



# Project : Advanced Lane Finding

05.12.2018

---

Submitted By:

Lalu Prasad Lenka

## Overview

The project aimed at creating a software pipeline to identify the lane boundaries in a video from a front-facing camera on a car.

## Goals

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

## Pipeline(Image)

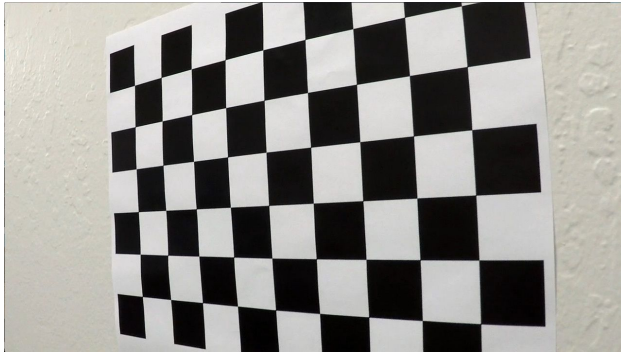
### I. Camera Calibration

OpenCV provides some really helpful built-in functions for the task on camera calibration. First of all, to detect the calibration pattern in the calibration images, we can use the function `cv2.findChessboardCorners(image, pattern_size)`.

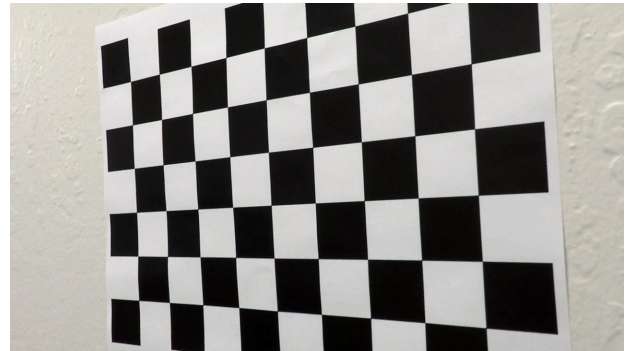
Once we have stored the correspondences between 3D world and 2D image points for a bunch of images, we can proceed to actually calibrate the camera through `cv2.calibrateCamera()`. Among other things, this function returns both the camera matrix and the distortion coefficients, which we can use to undistort the frames.

The code for this steps can be found in [Advanced Lane Finding.ipynb](#) cell 3.

I applied this distortion correction to the test image using the `cv2.undistort()` function in code **cell 3** `undistort_image()` and obtained the following result.



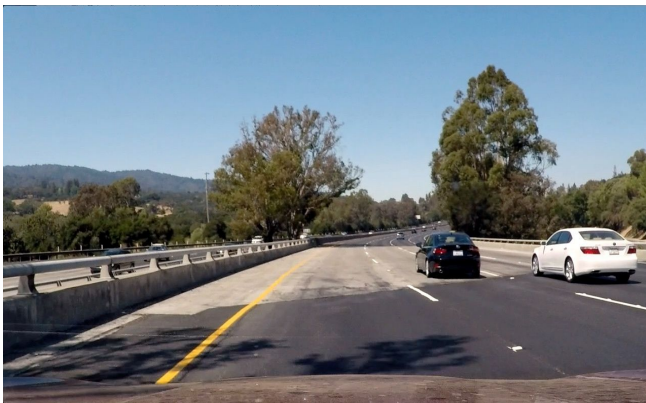
Chessboard image before calibration



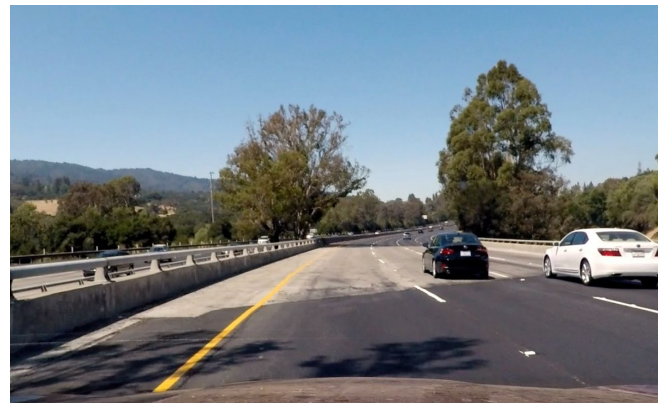
Chessboard after before calibration

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

Once the camera is calibrated, we can use the camera matrix and distortion coefficients we found to undistort also the test images. Indeed, if we want to study the geometry of the road, we have to be sure that the images we're processing do not present distortions. Here's the result of distortion-correction on one of the test images:



Test image before calibration

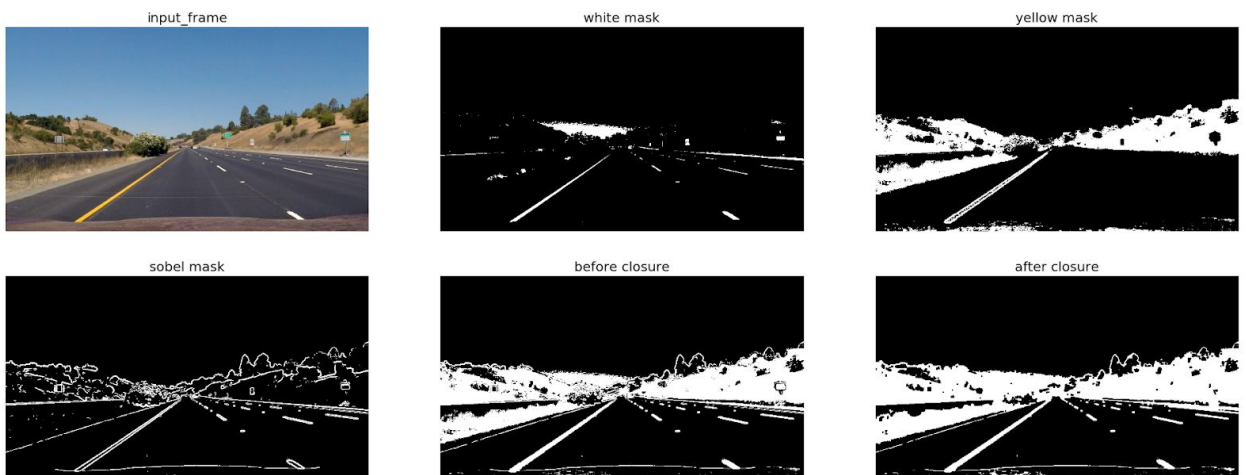


Test image after calibration

III. 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.

Correctly creating the binary image from the input frame is the very first step of the whole pipeline that will lead us to detect the lane. For this reason, I found that is also one of the most important. If the binary image is bad, it's very difficult to recover and to obtain good results in the successive steps of the pipeline. The code related to this part can be found [Advanced Lane Finding.ipynb cell 6](#).

I used a combination of color and gradient thresholds in `binarize()` to generate a binary image. In order to detect the white lines, I found that [equalizing the histogram](#) of the input frame before thresholding works really well to highlight the actual lane lines. For the yellow lines, I employed a threshold on V channel in [HSV](#) color space. Furthermore, I also convolve the input frame with Sobel kernel to get an estimate of the gradients of the lines. Finally, I make use of [morphological closure](#) to fill the gaps in my binary image. Here I show every substep and the final output:



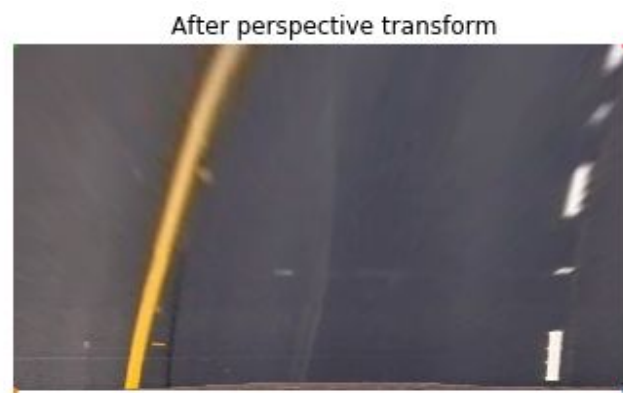
#### IV. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

Code relating to warping between the two perspective can be found in [Advanced Lane Finding.ipynb](#) cell 8.

. The function `birdseye()` takes as input the frame (either color or binary) and returns the bird's-eye view of the scene. In order to perform the perspective warping, we need to map 4 points in the original space and 4 points in the warped space. For this purpose, both source and destination points are hardcoded (ok, I said it) as follows:

```
src = np.float32([[w, h-10],      # bottom right
                  [0, h-10],      # bottom left
                  [546, 460],     # top left
                  [732, 460]])    # top right
dst = np.float32([[w, h],         # bottom right
                  [0, h],         # bottom left
                  [0, 0],         # top left
                  [w, 0]])        # top right
```

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



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

Code relating to lane detection can be found in [Advanced Lane Finding.ipynb](#) cell 10.

In order to identify which pixels of a given binary image belong to lane-lines, we have (at least) two possibilities. If we have a brand new frame, and we never identified where the lane-lines are, we must perform an exhaustive search on the frame. This search is implemented in `get_fits_by_sliding_windows()`: starting from the bottom of the image, precisely from the peaks location of the histogram of the binary image, we slide two windows towards the upper side of the image, deciding which pixels belong to which lane-line.

On the other hand, if we're processing a video and we confidently identified lane-lines on the previous frame, we can limit our search in the neighborhood of the lane-lines we detected before: after all we're going at 30fps, so the lines won't be so far, right? This second approach is implemented in `get_fits_by_previous_fits()`. In order to keep track of detected lines across successive frames, I employ a class defined in `Line`, which helps in keeping the code cleaner and access required properties of a `Line`.

```
class Line:

    def __init__(self, buffer_len=10):

        # flag to mark if the line was detected the last iteration
        self.detected = False

        # polynomial coefficients fitted on the last iteration
        self.last_fit_pixel = None
        self.last_fit_meter = None

        # list of polynomial coefficients of the last N iterations
        self.recent_fits_pixel = collections.deque(maxlen=buffer_len)
        self.recent_fits_meter = collections.deque(maxlen=2 * buffer_len)

        self.radius_of_curvature = None

        # store all pixels coords (x, y) of line detected
        self.all_x = None
```

```

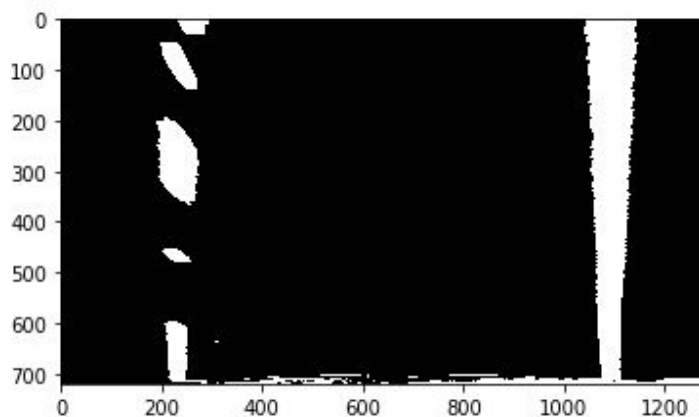
self.all_y = None

... methods ...

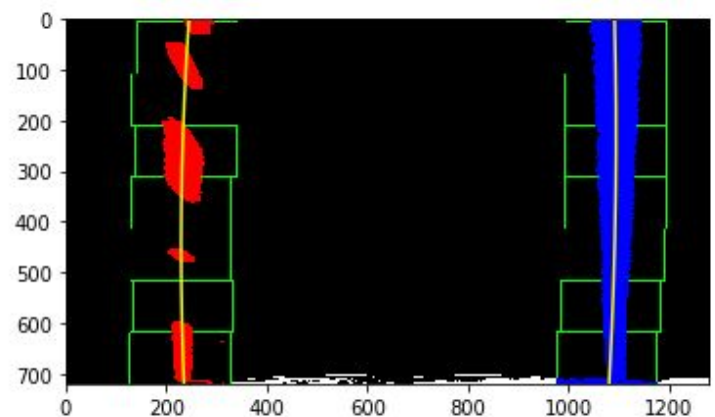
```

The actual processing pipeline is implemented in function `process_pipeline()`. As it can be seen, when a detection of lane-lines is available for a previous frame, new lane-lines are searched through `get_fits_by_previous_fits()`: otherwise, the more expensive sliding windows search is performed.

The qualitative result of this phase is shown here:



Bird's-eye view (binary detected)



Bird's-eye view (lane detected)

## VI. 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.

Offset from center of the lane is computed in `compute_offset_from_center()` as one of the step of the processing pipeline in **Advanced Lane Finding.ipynb cell 10**. The offset from the lane center can be computed under the hypothesis that the camera is fixed and mounted in the midpoint of the car roof. In this case, we can approximate the car's deviation from the lane center as the distance between the center of the image and the midpoint at the bottom of the image of the two lane-lines detected.



During the previous lane-line detection phase, a 2nd order polynomial is fitted to each lane-line using `np.polyfit()`. This function returns the 3 coefficients that describe the curve, namely the coefficients of both the 2nd and 1st order terms plus the bias. From this coefficients, following [this](#) equation, we can compute the radius of curvature of the curve. From an implementation standpoint, I decided to move this methods as properties of `Line` class.

```
class Line:
    ... other stuff before ...
    @property
    # average of polynomial coefficients of the last N iterations
    def average_fit(self):
        return np.mean(self.recent_fits_pixel, axis=0)

    @property
    # radius of curvature of the line (averaged)
    def curvature(self):
        y_eval = 0
        coeffs = self.average_fit
        return ((1 + (2 * coeffs[0] * y_eval + coeffs[1]) ** 2) ** 1.5) / np.absolute(2 * coeffs[0])

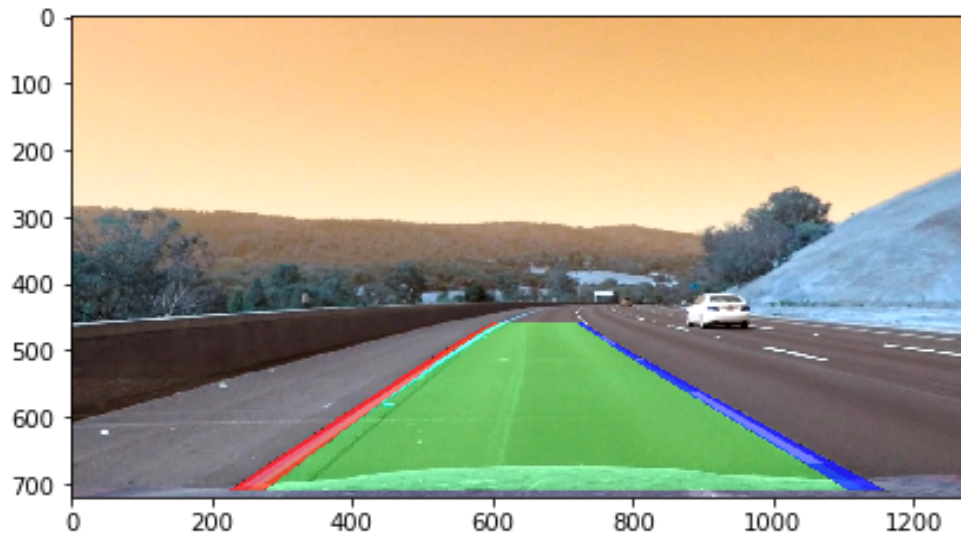
    @property
    # radius of curvature of the line (averaged)
    def curvature_meter(self):
        y_eval = 0
        coeffs = np.mean(self.recent_fits_meter, axis=0)
        return ((1 + (2 * coeffs[0] * y_eval + coeffs[1]) ** 2) ** 1.5) / np.absolute(2 * coeffs[0])
```

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

The whole processing pipeline, which starts from input frame and comprises undistortion, binarization, lane detection and de-warping back onto the original image, is implemented in function `process_pipeline()` in [Advanced Lane Finding.ipynb](#) cell 14.

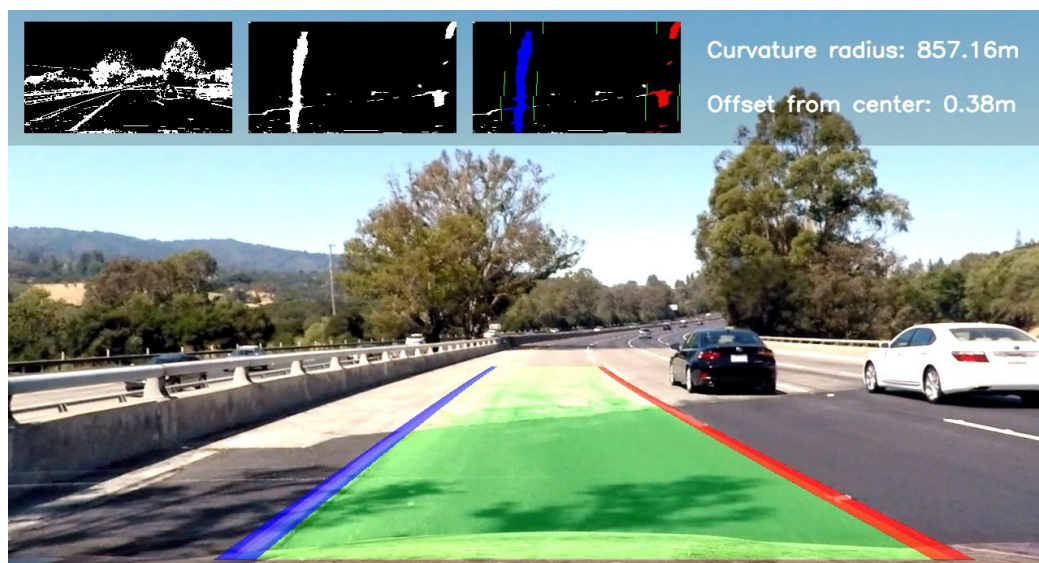


`get_fits_by_sliding_windows()` & `draw_back_onto_the_road()` in code **cell 10** help in finding and drawing the lane lines. The qualitative result for one of the given test images follows: Lane boundaries result for test3.jpg



### VIII. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

I thought of adding the binary, binary warped & lane detected images in the frame along with the curvature radius and offset from center. It dramatically improved the ability to find flaws in the pipeline and debug code. Here is one qualitative result for one of the given test images.



Lane boundaries along with other information result for test4.jpg

All other test images can be found in [./output\\_images/](#)

## IX. Sanity Check

I implemented a function to sanity check if the lane lines were being properly identified. This function did the following 3 checks:

- Left and right lane lines were identified (By checking if np.polyfit returned values for left and right lanes).
- If left and right lanes were identified, there average separation is in the range 150-950 pixels.
- If left and right lanes were identified then there are parallel to each other (difference in slope  $< 0.125$ ).

If an image fails any of the above checks, then identified lane lines are discarded and last good left and right lane values are used. This function was particularly useful in the challenge video where it was difficult to properly identify lane lines. The code for sanity check is in code **cell 13. Sanity Check** helped to counter variation in light and shadow.

## Pipeline(Video)

- If this is the first image then the lane lines are identified using a histogram search (checked using a counter variable `processed_frames`) in `get_fits_by_sliding_windows()` function in code **cell 10**.
- Otherwise `get_fits_by_previous_fits()` function starts from previous left and right fit are used to narrow the search window and identify lane lines (function implemented in code **cell 10**).
- if sanity check described above was failed, then the last good values of left and right fit were used (function implemented in code **cell 13**).

Finally moviepy editor was used to process the project video image by image through the pipeline. The result was pretty good. Lane lines were identified very well through the entire video. Here's a [link to my project video result](#).

I also tested the pipeline on the challenge video and it performed well, not as good as the project video but moderately good. Here's a [link to my challenge video result](#)

## Discussion


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?

- `np.polyfit` sometimes resulted in curves that were a very bad fit. This was especially happening for the left lane in the challenge video. To address this, I introduced a condition in sanity check that left and right lane slopes are roughly parallel (delta in slope is less than 0.125). If this condition failed then, last good value of left and right lane fits were used. This was very useful in improving performance on the project and challenge video.
- In challenge video probably due to low saturation the left lane is rarely detected and mostly the left road divider is detected. I will continue working on this issue even after submission. Probably providing a horizontal offset for `left_base` and `right_base` while making histogram of image.

My biggest concern with the approach here is that it relies heavily on tuning the parameters for thresholding/warping and those can be camera/track specific. I am afraid that this approach will not be able to generalize to a wide range of situations. And hence I am not very convinced that it can be used in practice for autonomously driving a car. This was apparent in the challenge video where parameters tuned for the project video resulted in an inferior performance on a different track in the challenge video.

Here are a few other situations where the pipeline might fail:

1. Presence of snow/debris on the road that makes it difficult to clearly see lane lines
2. Roads that are wavy or S shaped where a second order polynomial may not be able to properly fit lane lines



To make the code more robust we should try to incorporate the ability to test a wide range of parameters (for thresholding and warping) so that the code can generalize to a wider range of track and weather conditions. To do this we need a very powerful sanity check function that can identify lane lines that are impractical and discard them.

Or we can take a deep learning approach and try building a model with annotated lane positions.

## References

I have mostly taken help(code reference) from udacity classroom lectures and quizzes to build my pipeline.