# Advanced Lane Lines Project

STEPS

## 1. Camera Calibration

The code for this step is included in file `camera_calib.py`

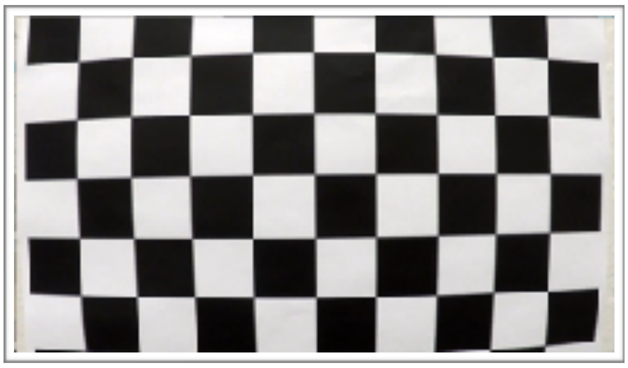
We start by preparing "object points", which are (x, y, z) coordinates of the chessboard corners in the world (assuming coordinates such that z=0). Thus, *objp* is just a replicated array of coordinates, and *objpoints* will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. *imgpoints* will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

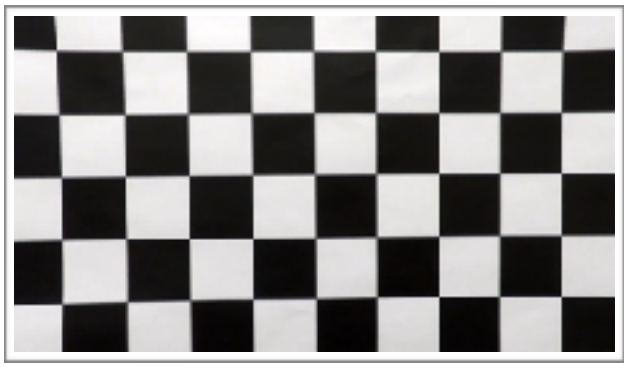
We obtain *objpoints* and *imgpoints* with method `get_calib_points` which is implemented from line 8 to line 38.

Method `get_coefficients` applies `cv2.calibrateCamera to` *objpoints* and *imgpoints* in order to get the coefficients needed to un-distort images.

The `undistort_image` method performs this task given the coefficient and the image to undistor

We can appreciate the results on the test image:


Original image


Undistorted image

Original image

Undistorted image

## 2. Threshold Binary Image

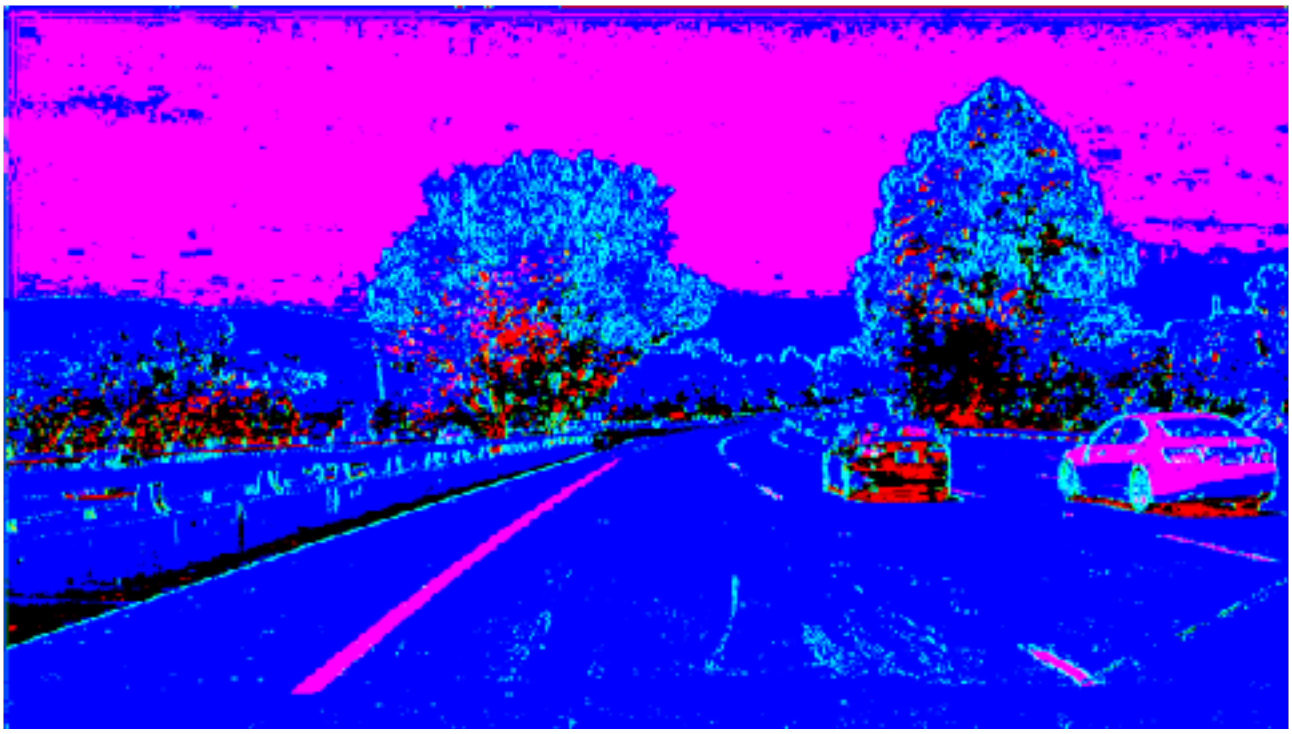
I used a combination of color and gradient thresholds to generate a binary image.
The code is in file `binary_image.py`

For colour thresholds I worked in HLS space.
I transformed the image from BGR to HLS and extracted the L and S channel: line 10 to 12.

To calculate the gradient threshold I used the Sobel operator along the x axis.: line 13 to 20

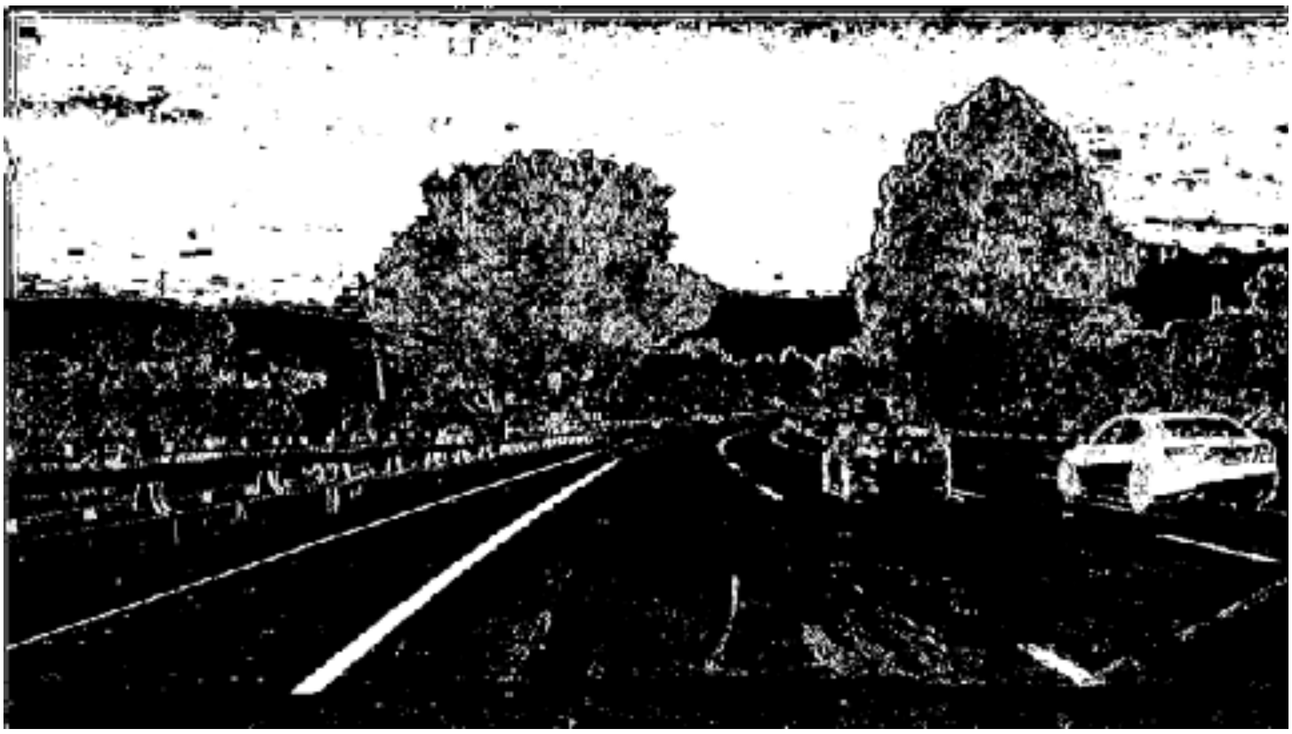
The threshold values (line 7) were found by manually trying different values and testing them on the given test images.

Finally I stacked these 3 channels together and generated a new 3 channel representation of the image:



*Example on test_images/test1.jpg*

The binarised version of the image (by applying the thresholds values) looks like this:



I applied this transformation to all test images and they can be found in the folder *output_images.*

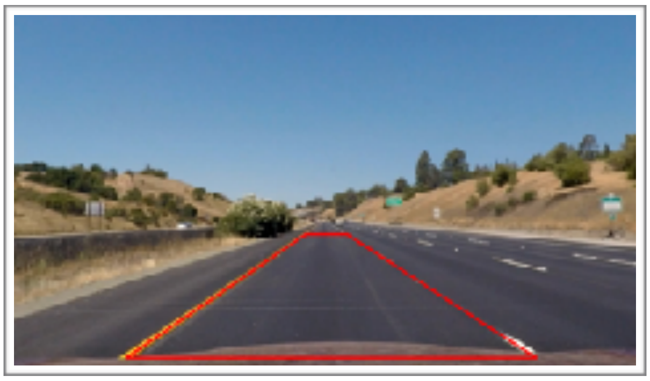
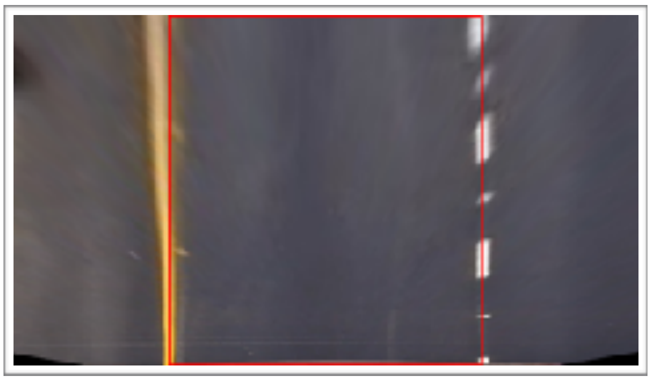
## 3. Perspective Transform

The code for my perspective transform is implemented in `perspective_transform.py`
The goal of this step is to get a “birdeye” view of the road so that it can be easier to calculate the curvature of the road.

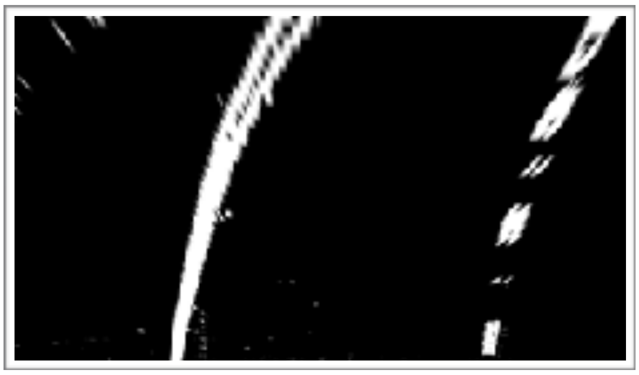
The core of the method is implemented from line 21 to 27.

In order to calculate the matrix transformation and its inverse we need to define 4 source points that will be mapped to 4 destination points in the transformed image.

The 4 source points were found manually using test image *straight_lines1.jpg*

The method was applied to all test images. Below an example of this method applied to the binarised test image number 3



## 4. Detect Lanes

Given a birdeye view binarised image, it is now easier to find the pixels of the image that belong to each lane.

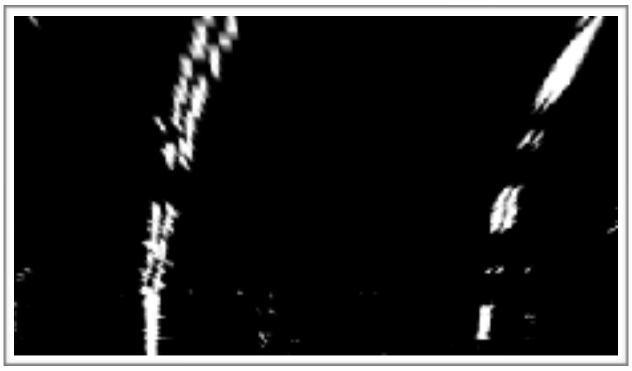
This is implemented in file `find_lanes.py`

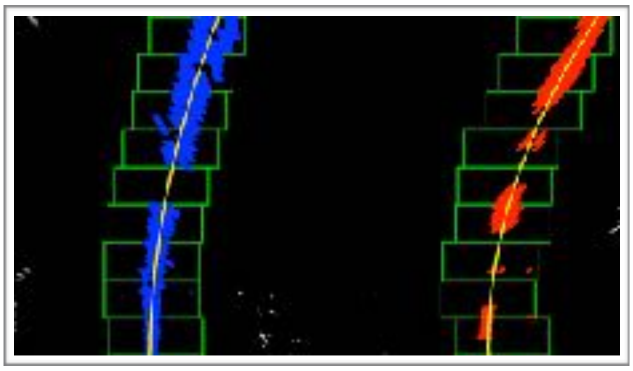
Most of the code was already given in the lessons.

The files contains 2 methods: `get_lanes_full_search` and `get_lanes_from_previous`.
The first method performs a search over the whole image divided in 9 horizontal windows.
For each windows we find the peacks in the histogram of sum - column wise- of the white pixels.
The peaks provide an area where the lanes can be located, and the white pixels withing a certain margin around that value, will belong to the lane.

The second method relies on the fact that, if we detected lanes from a previous frame, we can look for the lanes in the surroundings, without having to perform a blind-search again.

These 2 methods provide a "cloud of pixels", and we can use the numpy method `polyfit` to estimate a mathematical representation of the lanes, and draw them.
Below an example:





## 5. Find Curvature and Offset

The radius of the curvature and the offset are implemented in file `curvature.py`

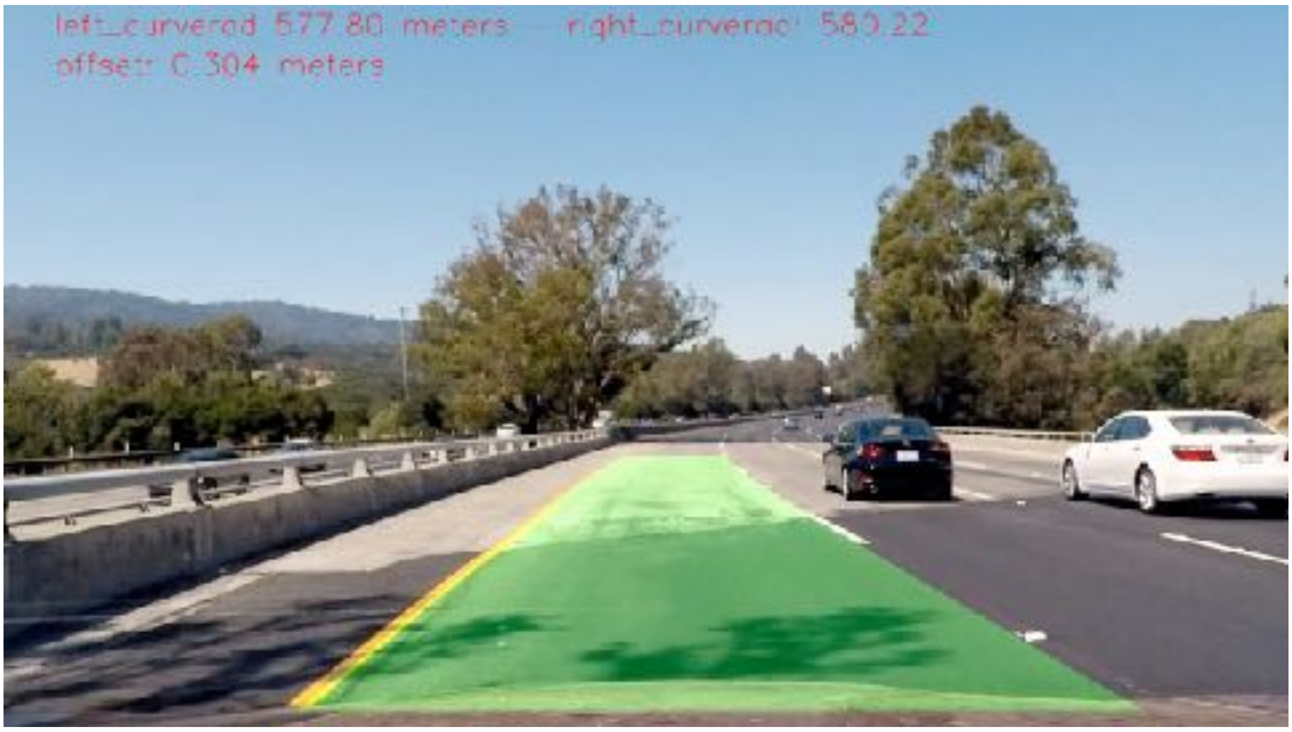
Given the second order polynomial `A y^2 +B y + C` found with the previous method, the curvature can be found with this formula: `R = [(1+(2 Ay +B)^2 )^3/2]/|2A|.`

The method is implemented from line 7 to get the curvature in meters. To get the curvature in pixels, the method is implemented at line 31.

The offset of the vehicle from the centre of the road is calculated by assuming that the camera is mounted at the centre of the vehicle, and the distance of the 2 detected lanes allowed us to find the offset with the following code:

```
camera_position = image.shape[1] / 2.
lane_center = (right_fitx[-1] + left_fitx[-1]) / 2.
center_offset_pixels = abs(camera_position - lane_center)
```

Finally the method `draw_lane` implemented in file `draw_lanes.py` uses all the method so far described. We can see the results for a test image:



## 6. Test Pipeline on Video

The whole pipeline is implemented in file `process_video.py` from line 57
The code contains a lot of comments in order to explain all the steps used to detect lanes.

### *Smoothing*

Smoothing between frames was applied. In particular check lines: 121 to 128 and 139 to 141.
The concept of an ***Infinite Impulse Response filter*** with coefficients `alpha = 0.1` and `beta = 0.9` was applied so to easily take into account the information about the location of the lane lines from the previous frames.

File `demo.avi` shows the results of the pipeline applied to the `project_video.mp4` file.

## 7. Discussion

- **Threshold Values**

  The first problem I encountered was to find the right threshold values both in the HLS space and in the gradient space.
  It took a while to find the right values, and I don't think I got the right idea on why the values I found worked well on the test images and test videos.
  **Lack of proper strategy:**
  I believe that defining a strategy in the search and an objective method to evaluate the result of binarizing the image would lead to better results in the project video and in the more challenging videos.

- **Video process pipeline - processing speed**

  Because I am using the functions that I wrote to test each step of the video pipeline, some operations are repeated and the whole pipeline is not optimised.
  This makes my pipeline unusable for realtime lane line detection.

- **Sanity check - missing lane detection not implemented**

  There is no way for the pipeline to robustly react when lane lines are not detected in certain frames.
  As long as lane lines are detected correctly enough, the whole pipeline works, but as soon as the detection fails for 1 or more frames, the whole detection in the future frames becomes unusable.

**Conclusions**:
I would start from refining the image binarization in order to become more robust to different lane line colors, possible shadows, almost invisible lane lines, etc…
Because this is the first step (after camera calibration), I believe that if this was more robust, the following steps in the pipeline would benefit a lot.

Secondly, I would have to implement sanity check and make sure that the detection is in line with where we expect the lane lines to be.

Lastly, I would like to test the whole pipeline on a live stream of a webcam mounted on a car, therefore the whole algorithm should be optimised and redundancies should be eliminated