Advanced Lane Finding – Project P4

The goals / steps of this project are:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboardimages.
- 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 centre.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature an vehicle position.

Rubric Points

In the following I will describe how I approached the different Rubic points

1. Camera Calibration

The camera calibration is required since the camera transforms a 3D image into 2D images. This transformation can create distortion i.e. pictures show different sizes or shapes as in reality. Distortion can be corrected by calibrating the camera on known distorted images: 2D image points (x,y) of a distorted image are calibrated with the correct 3D points (x',y',z') and can be corrected by a distortion matrix therefore.
In this project chessboard corners (9x6) are used for distortion correction and the python openCV functions: *findChessboardCorners(), cv2.calibrateCamera(), drawChessboardCorners(), cv2.calibrateCamera(), cv2.undistort().*
For the calibration I ran the following steps using the 20 test images of the ***camera_cal*** directory (code cells 2,3, 15,16 of jupyter notebook):

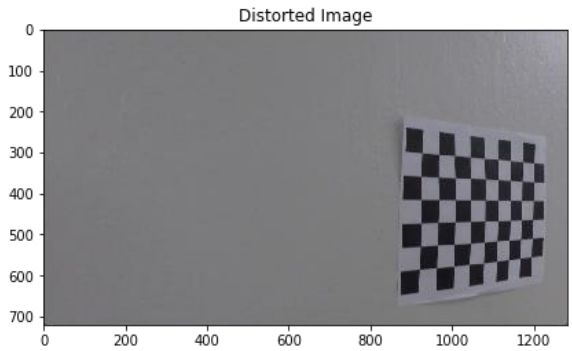
1) Build of the reference list ***ref_ob*** which contains the corner coordinates for a chessboard 9x6 (line). Note: the z coordinate is set to 0.
2) Read in the test image and convert it to grayscale (for easier corner detection)
3) Run *cv2.findChessboardCorners()* on the test image to find the chessboard corners. If found, two list are updated/appended: **obj_l** with the correct corner coordinates by copying ***ref_ob***, and ***img_l*** with the recognized corner coordinates of the function.
   (code cell 2 and 15 of jupyter notebook)

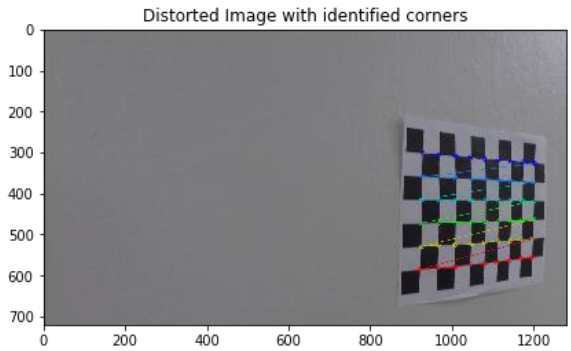
```
obj_l, img_l = runCalibration ("./camera_cal/*.jpg",9,6,2)
```

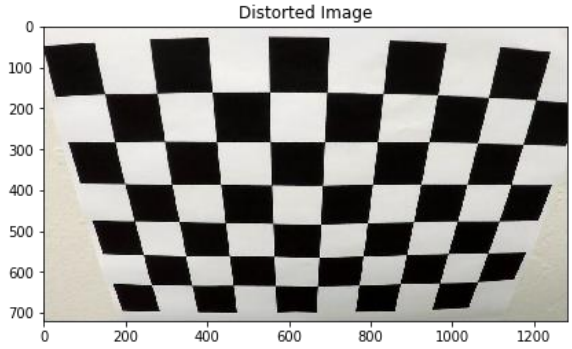
4) Run *cv2.calibrateCamera()* on the ***obj_l*** and ***img_l*** lists to generate the distortion matrixes.
5) Run *cv2.undistort()* on the distortion matrixes and the distorted image to get the undistorted image
   (code cell 3 and 16 of jupyter notebook).

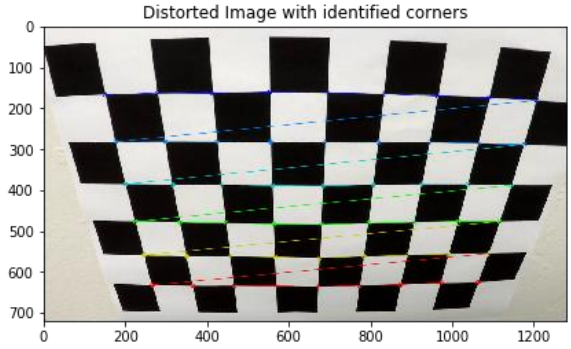
```
Image_und    = runUndistort("./test_images/test1.jpg",obj_l,img_l, 1)
```

The figures below present search of chessboard corner for distorted images.

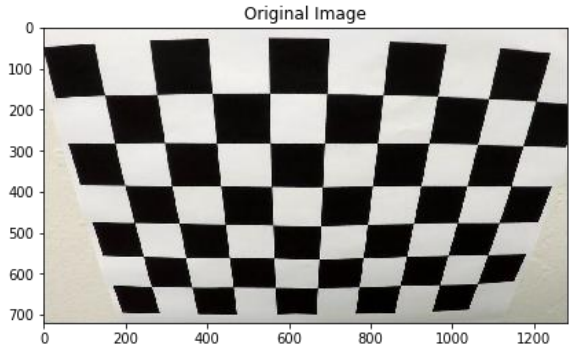


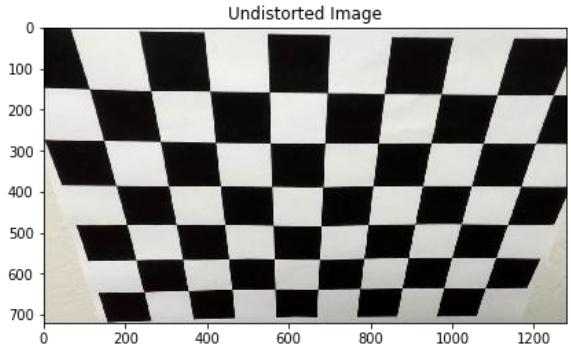




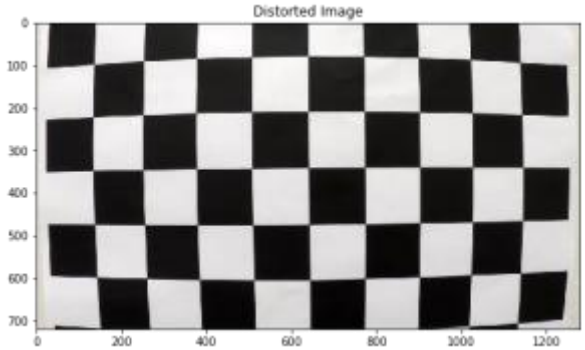


The image below present an undistorted chessboard image.





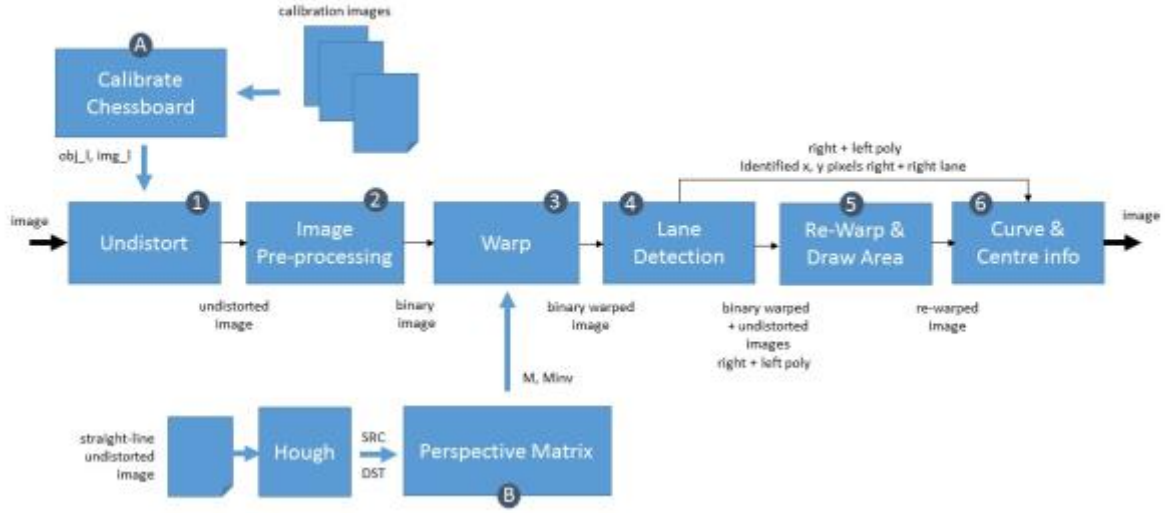
Note: Not for all 20 undistorted images the chessboard corners have been identified. Only 17.
Rationale: 3 test images (1,4,5) contain less than 9x6 chessboard corners, see image below



2. Pipeline

The figure below presents the pipeline I used for Lane detection (code cell 13 of jupyter notebook). I will discuss the different components in the following:

A) Calibrate Chessboard: generate the object (ref. corner) and image (distorted corners) list for calibrated chessboard corners
1) Undistort: Undistort an images using the object and image list from A)
2) Image Pre-processing: applies color conversion and Sobel operations on the image
B) Perspective Matrix: provide the Matrix & inverse for perspective transformation. The source (SRC) and destination (DST) coordinates are generated by applying a Hough transformation on an undistorted image which represents a straight line.
3) Warp: warps a binary image from 2) using perceptive transformation (bird view)
4) Lane Detection: runs lane detection on the warped image
5) Re-Warp & Draw Area: Re-warped identified lanes and draws it into the original image
6) Curve & Centre Info: calculates curve and vehicle offset and adds info into image



Note: Code cell 21 of the jupyter notebook contains all pipeline images for the test images

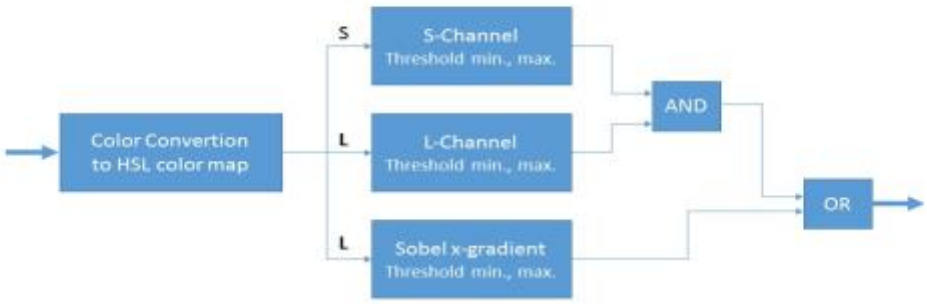
2.1 Undistort (Distortion Example)

Images are undistorted by the *runUndistort()* function which reads in the distorted image and the chessboard corner coordinates lists***: obj_l*** and ***img_l*** . The function applies distortion correction functions as described in chapter 1.

Pipeline (code cell 13, 15 of jupyter notebook)

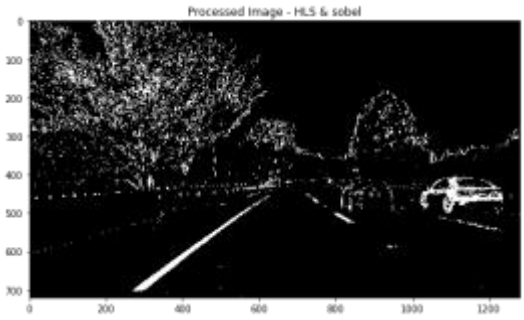
```
# 1. undistort image
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(obj_l, img_l,
                                   (image.shape[1], image.shape[0]),None,None)
 image_und = cv2.undistort(image, mtx, dist, None, mtx)
```

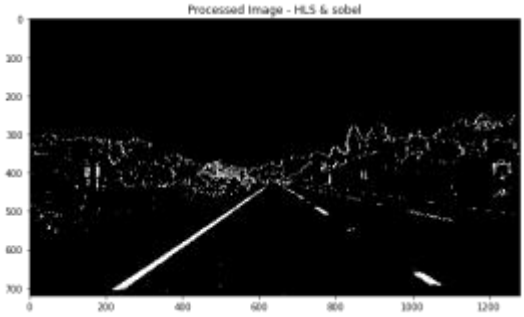
2.2 Image Pre-Processing (*combinedAdvanced2()* func. code cells 6 and 4, 5 of jupyter notebook)
For image pre-processing I used the Sobel x-gradient, L-channels and S-channels form a converted RGB to HSL image. The figure below presents how these elements are combined.



- The first image pre-processing step is a color conversion from RGB to HSL color map.
- On the L-channel there are applied:
  - Sobel x-gradient focusing to identify the very light white lines and edges thresholds are: minimum = 35, maximum = 255
  - a color transformation in order to remove unwanted object covered by shadows thresholds are: : minimum = 40, maximum = 255
- On the S-channel there is applied:
  - a color transformation in order to identify yellow and white lines thresholds are: minimum = 155, maximum = 255

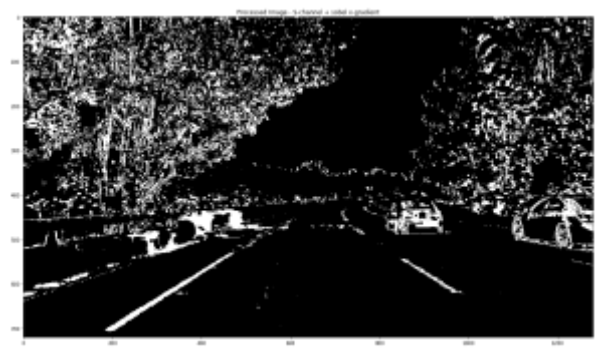
The images below present the image pre-processing of a test images (code cell 16,17 of jupyter notebook).

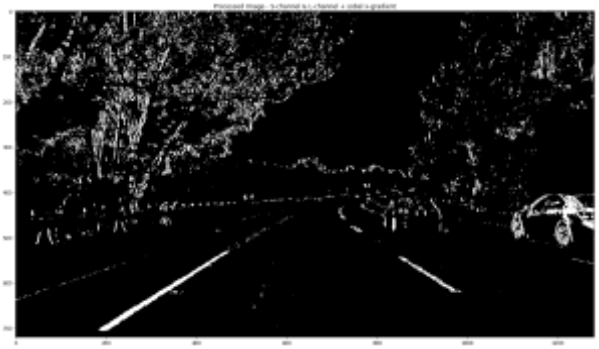




L-channel (code cell 18 of jupyter notebook)
I would like to emphasis the bit-wise AND combination of the S-channel and L-channel were removing unwanted object inside the shadow. The following pre-processing result with and without the L-channel bit-wise AND on an image from the project video

Image pre-processing without L-channel AND S-channel AND combination: left picture
Image pre-processing with L-channel AND S-channel AND combination: right picture - Final solution





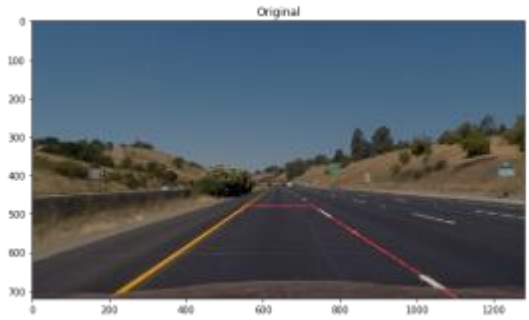
2.3 Perspective Transformation

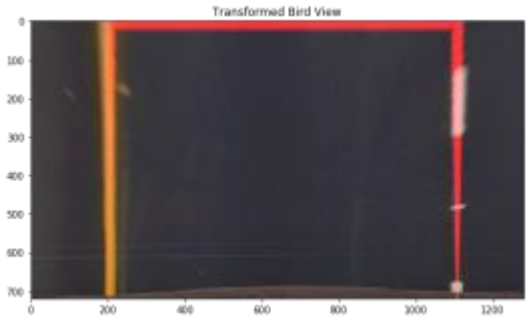
In order to get the source points for the perspective transformation, I ran a Hough transformation, as used in the project P1, on the undistorted and pre-processed image *straight_line1.jpg* (code cell 8, 19 of jupyter notebook). I received the following coordinates for the image below and defined the destination points by setting the lower y value to 0.

```
NOTE: SRC coordinates=
[[  204.   720.]
 [  556.   480.]
 [  730.   480.]
 [ 1105.   720.]]
NOTE: DST coordinates=
[[  204.   720.]
 [  204.     0.]
 [ 1105.     0.]
 [ 1105.   720.]]
NOTE: lane width [pixels] = 901
```

Then I generated the transformation matrix M & its inverse using the *cv2.getPerspectiveTransform()* on the source and destinations points. Then I ran the *cv2.warpPerspective()* on the undistorted image and verified visually the transformation result by checking that lines are parallel to the y axis (code cell 8 of jupyter notebook).

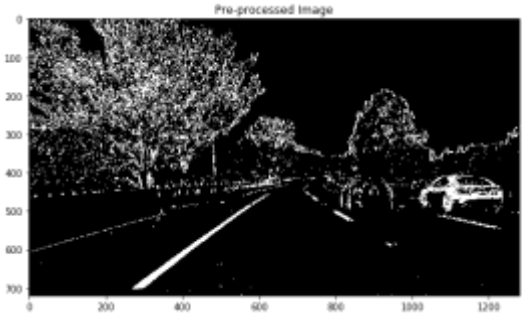
```
# Matrix
M    = cv2.getPerspectiveTransform(src, dst)
Minv = cv2.getPerspectiveTransform(dst, src)

# Warp the image using OpenCV warpPerspective()
warped = cv2.warpPerspective(rst, M, (rst.shape[1],rst.shape[0]))
```

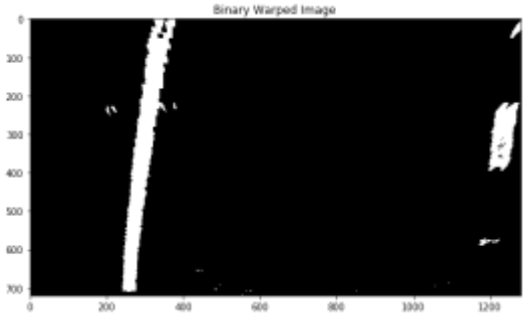




2.4 Lane line Identification (centre code of *pipeLine()* func, code cell 13,12, 11 of jupyter notebook)
I used for the lane identification a warped binary as input. The warped binary was generated by running the image pre-processing step as described before. Then I applied two lane identification methods as described in the udacity lessons 33,34 ,35. Both methods use the 2nd polynomial for y-values for the right and left lane separately.

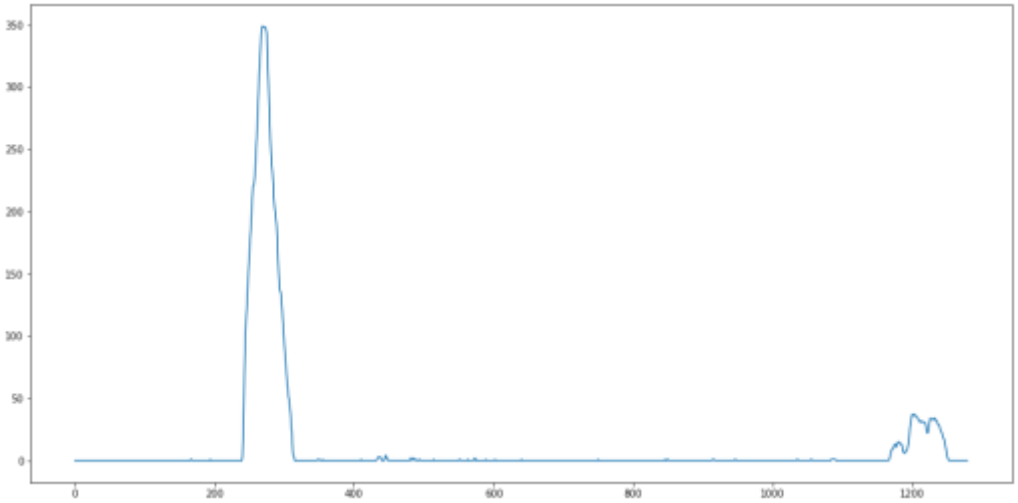
f(y) = A * y **2 + B * y + C





1. Sliding Approach - Blind search (func. *detectLaneSlide(),* code cell 10 of juypter notebook)

The first step of the sliding approach is to identify pixels'density peaks for the left and right side of the image in order to get a starting points for the left and right lane detection. The algorithm uses a histogram approach on the lower half of the image for this. The peak in the left hand-side is used for the left lane and the one in right hand-side for the right lane.

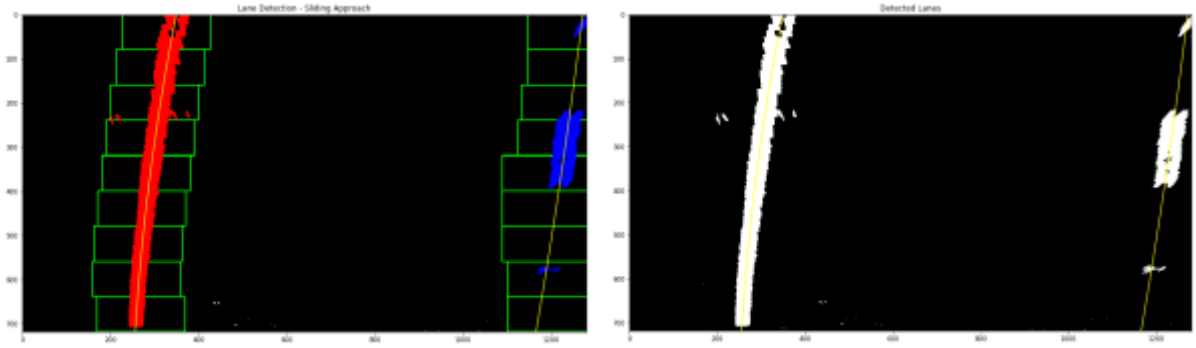


The window sliding starts inside the middle of image. It goes in up and down direction unit it reaches the end of the picture (sliding window height is multiple of image height). At each sliding step the identified pixels of window are saved and analysed:

- The position for the next window is calculated by meaning all identified x-pixels of the current window.
- The curve for the left and right lane is calculated after the sliding process by running the 2nd polynomial on the identified pixels
  (line func detectLaneSlide code section x of the

```
# Fit a second order polynomial to each
 left_fit = np.polyfit(lefty, leftx, 2)
 right_fit = np.polyfit(righty, rightx, 2)
```

The figures below present on the left hand-side, the sliding window (green) and the identified pixels for the left (red) and right lane (blue). On the right hand-side the detected curves of the input image.



2. Reusing previous Lane Positions ((func. *detectLaneReuse(),* code cell 10 of juypter notebook)

This method uses the 2nd polynomial of the left and right lane for the previous frame. It searches for
pixels around this polynomial according to a certain width (margin). After image was analysed the algorithm calculates the new curves for the left and right lane by running 2nd polynomial on the identified pixels (same way as for sliding approach)

2.5. Radius of curvature of the lane and the position of the vehicle with respect to Centre.

Vehicle Position

The function *centreOffset_m()* (code cell 12 of the jupyter notebook) calculates the vehicle position versus the centre in meter.
The function uses the2nd polynomial of the identified left and right lane curve. I calculates the position by subtracting the half image width from the calculated half lane width using y = max for the2nd polynomial and multiplying the result by the meter versus pixel correction factor (x: 3.7m / 900 pixels), see also python code below.

```
def centerOffset_m (left_fit, right_fit, width=1280, height=720, lane_mp=3.7/900)

    xl =  left_fit[0]*height**2 +  left_fit[1]*height + left_fit[2]
    xr = right_fit[0]*height**2 + right_fit[1]*height + right_fit[2]

    lane_center = (xr+xl)/2
    pic_center  =  width/2
    offset      =  lane_center - pic_center

    return offset * lane_mp
```

Curvature Radius

The radius for the left and right lane is calculated in the lane detections functions *laneDetectSlide()* and *laneDetectReuse()* (see code cell 11 of jupyter notebook) reusing the identified pixels of the detection process. The calculation process is done in the following way:

1. Re-calculate the the2nd polynomial for meter by multiplying the identified pixel with meter / pixels coefficients (y: 30 m / 720 pixels, x : 3.7 / 900 pixels)
2. Calculate the radius for the left and right lane by entering the y = max value in the new

the 2nd polynomial (step 1.)

The displayed curvature radius of the result image is calculated based on the left and right curve radius using the mean value between both. See also code of python pipeline function (code at the botton of *pipeLine()* func., code cell 13 of jupyter notebook)

```
# add curve + offset info
   av = np.copy(np.abs(curve_right_ + curve_left_))
   radius = int(av / 2)
```

6. Example image of your result plotted back down onto the road.

I implemented this step by:

- the function *drawLaneArea()* (code cell 12 of jupyter nooteboo) which draws the identified area back to the image, re-warping the identified lanes from the previous step 5. Lane Detection.
- and by using the *cv2.putText()* function (bottom code of pipeLine() func., code cell 13 of jupyter notebook) to annotate the curvature and offset value to the result image.

The images below present the input image for the pipeline and the plotted back one (code cell 21 of the jupyter notebook contains all plotted back test images)

3. Video

The processed video is provided with the GitHub.

4. Discussion

My main issue has been the Image pre-processing step , which was pretty difficult in order to cover all light scenarios, especially dark shadows or light grey roads with tire traces and light white line marks. The provided test image did not cover all situations therefore I used video images for tuning. The pipeline may fail to recognize lanes if the line marks are very light on a dirty road.
I think I can improve my pre-processing further and lane detection algorithm
May lane detection can be improved if the image pre-processing is dynamically adapted on weather conditions i.e. by a light sensor