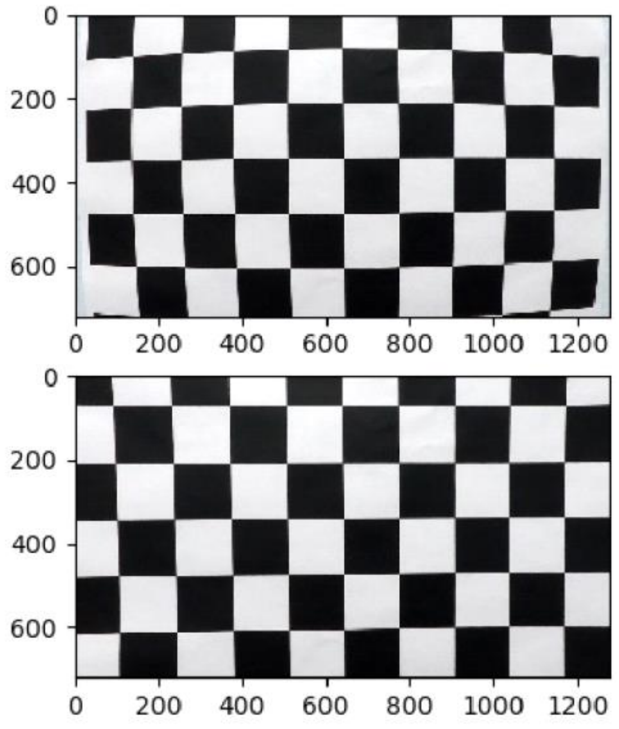
Project 2: Advanced Lane Finding

Dienstag, 24. Dezember 2019 01:14

Camera Matrix and Distortion Coefficients

The camera calibration was done with the given images from a chessboard with a 9x6 pattern. The pattern could be found in 17 of 20 images. The figure below shows a distorted (top) and an undistorted chessboard pattern image. In order to calibrate an image, all detected points in the image are mapped to 3d object space. The mapping between those spaces is calculated within "calibrateCamera"-function.
The camera calibration is implemented in "calc_calibration_coeff".



Pipeline for advanced lane finding

The pipeline contains following major steps which are shortly described in seperate chapters:

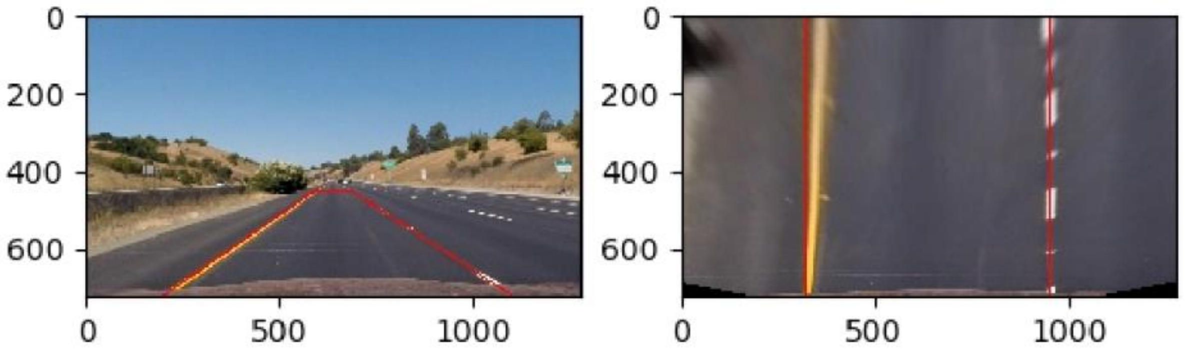
1. Perspective Transform
2. Color Transform and gradients
3. Identification of lane line pixels
4. Polynomial fitting of curvature
5. Calculation of Curvature and position of vehicle in lane
6. Final image with lane area (back-warping)

1. Perspective Transform

The perspective transform is needed in order to generate a top-view-image s.t. the curvature of the street can be measured. To find corresponding locations, a trapezoid was fitted on an image with straigt lines. These points were used as the mapping function between the "undistorted, undwarped" and "undistorted, warped" image. The mapping matrix is given below:

Original x	Original y	New x	New y	
200	720	320	720	Bottom left
1150	720	950	720	Bottom right
700	450	950	0	Top right
600	450	320	0	Top left

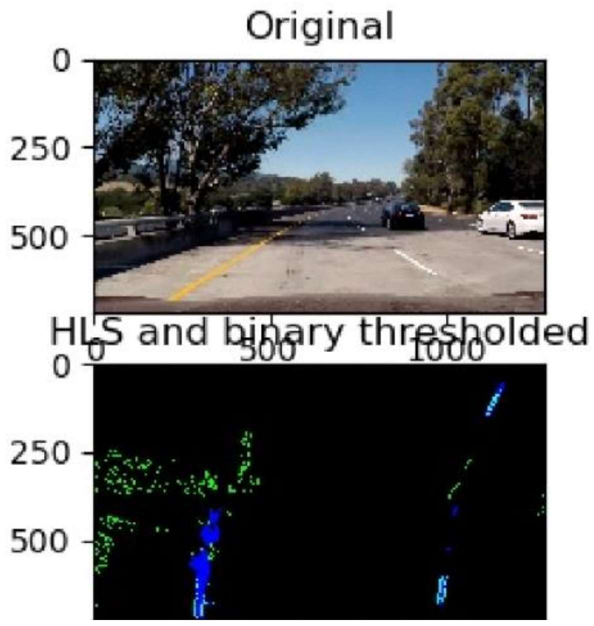
The output of the perspective transform can be seen in the image below:



The implementation of the perspective transform was done in "warp_undistort".

2. Color Transform and gradients

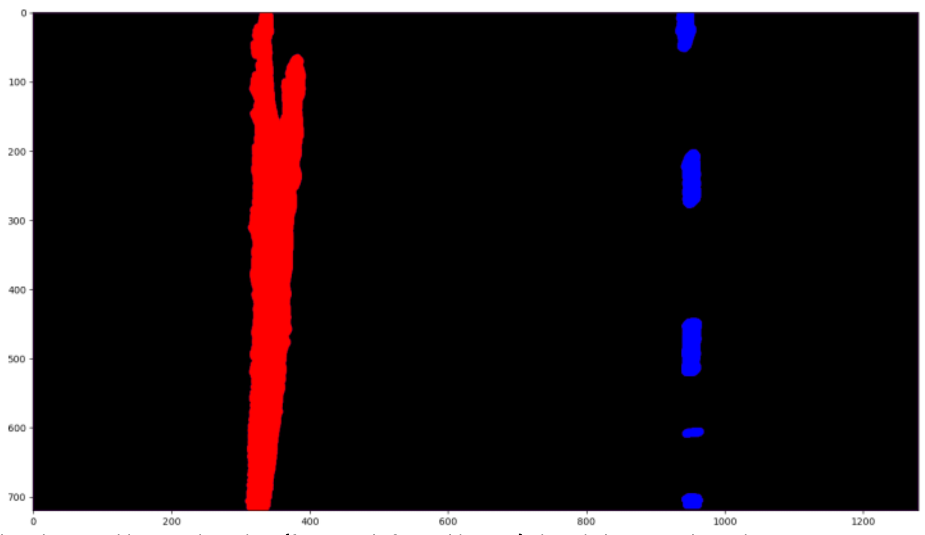
While experimenting with the s-channel after RGB-to-HLS-conversion I saw, that the s-channel is prone to shadows and lighting conditions. This leaded to wrong detections as seen in the image below.



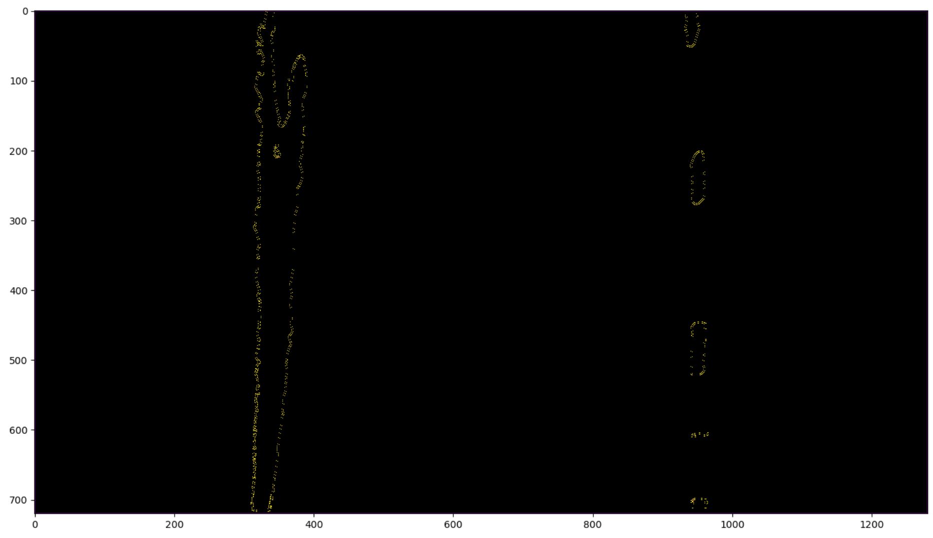
Due to this reason, white and yellow color is filtered within the image in RGB- and HSV-space with following threshold values:

White	180 <= R <= 255	180 <= G <= 255	180 <= B <= 255
Yellow	90 <= H <= 110	60 <= S <= 255	100 <= V <= 255

In order to avoid noisy data like seen in the binary thresholded picture above, a low-pass filter Median-Blur is used with a kernel-size of 11. Beneath, examples with color-thresholding are shown.



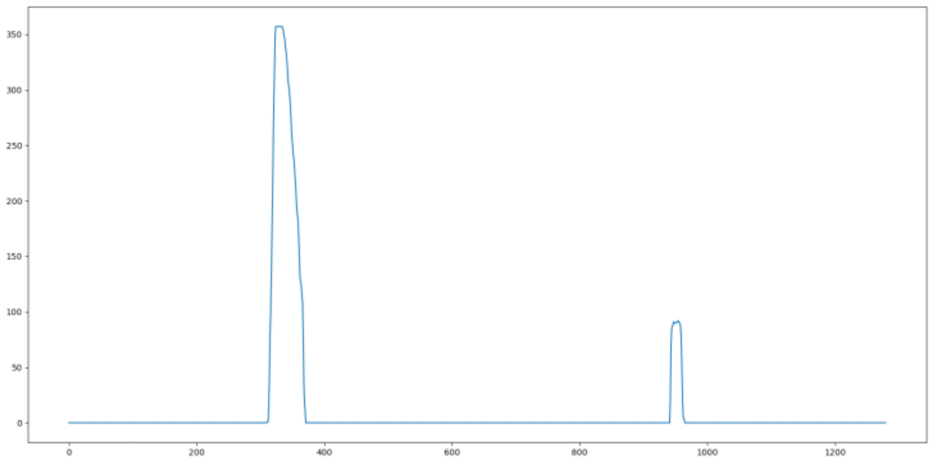
In order to avoid wrong detections (for example from white cars), the sobel-operator is used to highlight differences in x-direction. The result after applying Sobel is shown below.



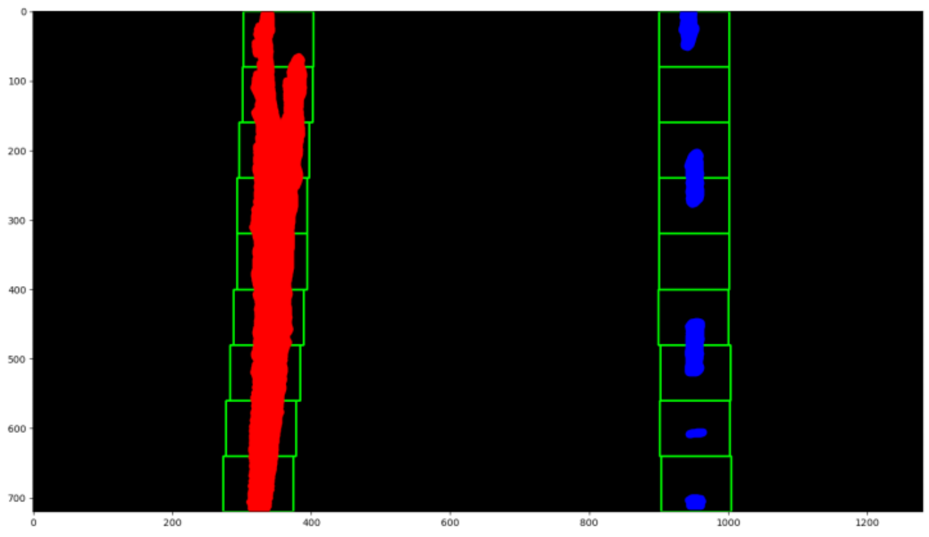
A disadvante when applying Sobel is that the number of valid lane pixels is much lower than the previously implemented color-thresholding. Due to this reason,

3. Identification of Lane line pixels

For the first image, the sliding window approach is used in order to find relevant lane pixels within a given region. The starting point for the x-position is determined with calculating the histogram seen below.



The new x-value of the sliding windows is determined by calculating the mean-values of each sliding window. An interesting parameter to play around with is the size of the margin within the relevant pixels are searched for. If the margin is too large, the measured curvature gets too inaccurate. If the margin is chosen too low, higher curvatures cannot be covered with the algorithm. I experimented with the margin-size of 50 until 100.



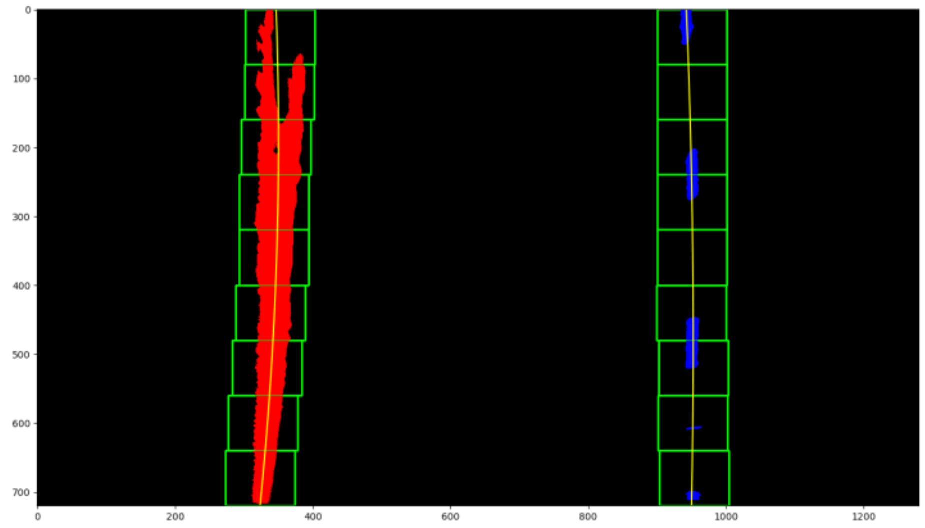
The implementation of color transform was done in "apply_HLS_threshold".

If the line was detected during the last frame, the change of curvature within the next frame will be very low. In my algorithm, the current determined polynomial coefficients are compared

to the mean-values of the last "n" iterations. If the difference is lower than a threshold, relevant pixels are chosen within a margin around the polynomial detected during the last frame. The tracking within the given margin is done in "search_around_poly".
The final polynomial coefficents are the mean-values of the last "n" coefficients, s.t. the output gets low-passed and outliers are filtered.

4. Polynomial fitting of curvature

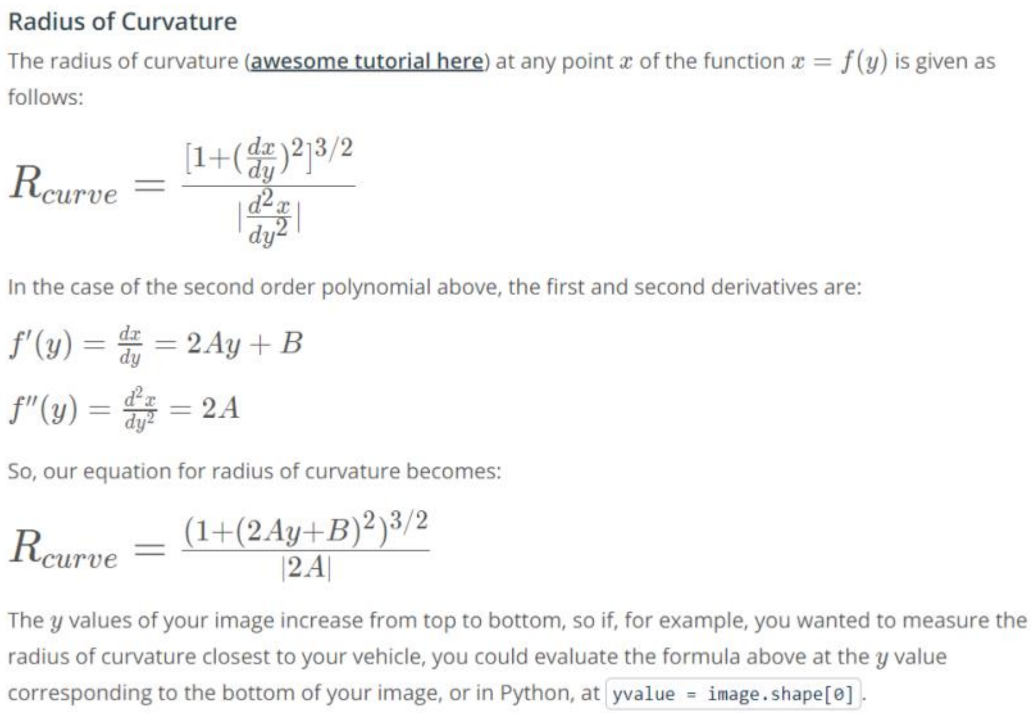
With the given relevant pixels for left and right line, a 2nd order polynomial function is fitted to the given pixel-values (x,y). This fitting is only done, if there are enough pixels. If there are not enough pixels, the lane is set to be not detected (..._lane.detected = false).



The polynomial fitting is implemented in "fit_polynomial".

5. Calculation of lane curvature and position within the lane

If the lanes are detected, the curvature and position within the lane can be calculated by the formula givin within the course (screenshot taken from course material "Measuring Curvature I":



Radius of Curvature

The radius of curvature (awesome tutorial here) at any point x of the function $x = f(y)$ is given as follows:

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

In the case of the second order polynomial above, the first and second derivatives are:

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

So, our equation for radius of curvature becomes:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

The y values of your image increase from top to bottom, so if, for example, you wanted to measure the radius of curvature closest to your vehicle, you could evaluate the formula above at the y value corresponding to the bottom of your image, or in Python, at `yvalue = image.shape[0]`.

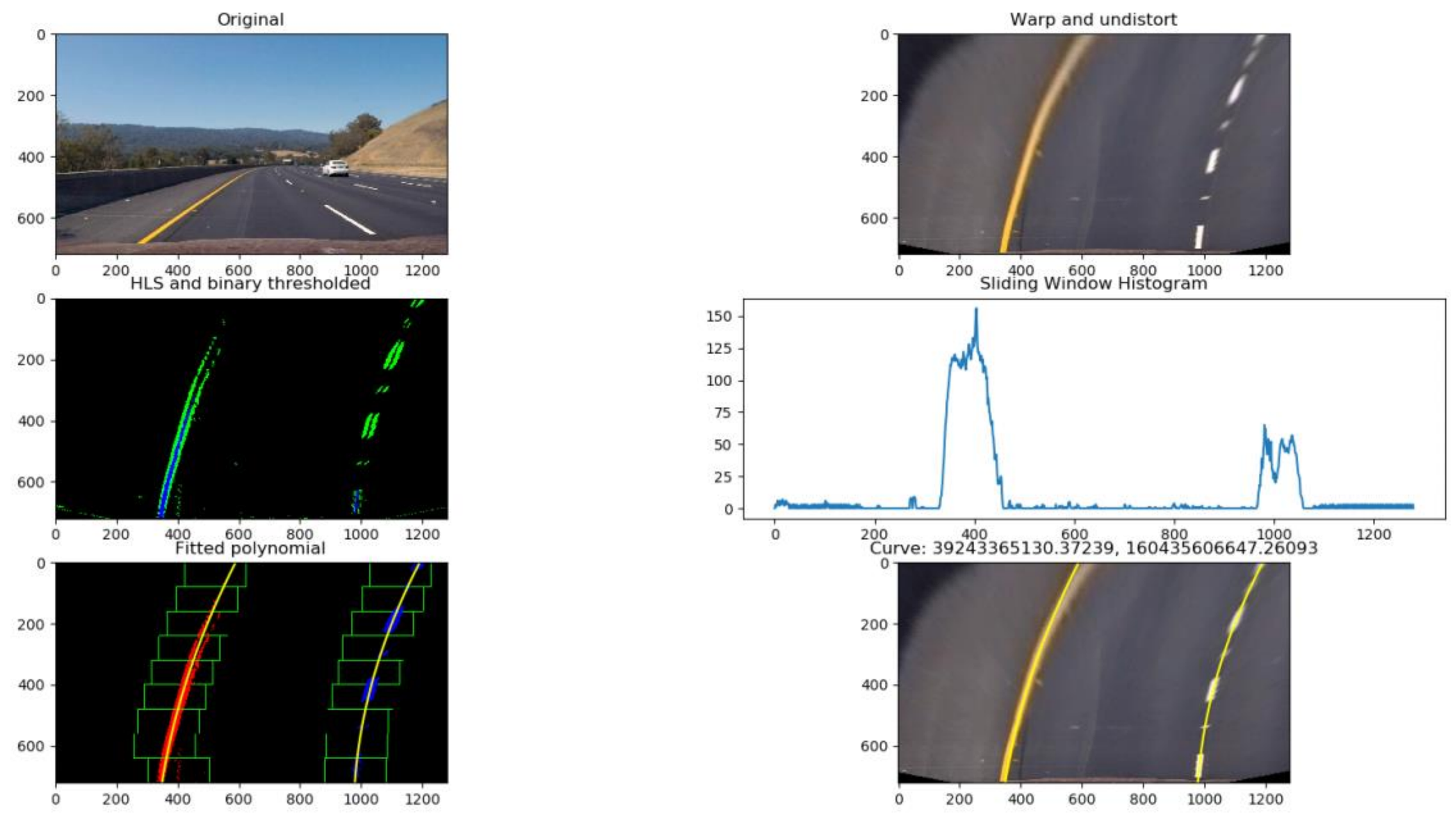
For calculating the position of the car within the lane, it is assumed that the camera is mounted in the middle of the car. The position is given by following formula:

pos = ([x_pos right_line(@bottom of image) - x_pos left_line(@bottom of image)] - img_width/2) * xm_per_pix

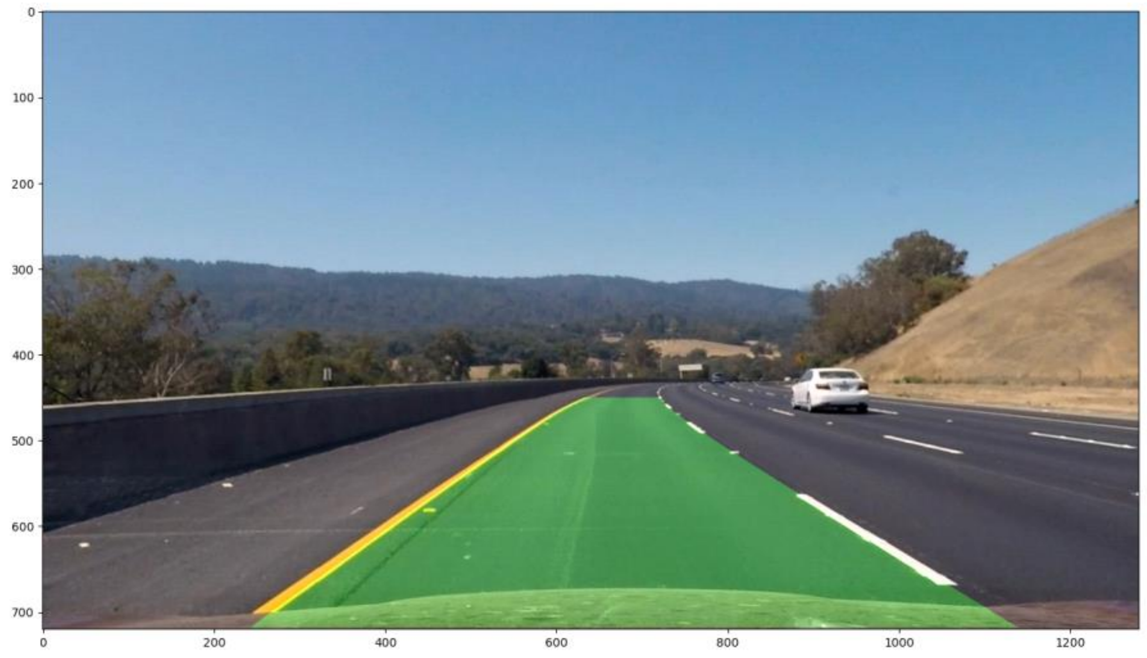
The calculation is done in "measure_curvature_real".

Summary

The following picture shows all these steps as a short summary:



Following picture shows detected curvature back-warped to the original image:



Sanity Checks

For robustness, some sanity checks were implemented:

- Look-ahead-filter: if a line was detected in the previous frame, the distance of the polynomial coefficients between the current and previous frame are lower than a certain threshold and both lane lines were successfully detected, the next frame will search for pixels within a margin around previously calculated polynomial. My assumption here is that if the polynomial coefficients are close to each other, the curvature will be close as well.
  If the difference between two frames gets too large, the algorithm gets a reset and starts with the sliding-window-approach for the next frame.
- Detection-success: if there are to less pixels to fit a polynomial, the line is not detected and the "..._line.detected"-variable is set to "False".

Discussion

Problems during implementation

I had some problems when implementing the color threshold, since the images from the different video streams are very different. They are very sensitive to lighting conditions, s.t.paraneters for color-thresholding were hard to determine. After searching online I found a solution from former Udacity-students, which I implemented and adopted in order to be able to solve "project_video" and "challenge_video" as well.

Problems with algorithm

In order to avoid the above mentioned problems, the video should be preprocessed with some equalizing algorithm (histogram-equalization on each channel, etc.). May be the exposure time of the camera could be adapted online during recording.