Advanced Line Project Writeup

Rubric Points : Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Notes :

- All my code is gathered in one unique jupyter notebook file, named "P2_vXX.ipynb" (XX being the version number of this file)
- Output images based on image examples from folder `./test_images` generated by my pipeline are in folder `./output_images` , in sub-folders for any specific processing tasks.
- Output videos are in folder `./output_videos` .

Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

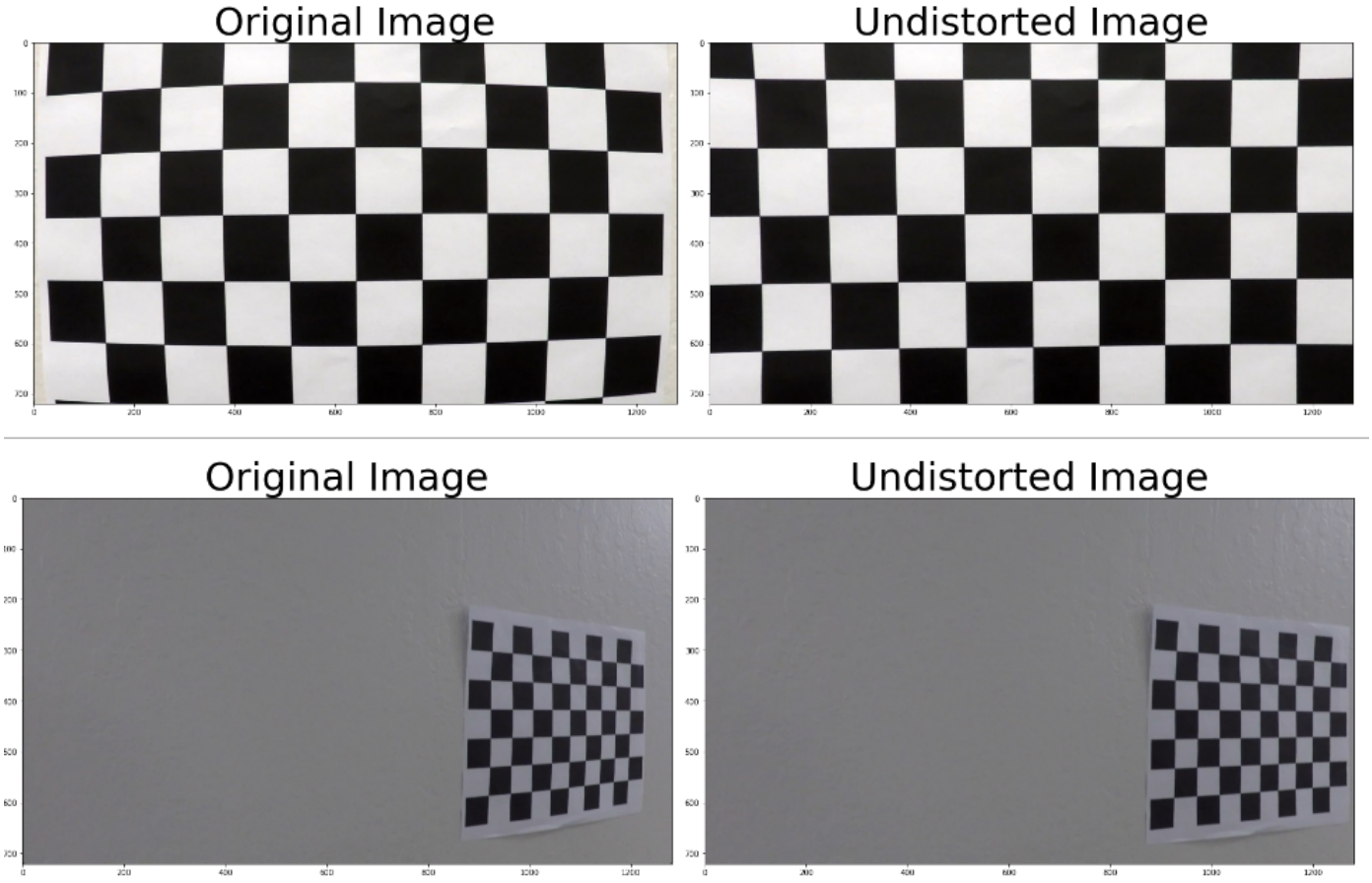
- The code for this step can be found in "P2_vXX.ipynb" : in paragraph titled **"Compute the camera calibration matrix and distortion coefficients given a set of chessboard images"**.
- It is divided in 2 parts :
  - Chessboard corner detection
  - Camera Calibration
- For Camera Calibration, we define 2 arrays of points, `objpoints[]` for 3d points in real world space, and `imgpoints[]` for 2d points in image plane.
- We have several chessboard images 'camera_cal/calibration*.jpg' taken from different directions, and we'll process them by doing the following : For each chessboard image :
  - Read it.
  - Using `cv2.findChessboardCorners()` , we'll find coordinates of chessboard corners.
    - Note : we can visualize identified corners overlayed on source image using openCV function `cv2.drawChessboardCorners(img, (9,6), corners, ret)`
  - Store and append the corners coordinates into array `imgpoints[]` (2D projection of the chessboard image).
  - Define ourselves coordinates of the chessboard corners as it should appear on a non distorted grid image, using `numpy.mgrid()` function. Once done, append them to `objpoints[]` .

After processing all those camera calibration chessboard images, we get arrays `imgpoints[]` and `objpoints[]` with all the real corners coordinates of the chessboards and their corresponding expected coordinates in a non distorted grid image of the chessboard.

With that :

- We apply openCV function `cv2.calibrateCamera(objpoints,imgpoints, (xSize,ySize),None,None)` to get the distortion matrix and coefficients which will be used to undistort any images taken with this particular camera. Cf notebook chapter 'Camera Calibration'

- Use distortion matrix and coefficients to distort any image taken from this camera (cf `cv2.undistort(img, mtx, dist, None, mtx)`)´
- cf examples to illustrate below, on front facing chessboard image and another not front facing chessboard image.



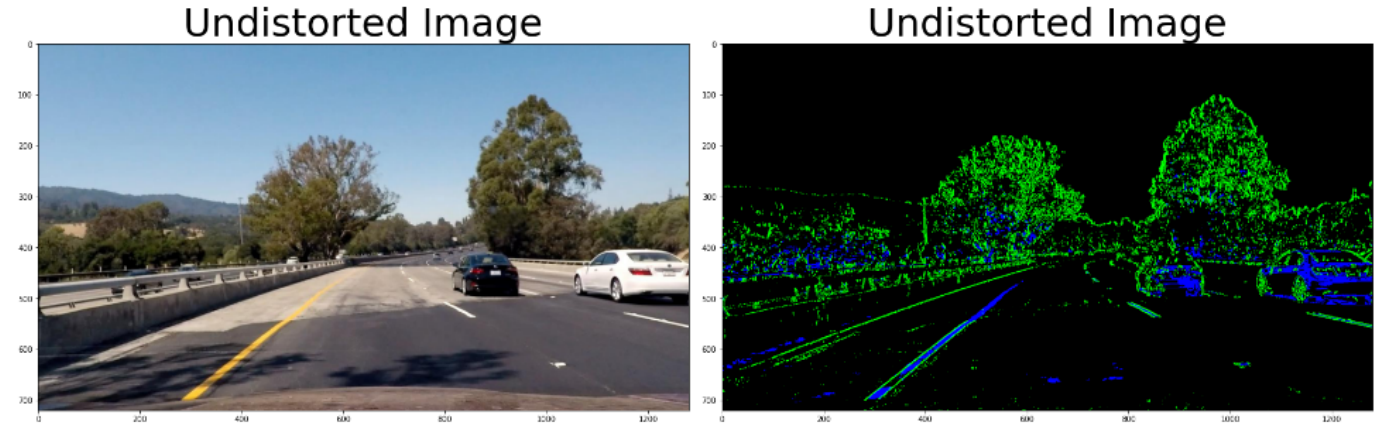
Pipeline (single images)

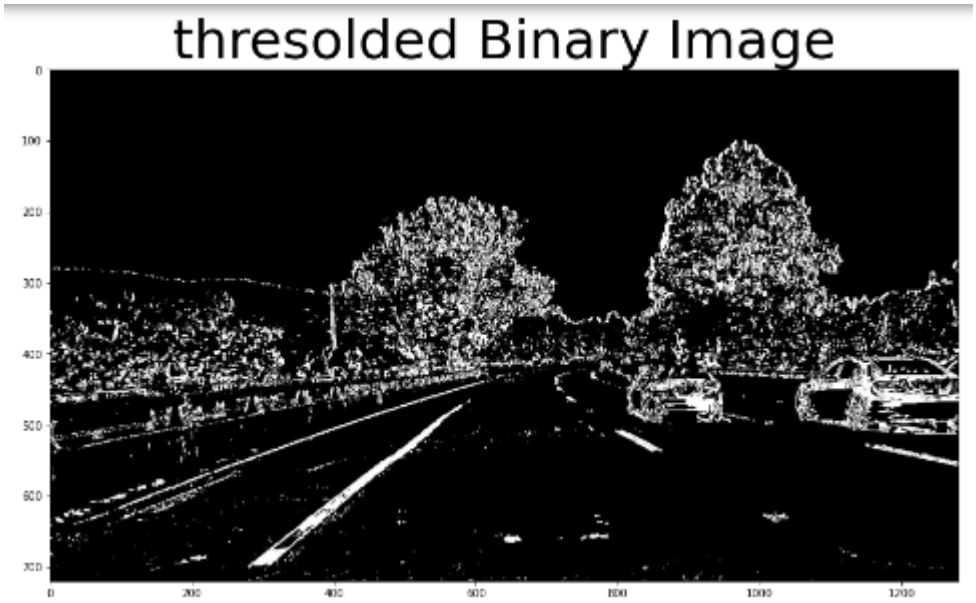
1. Provide an example of a distortion-corrected image.

- Done on my notebook chapter **'Apply a distortion correction to raw images'**, same undistortion parameters applied to images from `./test_images` folder, via `undist = cv2.undistort(img, mtx, dist, None, mtx)` , one example below :

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

- Covered in my notebook chapter **'Use color transforms, gradients, etc., to create a thresholded binary image'**
- Re-used examples from the course, using HLS color space on undistorted images from previous step, separating L and S channels, using Gradient/Sobelx on L channel, and color threshold on S channel, those 2 methods allowing to find 2 different sets of pixels, and rearranging them together on a single image, improving detection of borders for challenging images to better identify yellow/white lines in bright/shadow situations, using gray threshold image conversions.
- Example of output obtained, in colored threshold image to show with Green/Blue color results of different processes, and final image result in gray threshold binary :





3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

- Covered in my notebook chapter **"Apply a perspective transform to rectify binary image ("birds-eye view")"** and in 1st chapter **"Common imports, variables, functions"**
- I started by taking one of the straight line images `straight_lines1.jpg` .
- I picked 4 points to pick 2 coordinates on each left and right lines as source (src) coordinates : `[265,678],[1042,678],[582,460],[702,460]` to cover roughly 3 discontinuating lines

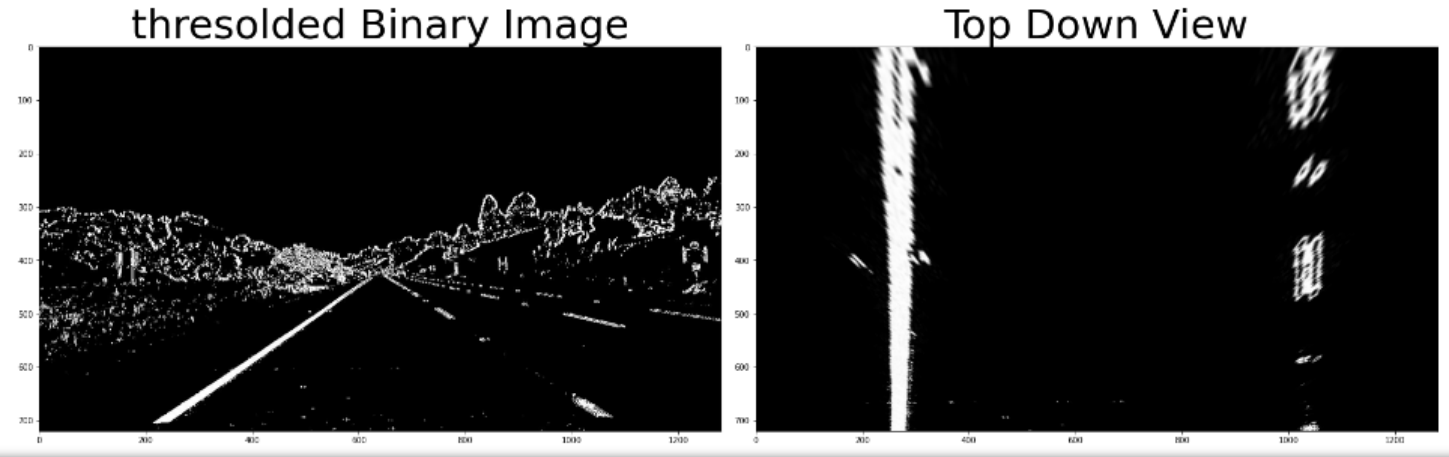
- I then created destination dst array of coordinates, representing where the destination points should be on a transformed image representing 2 parallel lines : `[[265,ySize-1],[1042,ySize-1],[265,0],[1042,0]]` , ySize being the height of the image.
- I then use openCV function to define the perspective transform Matrix to transform the source image and points into a "birds-eye view" where 2 lines are parallel like I defined in the destination points. All this is done via function `getPerspectiveTransformMatrix()` like below :

```
def getPerspectiveTransformMatrix():

    #define 4 source points src = np.float32([[,],[,],[,],[,]])
    src=np.float32([[265,678],[1042,678],[582,460],[702,460]])
    #define 4 destination points src = np.float32([[,],[,],[,],[,]])
    dst=np.float32([[265,ySize-1],[1042,ySize-1],[265,0],[1042,0]])

    # use cv2.getPerspectiveTransform() to get M, the transform matrix
    M = cv2.getPerspectiveTransform(src,dst)
    Minv = cv2.getPerspectiveTransform(dst,src)

    return M, Minv

M, Minv = getPerspectiveTransformMatrix()
```

- I then use the matrix M or the inversed matrix Minv to transform between image view to birds-eye view and inversely, using openCV function `cv2.warpPerspective(img,M,img_size,flags=cv2.INTER_LINEAR)` to obtain warped image using matrix M or to come back to original view image using matrix Minv.

Here are examples of test images, both with gray threshold binary images or original images transformed into bird-eyes view :

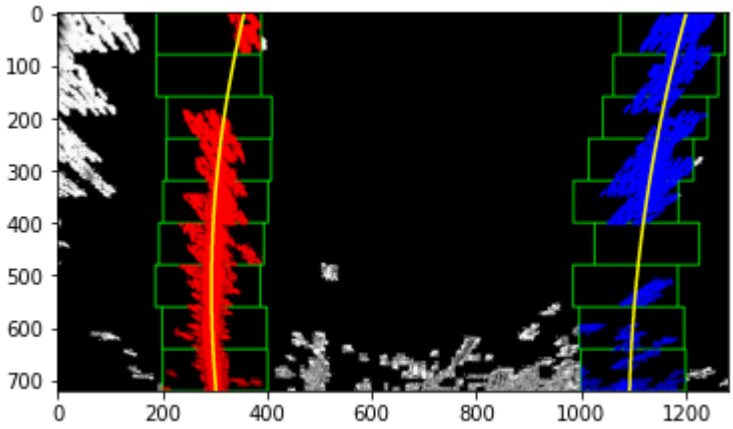
I also checked on another test image with straight lines that the lines are indeed straights after applying a perspective transform, cf below :



4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

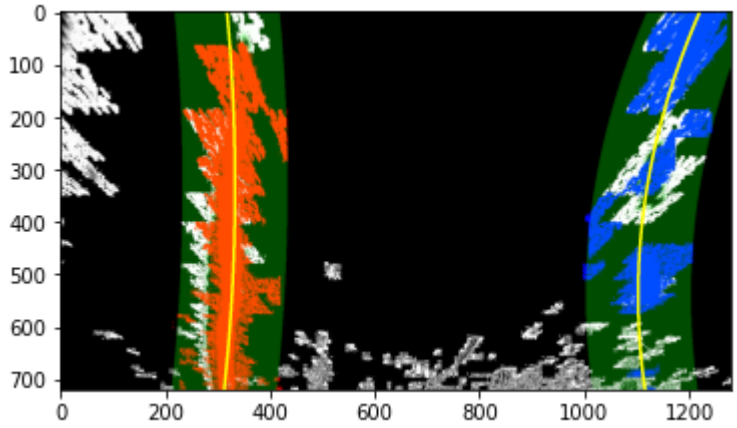
- Covered in my notebook chapter **"Detect lane pixels and fit to find the lane boundary."**
- Function `find_lane_pixels_slidingWindows()` uses the Sliding Window Search to identify left and right lines in an image.
    - First taking bottom half of gray binary threshold image, doing histogram to identify bottom x coordinates ranges of left and right lines by identifying the histogram peaks.

  - Then dividing image horizontally in nwindows = 9 parts, take bottom part, identifying position of all non zero pixels, identifying the ones inside virtual rectangle windows for each left and right lines, and if number of pixels in those 2 left/right rectangles are above a threshold, recenter the rectangles at the mean positions of pixels. Store the non-zero pixels inside those virtual rectangles.
  - Once bottom part window is done, take next part, and slide/recenter virtual rectangles to mach mean position of pixels on both left/right lines.
  - And so forth until top part is done with virtual rectangle.
  - Then use Numpy function `polyfit()` to get a 2nd degree polynomial curve passing through each left/right center of all pixels found. Done in function `fit_polynomial()`
  - Note : function `fit_polynomial()` is called first and then calls both functions `find_lane_pixels_slidingWindows()` and `polyfit()`



- Function `find_lane_pixels_fromPriorSearch()` can also be called from function `fit_polynomial()` as a way to reduce computing effort from function `find_lane_pixels_slidingWindows()` , by searching lines based on previous image lines detected, and limiting the search of line pixels within the polygons identified around previous image line curves for both left/right lines, using 2nd degree polynomial left and right line equation, +/- margin to define polygons with parallel borders to left and right lines. it is done by :

  - First having first image gone through `find_lane_pixels_slidingWindows()` .
  - For subsequent images, taking the lines found from previous image (represented by their 2nd degree polynomial parameters `left_fit` and `right_fit`), Designing a polygon around those lines with borders parallel to those lines (apply + or minus margin to find polygon borders).
  - Selecting pixels from binary gray warped image which are inside those polygons, and using those pixels to find the best line passing through most of those pixels using `np.poly()` function.
  - it then gives us 2nd degree polynomial parameters for each left or right lines being centered on points identified for the left and right line for the current image.
  - Following image represent the polygon search areas for those pixels with the yellow lines representing the left and right lines found using `np.poly()`



- Function `fit_polynomial()` is the main function called to detect the lines.
  - If no prior lines detected or if too much images line detection failed, it will decide to detect lines via Sliding Window Search ie call function `find_lane_pixels_slidingWindows()`.
  - If previous image line detections was acceptable, it will call `find_lane_pixels_fromPriorSearch()` to search for lines based on previously found lines from previous image.

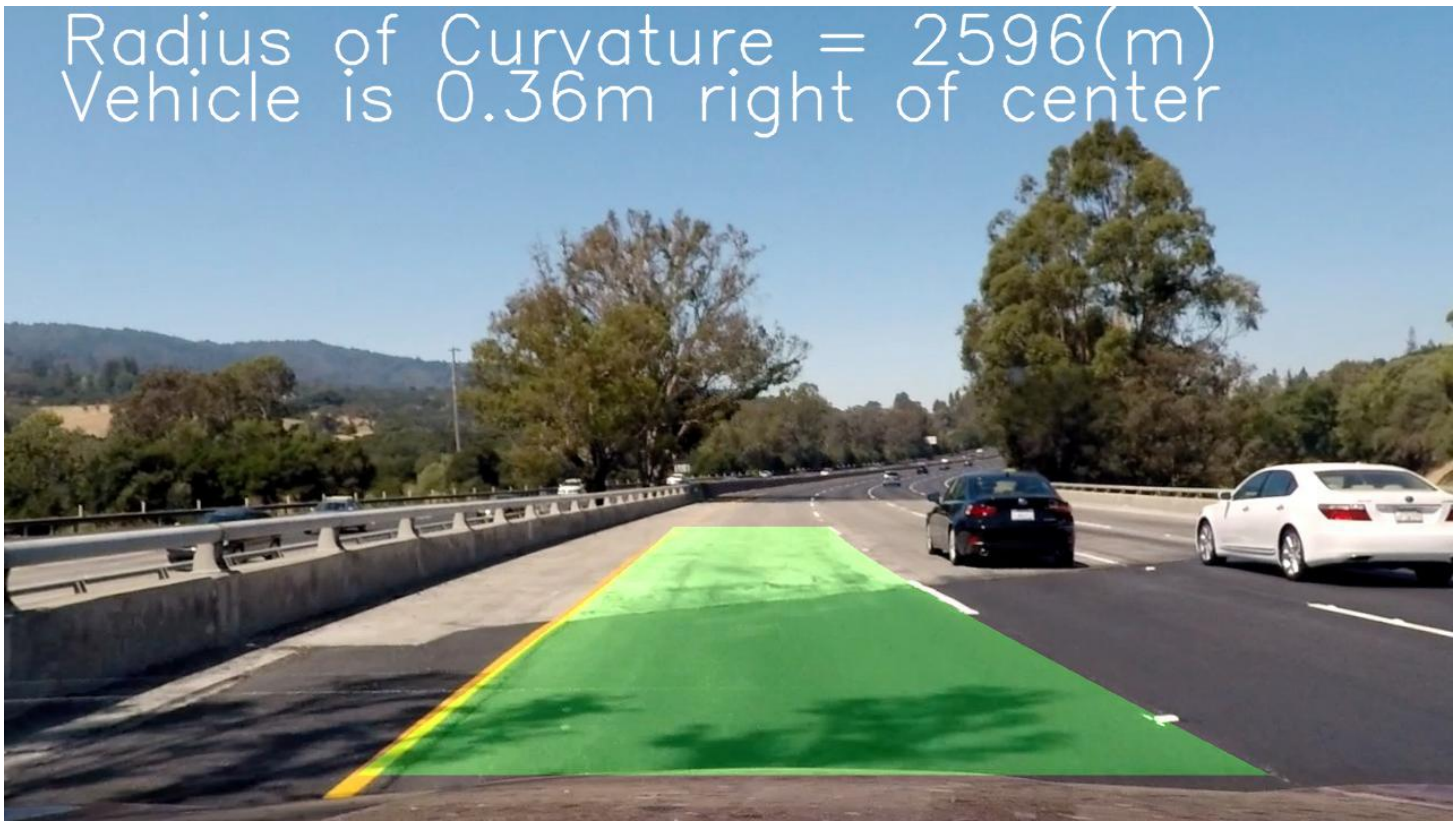
5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

- Radius of Curvature is covered in my notebook chapter **"Determine the curvature of the lane and vehicle position with respect to center. / Measuring Curvature"**
  - Done via the function `getCurvature()`, calling function `measure_curvature_real()`, all defined in earlier chapter and also used to perform sanity checks on line detections and detect false line detections when curvature deviations are to far away from previous image lines.
  - Computing curvature radius for both left and right lines defined by 2nd degree polynomial parameters (`left_fit` and `right_fit`, using formulas provided in the course).
- Position of the vehicle with respect to center, which I label as "deviation", is covered in the same chapter, in function `vehicle_position_vs_center()`.
  - It first calculate x coordinates for left and right lines at the bottom of the picture, using 2nd degree polynomial equation.
  - I calculate middle x coordinates between the left and right lines x coordinates.
  - I calculate also middle x coordinate of the image, knowing the image size.
  - I then subtract the 2 middle x coordinates, which is giving me the deviation of the car compared to the middle of the left/right lines, in pixel values.
  - I then convert into meters, using the pixel/meter calculations I did (knowing the standard distance between 2 lines - 3.7m, or the discontinued line length (3 m?)
  - And this is giving me the deviation of the center of the car, compared to the center of the lanes, in meters.
- Just as an example, my Jupyter notebook computes curvature radius and vehicle deviation for the test images :

```
left_curv   3635 m, right_curv   1556 m, deviation 0.36 m
left_curv   2359 m, right_curv   3232 m, deviation 0.28 m
left_curv  11250 m, right_curv   4927 m, deviation 0.06 m
left_curv   2508 m, right_curv  28153 m, deviation 0.11 m
left_curv   4595 m, right_curv   2616 m, deviation 0.37 m
left_curv   3549 m, right_curv   3649 m, deviation 0.25 m
left_curv   5076 m, right_curv   3041 m, deviation 0.11 m
left_curv   2663 m, right_curv   1370 m, deviation 0.28 m
```

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

- Covered in my notebook chapter **"Warp the detected lane boundaries back onto the original image."**
- Done for all test images in my image output folder : "./output_images/009_detectedLaneBoundaries"
- One example is below :



Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are OK but no catastrophic failures that would cause the car to drive off the road!).

Here's a link to my video result

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

- I mainly faced integration problems between re-using code I used in previous quizzes and putting it together in a single Jupyter Notebook. That by itself was challenging and caused many bugs I introduced, which lingered for many days before I could realize and find them, and therefore I lost many days because of those issues. For instance doing a Prior Line Search Method with the lines polynomial fit coefficient not set correctly to start with. Many issues as well derived from separating code to display examples, from real code for the image pipeline.
    - Then when trying to switch between all the separate steps (like camera calibration, undistort image, gradients and color threshold, line detections, unwraping images with lines) and building a unique image processing pipeline function, removing any display functionalities, went to be challenging as well.
    - I still see one more bug on a test image, for which if I apply the steps separately, the line detection is working ok, but if I apply all the steps together via the process_image() pipeline, I see a different result in line detections. I already spotted out it is coming from the Sliding Window search mechanism, which on one line does not slide the Window in the good direction, I would need to dig further to find this bug out, but time is running out and I do not want to spend days on it yet. I keep it for later debug.
- I also found out that sizing the polygon, selecting the points and the line length to cover in order to do the perspective transform could impact heavily of line detection quality. Picking points further on the road and lanes makes those lanes blurry when converting to gray color and applying gradients and color threshold, and therefore could end up having a polynomial line being curved at the top part of the line instead of straight. I fixed this by shortening the polygon I would choose to do the perspective transform.
- Building this pipeline was challenging enough due to integration bugs to reach an acceptable status on the 1st video project.video.mp4, so I did not have time to look at more challenging videos.
- One thing I had to skip because of lack of time is the smoothing part, ie storing the detected line pixels over several images and averaging them with the newly image lines found to smoothen out the line boundaries in the output videos. I keep that for later work when I'll come back on this project. Therefore it would be an item where my pipeline would fail if lines detected variate too much across images.
- Another thing I skipped and I would do in future when I have more time on this project would be to reduce number of parameters returned by functions or given in input functions, by gathering all those parameters in classes/objects so that it lighten the code between number of parameters passed into functions and returned by those functions.
- Points of failures in my pipeline would be :

- No real error management implemented yet.
- No optimization yet of gradient and thresholds and color transforms, which I feel could improve line detections in challenging images/videos/situations.
- One bug mentioned above to be fixed in the sliding windows search algorithm I have in my pipeline.
- Not yet challenged against more challenging videos like `challenge_video.mp4` and `harder_challenge_video.mp4`, ie no optimization yet about number of frames without lines detected before triggering a reset and starting again a sliding windows line search.