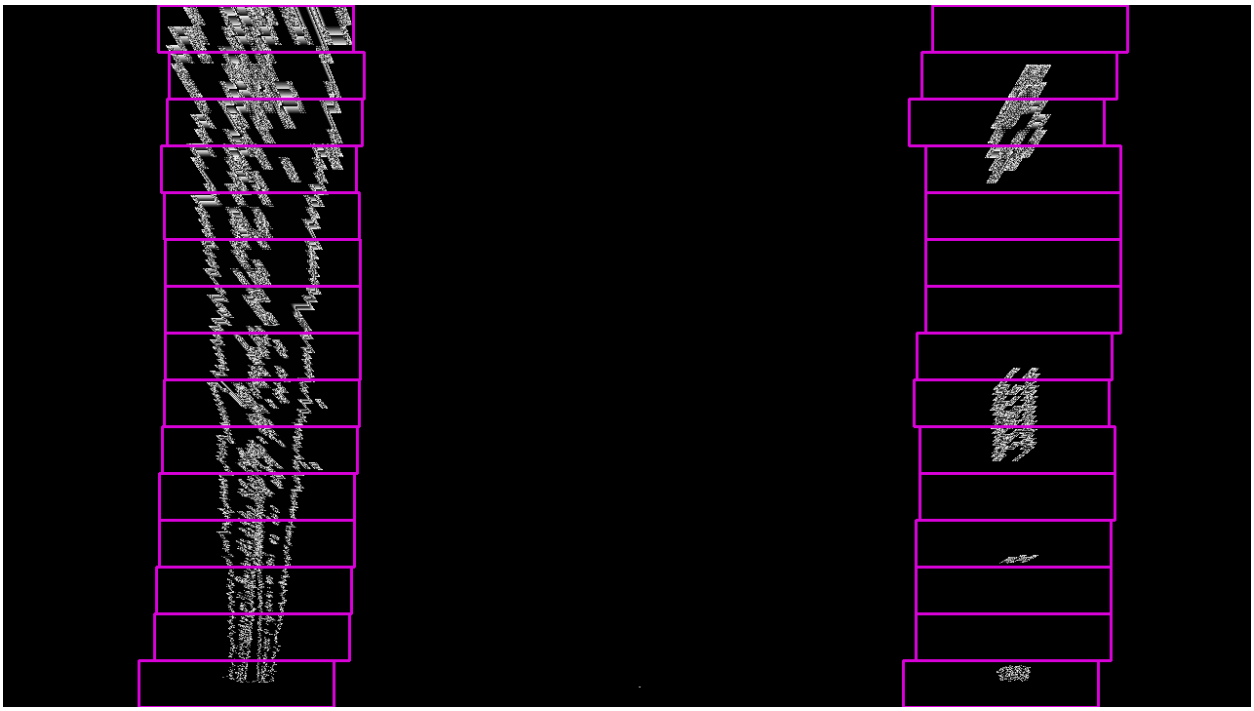

*Figure 7: Lane lines detected by the sliding window method*

**5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.**

In the same sliding_window_init and sliding_window_next functions, the radius of curvature for the more prominent lane is calculated along with vehicle position with respect to the center. The ReferenceLane is simply the lane with more pixels present. This selection is necessary because there can be situations where the detected curvature may be significantly different for the complete lane and the dashed line. The formula to calculate radius of curvature of a fitted $2^{nd}$ degree curve is given in the course.

**6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.**

Finally, the fill_lane_area function is used to shade the detected lane area and to display the computed radius of curvature and vehicle position with respect to center. A sign convention is used for the vehicle position value, negative means that the vehicle is to the left of center, and positive means that the vehicle is to the right of the center.

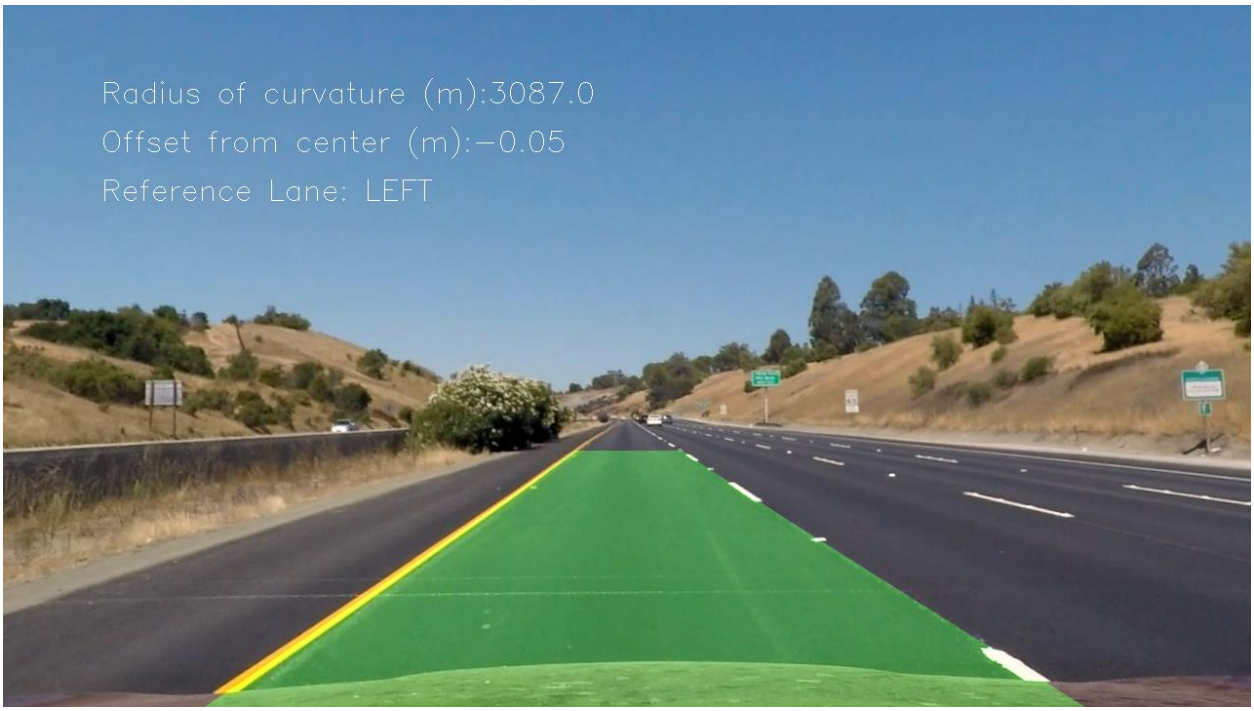


*Figure 8: Fully processed image*

---

## Pipeline (video)

### 1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

The final video result is the file [project_video_PROCESSED.mp4]

---

## Discussion

### 1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

The same pipeline unfortunately did not work too well for the challenge videos provided. One technique that could probably be used is the use of more sanity checks

to ensure that lane lines are more accurately classified. Neural networks could also be used for this purpose.