Advanced Lane Line Detection

Advanced Lane Finding Project

The goals / steps of this project are the following:

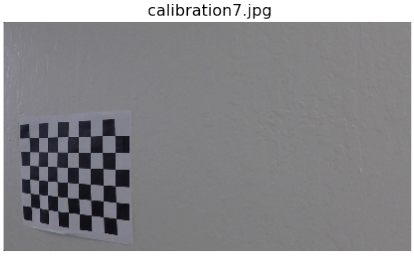
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").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

Camera Calibration

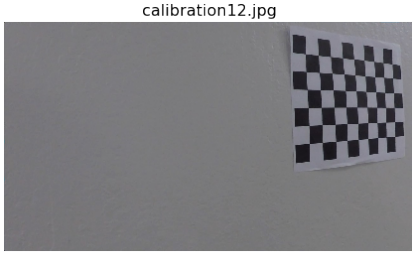
In order to detect lane lines correctly, we need undistorted images in terms that straight lines really are shown as straight lines. The code for this step is contained in the first code cell of the file in `src/laneliones/preprocessing.py` (lines 8 through 63). It requires a set of chessboard images in `data/raw/camera/calibration/*jpg` that were taken from different angles and different distances:

calibration2.jpg

calibration7.jpg



calibration12.jpg



I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. 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.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

Especially when looking at the lower parts of the image, the effect of the undistortion gets ovious.

The processing of `cv2.calibrateCamera()` takes several seconds and shall only be done once. Thus I created a class `DistortionCorrector` and a `namedtuple` called `CameraCalibration` to save the distortion coefficients:

```
# Data structure to save the results of cv2.calibrateCamera()
CameraCalibration = namedtuple("CameraCalibratioon",\
    ["ret", "mtx", "dist", "rvecs", "tvecs"])

# Class to do image undistortion
class DistortionCorrector(object):
    def __init__(self, _camera_calibration=None):
        ...

    # Read in chessboard images, and create a CameraCalibration
    def fit(self):
        ...

    # Transform a given input image based on the CameraCalibration
    def transform(self, _img):
        ...

    get_info(self):
        ...
```

Pipeline (single images)

1. Distortion correction of input images

The undistortion is simply applied by calling the `.transform(...)` method on an instance of the `DistortionCorrector` class. Here, an example for the image demonstrated in this pipeline:

test4.jpg Original

test4.jpg Undistorted

2. Creating a thresholded binary image

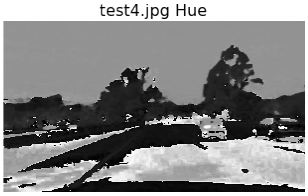
I used a combination of color and gradient thresholds to generate a thresholded binary image.

2.1 Color

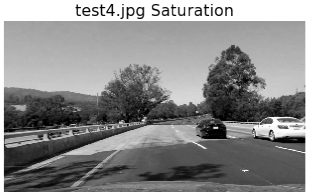
The color thresholding was inspired on the example given in the class. Thus, I first converted the images to the HLS color space and inspected the result of each channel:

test4.jpg Original

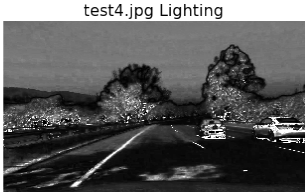
test4.jpg Hue



test4.jpg Saturation



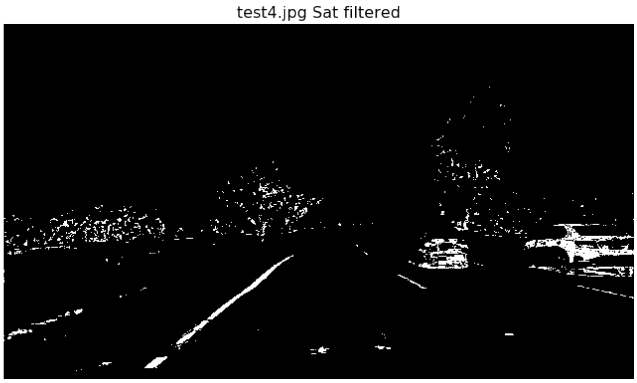
test4.jpg Lighting



As mentioned in the class, it was clear that the saturation channel shows the best resembling of the lane lines. For that reason, I decided to threshold the s-channel. The range of pixel values to keep was chosen as `[170, 255]`, which was empirically found. However, for a real productive application, such hard-coded thresholds would be very dangerous and a proper calibration method should be designed instead. Ideas that could be investigated are:

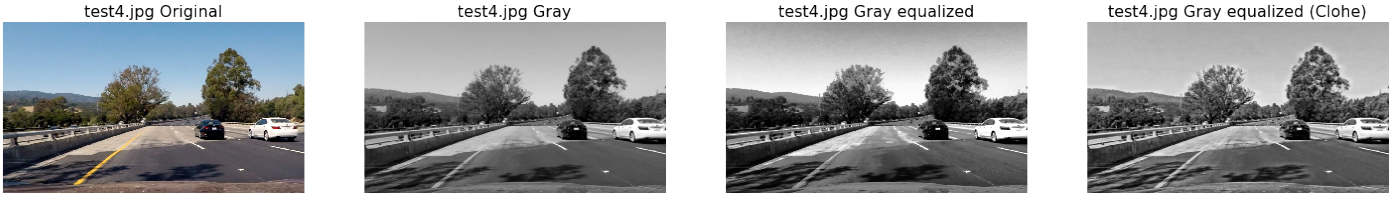
- Do the perspective transformation before thresholding an work with grid search and the correlation coefficient of x- and y- values on the thresholded image to find good values.
- Working with the variance of x-values could be investigated.
- Plot the histogram of thresholded images and investigate which parts we are cutting out with our thresholding. Depending on the maxima of the histogram, the thresholds could be adapted
- Experiment with adaptive histogram thresholding on the image channels
- All these steps could be done based on the area around the previously found lane lines. This could resemble what a human does while driving, when the lane lines suddenly look differently: We know where the line should be and immediatley understand the new "look" of the line.

The code for this is step is located in `src/lanelines/preprocessing.py` (lines 99 - 176). The result is acutally a numpy array with values `0` or `1`, i.e. a binary image:

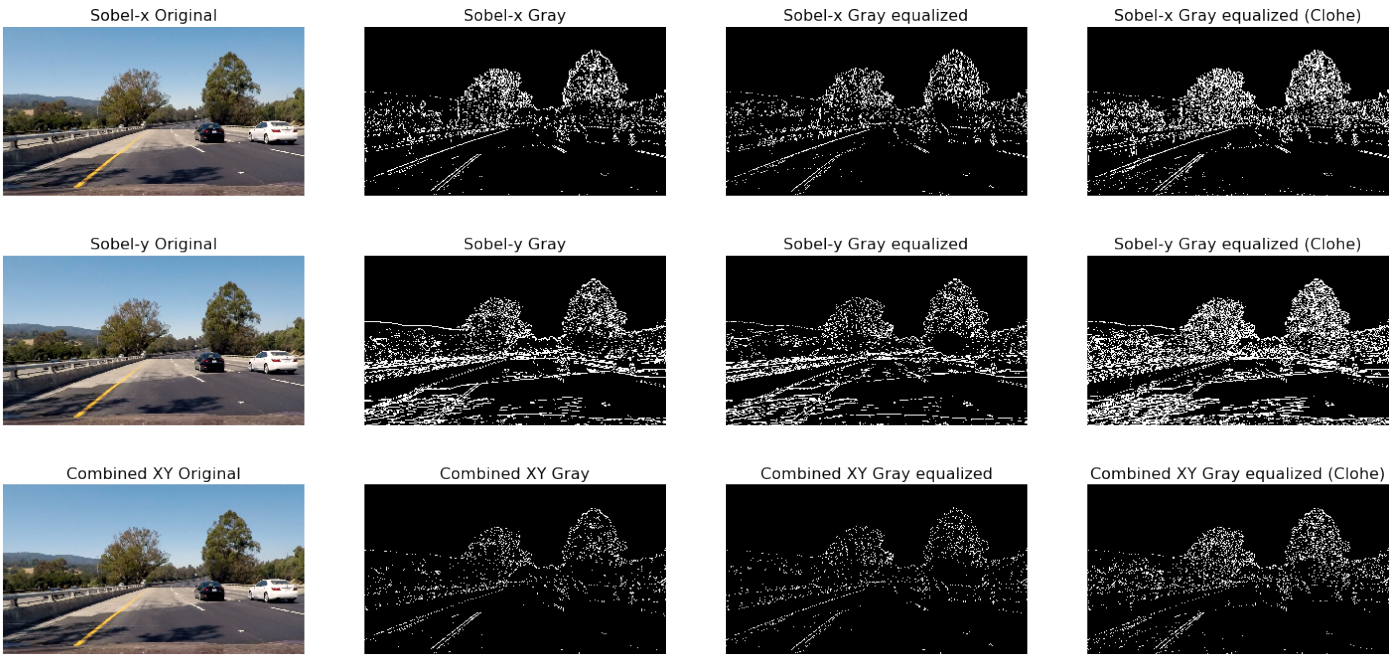


2.2 Gradient

Before computing gradients, the images were transformed to gray-scale and smoothened by a Gaussian blur with a `(3, 3)` kernel, to reduce noise. I experimented with applying histogram equalization and adaptive histogram equalization, as well:



Next, the Sobel operator was applied to the images on x- and y-axis. The images were thresholded to binary thresholded versions, where onle pixels with a gradient in the range of `[20, 100]` were kept. The thresholded x- and y-Sobel results were combined with the logical `and` operator. The results for each of the gray-scaled images are shown below:

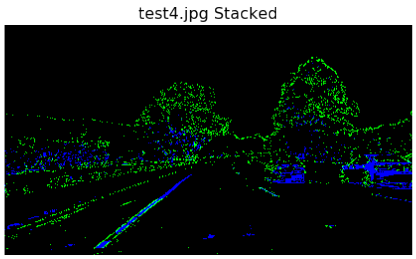


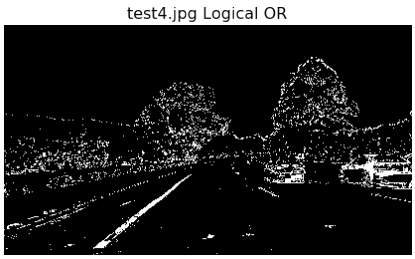
It turned out, that the equalization had a negative impact on the Sobel values, as more noise is generated.

However, it could be further analyzed if other thresholds lead to better results on the equalized versions - in first experiments no combination was found that could resemble the lane lines as clear as the simple gray-scale version.

2.3 Combination of color and gradient

In the final step, the two binary thresholded images were combined by the logical `OR` operator. The following figure shows the contribution of Sobel on the green channel and the contribution of the thresholded s-channel on blue the blue channel:





Except from some noise, the results looks good to go! However, we need to be sure that the thresholds may not be transferable to other lighting or road conditions due to hard-coded values.

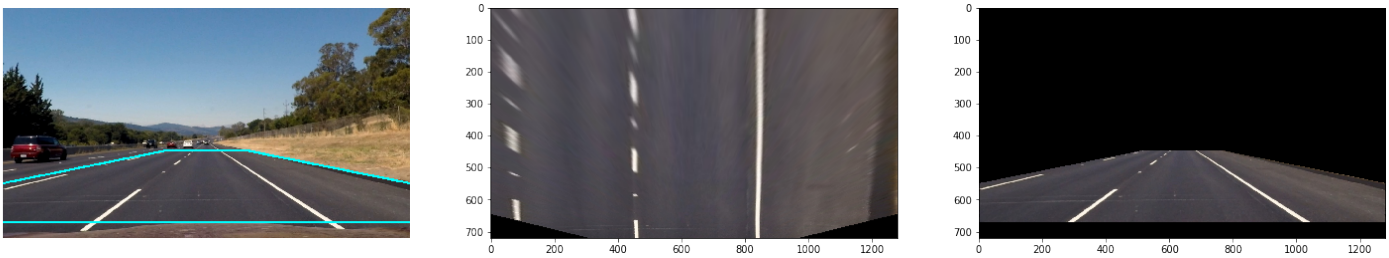
3. Perspective Transform

The code for perspective transform is located in `src/lanelines/preprocessing.py` (lines 69 - 94). It consists of three methods and one `namedtuple`:

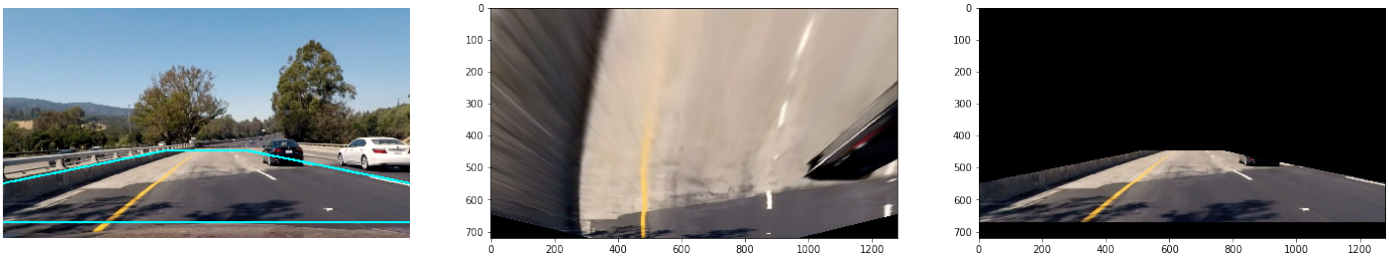
- `PerspectiveTransformationConfig`: a `namedtuple` to store the output of `cv2.getPerspectiveTransform(src, dst)` along with the height, width and src points and dst points of the transformation.
- `init_birdeye(...)`: Takes the sizes of a trapezoid on the input image, as well as the height and width of the input image as parameters. It acually 'pulls' the corners of this trapezoid to the corners of the target image. The result is returned as a `PerspectiveTransformationConfig` based on the given
- `image2birdeye(...)`: Apply the method `cv2.warpPerspective(...)` to a given input image based on the obtained `PerspectiveTransformationConfig`. The result is the warped image.
- `birdeye2image(...)`: Unwarp a given warped image by applying the method `cv2.warpPerspective(..., flags=cv2.WARP_INVERSE_MAP)` based on the obtained `PerspectiveTransformationConfig`. The flag actually unwarps the image.

An examplary application of these methods is shown below. An image with absolutely straight lines is

needed to find the right calibration of the perspective transform. Here, the first image is used to manually 'fit' the trapezoid. This was done by aligining the lines if the trapezoid parallel to the lane lines or street boundaries. The middle column shows the resulting perspective transformed verison, also called warped image. Thre rightmost column shows the unwarped image.



When applied to our example image from above, the following output is achieved:



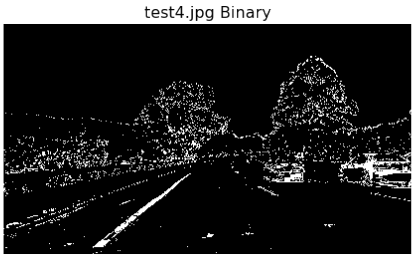
```
h, w = img_h, img_w
center = int(w/2)
src = np.array([[center-int(trpzd_width_low/2), trpzd_y_start],
                [center+int(trpzd_width_low/2), trpzd_y_start],
                [center+int(trpzd_width_high/2), trpzd_y_start + trpzd_height],
                [center-int(trpzd_width_high/2), trpzd_y_start + trpzd_height]], n
p.float32)

    dst = np.array([[0, 0],
                    [w, 0],
                    [w, h],
                    [0, h]], np.float32)
```

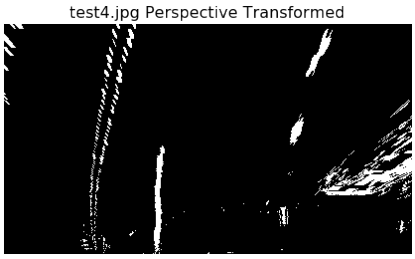
This resulted in the following source and destination points:

Source	Destination
515., 445.	0., 0.
765., 445.	1280., 0.
1890., 670.	1280., 720.
-610., 670.	0., 720.

The results were empirically verified on the given "straight_lines*.jpg" images. Applied to the binary

thresholded example, we receive the following result:





4. Fitting lane lines with a polynomial

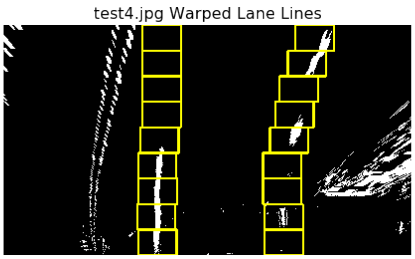
The code for fitting the lane lines is located in `src/lanelines/detection.py`.

The basic principle is as follows:

- Locate potential pixels of left and right lane lines
  - Sliding Window Search: Locate the start of the lane lines and perform a sliding window search. This is the preferred method for the first run, or if the search needs to be reinitialized.
  - Polynomial Search: Search around in the area around the last fitted lane lines, if possible. Backup: Do a Sliding Window search.
- Fit a polynomial to both, the left and right pixels

4.1 The Sliding Window Search

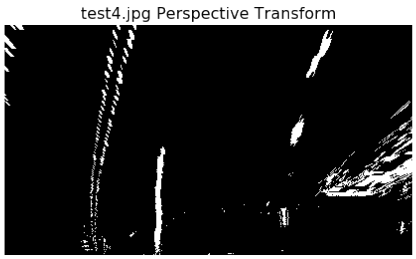
The principle of the sliding window search is to define two windows, one for the left and one for the right lane line. These window will be sliding from the bottom of the image to the top of the image, recentering after each step based on the pixels found in the previous step. For our example, this is illustrated below:

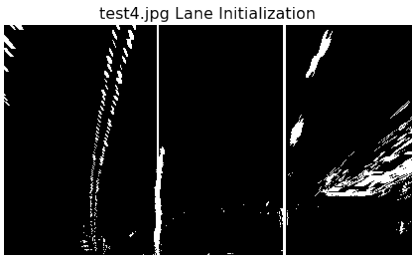


The sliding window search relies on the information where the windows should be initialized. The center of these windows shall be located on the left and right lane line. In order to do this initialization, the method `init_lane_lines(...)` (lines 9 - 51) searches the peaks with help of the scikit-learn library. The code defines a ROI in the lower half of the image and computes the pixel sum for each column. This gives us an overview on the intensity on the x-axis within this ROI. As in our binary thresholded image only potential lane line pixels have a value >0, the approach is to detect the peaks on the x-axis, which will be done with `find_peaks()`. As parameter, we define to only allow those peaks, that have a minimum distance of 280pxl to neighboring peaks.

```
from scipy.signal import find_peaks

_roi_width=800
_min_lane_distance=280
_h, _w = _img.shape

center = int(_w/2)
_roi_height = int(_h/2)
roi_x= [int(center - _roi_width/2), int(center + _roi_width/2)]

roi_y = [_img.shape[0] - _roi_height, _img.shape[0]]

window_hist = np.sum(_img[roi_y[0] : roi_y[1], roi_x[0] : roi_x[1]], axis=0)
peaks = find_peaks(window_hist, distance=_min_lane_distance)
```

Additionally, there is a filter for maximum distance of 500pxl applied in lines (32 -25). It is important to notice that the minimum and maximum distance shall not directly match the official lane widths on highways, as there might be effects in curves etc. The found start of the lane lines on the bottom of the image is shown by vertical lines:

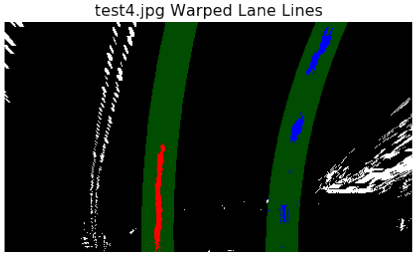


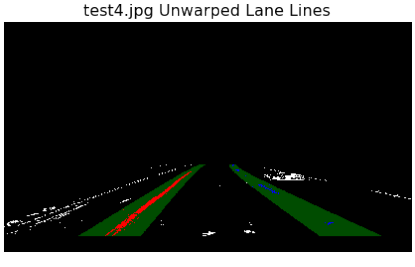


Important: If no valid left or right lane is found, the missing value is replaced by `center + (center - _min_lane_distance/2)` for the left and `center + (center + _min_lane_distance/2)` for the right.

4.2 Polygon Search

To improve efficiency, the polygon search can be applied when there is an existing, valid fit available for the previous frame. The core idea is, that such a previous fit defines to 2nd order polynomials and only those pixels shall be considered, that are within a defined margin around those polynomial lines, 50pxl in our case. The result is shown below:





The code for polygon search is found in lines 162 - 221. It is widely based on the code given in the classroom, except for the graphical parts.

4.3 Fitting Polynomials

The core function for lane line detection is `fit_polynomial(binary_warped, lane_history=None)` in lines 226 - 279. Both, the sliding window search and the polynomial search return the x- and y-indices of non-zero pixels on the left lane and sperately also on the right lane line: `leftx, lefty, rightx, righty`

The core functionality of `fit_polynomial(...)` is to fit two 2nd order polynomials for the mentioned pixel indices found for each line. This is done via:

```
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

Note: Here y is given upfront, as we lines are vertical most of the time. So our found polynomials can predict the x value of the lane line given the y value!

In practical application it turned out, that is necessary to apply a low-pass filter in order to get good results. To do this, a `recordclass` is created that stores the last 10 `left_fits` and `right_fits`, as well as the history whether the `leftx, lefty, rightx, righty` found in the previuos frame could be fitted to a 2nd order polynomial successfully:

```
LaneLineHistory = recordclass("LaneLineHistory", ["left_fits", "right_fits", "fit_
found"])
```

4.3.1 Moving Median

The above-mentioned `left_fit` and `right_fit` is thus not directly used, but first concatenated with the 10 last found fits for the respective side in the `LaneLineHistory` record. From these two lists, the medians are chosen as the final fits (235 - 244), so we implement a moving median. Here, compared to average, the median is prefered in order to reduce the impact of sudden outliers, i.e. when lighting conditions change promptly.

```
# Smoothen the fit if possible
if not lane_history is None:
    left = np.vstack((np.array([left_fit]), np.array(list(lane_history.left_fits))
))
    right = np.vstack((np.array([right_fit]), np.array(list(lane_history.right_fit
s))))

    left_fit_avg = np.median(left, axis=0)
    right_fit_avg = np.median(right, axis=0)
else:
    left_fit_avg = left_fit
    right_fit_avg = right_fit
```

4.3.2 Deciding for Sliding Window or Polynomial Search

The values of `leftx, lefty, rightx, righty` are provided to `fit_polynomial(binary_warped, lane_history=None)` by the helper function `find_lane_pixels(binary_warped, lane_history=None)` in lines (136 - 159). This class actually decides whether to use the sliding window search, or the polygon search.

If the `lane_history` object is existing and the fit of polynomials on the previous image was successful, i.e. `lane_history.fit_found == True`, the polygon search is applied. The polynomial curves to search along are also created out of the `lane_history` using the same sliding median technique mentioned above.

In line 148 - 154, there is an additional sanity check on the found polynomial curves. Only if there upper and lower distance is within the boundary of `[200, 600]` pxl, the polynomial search cabn be performed.

In all other cases, the full sliding window search is performed.

5. Radius of curvature of the lane and the position of the vehicle with respect to center.

I did this in lines 285 through 311 in my code in `detection.py`. It is mainly based on the code of the classroom with two extensions: - It uses the last element of `lane_history.left_fits` and `lane_history.right_fits` and averages them - A transformation from pixels to meters is done.

According to https://www.answers.com/Q/How*long*are*the*white*lines*on*the*interstate_highways, the white dashes have a length of 3m. Thus, I define `y_meter_per_pxl = 3m / 80pxl`.

According to Wikipedia, the Interstate Highway standards for the U.S. Interstate Highway System use a 12-foot (3.7 m) standard lane width. Thus, I define `x_meter_per_pxl = 3.7m / 375pxl`

The m/pxl scaling is used by scaling the pxl indices stored in `lane_history` based on the last image to

real-world coordinates. Then, a new polynomial is fitted on the left and right lane line. The final result is the mean of both curvatures.

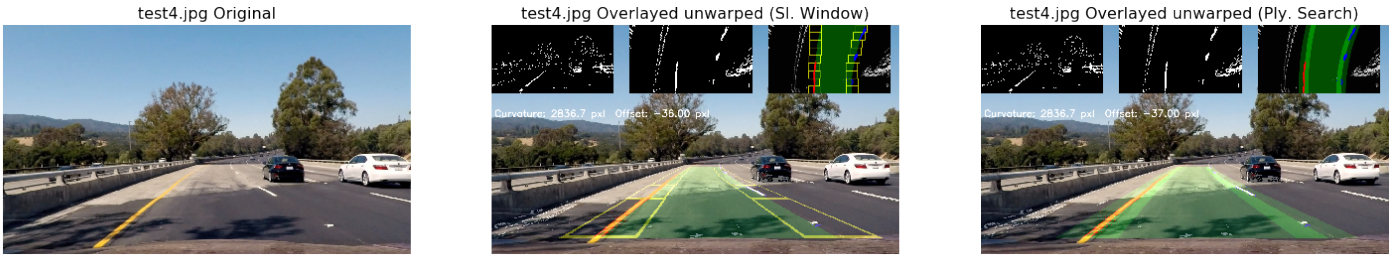
The offset from the center of the lane line is based on the assumption, that the camera was mounted in the center of the vehicle:

```
#_config.w holds the image width.
car_position = int(_config.w/2)

# Current lane lines position
left_fit = lane_history.left_fits[-1]
right_fit = lane_history.right_fits[-1]

left_lane_start = left_fit[0] * (_config.h -1)**2 + left_fit[1] * (_config.h-1) +
left_fit[2]
right_lane_start = right_fit[0] * (_config.h -1)**2 + right_fit[1] * (_config.h-1)
 + right_fit[2]

lane_center = int(left_lane_start + (right_lane_start - left_lane_start)/2)

# Offset
offset =  car_position - lane_center

# In meters
x_meter_per_pxl = 0.00986667
offset = offset * x_meter_per_pxl
```

Is curvature calculation is not very stable, as it also shall be done on a moving median.

6. Final Result

I implemented this step in lines 13 through 44 in my code in `pipeline.py` in the function `transform_img(...)`. Here is an example of my result on a test image:



Pipeline (video)

The pipeline was applied to the video `project_video.mp4`. The result can be found here:

Here's a link to my video result

Discussion

Problems faced in this project were, that I worked on the iPython notebook with the provided test images without problems, but in the real video, the lines were not correctly detected. One of the main reason for this was already mentioned above in Section 2.1: There are a lot of hardcoded threshold values in the code, that are not directly applicable all kind of scenes. This should rather be solved by more sophisticated calibration methods.

My pipeline particulary fails in steep curves (moving median prevents adaption), closly parallel lines like in the `challange_video.mp4` (closly parallel lines confusing the pipeline), obstacles blocking the lane lines, varying inclination of the road, bumpy roads that shake the vehicle, changing weather conditions and esp. night.

The approach I took to solve the detection for the `project_video.mp4` was described in detail above. Ideas to improve the pipeline shall include information about the road. However, besides the improvements outlined in Section 2.1, the following ideas could be explored:

- The color of lane lines
  - Color filtering on the yellow and white channel
  - In case of varying light, recalibrate to find the line where we assume it to be
- Lane lines run in parallel
  - Adapt the sliding window or the polygon search to respect that
  - Estimate the position of the other line, if only one was detected
- Lane lines have a sharp border compared to the road
  - Apply more image thresholding techniques, like Sobel magnitude and Sobel angle
- How quickly the curvature changes on which road
  - Adapt the length of the moving median based on the curvature gradient
  - Use information about possible curvature ranges based on the speed in order to prevent unrealistic matches.