Advanced Lane Finding Project

1. The goals / steps of this project are the following:
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. Smoothing and Filtering: Averaging lane curves across frames and drop those with poor lane detections.
9. Warp the detected lane boundaries back onto the original image.
10. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

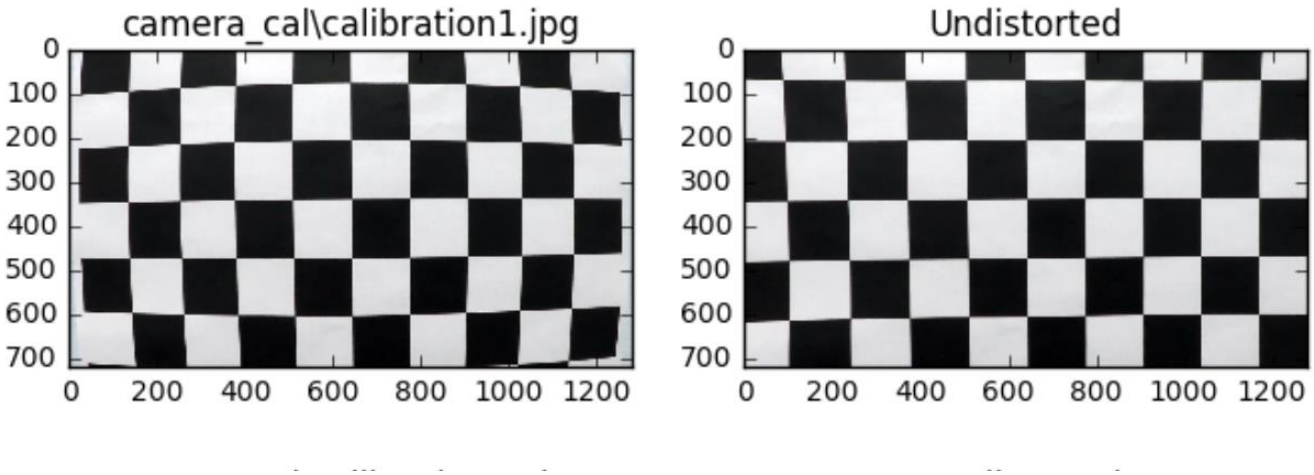
IPython Notebooks

- AdvancedLaneFinding - Exploration.ipynb. This notebook contains an exploration of algorithms for the step 1 to 6 mentioned above.
- AdvancedLaneFinding.ipynb. This notebook contains the processing pipeline with all algorithms for the lane finding.

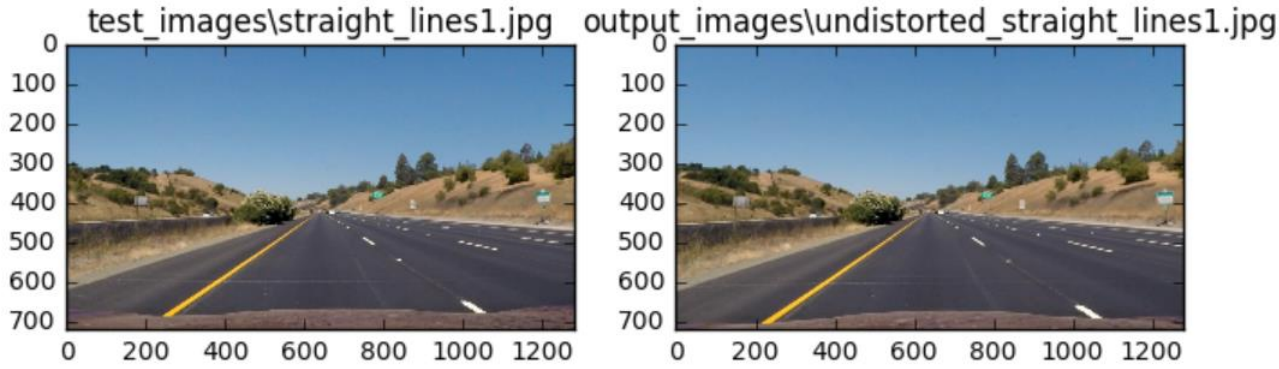
Camera Calibration and Distortion Coefficients

Correcting camera image distortion involves using calibration images to derive the camera matrix and distortion coefficients, and then using those parameters to undistort newly acquired images. For the purpose for camera calibration a set of 20 test images are provided by Udacity in the subfolder “camera_cal”. The code for calibrating the camera and calculating the distortion matrix can be found in both notebooks in the section “Compute the camera calibration using chessboard images”. I used the OpenCV function cv2.calibrateCamera to implement this functionality. Then I tested the distortion correction using the OpenCV function “cv2.undistort“ for the chessboard images (see section Apply a distortion correction to chessboard images in the notebook “AdvancedLaneFinding - Exploration.ipynb”). The undistorted images can be found in the folder “output_images”. The filenames start all with “undistorted”. Then I apply the distortion correction also to the test images in the folder “test_images”. The results can be found too in the folder “output_images”.

Here is an example for the distortion correction of the chessboard images:



And here for two of the test images:



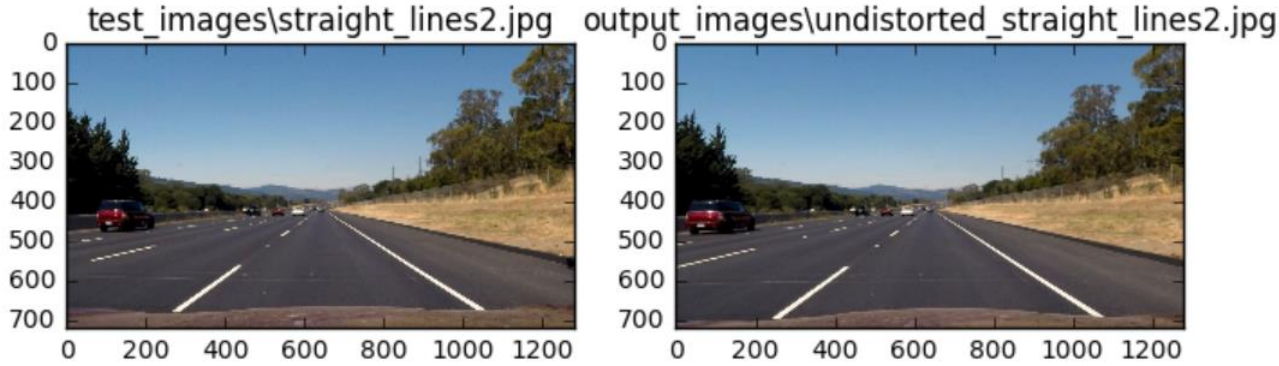


Image Thresholding - Exploration

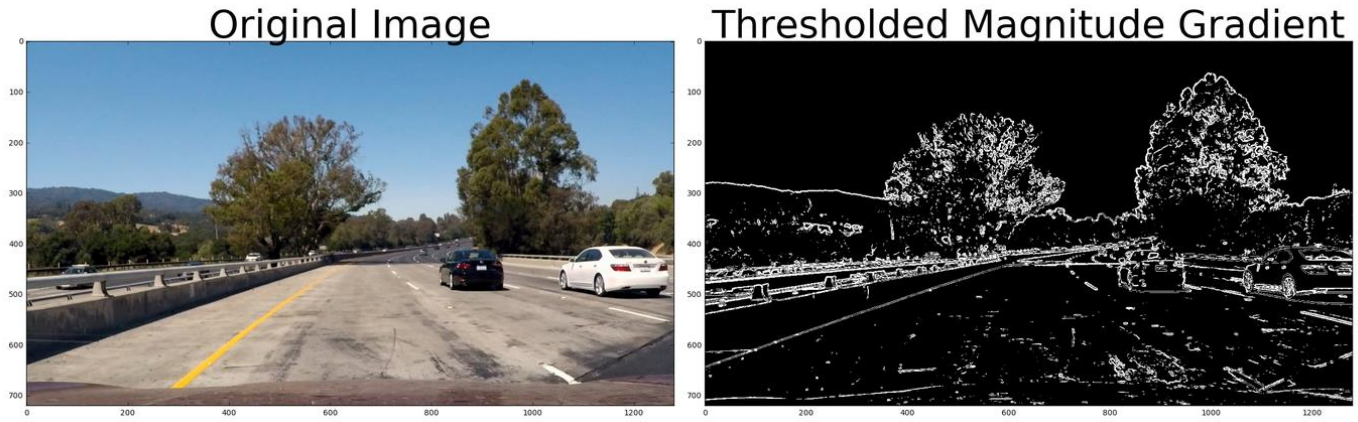
To make the lane detection easier some gradient calculation and color thresholding is very helpfull to generate a binary image which contains at best only the lanes as white lines. The notebook “AdvancedLaneFinding - Exploration.ipynb” contains the code from the Udacity Lessons in the section “2) Describe how (and identify where in your code) you used color transforms…”

Thresholded gradient applying the Sobel operator in x orientation

The following image shows that this not works well enough for our purposes. The yellow lane is not visible after processing

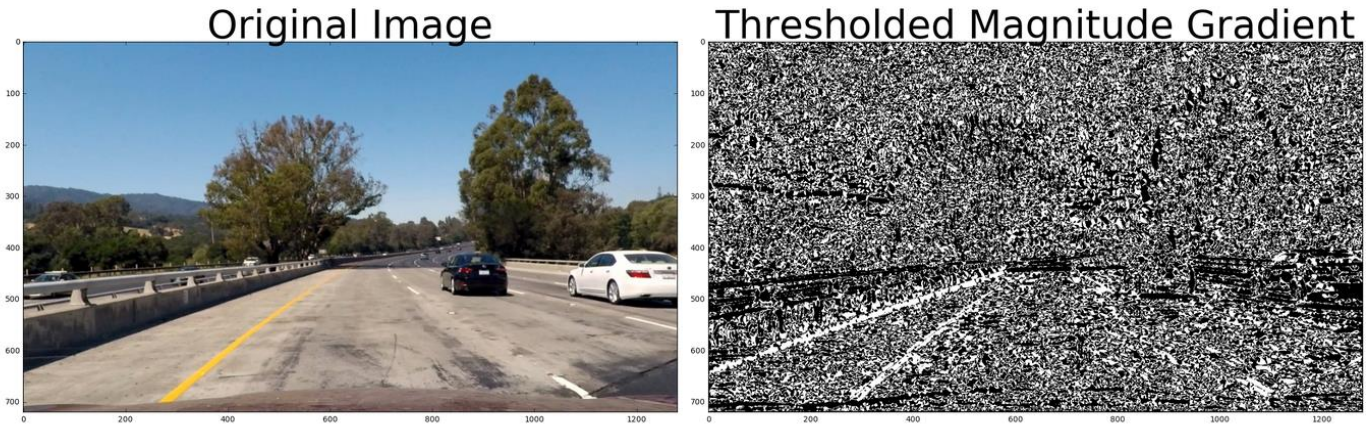
Magnitude of gradient

Next we are using the magnitude of the gradient using the Sobel operator in both the x and y direction. However, there are still some thick lines in the middle that could confuse the lane detector. We also notice that both gradients fail to detect the yellow lane at all. We'll need to fix this if we want to be able to drive anywhere with yellow lanes).



Direction of the Gradient

In order to detect those yellow lanes, we can compute the direction of the gradient as the arctangent of the gradient in the y direction divided by the gradient in the x direction. This is a much noisier gradient than our magnitude gradient, but it accurately captures the identical direction of the pixels from the yellow lane.



Combining Thresholds

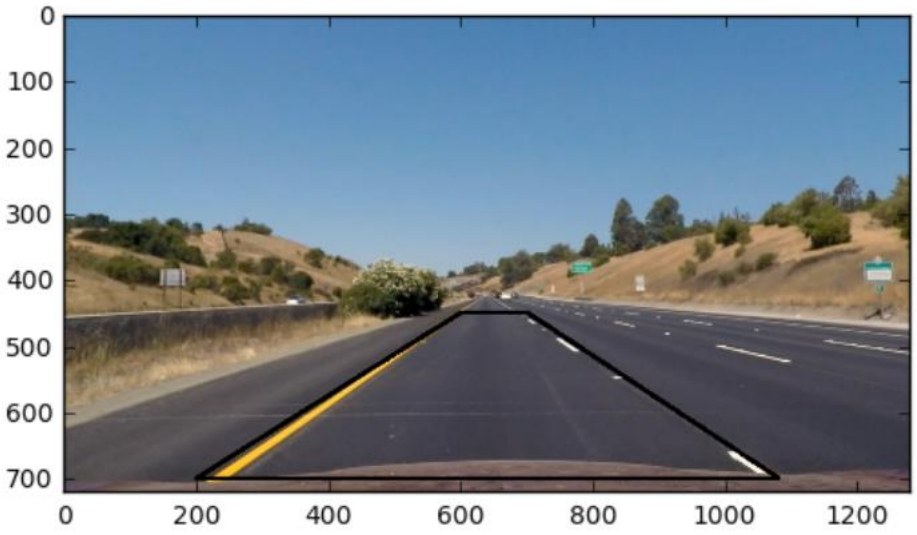
Now that we've isolated the white lanes with magnitude thresholding and the yellow lanes with direction thresholding, we can combine those images with base x/y sobel thresholds to get a result that captures both lanes. The result is not really well suited to capture the lane lines.

Color Thresholding

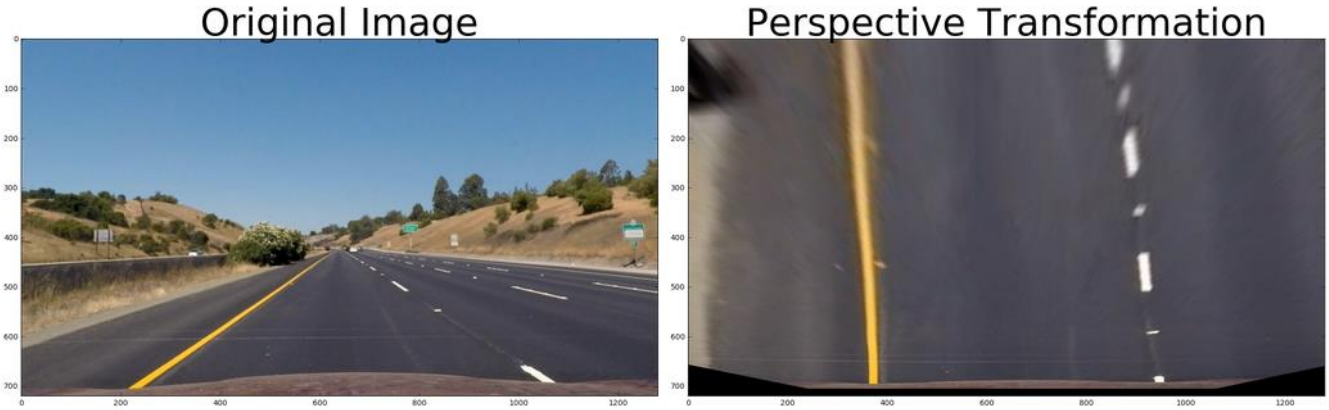
In this step, we isolate the lightness and saturation channels of the color image and then take an absolute Sobel in the x direction of the lightness channel. Finally we compute a binary that activates a pixel if it's saturated pixel is within the hardcoded threshold OR if it's scaled Sobel pixel of the lightness channel is within a separate hardcoded threshold. The result is quite good – the yellow and the white lane line are clearly visible with not too much noise around. We will use this color thresholding processing step in our final processing pipeline.

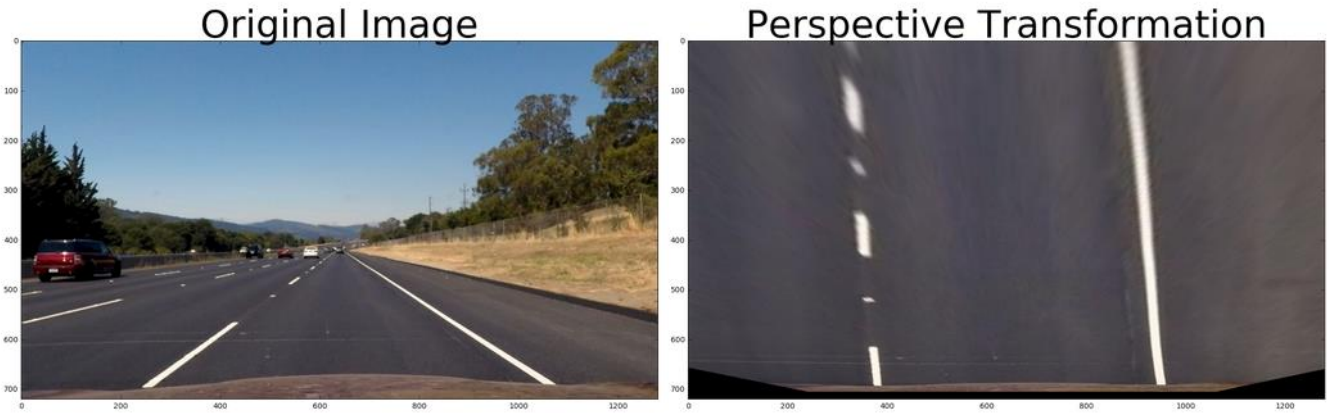
Perspective Transformation

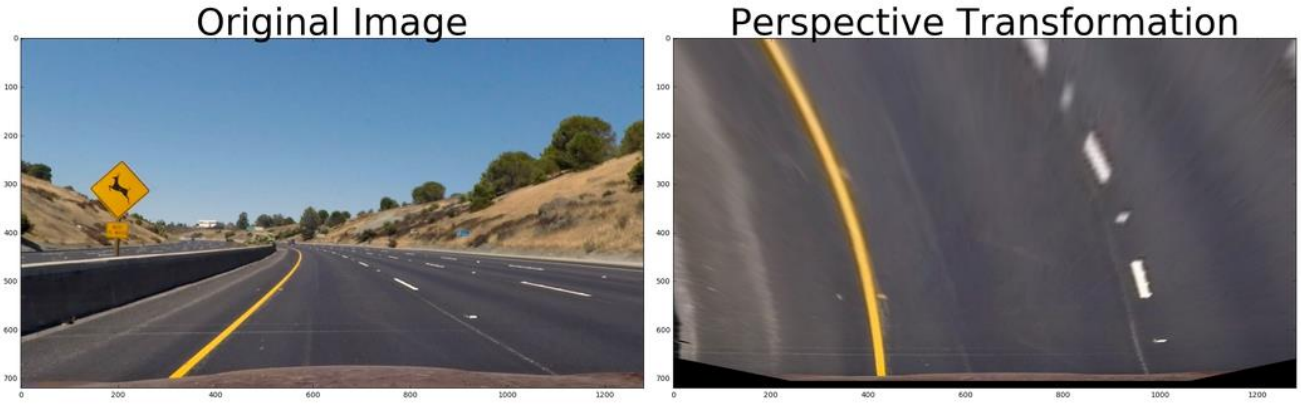
The next step is a perspective transformation to transform the binary image into a bird's eye-view. To accomplish this, one must first derive a transformation matrix using four source and destination points. The source points are from the camera view (after correcting distortion) while the destination points specify where in the bird's eye-view to project the source points. If we select four points defining a trapezoid in an image with parallel, we know that trapezoid will be transformed into a rectangle in a bird's eye-view. Consequently, the four vertices of the trapezoid are the source points and the four corners of the rectangle are the destination points. The following image shows the source trapezoid in black.



The following images shows some of the test images before and after perspective transformation. The lane lines are almost parallel which is important for further processing.







Region of Interest (ROI) Method

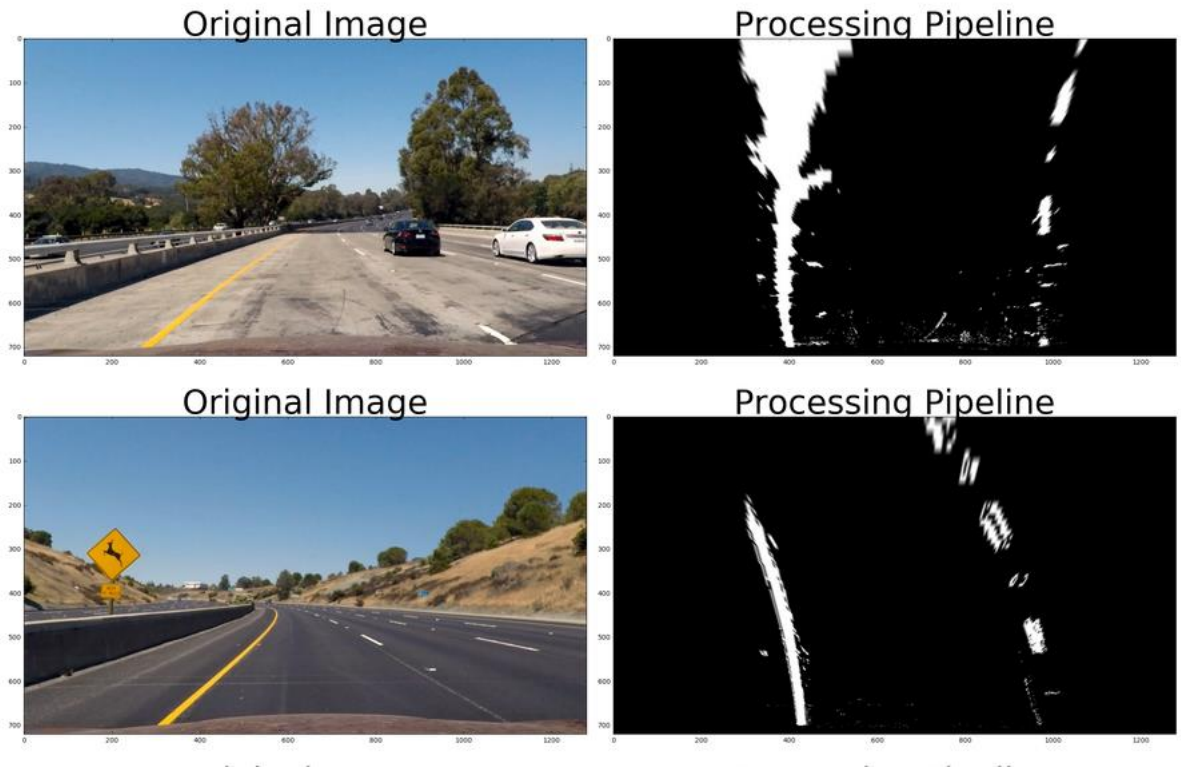
An important step is the to define a region of interest to suppress the parts of the images which are not important for lane detection. We define a trapezoid and apply that is image mask. The following image shows the original image and the defined ROI.

The Pipeline so far

We have now the following processing steps:

1. Camera correction
2. Color thresholding
3. Region of interest
4. Perspective transformation

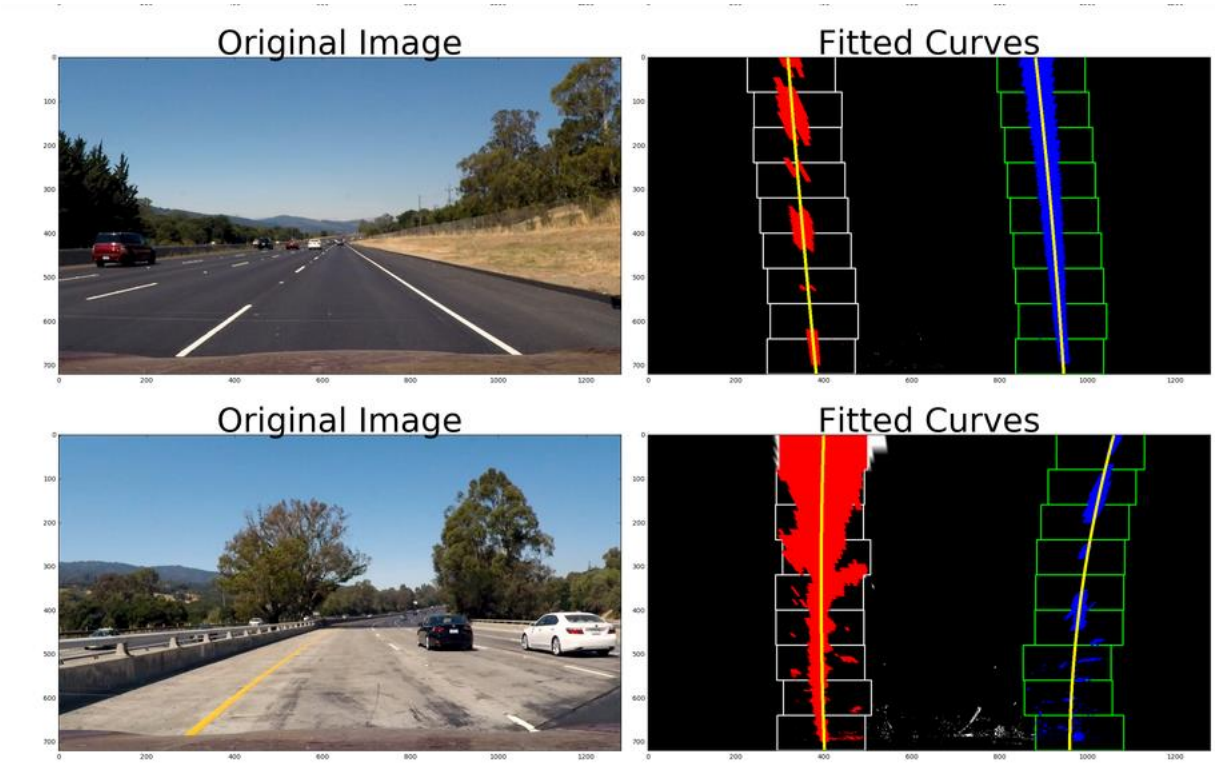
The following images shows some of the test images before and after these steps.



The lane lines are isolated quite well.

Lane Line Detection

First we use some of the sample code provided by udacity. It uses a histogram of the thresholded image to detect the line lanes. Then a sliding window is used to find the pixels which contributes to the lane lines. Then a second order polynom is fitted to these pixels. The following images shows this method applied to some test images where the sliding windows and the fitted lines are shown.



Overall Processing Pipeline

Based on the exploratory work the processing pipeline was implemented in the IPython notebook AdvancedLaneFinding.ipynb. The processing steps are (see the sections in the notebook):

1. Compute the camera calibration using chessboard images
2. Color Thresholding
3. Region of interest
4. Perspective transformation
5. Lane detection
6. Warp the detected lane boundaries back onto the original image.
7. Output visual display of the lane boundaries
8. Output numerical estimation of lane curvature and vehicle position.

The steps 1 to 4 are already described above and we will focus now on the implementation of the lane detection. For lane detection, a class **Line** was implemented. This class implements the following methods:

1. **sanity_check_lane**. This method implements a simple test if the calculated lane curvature of a new picture frame is not too different from the one before.
2. **measure_curvature**. This method calculates the curvature of a lane line in meters.
3. **set_line_base_pos**. This method is used to calculate the position of the car relative to the lane line.
4. **avgcoeffs**. This method calculates the average of the last N lane line fitting coefficients. To smooth the lane line detection a DQueue of the last N successful lane line fittings are used.
5. **fit_line**. This method fits a second order polynom to the lane line pixels and calculates the curvature. If the fitting is successful, the coefficients are stored in a DQueue and the flag detection is set to True. Otherwise the flag will set to False and the number of dropped frames will be increased by one.
6. **dectect_line_sliding_window**. This method implements the histogram and sliding windows method described above.
7. **dectect_line_fitted.** This method uses the successful fitting of previous frames for searching for the lane pixels in the new frame.
8. **dectect_line**: This method calls either the dectect_line_sliding_window method in the case that no fitting exists or in the case that too much frames were skipped. Otherwise the method dectect_line_fitted is used to detect the lane line.

The function **create_result_image** implements the step 6 and 7. The function detect_lanes implements the overall function for left and right lane detection using the preprocessing steps and the class Line. It implements also the step 8. The following images shows some of the test images and the output generated by lane detection.


Curvature: Left = 2644.57 m and Right = 2177.41 m
Lane Position: 7.00 cm right of center


Curvature: Left = 1790.93 m and Right = 2013.27 m
Lane Position: 7.00 cm right of center

Pipeline

I have applied the processing pipeline on the following videos:

1. **project_video**. The result of the lane detection is the video **result_project**. It works quite well without any faults.
2. **challenge_video**. The result of the lane detection is the video **result_challenge**. The lane detection fails several times. Sometimes the left border of the lane instead of the lane line is detected, sometimes the boundary between a darker part and lighter part of the lane is identified as lane line.
3. **Harder_challenge_video**. The result of the lane detection is the video **result_harder_challenge**. The lane detection works for some parts quite well for other parts it failed completely.

Discussion

It is relatively easy to process images/videos where road condition is nice and clean. However, when it comes to shadows, road material change or cases where background road color is relative light, the pipeline does not work as well. One reason is that the histogram method for lane detection assumes that there is only one peak in the histogram. This could explain for example the failures in the challenge video and partly in the harder challenge video. It would be better to detect all main peaks and then use some algorithm to find out which is the lane line with high probability. For example, if we are searching for the left lane and we get to peaks, the peak for the lane line should be nearer to the left side of the car. Another area of improvement would be the image preprocessing. Areas with low contrast (dark or very light) should use an adaptive contrast enhancement.