Advanced Lane Finding Project

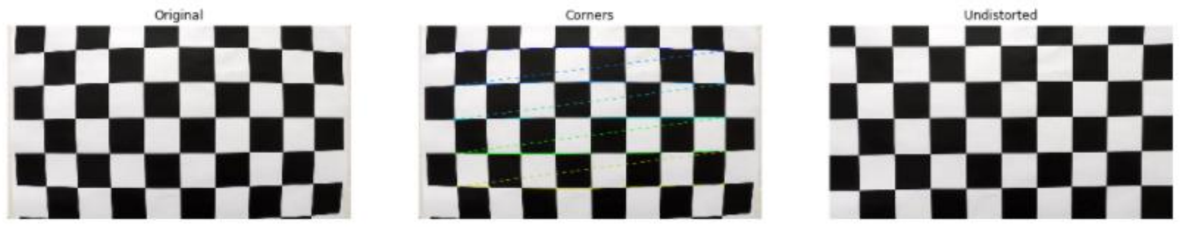
The goals / steps of this project are the following:

- 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 position.

Camera Calibration

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

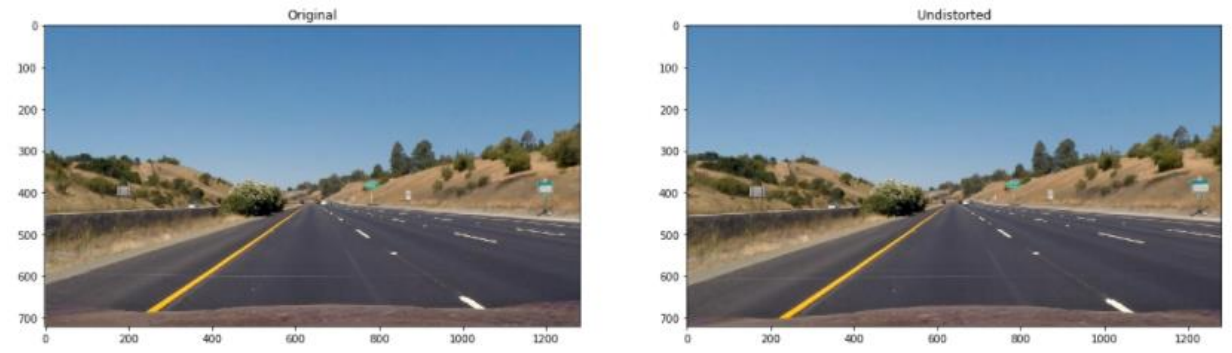
The code for this question is provided in the cells 2,3,4 & 5 of the IPython notebook: P4.ipynb.
The chessboard images were used to calibrate the camera and calculate the camera matrix, distortion coefficients. The object points which are a vector of pattern points in the calibration space are calculated using the expected number of rows and columns in the chessboard which is 5 and 9 respectively in the image below. The image points which represent the projection of the object points in the image space for the chessboard are nothing but the inner corners of the chessboard which are detected using the OpenCV function cv2.findChessBoardCorners(). The camera calibration matrix and the distortion coefficients are calculated using the OpenCV function cv2.calibrateCamera(), the object points and image points. The camera calibration matrix and distortion coefficients are then used to undistort the image by applying the cv2.undistort() function as shown below.



Pipeline (test images)

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

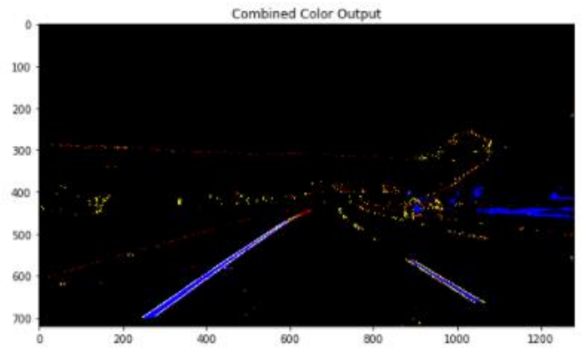
The following images displays the distortion-corrected test lane image. The code for this question is provided in cell 6 of the IPython notebook: P4.ipynb.

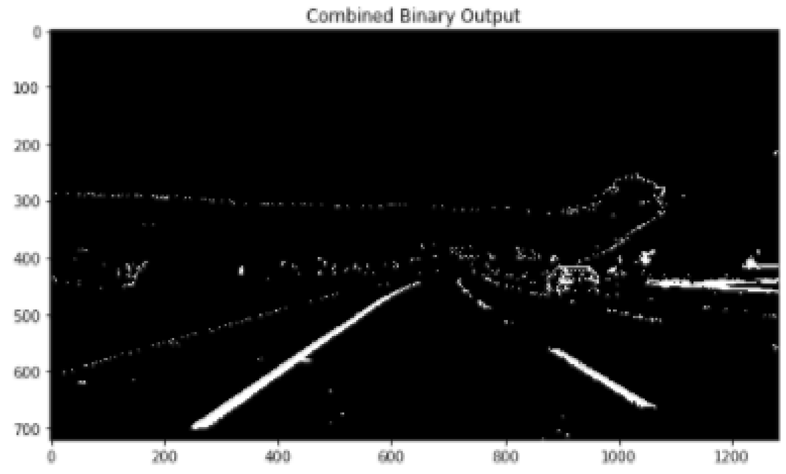


2. **Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.**

   The code for this question is provided in the cells 7, 8 & 9 of the IPython notebook: P4.ipynb. The gradients, the color transform, the magnitudes and the directions were used to create the thresholded binary image. The gradient threshold helped define the definition of the lanes. The x gradient was used as it captured the vertical lane lines accurately without any horizontal gradient noise. The color transform was performed on the HLS color space. All three channels were used to identify yellow and white lane lines only. The magnitude and direction thresholds helped to capture the smaller sections of the lane lines which weren’t detected in the gradient and color thresholds. The images below display the original image, the contribution of individual thresholds and the combined binary output. The combinedColorGradientThreshold() function performs the above operations.

Parameter	**Threshold Limits**
Gradient	(50, 150)
Magnitude	(50, 150)
Direction	(0.7, 1.3)
H Channel	(0, 35)
L Channel	(90, 255)
S Channel	(65, 255)

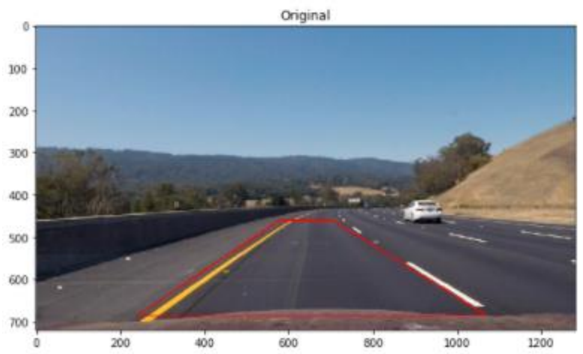


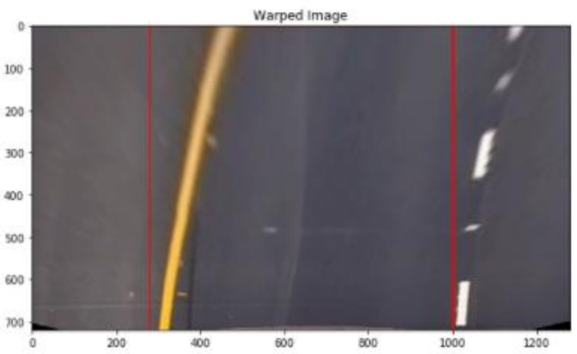


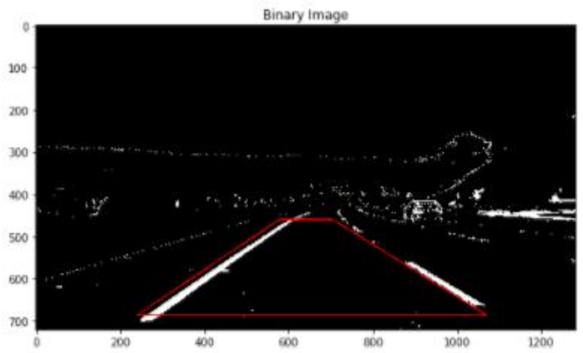
3. **Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.**

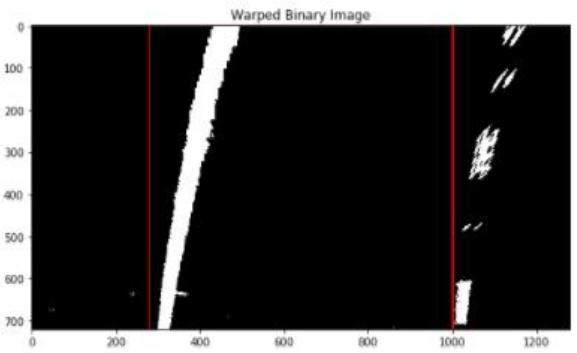
   The code for this question is provided in the cells 7 & 10 of the IPython notebook: P4.ipynb. The perspective transform was performed using geometric transformations in OpenCV. The functions cv2.getPerspectiveTransform() and cv2.warpPerspective() were used to calculate the transformation matrix and transform the image respectively. The source points were selected in a trapezoidal pattern with the non-parallel sides to be as parallel to the left and right lanes each. The destination points were selected in a rectangle. The following table lists the source points and their corresponding destination points. The images below display the perspective transform of the original image and binary thresholded image. The polylines function is used to plot the source and destination points on the respective images.

Source	**Destination**
(578, 460)	(280, 0)
(704, 460)	(1000, 0)
(1070, 686)	(1000, 720)
(240, 686)	(280, 720)



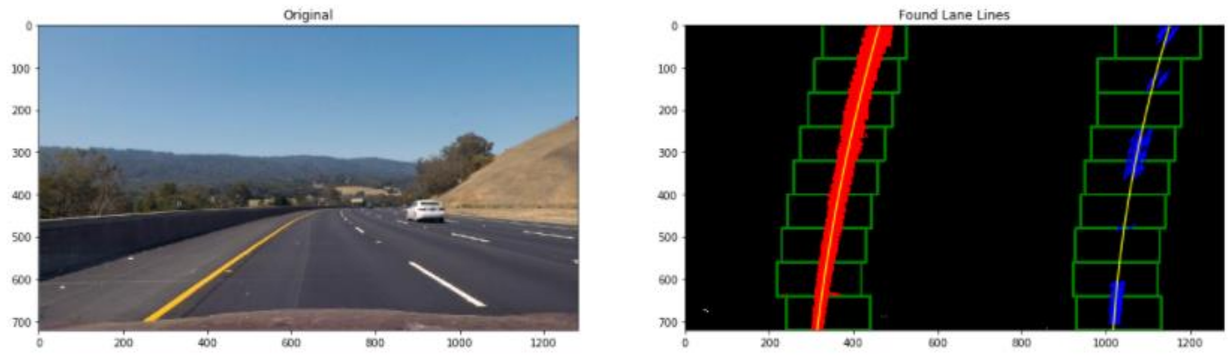






4. **Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?**

   The code for this question is provided in the cells 11, 12, 13, 14 & 15 of the IPython notebook: P4.ipynb. The class Line() was used to identify the lane lines. The class defines a line object with attributes that store the lane line pixels, the best fit polynomial, the radius of curvature and the vehicle position. The class also defines the blindSearch(), knownSearch() and findLanes() functions to calculate the lane lines associated with the object. The blindSearch() function searches for lane pixels with no prior knowledge about the lane position in the image. It is used when there aren't any lane lines detected in the frame. If the algorithm detects lanes in the previous image, the knownSearch function is used which is computationally less expensive. The knownSearch function uses the fit polynomial of the previous image and searches just in the vicinity for the lane pixels. The lane pixels are used to calculate a second order polynomial function that fit the lane lines using the np.polyfit() function. The fit polynomial is used to calculate the extreme intercepts, the radius of curvature and the vehicle position. A sanity check is performed that compares the radius of curvature of the left, right lanes and the current, previous image lanes. The findLanes() function calls the blindSearch() and the knownSearch() functions. The processImage function function is used to call the findLanes() function, the radius() function, the sanityCheck() function and returns the original, binary images with lane lines drawn. The images below display the lane lines and fit polynomial.



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.**

   The code for this question is provided in the cells 12 & 13 of the IPython notebook: P4.ipynb. The radius of curvature was calculated by plotting points around the calculated fit of the current lane line, fitting a polynomial and applying the formula as shown below:

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

$$For\ a\ second\ order\ polynomial: Ay^2 + By + C$$

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

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

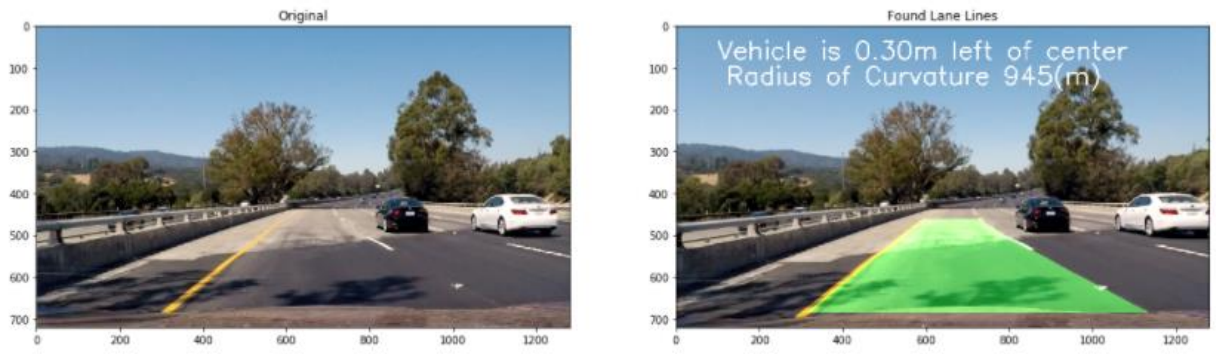
   Algorithm used:

```
# Calculate an array of y coordinates of the image
 ploty = np.linspace(0, 719, num=720) # to cover same y-range as image

 lcoeff = leftfit[0] # Current left lane fit coefficient
 rcoeff = rightfit[0] # Current right lane fit coefficient

# Calculate the x coordinates of the points around the fit function
 leftx = np.array([left_bottom + (y**2)*lcoeff + np.random.randint(-50, hi
g    h=51) for y in ploty])

 rightx = np.array([right_bottom + (y**2)*rcoeff + np.random.randint(-50,
high=51) for y in ploty])
```

```
 leftx = leftx[::-1]  # Reverse to match top-to-bottom in y
 rightx = rightx[::-1]  # Reverse to match top-to-bottom in y

# Fit a second order polynomial to pixel positions in each fake lane line
 left_fit = np.polyfit(ploty, leftx, 2)
 left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
 right_fit = np.polyfit(ploty, rightx, 2)
 right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]

 y_eval = np.max(ploty)

# Calculate radius of curvature using
 left_curverad = ((1 + (2*left_fit[0]*y_eval + left_fit[1])**2)**1.5) / np
.absolute(2*left_fit[0])
 right_curverad = ((1 + (2*right_fit[0]*y_eval + right_fit[1])**2)**1.5) /
np.absolute(2*right_fit[0])
```

The vehicle position is calculated by assuming that the camera is at the center of the image. So the difference between the center of the image and the center of the lanes is the deviation of vehicle from the center. The vehiclePosition() function does this calculation.

6. **Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.**
   The following image displays the lanes plotted back onto the original perspective. This is done by using the previous source points as destination points and vice versa.



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!)**

   Click here to access the project video output.

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**
   A lot of the problems faced were related to isolating the lane lines using thresholds. A single s channel failed in shadows. So a combination of hls channel and the gradient was used to threshold the image. However, the pipeline is still weak on high contrast images and images that have some additional noise, for example, the challenge videos. Also, the algorithm relies on both left and right lane lines. So if one of the lanes is missing or goes out of vision, the algorithm wouldn't display the lanes properly. Another thing is that only a second order function is used to fit the lines; so, if the lanes have larger curves it wouldn't be able to fit the lane lines.