Advanced Lane Finding Project

- The goals / 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 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.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle

Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. You can use this template as a guide for writing the report. The submission includes the project code.

The project submission includes the following files:

- Ipython code
- PDF report detailing the rationale behind the current code
- Example Output Images
- Output Video

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

The OpenCV functions: FindChessboardcorners and calibratecamera functions provide us with an efficient method to remove distortion in images.

1. Object points are a grid of images corresponding to the same number of corners as an image. FindingChessboardCorners function involves using a variety of chessboard images to quantify the corners in each image. These corners are the image points representing how the object points appear in each image.
2. Using both the object points and image points from all different images, calibrateCamera function determines the distortion matrix and coefficients which help in undistorting the images taken by the same camera.

Typically, the distortion matrix and coefficients will not change for a given camera (or lens).

The image depicted below represents an original and undistorted image based on the steps as discussed above:

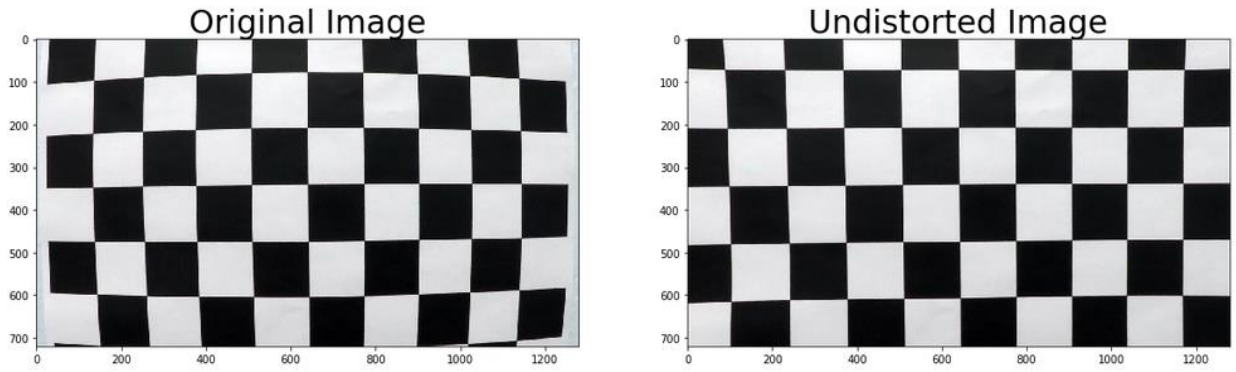

Fig 1: Original and Distorted image of Calibration1.jpg

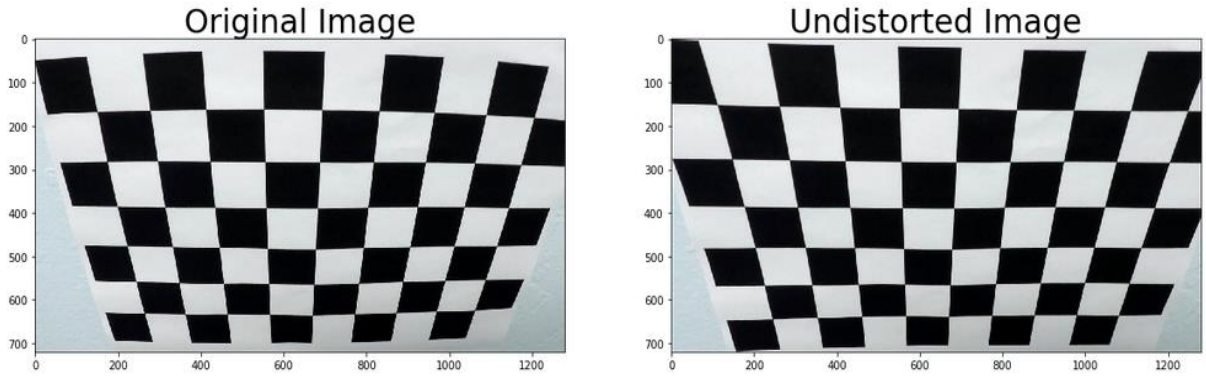

Fig 2: Original and Distorted image of Calibration2.jpg

(***Note***: The code for this step is contained in the first 2 cells of the iPython notebook. Also, all the undistorted images are present in the camera_cal folder.)

(***Note***: Another important point to be noted is that not all calibration images can have chess board corners to be found by the open Cv functions. Refer to the finding chess board corner folder under camera_cal folder to understand this note.)

Provide an example of a distortion-corrected image

The lane images were corrected for distortion based on the camera calibration matrix and distortion coefficients. Some examples of the corrected image are:

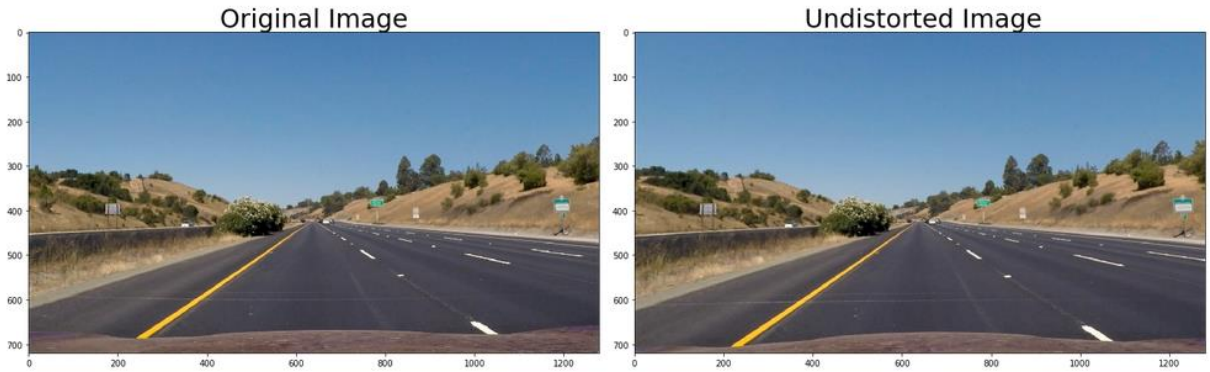

Fig 3: Original and Distorted image of StraightLines1.jpg

Fig 4: Original and Distorted image of Test5.jpg

(***Note***: The code for this step is contained in the 4th cell of the iPython notebook. Also, all the undistorted images are present in the test_images folder.)

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.

This portion of the code is in the 6th & 7th cell (under the heading - "Lane Visualization through color and gradient thresholds") of the Ipython notebook.

In the code, several color and gradient thresholds were explored. Some of the explored methods are:

1. Sobel X Gradient Threshold
2. Sobel Y Gradient Threshold
3. Absolute Magnitude Gradient Threshold
4. Direction Gradient Threshold
5. H Channel Color Threshold
6. L Channel Color Threshold
7. S Channel Color Threshold

(***Note***: The corresponding functions for the different thresholds are enclosed in 3rd cell of the code – under bin_threshold function)

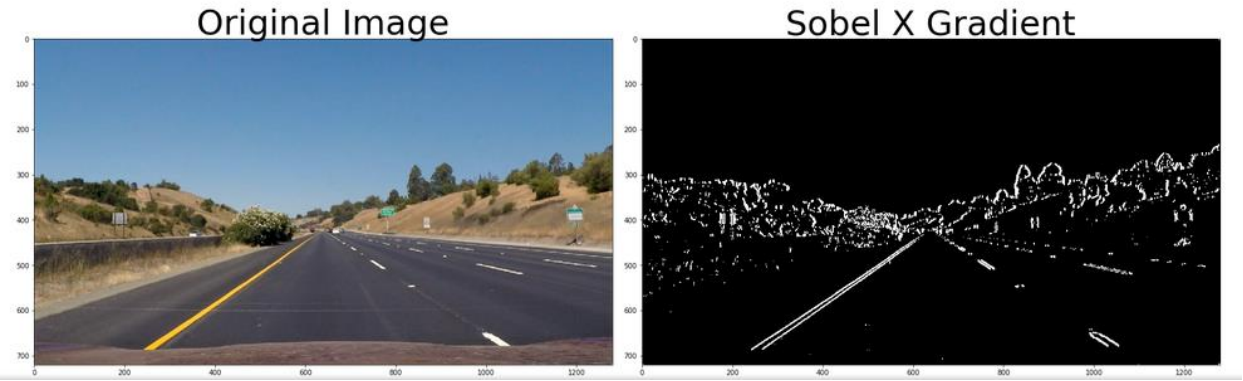

Fig 5: Sobel X Gradient threshold

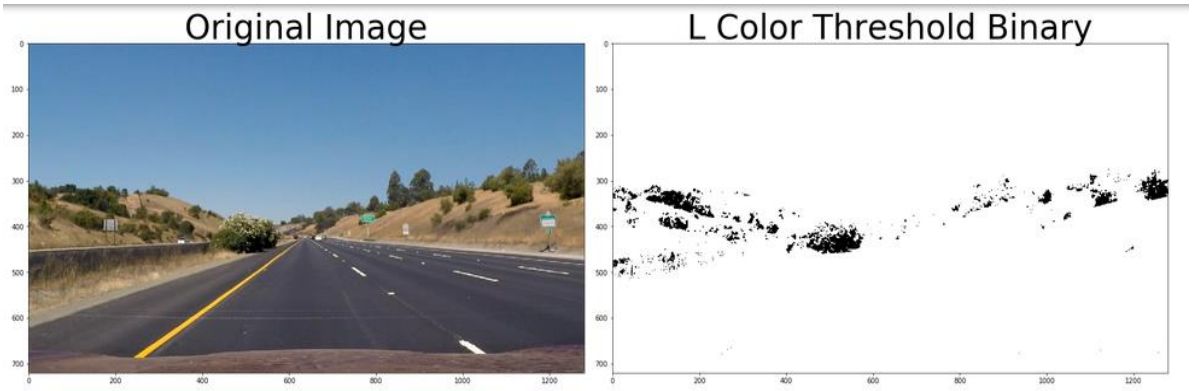

Fig 6: L Channel Color threshold

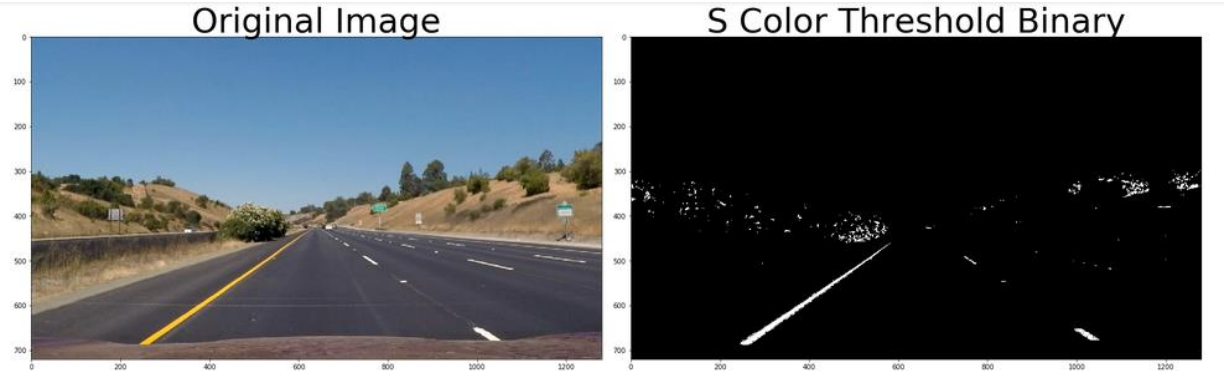

Fig 7: S Channel Color threshold

Ultimately, a combination of the above listed: Sobel X, L Channel and S channel were used in pipeline identification. Sobel X gradient and S Channel color thresholds already capture the lane lines well. However, to increase robustness with respect to different light levels in an image frame, the L channel really helps in smoothening out images and removing some noise.

For the final pipeline implementation, the combination of the 3 channel thresholds are as mentioned below:

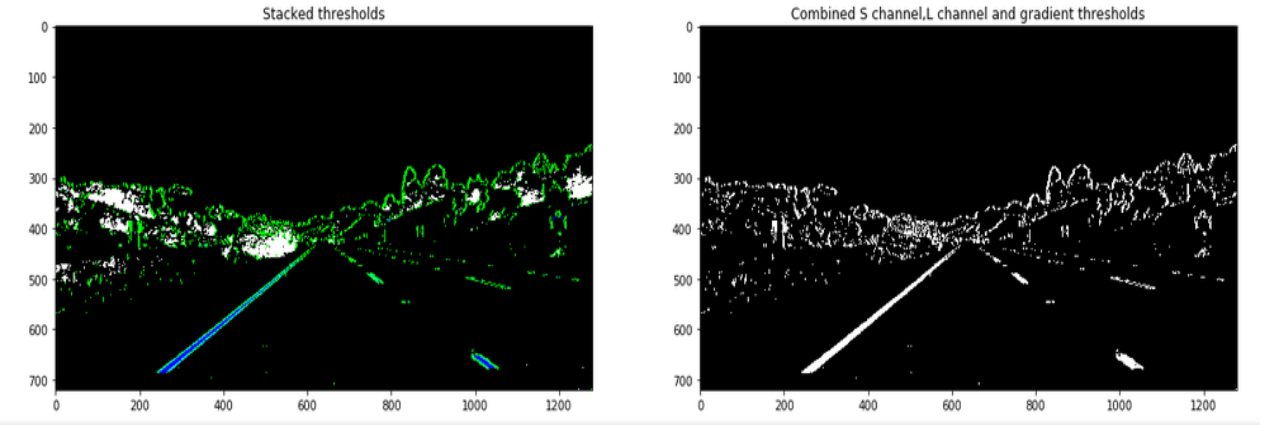

Fig 8: Combined Stacked and binary images based on color and gradient threshold

(***Note***: Please refer to the output of 6th cell in the html document to explore the color and gradient threshold combinations explored and how they can be combined to generate appropriate results. There can be more color schemes and isolation that can be considered and is described in the deficiencies and next steps section)

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

This portion of the code is in the 5th cell (under the heading - "Perspective Transform applied to images") of the iPython notebook. The function call for the perspective transform is defined in the 3rd cell of the iPython notebook(persp_transform function).

The Perspective Transform modifies a perspective of an image from source image points to a destination image points. Depending upon the source and destination image points, an image is transformed from the car dash cam perspective to a "bird's eye" view perspective. This perspective is better to obtain and detect the lane lines.

Source	Destination
710, 450	960, 0
1150, 720	960, 720
230, 720	320, 720
600, 450	320, 0

The destination points are hardcoded with a hardcoded offset from image size whereas the source points can be modified continuously. Multiple source points were attempted in the code for different images and these points were zeroed on for the image and video pipeline.

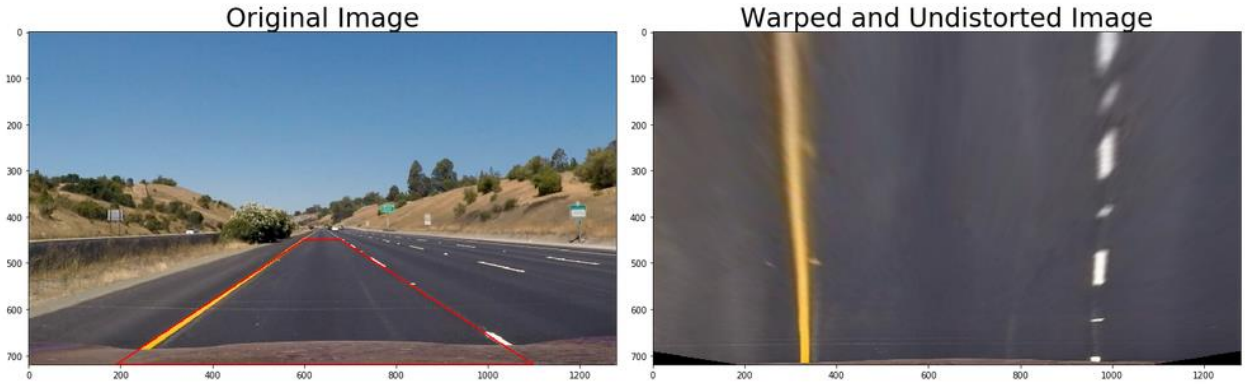

Fig 9: Original and Warped & Distorted image of StraightLines1.jpg

Fig 10: Original and Warped & Distorted image of Test2.jpg

The above 2 examples depict a birds-eye view of the original images. The lines are found to be parallel in the warped image as shown above. The dst and src points were used to verify that a warped image can be inversely warped into the original image using the perspective transform.

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

This portion of the code is in the 8th and 9th cell (under the heading - "Lane Detection on Images") of the Ipython notebook. The polyfit using previous fit is performed in 13th cell.

(***Note***: The corresponding functions for the different thresholds are enclosed in 3rd cell of the code – under sliding_window_polyfit and polyfit_prevfit function)

sliding_window_polyfit - This function computes a histogram of the bottom half of the image and finds the bottom-most x position (or "base") of the left and right lane lines. This implementation taught in the lesson was modified in the final implementation to start from a point just left and right of the midpoint. This helped reject lines from adjacent lanes and corner lines as well.

```
# Find the peak of the left and right halves of the histogram :
# Modify this to look not from the edges but from a slight offset of 100 pixels
# These will be the starting point for the left and right lines
offset=50
midpoint = np.int(histogram.shape[0]//2)
quarterpt= np.int(midpoint//2)-offset
leftx_base = np.argmax(histogram[quarterpt-offset:midpoint]) + quarterpt - offset
rightx_base = np.argmax(histogram[midpoint:midpoint+quarterpt+offset]) + midpoint
```

Fig 11: Example Code to introduce variations in areas to find histogram peaks

The offset is hardcoded and chosen to be 50 after trying multiple images. This can be a potential for narrowing the search more along the base of the image to begin the search more effectively.

The function then identifies 9 windows from which to identify lane pixels, each one centered on the midpoint of the pixels from the window below. This effectively tracks the lane lines up to the top of the binary image, and narrows the area of search to a small portion of the image

The above explained description can be observed in:

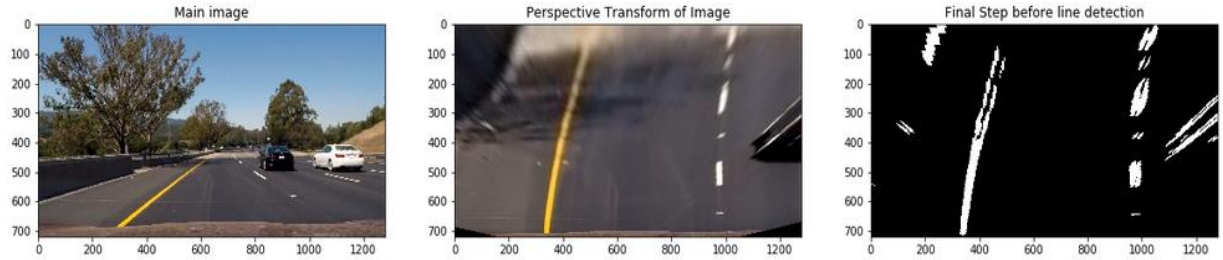

Fig 11: Example image for lane detection – test6.jpg

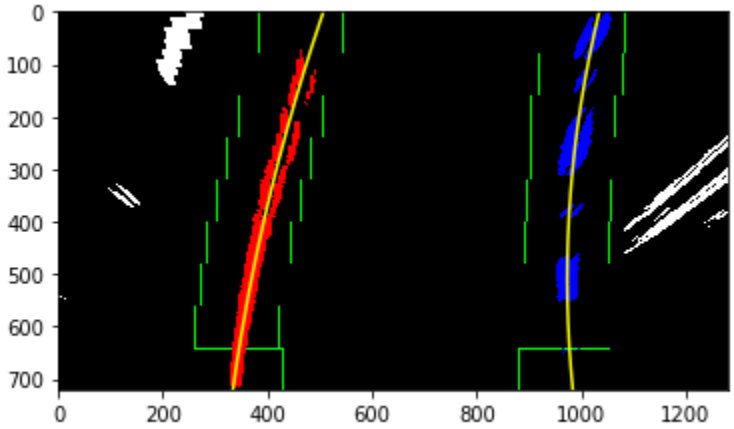

Fig 12: Example polyfit line from the sliding_window_polyfit function

The histograms along the x axis of the image can be observed as:

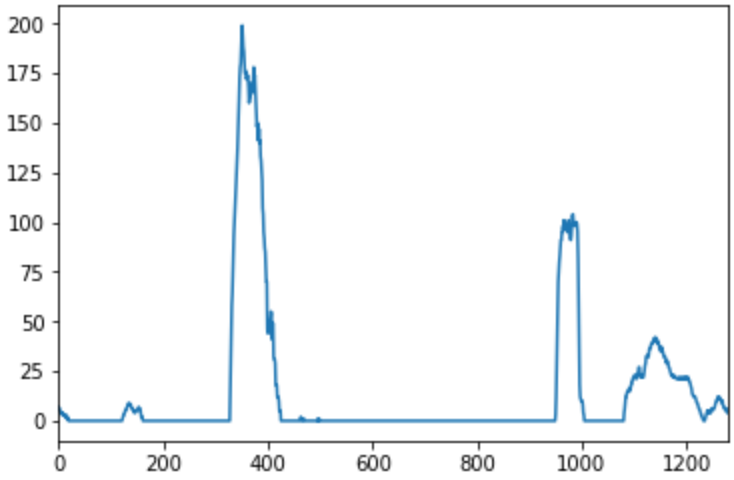

Fig 13: Example histogram line from the sliding_window_polyfit function

polyfit_prevfit – This function performs similar task but makes it easy for the searching process to be more focused with the certain range of a previous fit (eg: from the previous frame of a video). The image below demonstrates this - the green shaded area is the range from the previous fit, and the yellow lines and red and blue pixels are from the current image:

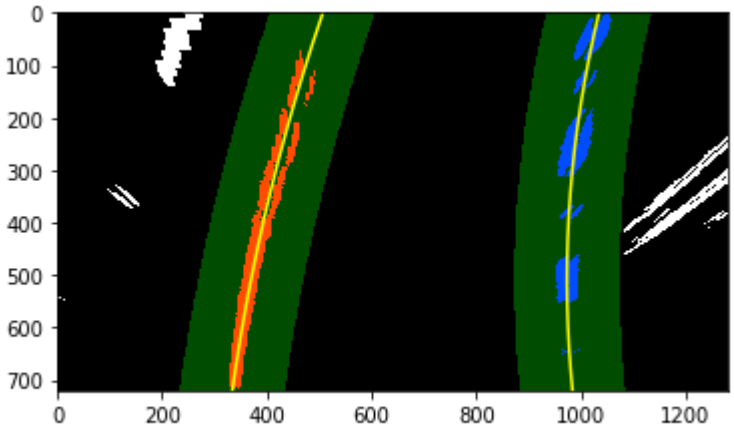

Fig 14: Example histogram line from the polyfit_prevfit function

(***Note***: This function is defined but not utilized during the video processing due to lack of time to complete the project. This will be further discussed in deficiencies and further improvements section)

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.'

This portion of the code is in the 11th cell (under the heading - "Lane Detection on Images") of the Ipython notebook.

(***Note***: The corresponding functions for the different thresholds are enclosed in 3rd cell of the code – under rad_curvature function)

Radius of curvature was calculated based on the content from the lessons. The curve radius is calculated separately for left and right lanes. This can be averaged for overall radius of curvature to be output at the end of the pipeline. The formula used was:

```
# Calculate the new radii of curvature
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]
```

Fig 15: Radius of curvature formula applied to left and right curves

The position of the vehicle with respect to center of lane is identified by following lines of code:

```
car_position = img.shape[1]/2
l_fit_x_int = left_fit[0]*x**2 + left_fit[1]*x + left_fit[2]
r_fit_x_int = right_fit[0]*x**2 + right_fit[1]*x + right_fit[2]
lane_center_position = (r_fit_x_int + l_fit_x_int) /2
center_dist = (car_position - lane_center_position) * xm_per_pix
```

Fig 16: Radius of curvature formula applied to left and right curves

There is an inherent assumption that the camera is mounted at the center of the image. To determine the relative position with respect to the center of the lane, the intercepts of the left and right polynomial fits of lane lines are computed using the polynomial generated in the function. The difference between the center of the image and average of the intercepts provides the position of the car relative to the center of the line.

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

This portion of the code is in the 12th cell (under the heading - "Lane Detection on Images") of the Ipython notebook.

(***Note***: The corresponding functions for the different thresholds are enclosed in 3rd cell of the code – under draw_lane function)

All the above steps are combined with a process_image function in14th cell of the notebook. The output images are stored in the output_images folder in the main directory.

Fig 17: Final processed images for Couple of images

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!)

The project video is available in test_video_output folder.

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?

This was one of the most challenging projects so far personally. The background of my knowledge is not in computer vision and it was quite good trying to get this to this stage. It is not probably over since this is a very good platform for me to try, learn and apply new variations not provided in the lessons.

The pipeline was created using very basic S channel ,L channel and Sobel X thresholds. The image pipeline works well but can still be improved. Working on the challenge video and the harder challenge video was almost nightmarish due to the amount of accidents that would have happened if the current algorithm was implemented. This submission is intend for minimum qualifications due to lack of time and I intend to explore the pipeline more slowly.

Shortcomings:

1. The problems encountered were almost exclusively due to lighting conditions, shadows, discoloration, etc.
2. Images of snow might be confused as lane lines and can confuse the pipeline.
3. Video output has a specific region where the pipeline gets confused due to the lack of a missing dotted lane line.
4. Narrower Lanes can be a challenge since we assume almost similar width

Further improvements:

1. Using polyfit_prevfit has not been effectively implemented while processing video output – This will greatly reduce the wobbles as seen in the video output
2. Better dynamic thresholding using different color spaces such as LAB, RGB,etc.
3. Dynamically optimizing the source, destination and offset in the perspective transform. This can be very useful with respect to speed of the vehicle- Greater the speed of the vehicle, more look ahead distance, slower the vehicle, look ahead slowly due to ability for sharp turns.

4. Sanity checks and Line objects have not been defined as described in the tips and tricks of the projects. This will be a huge help going further to ignore line segments which do not make sense.
5. Implementation of a sliding window search using convolution approach- This can be implemented parallelly with the existing approach to add more robustness and checking conditions to validate the detected line.