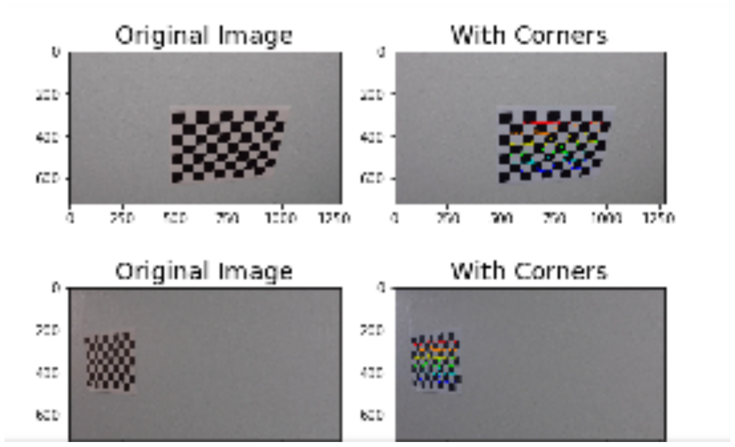
The goals / steps of this project are the following:

- **Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.**

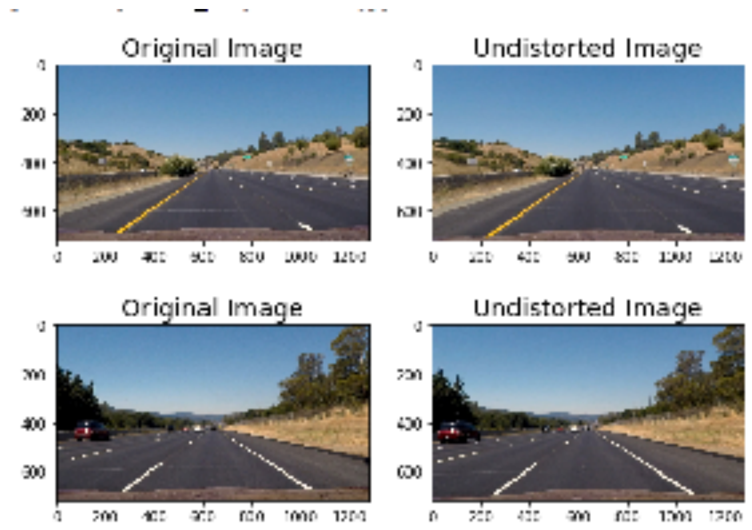
Camera calibration uses the provided checkerboard calibration images using findCheckerboardCorners and matches them with a square grid. This calculates a perspective transform matrix which can be used to correct for radial distortion. There are 20 images of which 2 are shown below. The rest are shown in the ipython notebook.



- **Apply a distortion correction to raw images.**

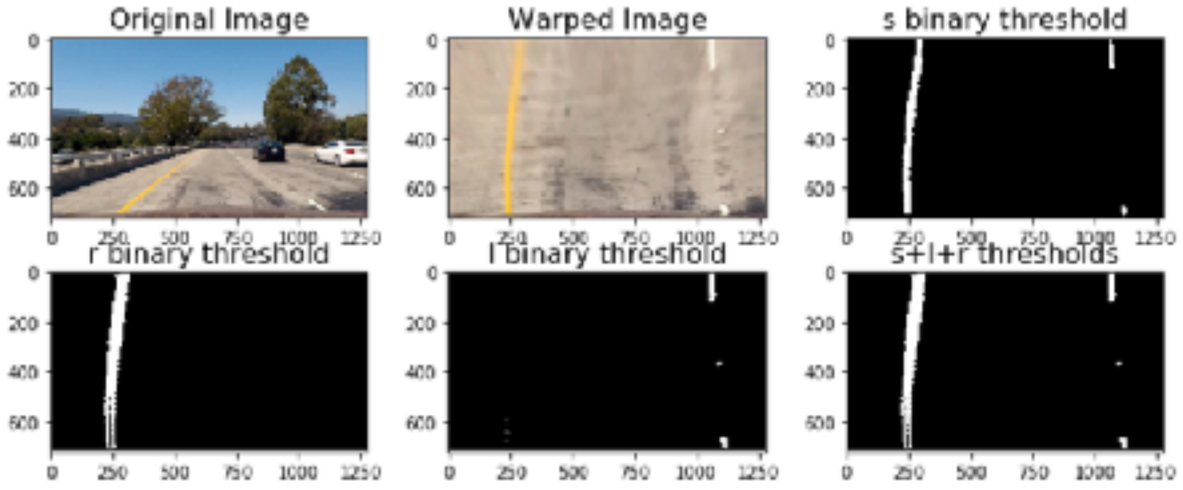
The distortion corrections are computed using cv2.calibrateCamera with datapoints computed from the camera calibration step above.

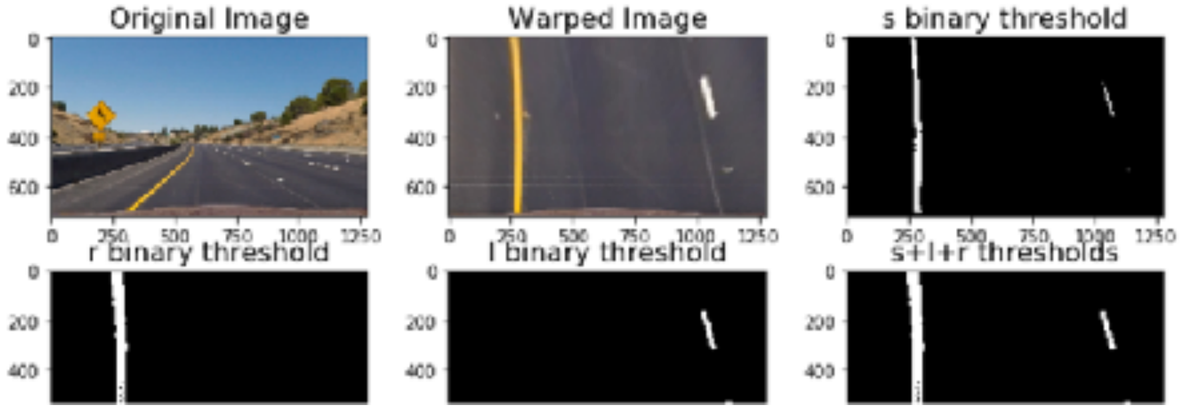
The distortion correction is are applied to 6 test images of which 2 are shown below. The rest are shown in the ipython notebook.



- **Use color transforms, gradients, etc., to create a thresholded binary image.**

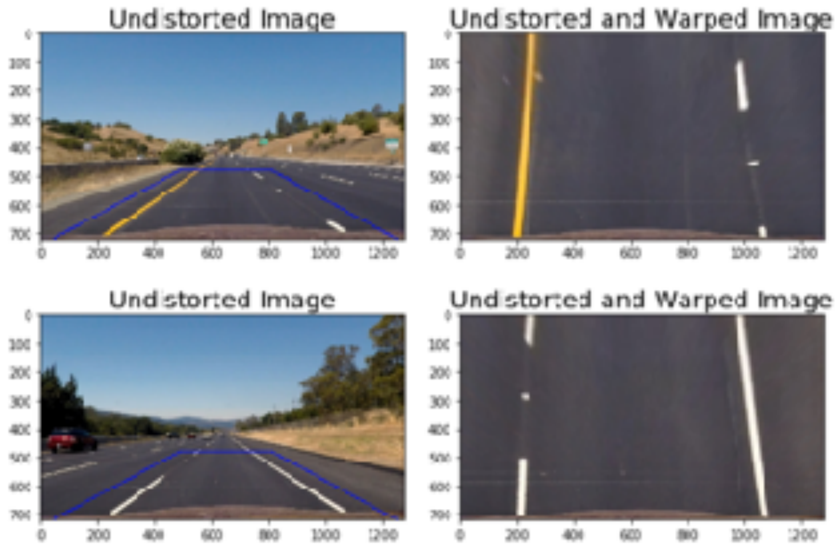
The lecture shows a combination of gradients + color thresholding for detecting edges. A combination of x and y gradients would work for detecting edges and the color thresholding to recognize the lane coloring. I found the gradients to not be necessary. The gradient code is in a separate notebook; lecturenotes. Gradient code would be more useful for sharper curves where the magnitude would be more important than a horizontal or vertical gradient. Blending R and L channels was enough using the test images as a metric. This assumes the 6 test images are representative of the video input. The output of 2 test images are shown below; the rest are in the notebook. The key to determining which are sufficient is to compare visually which combination of thresholded images matches the original without removing data which would cause the curvature calculation to be incorrect.





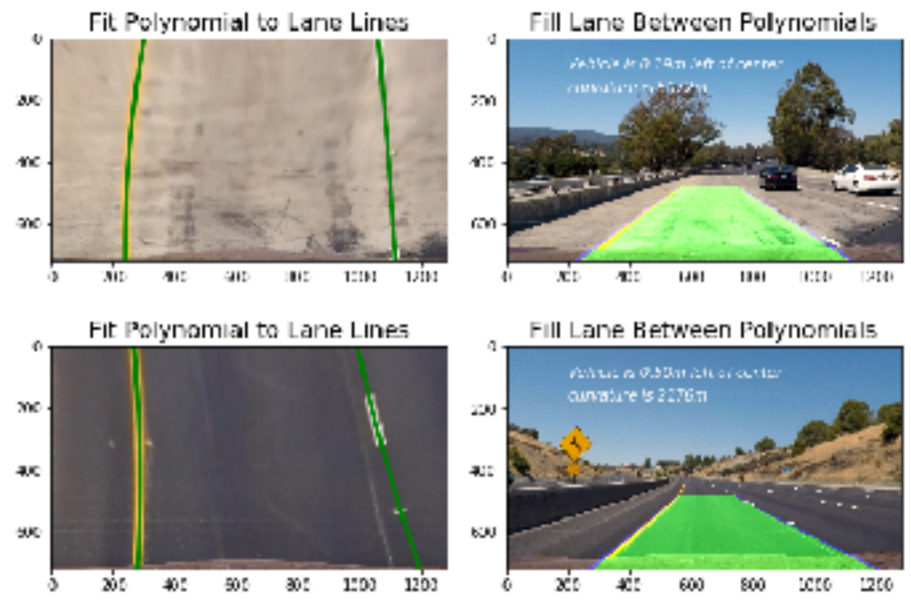
Apply a perspective transform to rectify binary image ("birds-eye view").

The birds eye view is calculated using the ROI shown in blue in the test images below. The ROI is supposed to bound the lane lines as in the first assignment on lane lines. The unwarping is done by assigning points from the ROI to a square which is smaller than the display window. The BEV is shown in the images below; the rest are in the notebook.



Detect lane pixels and fit to find the lane boundary.

Detecting lane pixels follows the lecture code; where we generate a histogram to find the left and right lanes then use a sliding window to find the pixels in the sliding window and uses lane centering to move the search window to the middle. The risk with this approach is it assumes the lanes are within the hard coded margins. This works great for the sample demo but for cases like U-turns or lane changing scenarios a different algorithm may be required. Test cases shown below; rest in notebook. Code copied from lecture examples. Not much work in this case except cut and paste.



Determine the curvature of the lane and vehicle position with respect to center.

The curvature is calculated from the polynomial fit. Then the coefficients from the polyfit are used to draw the curve onto the warped space image. The vehicle position is determined by splitting lhe x coordiante in half from the left/right x coordiante.

Warp the detected lane boundaries back onto the original image.

The distortion transform matrix from the step after the camera calibration is used to reverse the transform. Use another polyfit in the undistorted space to draw the lane lines.

Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

The video is calculated and the link is under result.mp4 and challenge_result.mp4.