Project 4. Advanced Lane Detection. Writeup.

Goal of the project: Build a pipeline for detecting the ego lane (current lane of the image capturing car) in the camera captured image and use it for road lane detection in the video file/stream.

The 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 camera-captured image.
- Use color transforms and/or gradients, to create a thresholded binary image from undistorted image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect pixels of left and right line and fit them to find the lane boundaries.
- Determine the curvature of the lane and vehicle position with respect to center of the lane
- Warp detected lane boundaries back onto original image and overlay the numerical estimation of lane curvature and vehicle position.
- Apply above defined lane detection pipeline to provided test video file **project_video.mp4**
- (Optional) Optimize detection pipeline for video stream use.

Project resources

My project includes the following required files:

- **pipeline.ipynb** - Jupiter notebook file containing the implementation of pipeline for detecting ego lane in a video file
- **output images/** - folder containing the images that visualize the results of each step of detection pipeline execution. Description of each image (as well as a copy of image itself) is provided in a writeup report.
- **writeup.pdf** - summarizing report of the work done in scope of this project (this document)
- **project_video_out.mp4** - video recording of applying the implemented lane detection pipeline to provided **project_video.mp4** test video file.

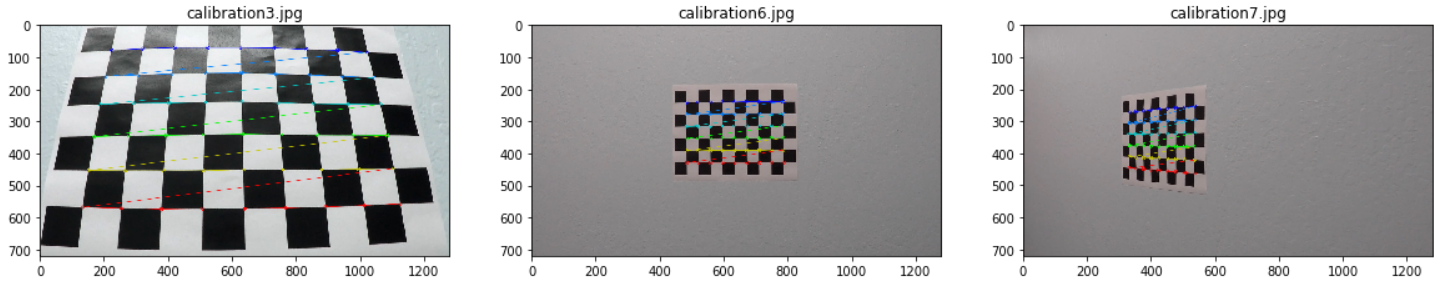
Additionally project includes the GUI tool script image_thresholding_lab.ipynb for fine-tuning the various sort of parameters for step 3.*Construction of thresholded binary image*

All above mentioned project recourses are can be found here.

All visualizations used in this write-up document are implemented with means of matplotlib.pyplot library.

1. Camera calibration

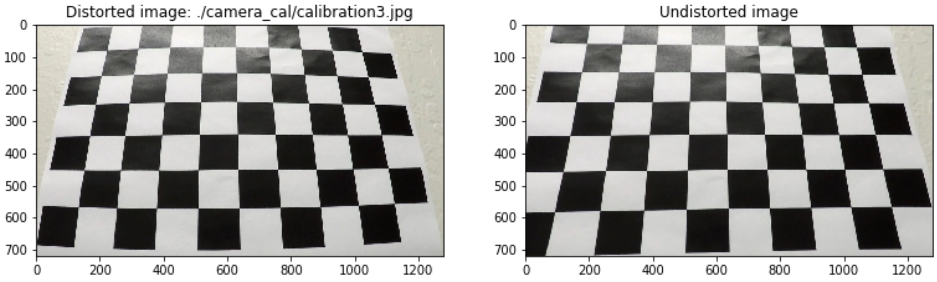
To calibrate the camera I use the chessboard corners detection method when same picture of chessboard is captured by camera from different angles and resulting images are used by OpenCV function findChessboardCorners() to find the chessboard corners in calibration images. Here is the result of detection the chessboard corners on several of calibration images:



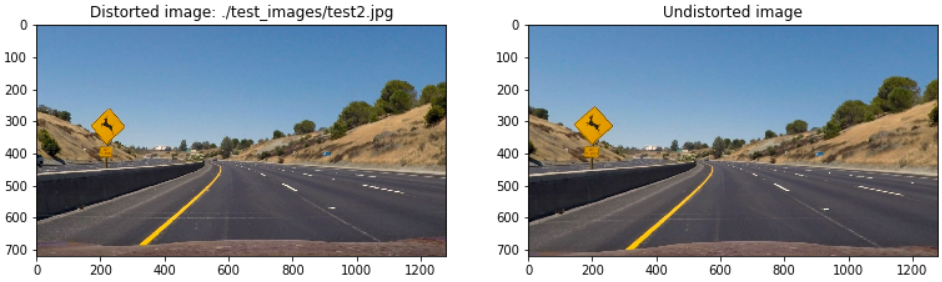
Then detected coordinates of the chessboard corners from all the calibration images together with reference (ground truth) coordinates of the corners are used to compute the camera calibration matrix and distortion coefficients using the cv2.calibrateCamera() function (see block **1. Camera calibarion** in pipleline.ipynb).

2. Correcting the image distortion

To correct the distortion of images captured by camera I use the Camera Calibration Matrix and Distortion Coefficients calculated in previous step and feed them together with original image into OpenCV function cv2.undistort(), which generates undistorted image. (see block **2. Correcting image distortion** in pipleline.ipynb). Here the result of distortion correction for one of calibration images:



And same distortion correction method applied to one of test images extracted from target video file **project_video.mp4**



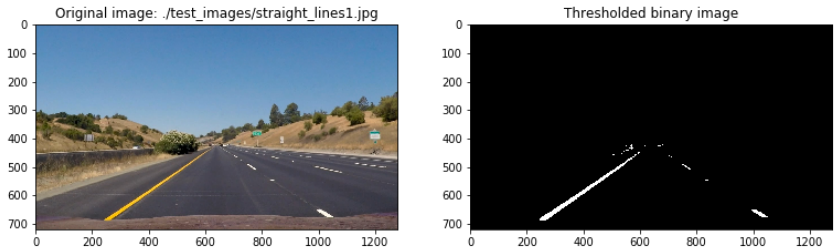
3. Construction of thresholded binary image

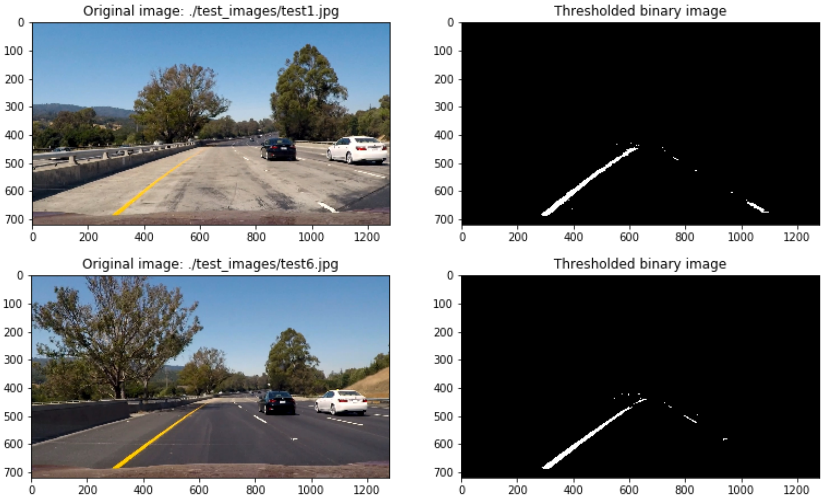
To generate a binary image that would contain mostly lane line pixels I have used following components:

- combination of multiple color channels (R, G, B, L,S,V, grayscale) thresholded by pixel value using different (empirically selected) pixel value ranges (one range per channel) (see lines 80-87 of block **3. Construction of thresholded binary image** in pipleline.ipynb). Thresholded color channels are then combined into single binary image using Boolean OR condition, means where one channels are weak to identify the line pixels, another channels are supposed to compensate this weakness. In total this combination should give consistent easily recognizable right and left lines of white color of the black background (see lines 96-98 of same block).
- The drawback of combining so many color channels is that additionally to lane line there are also lot of noise in generated binary image caused by shadows from the trees for example. Thus additionally to above combination I use a combination of various gradients of pixels in the mage (Sobel by X orientation, Sobel by Y orientation, combined gradient magnitude, and gradient direction) thresholded by empirically selected value ranges (see lines 76, 89-92 of block **3. Construction of thresholded binary image** in pipleline.ipynb). Thresholded gradient values are then combined into single 'gradient filtering' binary image using logical condition expression (see line 100 in the same block) selected empirically after trying different combinations.

The final binary image is the result of logical AND combination of binary images thresholded color channels and thresholded gradients. (see line 102 of same block.). Additionally I cut out (turn in black) from generated binary image all white pixels that do not belong to the area of interest (the region in the image where I expect to detect the lane boundaries + small margin fur curvy road segments) (see line lines 103, 51-73 of same block). All threshold values have been empirically selected using the convenient GUI based tool script, which I have implemented using ipywidgets library (see image_thresholding_lab.ipynb)

Here's are several examples of binary images generated from test images:



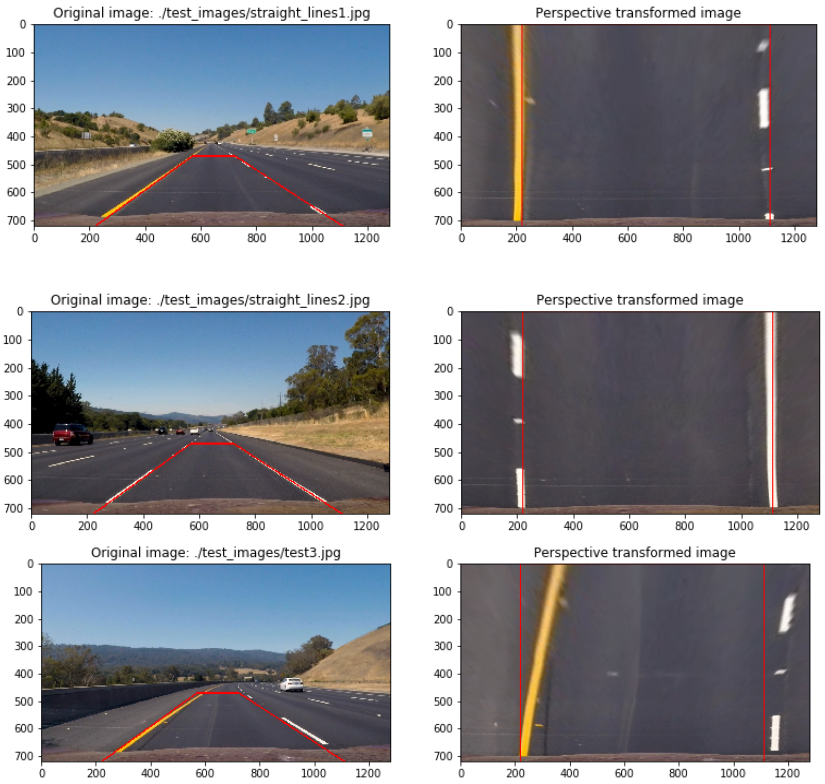


4. Perspective transformation of binary image

To make a perspective transformation of the images I have implemented following steps: (see lines 1-38 of block **4. Perspective transformation of images** in pipleline.ipynb)

- empirically defined the reference points in original image (SRC). Note, that upper reference points are located not on upper boundary of the image, but slightly below the level of horizon. In this way the unimportant part of the original image (e.g. sky, trees, etc..) is being automatically cut out from resulting warped image
- defined the reference points for target warped image (DST), by taking the points where lane lines are touching the bottom boundary of the original (undistorted) image and projecting them to the upper boundary of the image
- used SRC and DST reference points to calculate transformation and inverse transformation matrices using OpenCV function cv2.getPerspectiveTransform()
- generated perspective transformed image from original image using calculated transformation matrix and OpenCV function cv2.warpPerspective()

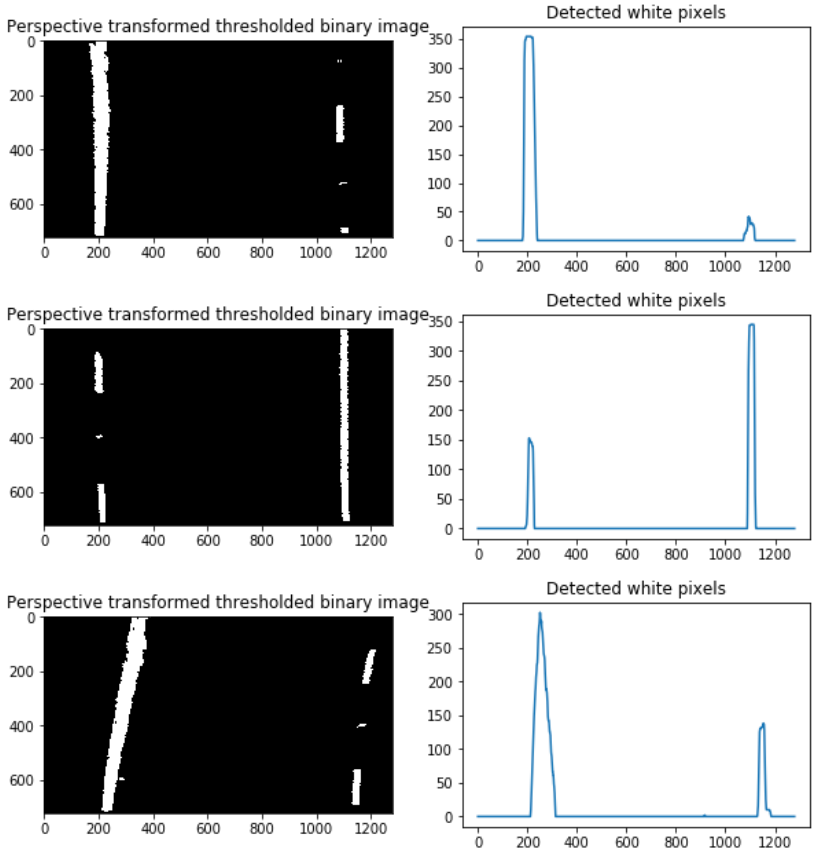
I have verified that my perspective transform was working as expected by drawing the lines connecting source points onto a test image and lines connecting destination reference points on the generated warped image to verify that the lines appear parallel in it. Here are several of examples (including one representing curved lane use case):



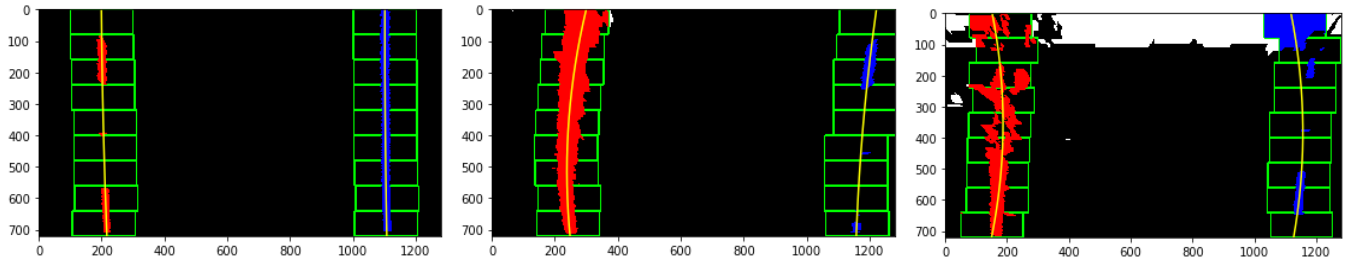
5. Detection of line pixels in perspective transformed binary image

To detect the lane lines on warped binary image I have defined following steps:

- **Find the left and right line search starting points.** This is achieved by take a non-zero pixel histogram of the bottom half of the binary image and find the peaks of the left and right halves of the histogram (see lines 7-13 in block **6. Detection of lane boundaries using Sliding Window search method** in pipleline.ipynb). Also below ere are visualized several examples of this step execution: (see visualization details in block **5. Detection of starting points of lane lines** in pipleline.ipynb)



- **Find the pixels of the left and right lines using Sliding Window search method.** Having the starting points for my search from previous step I define two sliding windows, placed around the these points, to find and follow the lines up to the top of the frame (see lines 15-74 of block **6 Detection of lane boundaries using Sliding Window search method** in pipleline.ipynb)
- **Fit second order polynomial along the discovered pixels of left and right line** which will allow to calculate the x coordinates of left and right lines for every y value from the frame vertical size (height) range (lines 76-78 of block **6 Detection of lane boundaries using Sliding Window search method** in pipleline.ipynb). Below are the visualizations of applying this search algorithm to several generated binary images (detected line pixels are displayed red and blue for left and right lines accordingly, sliding windows are displayed green, lines fit from calculated second order polynomials displayed yellow color):



7. Calculating a radius of ego lane curvature and vehicle position with respect to center of ego lane

In previous step I have calculated the second order polynomials for left and right lines that have following formula:

$$f(y) = Ay^2 + By + C$$

(the need to fit by y value instead of x value is caused by almost vertical orientation of detected lane lines)

In accordance to following reference the curvature of the fited lines can be then calculated by following formula:

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

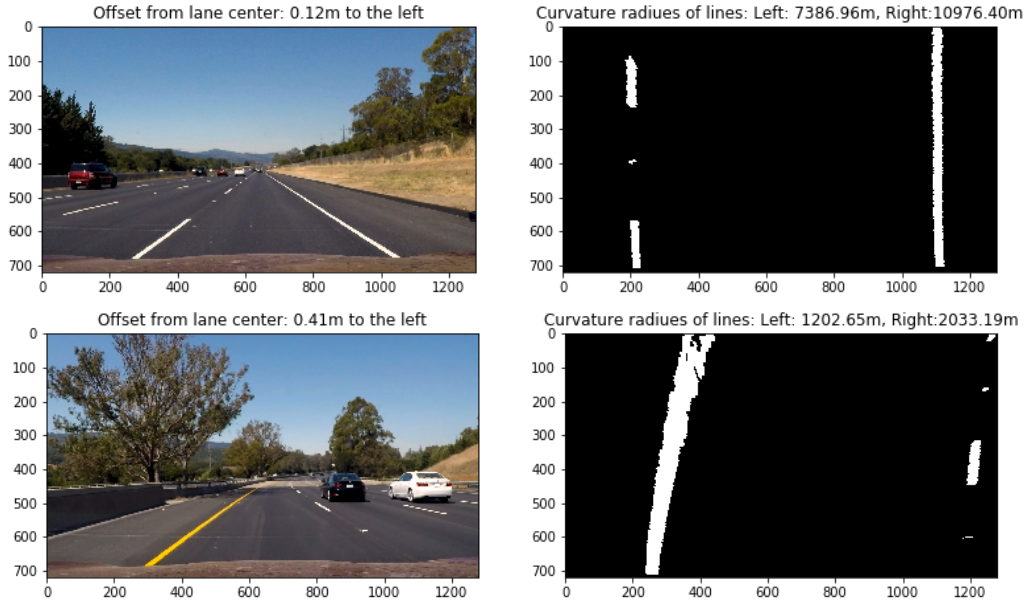
I have implemented this calculation in function *CalculateLineCurvature()* (see lines 5-14 of block **7 Calculation of the radius of lane curvature and vehicle position with respect to lane center** in pipleline.ipynb).

Calculation of offset of the vehicle from the center of ego lane is implemented in function CalculateVehicleOffset() (see lines 16-37 of block **7 Calculation of the radius of lane curvature and vehicle position with respect to lane center** in pipleline.ipynb). The idea behind is pretty simple: calculate the distance (in meters) from left image boundary to the center of the image. This will be considered as the vehicle center point. Then find the distance from the left image boundary to left and right detected lines (in meters) and calculate the middle point between the lines. This will be a center of detected ego lane. The offset of the vehicle from this center is the calculated as difference between vehicle center point and lane center point.

Both radius and offset value are returned in meters (conversion from pixels to meters is done inside mentioned functions using provided conversion rates:

```
ym_per_pix = 30/720  # meters per pixel in y dimension
xm_per_pix = 3.7/700  # meters per pixel in x dimension
```

Following are several results of calculation of curvature radius for the left and right detected lines and vehicle offset from center of the ego lane:



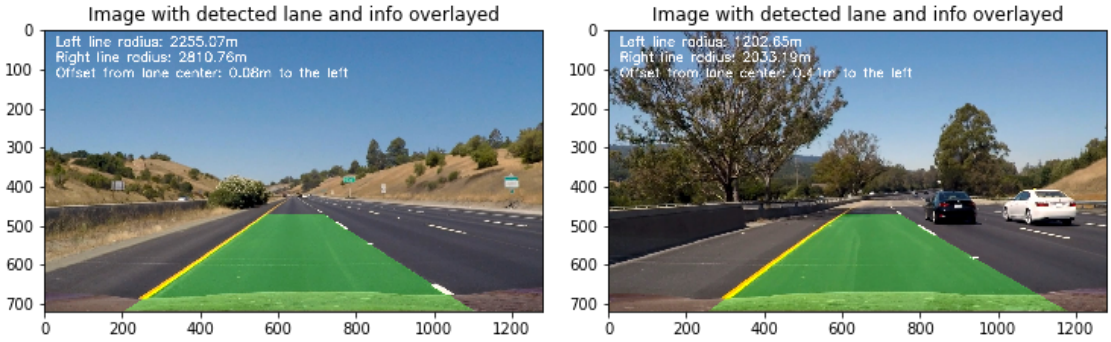
8. Overlaying of detected lane boundaries and lane information onto original images

To project a detected lane boundaries back on the original image I have implemented the function *OverlayDetectedLane() (see lines 1-25* of block **8 Projecting detected lane boundaries and lane info onto original images** in pipleline.ipynb*),* which consist of following steps:

- take the binary warped image and both left and right polynomials of detected lines and generate the black image of same size as binary image with color marked space limited by left and right lines using OpenCV function cv2.fillPoly()
- warp back the resulting image using Inverse transformation matrix (calculated on step *4. Perspective transformation of binary image)* and OpenCV function cv2.warpPerspective()
- overlay unwarped lane boundaries image onto original image using OpenCV function cv2.addWeighted()

To overlay the debug information about the line calculated in *step 7. Calculating a radius of ego lane curvature and vehicle position with respect to center of ego lane*, I have implemented the function *OverlayLaneInfo() (see lines 27-33* of block **8 Projecting detected lane boundaries and lane info onto original images** in pipleline.ipynb*),* which display all mentioned information in text form in upper left corner of original image using OpenCV function cv2.putText().

Results of applying these two functions to a couple of test images can be seen below:



9. Applying the lane detection pipeline to target video file

To detect the lane lines in provided for this project test video file **project_video.mp4** I have implemented the basic version of pipeline for detecting the ego lane in the single image. (see lines 1-25 of block **9. Basic pipeline for detecting ego lane in single image** in pipleline.ipynb) and used it as a lambda function for application to each frame of the video file (see lines 4-13 of block **10. Detecting lane boundaries in a video stream/file** in pipleline.ipynb). The video with visualization of the results of pipeline application to the video file **project_video.mp4** can be checked out in the file project_video_out_basic.mp4.

10. Optimization of detection pipeline

In order to improve the performance of the of the pipeline for use case when it is applied to a video stream, which is represented as a continuous sequence of very similar, slowly changing images, I decided to implement optimized version of detection pipeline (see function DetectLaneLinesOptimized() of block **11. Optimizing lane detection for sequences of similar images** in pipleline.ipynb). This version runs the full algorithms for calculation of polynomials of detected lanes based on Sliding Windows search (function DetectLaneLinesInFirstFrame(), block **6. Detection of lane boundaries using Sliding Window search method** of pipleline.ipynb) only in the beginning of the video, when there is no information about the location of the lines.
For all other consecutive frames of video file the pipeline runs simplified algorithms for calculation of polynomials of detected lanes (see function *DetectLaneLinesInNextFrame() in* block **11. Optimizing lane detection for sequences of similar images** in pipleline.ipynb). This algorithms consumes the line lane polynomials calculated for previous video frame for definition of the left and right 'corridors' of interest (where to look for the line pixels in current frame), since the line shapes in two consecutive video frames will differ very little (to take into account this difference a constant margin is added on the left and right side of previously detected coordinates of left and right lines).
The final result of applying implemented optimized pipeline to provided video file **project_video.mp4** can be checked out in the captured output video file project_video_out.mp4.

11. Conclusions

Implemented pipeline was quite successful in proper detection of the boundaries of ego lane in the provided video file **project_video.mp4** in all kinds of road conditions combinations (e.g. dark/bright surface, yellow/white lane markings, solid/dashed lines, sunny area/shadow). Both straight and curvy segments of the road could be handled properly by pipeline and overlaid lane boundaries were exactly matching the lane lines most of the time. Also the offset of the car from the lane center was visually always matching the corresponding calculated value

On other hand the pipeline had constant troubles with matching the calculated curvature radiuses of left and right lines. One the segments of the road, where the line marking is dashed, the curvature radius value were constantly changing with big amplitude and some calculated values were definitely far above the meaningful ranges. Also in one of the 'challenging' places (where the bright surface of the road with almost invisible yellow line switches to the dark surface)

the top part of left boundary of detected lane slightly warps and deviates from the real line of the lane for a fraction of second.

Taking into account above-mentioned identified issues, I would can point out following places for improvement:

- One of the first steps in this direction would be improving of calculation of the curvature radius value by using smoothing over several previously calculated values and also rejecting the calculated values which stand far beyond the standard deviation of previously calculated values
- Same ‘smoothing’ and ‘rejection of outliers’ mechanism may be also applied to the calculation of line polynomials to make the lane boundaries shape calculation more sturdy in ‘challenging’ road conditions