REPORT: PROJECT 4. ADVANCED LANE DETECTION

Author: Markus Buchholz

Project consist of following files:

1. Project4_SDCar_markus_buchholz. pdf => project report
2. Project4_markus_buchholz.ipynb => project code in jupyter
3. output videos:
    a. project_video_detected.mp4
    b. challenge_video_detected.mp4
    c. harder_challenge_video_detected.mp4

1. Introduction

This report discusses the approach of the lane detector based on computer vision (OpenCV). Performance of the lane detector was conducted sample image and 3 videos, which 2 of them are considered as a challenge. Beside the final test, the partial check was also performed (after each step of presented approach).
Lane detector was approached throughout the iteration design process securing (as an author goal) lanes detection on challenge video (mainly first).
To achieve sufficient lane detection on challenge video it was necessary to apply also sanity check and buffer (in order to average the estimated line polynomials). Sanity check and buffer are not necessary for lane detection on first video.

The discussed approach consists of following steps:

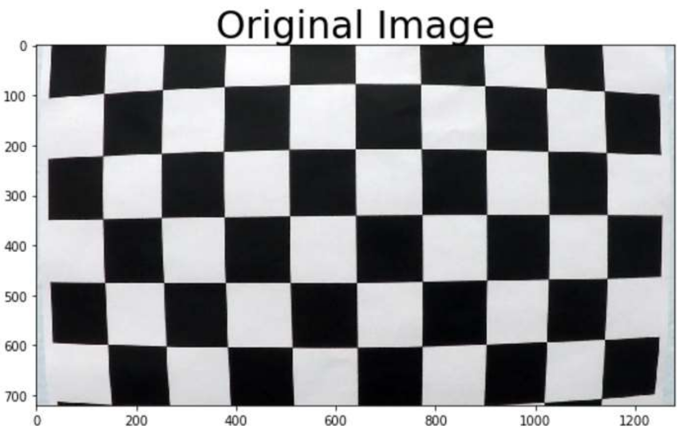
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- 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.
- Apply sanity check of estimated polynomials (left and right).
- Apply buffer in order to smooth over the last *n* frames of video and obtain a cleaner result.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

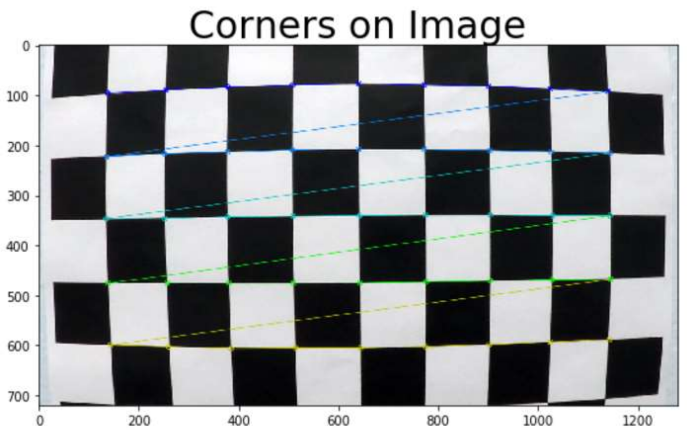
During the iteration design the recommended: look – ahead filter was also applied. Filter, it is still incorporated into the final code but flag: *loop_first = True* forces that the program bypasses this part of the lane detector code. The reason of not using this filter, was completely not satisfactory lane detection while applying detector on challenging video (almost no line were detected). Finally, the lane detector was verified on harder challenge video. In this case the test results can be classified as not satisfactory.

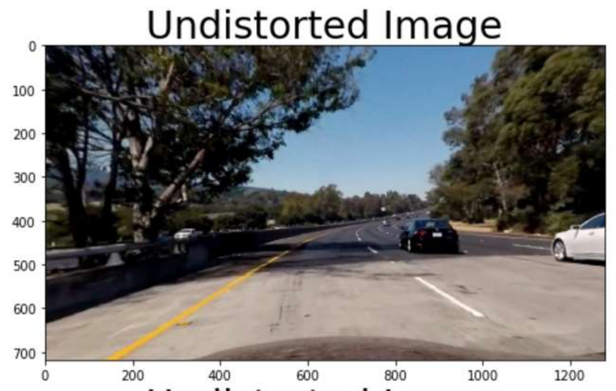
2. Camera calibration and distortion correction

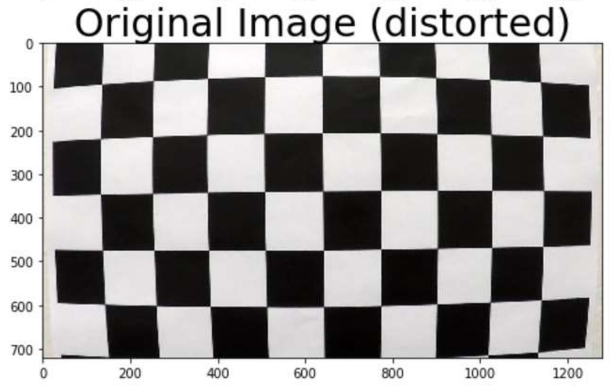
Each ordinary camera introduces distortion (not good enough camera lenses). Therefore, in order to receive correct measurements, before the image processing this error should be removed. At the beginning, however the calibration of the camera should be applied. Thank to carried out calibration camera process the image comparison is feasible.
In this project the printed 9x6 chessboard displayed with different angles was used (the camera calibration process was performed for 20 chessboard images). There were applied incorporated OpenCV functions to calculate the distortions introduced by the camera.
First it is created object array (image objects) which holds 3D coordination points for undistorted image (undistorted chessboard corners) and 2D array (image points) for distorted image which holds the position of detected corners.
Detection of corners (on distorted image which should be in gray scale) is performed by cv2. findChessboardCorners ().

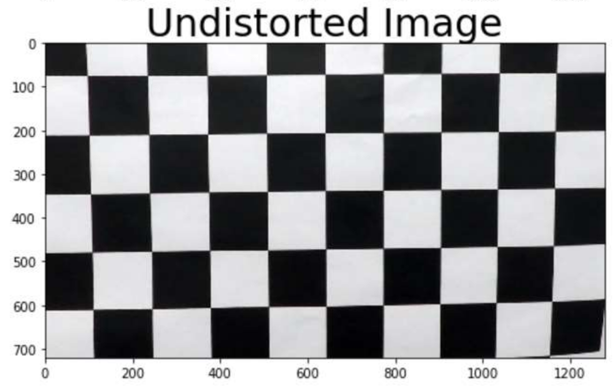
These steps are repeated for all delivered chessboard calibration (20) images. Having object points, image points and shape is it possible to calibrate the camera by applying cv2.calibrateCamera () function which returns distortion coefficient (dist), camera matrix (mtx) – which allows to transform 3D image objects to 2D image points. In order to undistort image (dst) is in necessary to apply the cv2.undistort(distorted_image, mtx, dist,…) function which takes as arguments: distorted image, dst and mtx.
Applying regarding steps for sample image it was possible to undistort sample image (calibration of camera is performed as a "background" process).







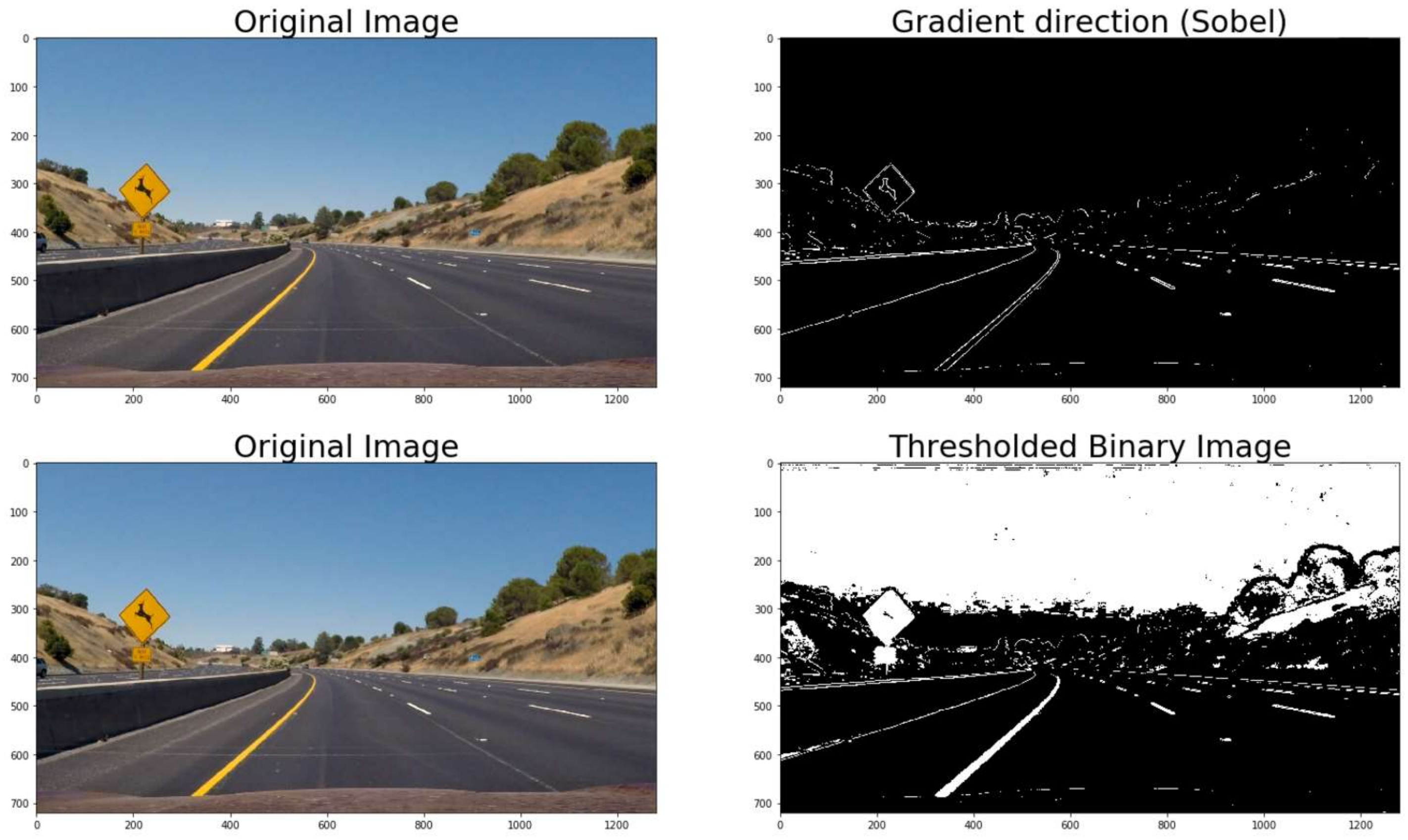




3. Thresholded binary image

In this project, removal of "noise" (unwanted information) from the image was performed by applying the combination of gradient direction threshold (Sobel operator) and color threshold. Threshold values for both techniques was chosen in the way of iteration approach process, bearing in mind the results presented during lectures.
Sobel operator (kernel =3) helps to detect a gradient of the image change so in the end it was possible to detect image edges. Project scope of work demands lane detection which are painted by light colors: white or yellow. Therefore, in

parallel to Sobel operator the color threshold (detecting these two “light” colors) was used. Here the conversion to HLS color space was applied in order to extract S channel.

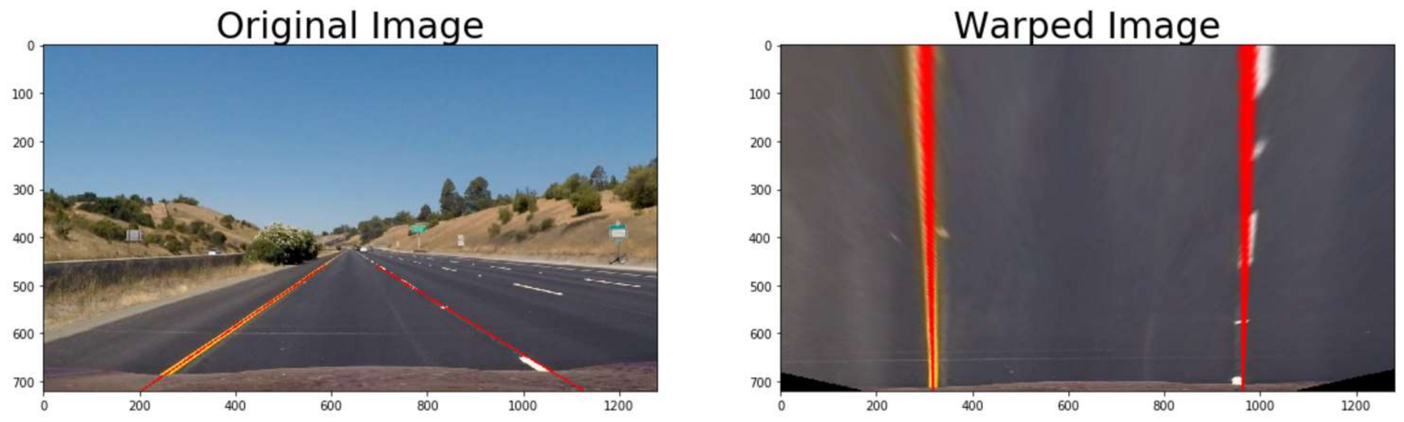


4. Perspective transform

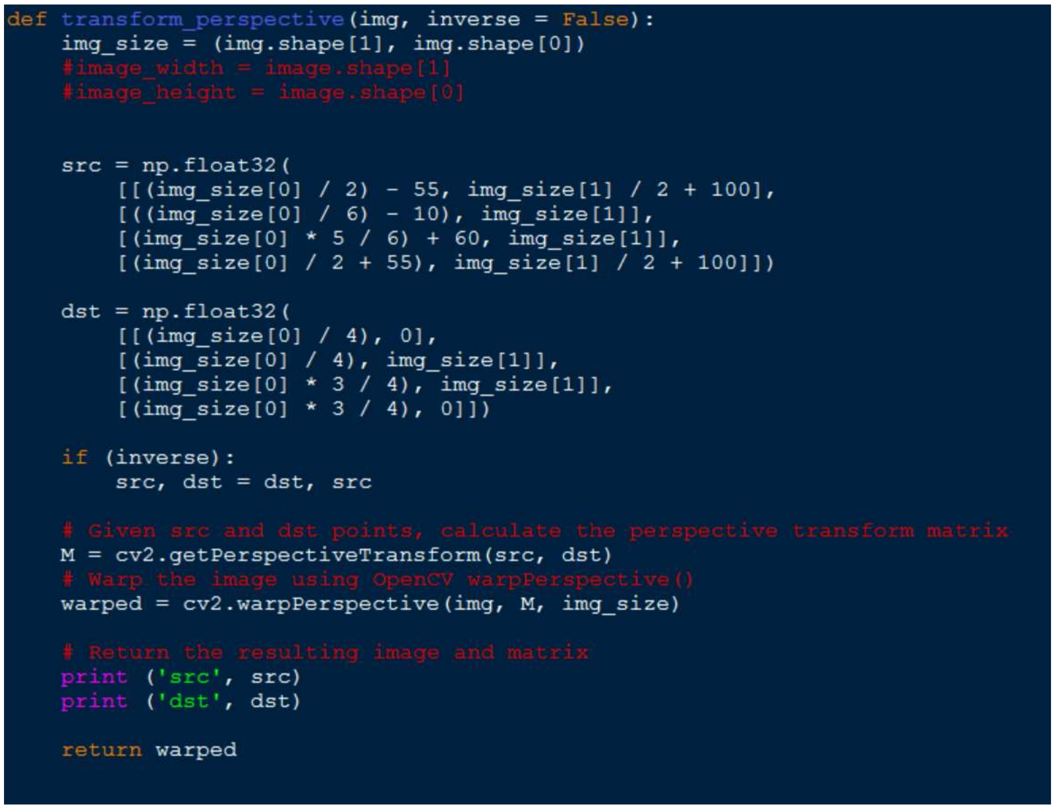
In order to boost detector lane performance it was necessity to focus only on important (and critical for the project goal) image part – here the road lines. For this reason, the image perspective transform was applied. In this case the transformation maps the given image (chosen points) to bird’s-eye view (desired). Applied transformation makes easier to estimate the line polynomials and compute their curvature.
In order to succeed with image transformation is necessary first to choose correctly the points on original image (sources) which should be seen (after transformation) from bird- view perspective (map source points are mapped to the destination points).
Here for the test image the source and destination point are given in the following table (x,y):

src	dst
585, 460	320, 0
203,720	320, 720
1127,720	960, 720
695, 460	960, 0

In order to apply correctly the transformation, the processed image should be undistorted (cv2.undistort()). Afterwards the transformation matrix M can be computed (cv2.getPerspectiveTransform(src, dst)). Having the transformation matrix M it is possible to transform (warp) each image later on (cv2.warpPerspective ()).
Sample result for described above transformation is depicted below.



Python function can be presented as follows:



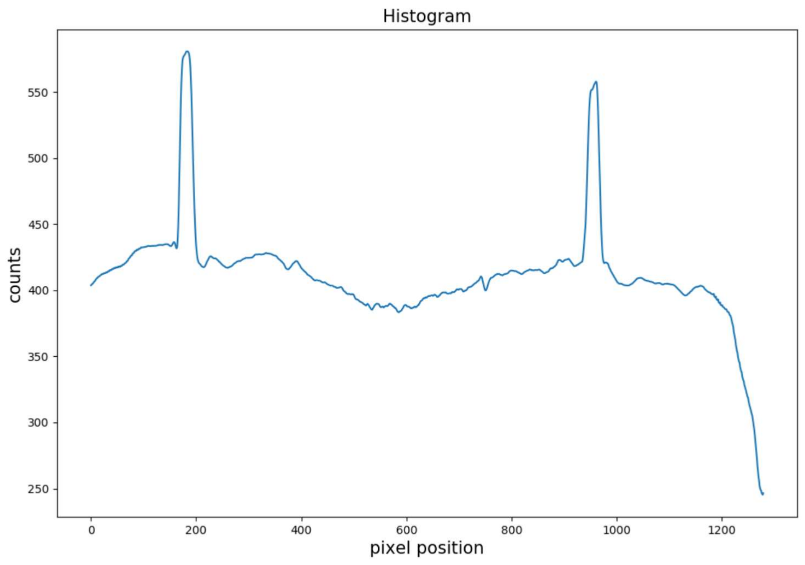
```python
def transform_perspective(img, inverse = False):
    img_size = (img.shape[1], img.shape[0])
    #image_width = image.shape[1]
    #image_height = image.shape[0]


    src = np.float32(
        [[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
        [((img_size[0] / 6) - 10), img_size[1]],
        [(img_size[0] * 5 / 6) + 60, img_size[1]],
        [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])

    dst = np.float32(
        [[(img_size[0] / 4), 0],
        [(img_size[0] / 4), img_size[1]],
        [(img_size[0] * 3 / 4), img_size[1]],
        [(img_size[0] * 3 / 4), 0]])

    if (inverse):
        src, dst = dst, src

    # Given src and dst points, calculate the perspective transform matrix
    M = cv2.getPerspectiveTransform(src, dst)
    # Warp the image using OpenCV warpPerspective()
    warped = cv2.warpPerspective(img, M, img_size)

    # Return the resulting image and matrix
    print ('src', src)
    print ('dst', dst)

    return warped
```

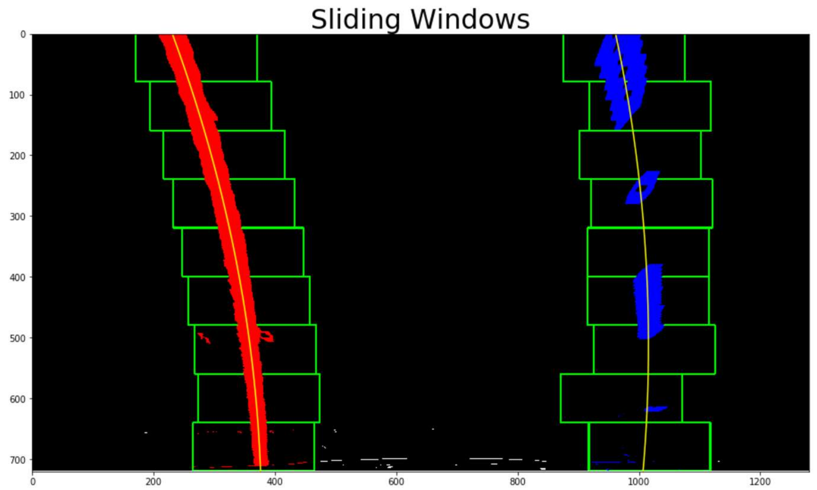
5. Identification of lane line pixels

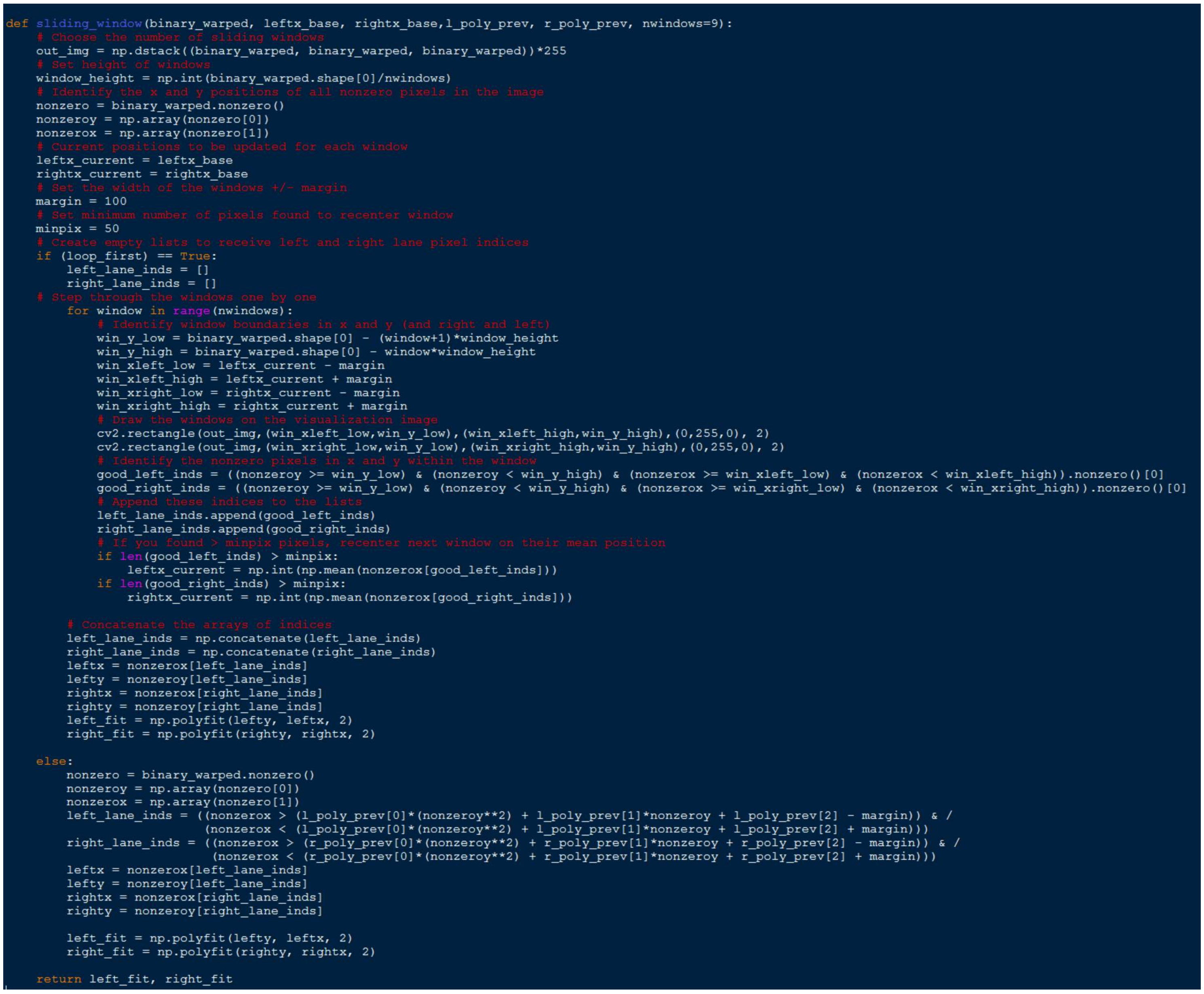
In this project road lines are approximated by second order polynomial: $x = A \cdot y^2 + B \cdot y + C$ (the coefficients A, B and C are unknow). Numpy function polyfit() based on given points x and y returns correct coefficient, which later on are used to draw the lines and compute their curvatures.

The image points fitted to the polyfit() function are extracted from binary wrapped image (perspective transformed image with lines founded by use of threshold). Extraction of pixels is perform first by finding the histogram of the image. Afterwards, the maximal values (peaks) which indicate the position of the lines (left and right) in image perspective are computed.

Further, by use of sliding windows technique it is possible to detect the nonzero image pixels – (pixels contributing in lines image are detected and selected by sliding windows – search is performed inside defined window boundary). Finally, all detected pixel (x,y) points are fitted to polyfit() function. The number of sliding windows was adjusted to 9 ,which cover the whole height of given image.







```python
def sliding_window(binary_warped, leftx_base, rightx_base,l_poly_prev, r_poly_prev, nwindows=9):
    # Choose the number of sliding windows
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
    # Set height of windows
    window_height = np.int(binary_warped.shape[0]/nwindows)
    # Identify the x and y positions of all nonzero pixels in the image
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
    # Current positions to be updated for each window
    leftx_current = leftx_base
    rightx_current = rightx_base
    # Set the width of the windows +/- margin
    margin = 100
    # Set minimum number of pixels found to recenter window
    minpix = 50
    # Create empty lists to receive left and right lane pixel indices
    if (loop_first) == True:
        left_lane_inds = []
        right_lane_inds = []
    # Step through the windows one by one
        for window in range(nwindows):
            # Identify window boundaries in x and y (and right and left)
            win_y_low = binary_warped.shape[0] - (window+1)*window_height
            win_y_high = binary_warped.shape[0] - window*window_height
            win_xleft_low = leftx_current - margin
            win_xleft_high = leftx_current + margin
            win_xright_low = rightx_current - margin
            win_xright_high = rightx_current + margin
            # Draw the windows on the visualization image
            cv2.rectangle(out_img,(win_xleft_low,win_y_low),(win_xleft_high,win_y_high),(0,255,0), 2)
            cv2.rectangle(out_img,(win_xright_low,win_y_low),(win_xright_high,win_y_high),(0,255,0), 2)
            # Identify the nonzero pixels in x and y within the window
            good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) & (nonzerox >= win_xleft_low) & (nonzerox < win_xleft_high)).nonzero()[0]
            good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) & (nonzerox >= win_xright_low) & (nonzerox < win_xright_high)).nonzero()[0]
            # Append these indices to the lists
            left_lane_inds.append(good_left_inds)
            right_lane_inds.append(good_right_inds)
            # If you found > minpix pixels, recenter next window on their mean position
            if len(good_left_inds) > minpix:
                leftx_current = np.int(np.mean(nonzerox[good_left_inds]))
            if len(good_right_inds) > minpix:
                rightx_current = np.int(np.mean(nonzerox[good_right_inds]))

        # Concatenate the arrays of indices
        left_lane_inds = np.concatenate(left_lane_inds)
        right_lane_inds = np.concatenate(right_lane_inds)
        leftx = nonzerox[left_lane_inds]
        lefty = nonzeroy[left_lane_inds]
        rightx = nonzerox[right_lane_inds]
        righty = nonzeroy[right_lane_inds]
        left_fit = np.polyfit(lefty, leftx, 2)
        right_fit = np.polyfit(righty, rightx, 2)

    else:
        nonzero = binary_warped.nonzero()
        nonzeroy = np.array(nonzero[0])
        nonzerox = np.array(nonzero[1])
        left_lane_inds = ((nonzerox > (l_poly_prev[0]*(nonzeroy**2) + l_poly_prev[1]*nonzeroy + l_poly_prev[2] - margin)) & /
                          (nonzerox < (l_poly_prev[0]*(nonzeroy**2) + l_poly_prev[1]*nonzeroy + l_poly_prev[2] + margin)))
        right_lane_inds = ((nonzerox > (r_poly_prev[0]*(nonzeroy**2) + r_poly_prev[1]*nonzeroy + r_poly_prev[2] - margin)) & /
                           (nonzerox < (r_poly_prev[0]*(nonzeroy**2) + r_poly_prev[1]*nonzeroy + r_poly_prev[2] + margin)))
        leftx = nonzerox[left_lane_inds]
        lefty = nonzeroy[left_lane_inds]
        rightx = nonzerox[right_lane_inds]
        righty = nonzeroy[right_lane_inds]

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

    return left_fit, right_fit
```

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

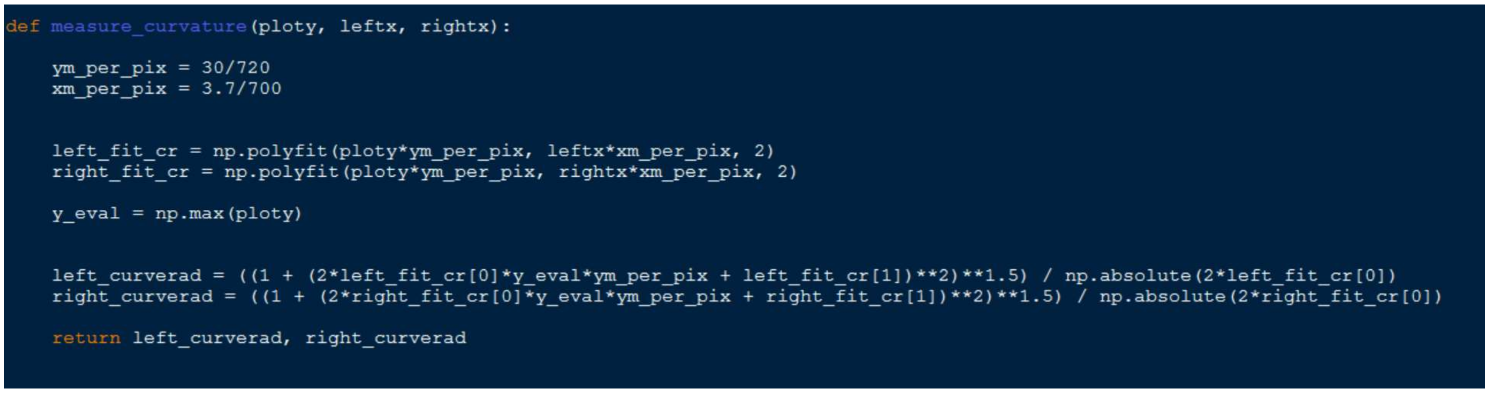
Polyfit() function applied in previous step returns A, B and C coefficient in second order polynomial (fitted to detected x, y line points). The curvature of applied polynomial can be described by following formula:

$$R_{curve} = \frac{[1 + (\frac{\partial x}{\partial y})^2]^{\frac{3}{2}}}{\left|\frac{\partial^2 x}{\partial y^2}\right|}$$

Computing first and second derivative of given polynomial $x = A \cdot y^2 + B \cdot y + C$ it is possible to compute curvature.

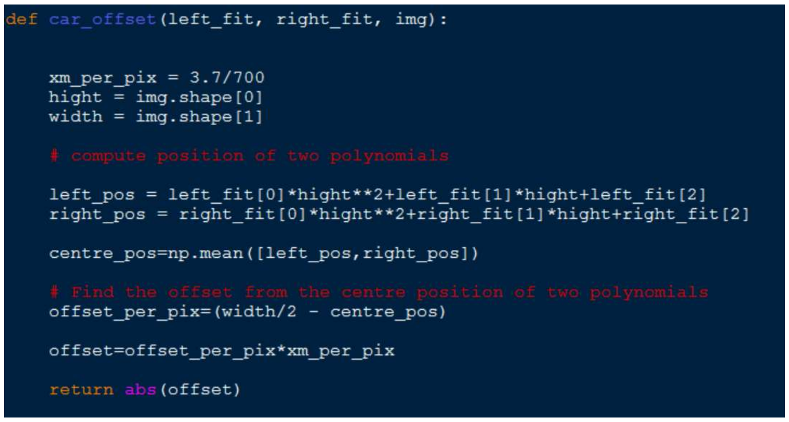
$$R_{curve} = \frac{[1 + (2Ay + B)^2]^{\frac{3}{2}}}{|2A|}$$

Above formula was implemented into Python code.



```python
def measure_curvature(ploty, leftx, rightx):

    ym_per_pix = 30/720
    xm_per_pix = 3.7/700


    left_fit_cr = np.polyfit(ploty*ym_per_pix, leftx*xm_per_pix, 2)
    right_fit_cr = np.polyfit(ploty*ym_per_pix, rightx*xm_per_pix, 2)

    y_eval = np.max(ploty)


    left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])
    right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])

    return left_curverad, right_curverad
```

Computation of car position (car offset) is performed by depicted below Python function. First the x points (for left and right polynomial) are calculated (for given y which equals height of the image). Taking average of this two values it is possible to compute exact position of the car (offset/difference between middle of the image and computed middle position of detected polynomials). In order to reflect image position with corresponding road “coordination system” it was necessary to scale last result.



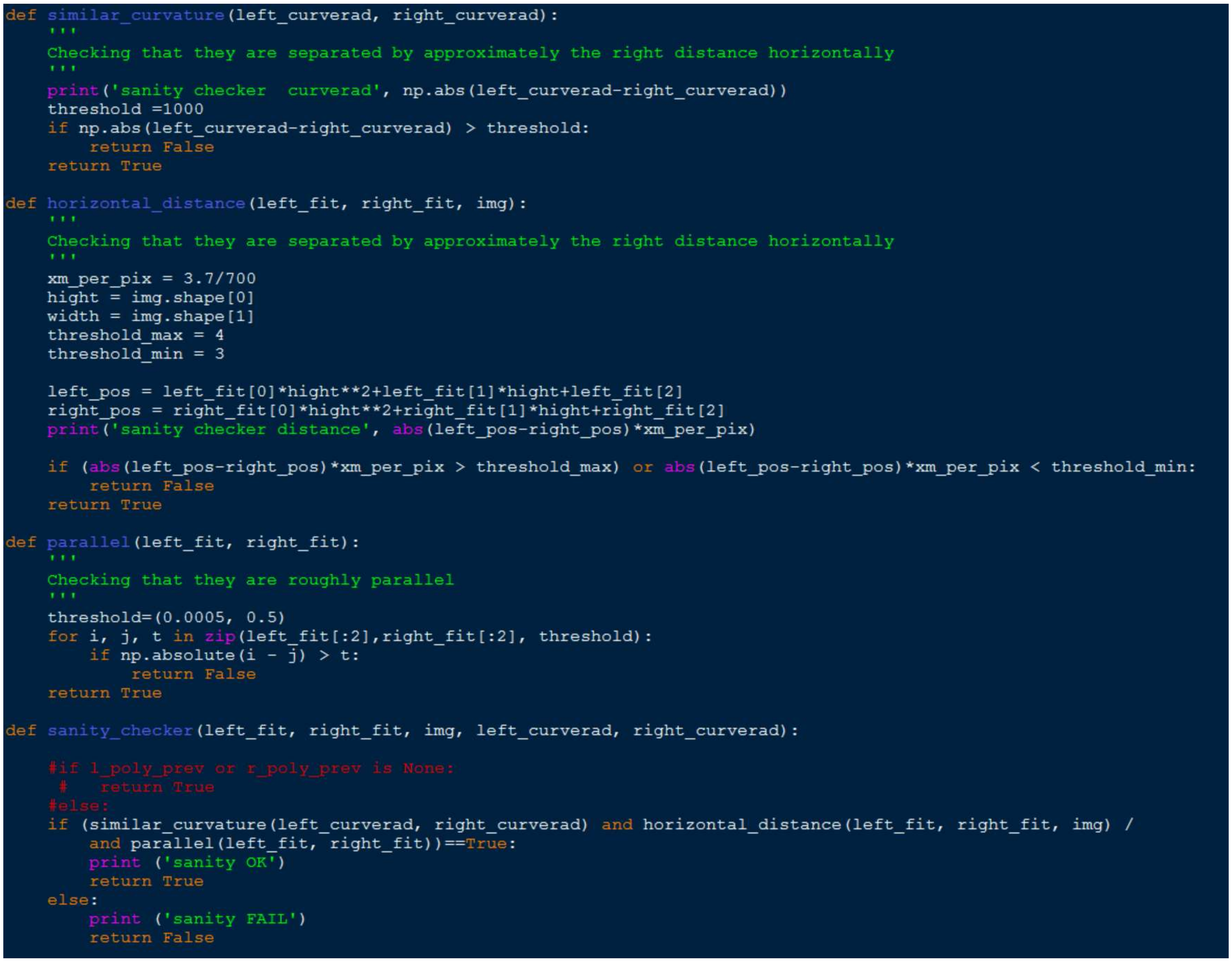
```python
def car_offset(left_fit, right_fit, img):


    xm_per_pix = 3.7/700
    hight = img.shape[0]
    width = img.shape[1]

    # compute position of two polynomials

    left_pos = left_fit[0]*hight**2+left_fit[1]*hight+left_fit[2]
    right_pos = right_fit[0]*hight**2+right_fit[1]*hight+right_fit[2]

    centre_pos=np.mean([left_pos,right_pos])

    # Find the offset from the centre position of two polynomials
    offset_per_pix=(width/2 - centre_pos)

    offset=offset_per_pix*xm_per_pix

    return abs(offset)
```

7. Sanity check

In this project the best Udacity practices (recommendations) were used. At this stage, the detected lines (before drawing them on the processed image) are checked if they reflect real road lines. In each step of image processing, following features of the lines are checked:

1. Checking that they have similar curvature
2. Checking that they are separated by approximately the right distance horizontally
3. Checking that they are roughly parallel

Parameters (threshold) to be verified by sanity checker, were estimated by running video and analyzing the measurements image by image. Deviation was noted and taken into consideration for setting correct threshold value.
Sanity check function is called from pipeline function and can be represented as follows:



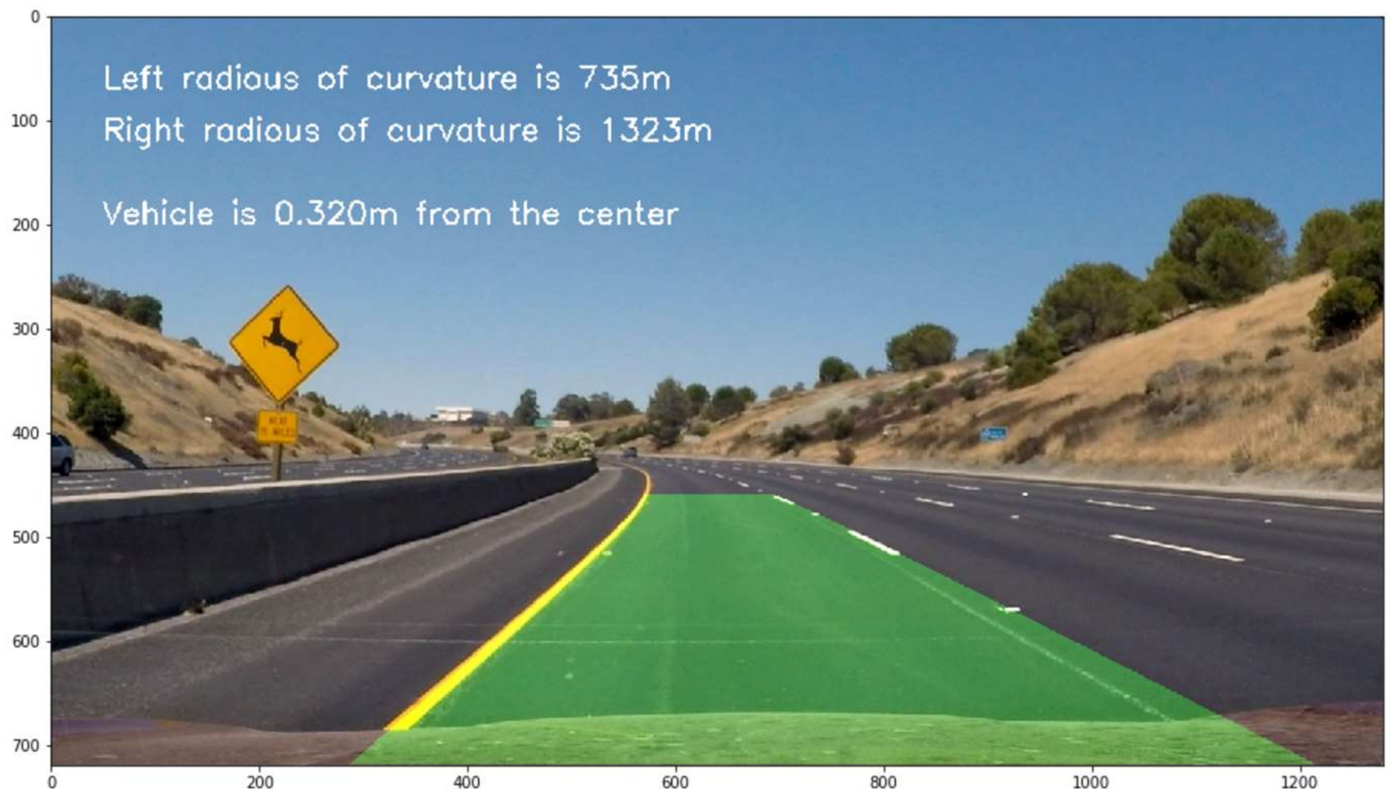
```python
def similar_curvature(left_curverad, right_curverad):
    '''
    Checking that they are separated by approximately the right distance horizontally
    '''
    print('sanity checker  curverad', np.abs(left_curverad-right_curverad))
    threshold =1000
    if np.abs(left_curverad-right_curverad) > threshold:
        return False
    return True

def horizontal_distance(left_fit, right_fit, img):
    '''
    Checking that they are separated by approximately the right distance horizontally
    '''
    xm_per_pix = 3.7/700
    hight = img.shape[0]
    width = img.shape[1]
    threshold_max = 4
    threshold_min = 3

    left_pos = left_fit[0]*hight**2+left_fit[1]*hight+left_fit[2]
    right_pos = right_fit[0]*hight**2+right_fit[1]*hight+right_fit[2]
    print('sanity checker distance', abs(left_pos-right_pos)*xm_per_pix)

    if (abs(left_pos-right_pos)*xm_per_pix > threshold_max) or abs(left_pos-right_pos)*xm_per_pix < threshold_min:
        return False
    return True

def parallel(left_fit, right_fit):
    '''
    Checking that they are roughly parallel
    '''
    threshold=(0.0005, 0.5)
    for i, j, t in zip(left_fit[:2],right_fit[:2], threshold):
        if np.absolute(i - j) > t:
            return False
    return True

def sanity_checker(left_fit, right_fit, img, left_curverad, right_curverad):

    #if l_poly_prev or r_poly_prev is None:
     #   return True
    #else:
    if (similar_curvature(left_curverad, right_curverad) and horizontal_distance(left_fit, right_fit, img) /
        and parallel(left_fit, right_fit))==True:
        print ('sanity OK')
        return True
    else:
        print ('sanity FAIL')
        return False
```

8. Smoothing

In order to stabilize the line image the buffer was applied. Buffer holds the samples/values of coefficient of fitted polynomials (in a way of verification it was adjusted to 3 samples) and releases the average value of final polynomial coefficients (polynomial coefficients are computed as an averge of 3 buffer samples). There are two buffers, one for left polynomial and one for the right.

9. Image results

Below image depicts result of implemented line detection. Curvature of detected lines, together with position of the car are given as imprinted text on the sample image.


Left radious of curvature is 735m
Right radious of curvature is 1323m
Vehicle is 0.320m from the center
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

10. Test of lane detector on videos

Designed lane detector was verified on videos. Detection of road lines on first delivered by Udacity test video is correctly and as expected. Some small deviations (when the rapid brightness) or line sharpness is changing) can be seen. Enlarging of buffer size make the line “stiff” – the detected line are longer frozen on given image. Lane detection on challenge video (bearing in mind the sudden changes of line size , sharpness, extra distortion, etc.) is quiet correct (lower performance then on first video). Detector is rather insufficient on last video (harder challenge video).

11. Discussion

The most challenging problem occurred and faced in this project was connected with adjusting correct thresholds in order to estimate correctly polynomials. Secondly, the problem with buffer implementation was also encountered (at the early stage of project the average of the buffer value was taken at each step the new sample/image came. It was completely wrongly approach). Additionally, the Look – Ahead Filter made insufficient lane detection for challenging videos, therefore it is not used (but it can be switch ON).
Designed lane detector is a good approach for next project. In order to make it robust the “intelligence” should be deployed (thresholds should not be “stiff” and other parameters should be adaptable to changing road conditions). In this project, the detector was mainly design for specific road conditions, making him insufficient for new data set. It can be also considered to implement controller like (PID or feed forward) to compensate overshoots - instead of implemented buffer.