Advanced Lane Finding Project

The main repository is present at this location:
https://github.com/sanyam89/Project_2_advanced_lane_lines

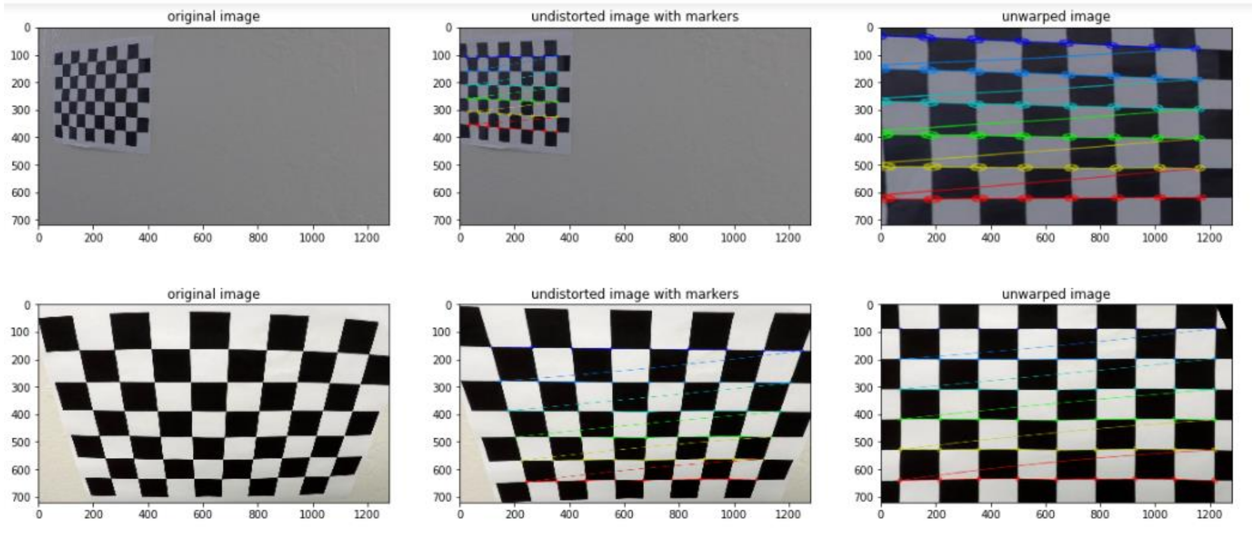
Camera Calibration

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

The code for camera calibration can be found **here**

The following steps were involved in the process:

1. Input 20 9x6 chessboard images
2. Find chessboard corners (white and black intersections) using cv2.findChessboardCorners() function on a grayscale image
3. Draw the found chessboard corners on those images using cv2.drawChessboardCorners() function
4. Then use the cv2.calibrateCamera and cv2.undistort function using the pre-defined object points and image points that were found using the findChessboardCorners() function
5. I also performed the perspective transform on the undistorted images to verify if the chessboards were being warped in the correct manner.
6. I saved all the image points and object points from all these images so that they could be used in the main program to calibrate the camera.

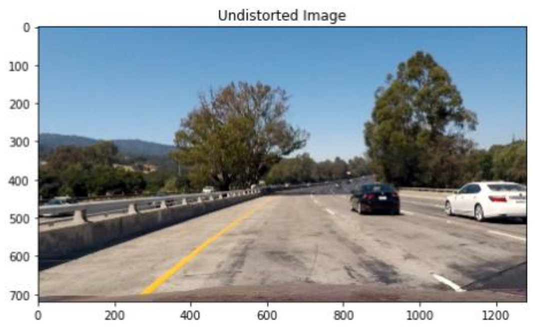


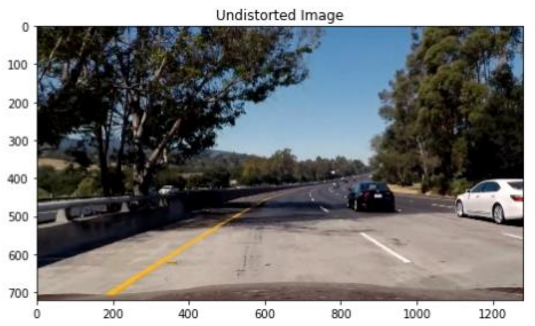
The code for the main program can be found here.

Pipeline (test images)

Provide an example of a distortion-corrected image.

Distortion correction that was calculated via camera calibration has been correctly applied to each image. Below are the 2 examples of a distortion corrected image



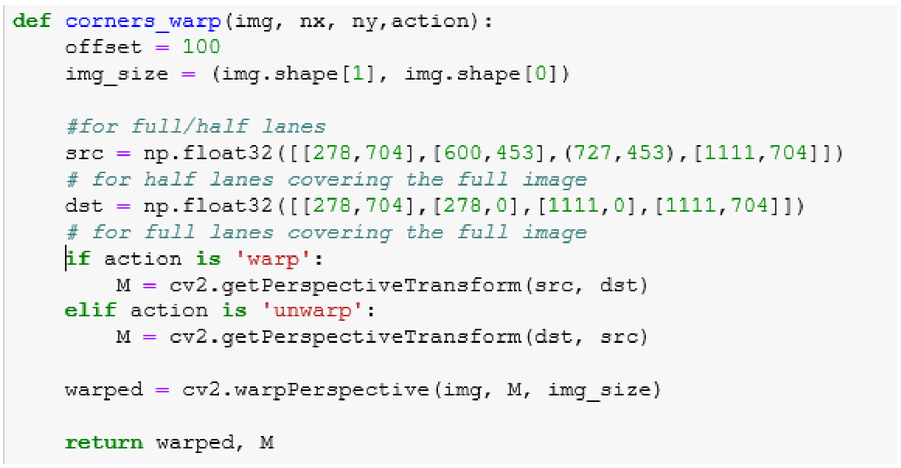


Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

I performed the perspective transform before applying the color thresholds and gradients to get the binary images because I wanted to get rid of the information that I would not need for finding the lanes on the road and its curvature and direction.

This is the snippet of the code where I selected 4 points of lane in the shape of a trapezoid in the image that I wanted to warp and saved them in the *src* array. Then I defined destination array where the upper two points were stretched to the top of the image to make a rectangle.

Finally, I performed perspective transform using these 2 arrays. The same function is used at the end of the code to unwarp the image to its original shape for the video.

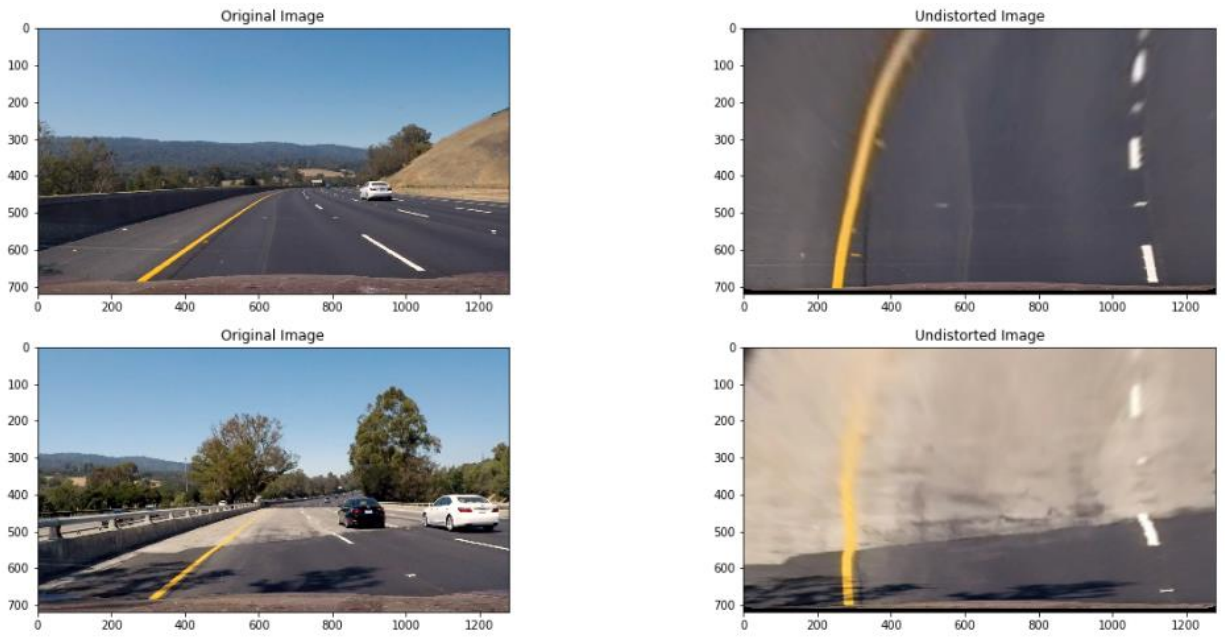

```python
def corners_warp(img, nx, ny,action):
    offset = 100
    img_size = (img.shape[1], img.shape[0])

    #for full/half lanes
    src = np.float32([[278,704],[600,453],(727,453),[1111,704]])
    # for half lanes covering the full image
    dst = np.float32([[278,704],[278,0],[1111,0],[1111,704]])
    # for full lanes covering the full image
    if action is 'warp':
        M = cv2.getPerspectiveTransform(src, dst)
    elif action is 'unwarp':
        M = cv2.getPerspectiveTransform(dst, src)

    warped = cv2.warpPerspective(img, M, img_size)

    return warped, M
```

Below is an example of the transformed images.



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.

I performed various iterations with different gradients and colorspaces like HSV, HSL color spaces, Gradient along X and Y-axes and their direction and magnitude gradients.

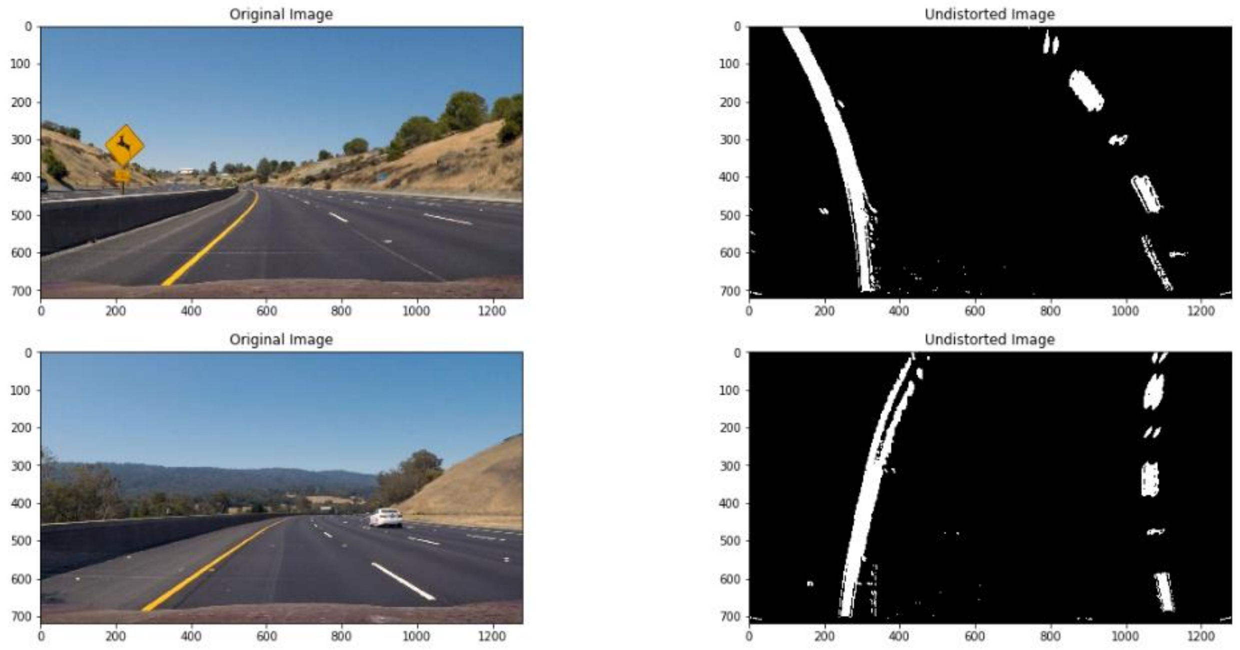
What provided the best isolation of the lane lines were HSL colorspace (S binary, L binary) and Gradient along X-axis. Then I OR'ed all three of these binaries to get my final output binary image *combined_LorSorX*.

This can be found in the process_images() function in my main code.

The thresholds are defined below:

Binary type	Minimum threshold	Maximum threshold
S Binary in grayscale HSL image	180	255
L Binary in in grayscale HSL image	180	255
Gradient X	20	100

The warped and binary thresholded images are displayed below.

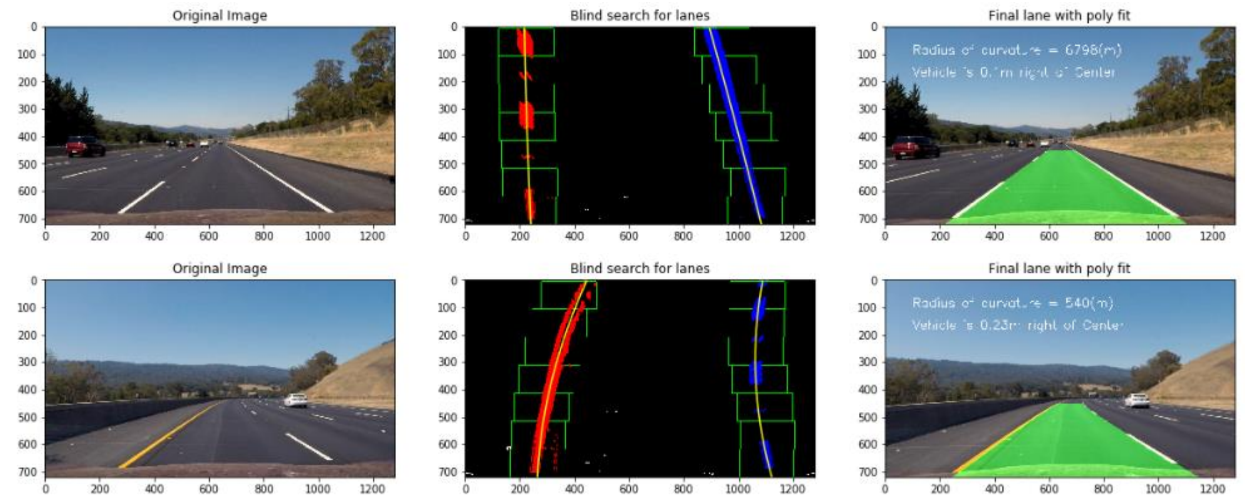


Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

After identifying the lanes in the binary image I used two functions: blind_lane_search() and focused_lane_search() to identify the lane pixels and fit their positions with a second order polynomial.

In *blind_lane_search()* function*,* I created a histogram of the bottom 75% of the image by adding the nonzero pixels along X-axis and then divided the lanes in half to segregate left and right lanes. Then I chose the highest points in the histogram arrays to act as the center of the lanes. The image was divided into 10 sections along Y-axis, from there I searched for lane pixels in a +/- 100 pixels window in each section. All the nonzero pixels were appended to two separate arrays for left and right lane and if the average of these pixels in the section were more than minimum pixels (50) then the center point of next section was shifted to the center point of the pixels found in the current section. After finding all the nonzero pixels, I called fit_polynomial() function that outputs a second order polynomial fit along the X and Y coordinates sent to it. Which was then used to plot the result on the warped image as shown below.

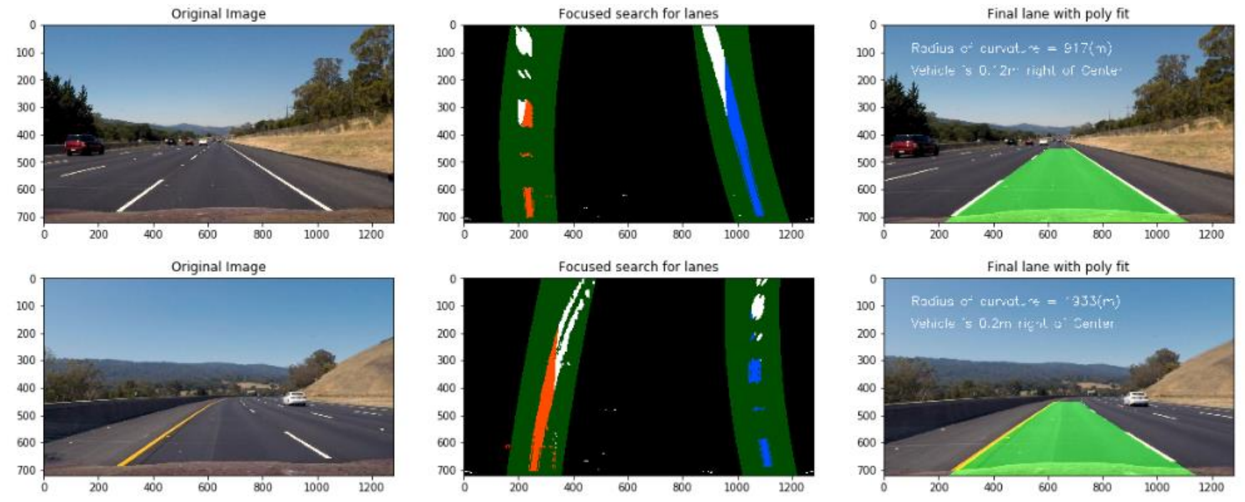
The relevant code is in *blind_lane_search()* function provided in the main code*,*



In *focusedd_lane_search()* function, I used the lane fit defined from the previous frame and added margin of +/- 100 pixels along those polyfit lanes to conduct a faster, more focused search. Fit_polynomial() function was again called to fit a new second order polynomial along the nonzero pixels found in the focused area.

The relevant code can be found in the main file under focused_lane_search() function.

The image result is shown below.



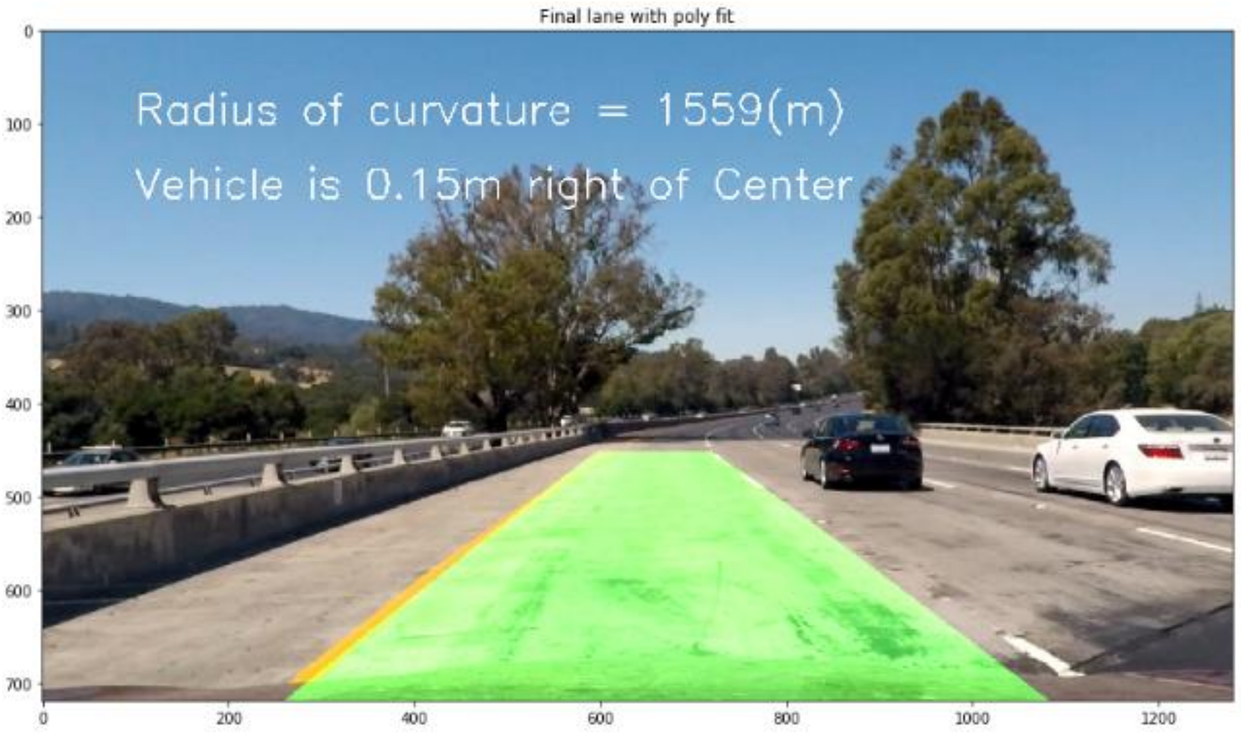
Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

The radius of curvature was calculated in *measure_curvature_real()* function. The fitted X and Y coordinates for left and right lanes were used to calculate left lane and right lane radii using the formula provided in the lessons.

left_radius = ((1 + (2*left_fit_cr[0]*y_eval + left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])

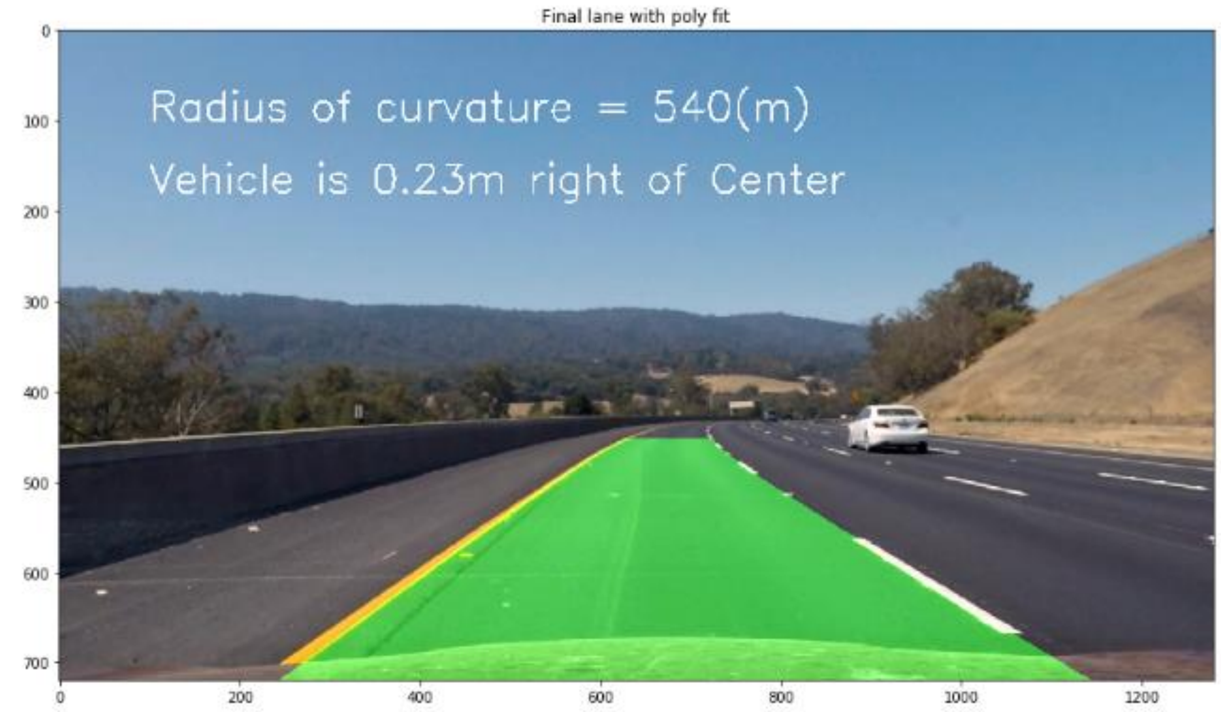
Where left_fit_cr is the polynomial fit that was obtained by the X and Y coordinated found using the lane search functions above, after converting them from pixels to meters. Then I averaged the left and right lane radii to obtain the final radius in meters.

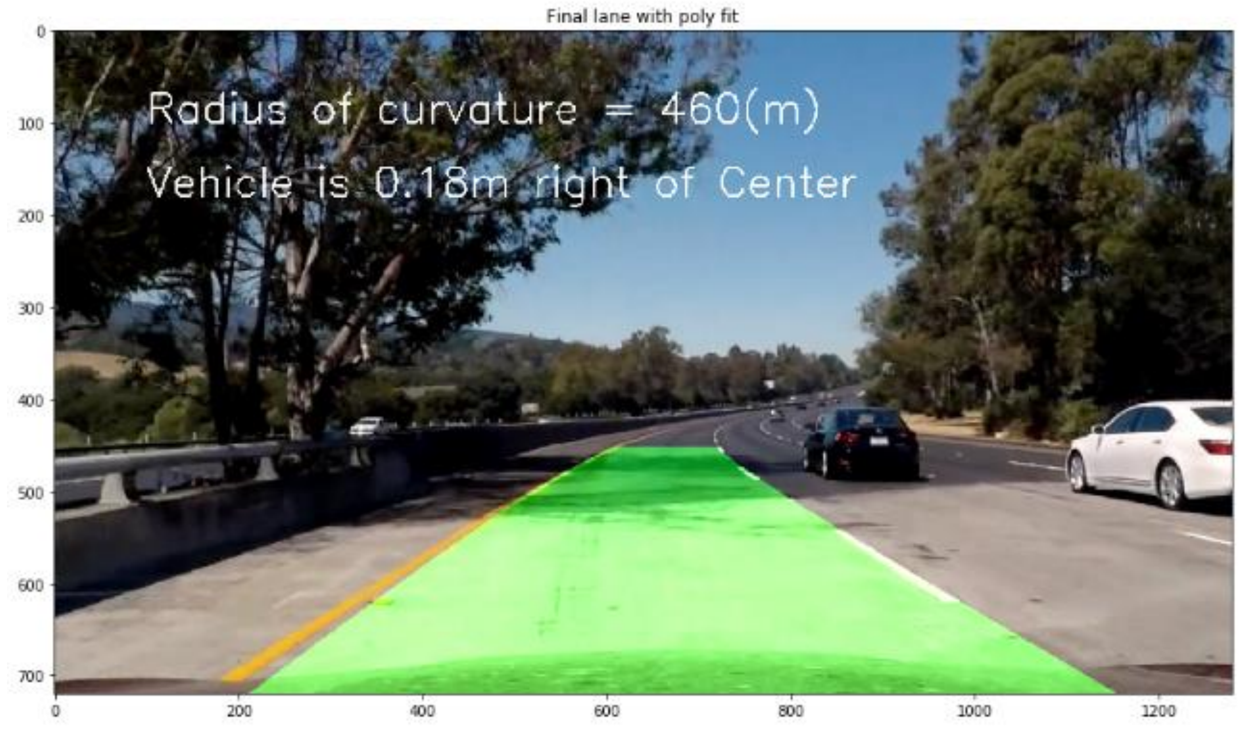
For the position of the vehicle, I assumed that the camera is mounted at the center of the car and the deviation of the midpoint of the lane from the center of the image is the offset I am looking for. As with the polynomial fitting, convert from pixels to meters. The calculation is performed in *offset_calculator()* function. I found the center of the lane by taking an average of the left and right lane points at the bottom of the image (i.e. where the vehicle is) and for camera center, I took the middle point of the image. Then I calculated the distance in pixels between these two points and converted it into meters. My code also shows if the vehicle is in the right of the center of the lane or left side.



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

The fit from the rectified image has been warped back onto the original image and plotted to identify the lane boundaries. This should demonstrate that the lane boundaries were correctly identified. An example image with lanes, curvature, and position from center is included below.


Final lane with poly fit
Radius of curvature = 540(m)
Vehicle is 0.23m right of Center
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


Final lane with poly fit
Radius of curvature = 460(m)
Vehicle is 0.18m right of Center
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Sanity Check

I added 2 sanity check in my code.

1. Any point on left lane cannot go to the right side of the corresponding point on the left lane. In this case then restart from the blind search method.
2. If no pixel points were identified in the binary thresholded images then use the previous frame's lane markers. In this case, use the lane line defined in the previous frames.

These helped me during the challenge video.

Pipeline (Video)

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 image processing pipeline that was established to find the lane lines in images successfully processes the video. The Project output video can be found at this location:

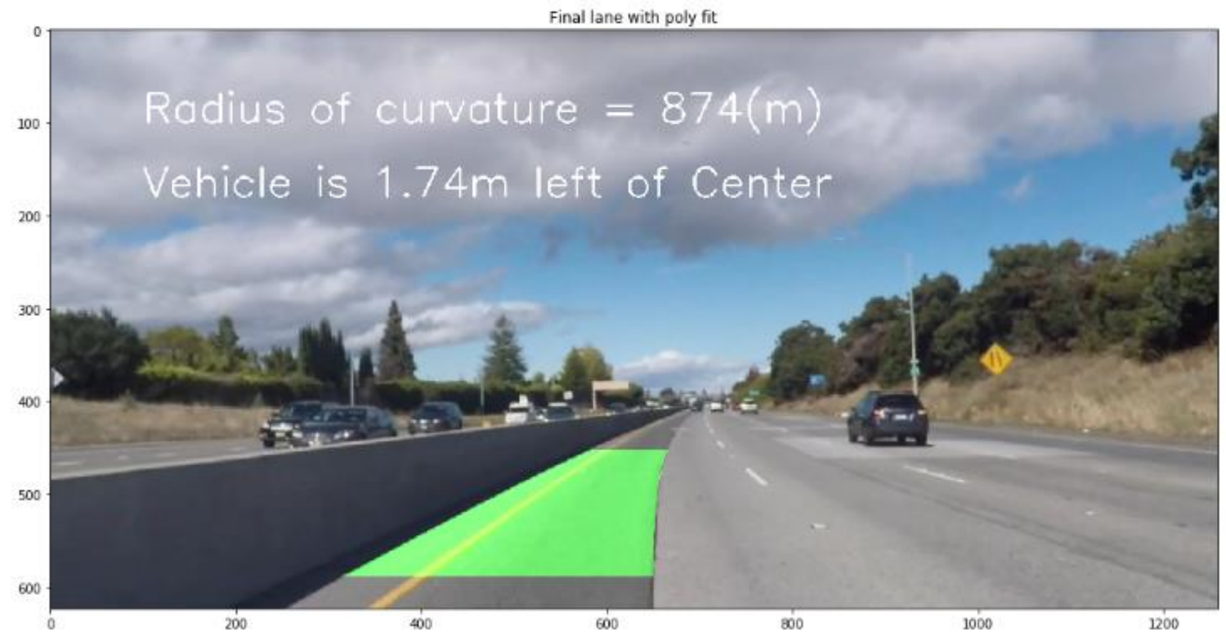
Discussion

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?

The challenge output video is located at this location:

For challenge video, my pipeline gets trapped into the road partition and then after some time it comes back to the real lanes. The problem I was having with this was that the gradient along X-axis identifies this as well as the yellow and white lanes and it’s hard to distinguish. If I don’t use gradient (sobel_X) then I don’t have enough points to form a line just by using S and L threshold in HSL image. If I use the lines defined in the previous frames then I get somewhat good result, but still not as concrete as I would like them to be. I thought of using a weightage while I am Or’ing the three threshold gradients I picked (80% for the SL and L and 20% for the gradX), but I couldn’t figure out how can I do that. Any help on this would be really appreciated.

Here is an example of the wrong lane identification:



Also, for harder challenge video, I was thinking that if left or right lane is not detected but the other one is clearly detected, then my code should add the other lane at a distance (3.7m) after warping to define the area where vehicle could be driven. Is there a better way to do it?