Camera Matrix and Distortion Coefficients

We are provided with 20 chessboard images for camera calibration purpose. We used number of corner in x (i.e. nx) = 9, and number of corner in y (i.e. ny) = 6. First off, we find chessboard corners in these 20 images. If found, these corners are stored as “image-points”.

We assume “object-points” to be present on a uniform nx x ny grid. Furthermore, we use OpenCV’s function *calibrateCamera* to find the Camera matrix and Distortion coefficient from the above computed “image-points” and “object-points”.

We can call OpenCV’s function *undistort* to un-distort any given image using the above Camera matrix and Distortion coefficients.

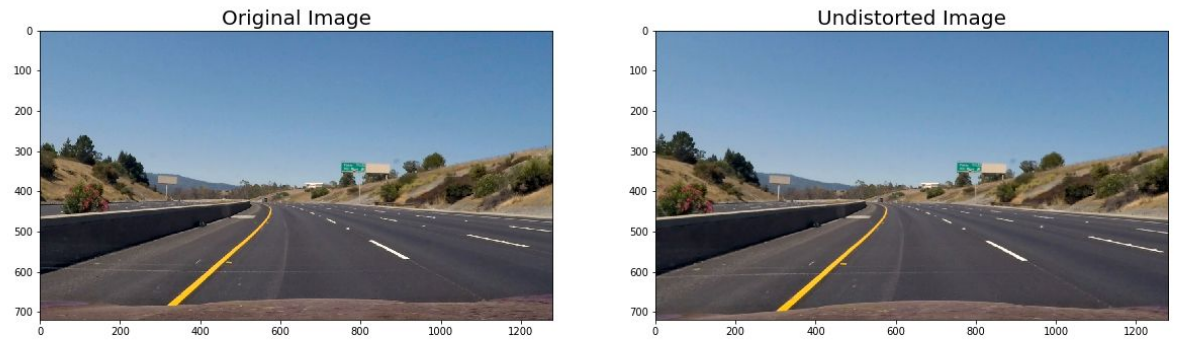
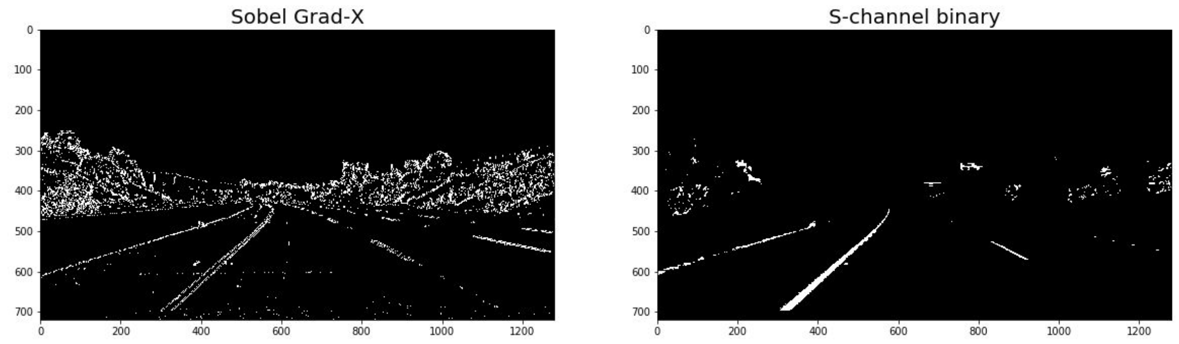
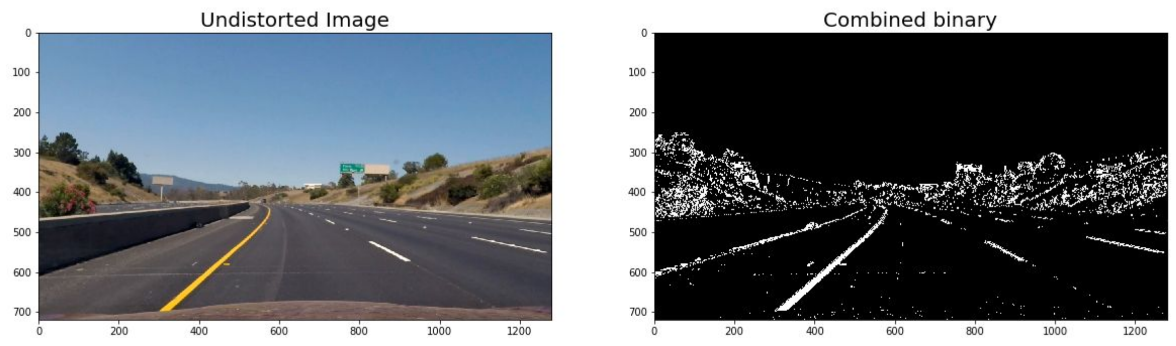


Image Binarization

I tested with sobel gradient in x/y direction, magnitude of result of sobel operator, direction of result of sobel operator, S-channel thresholding to create binary images with clear lane lines. Finally, I settled for a combination of thresholded Sobel-GradX (thresh=(20, 100)) and S-channel (thresh=(150, 255)) images.

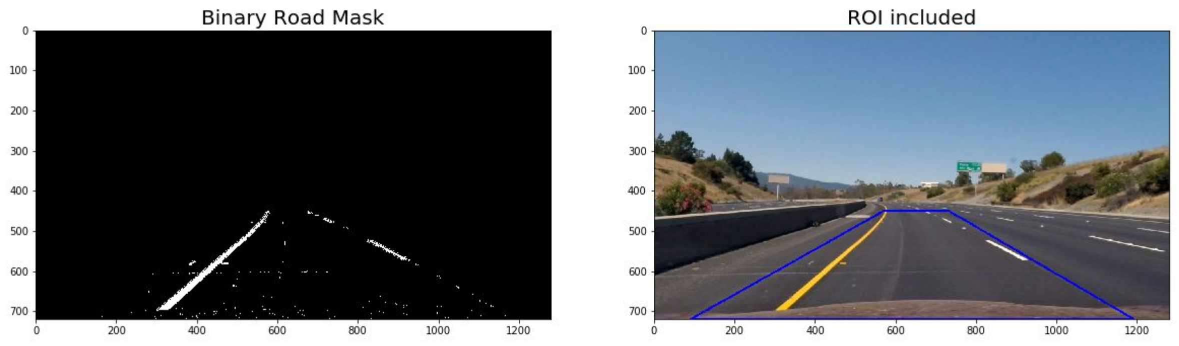
Any of the detected edge pixel in the above 2 images was accepted as edge in the final result.





ROI Selection

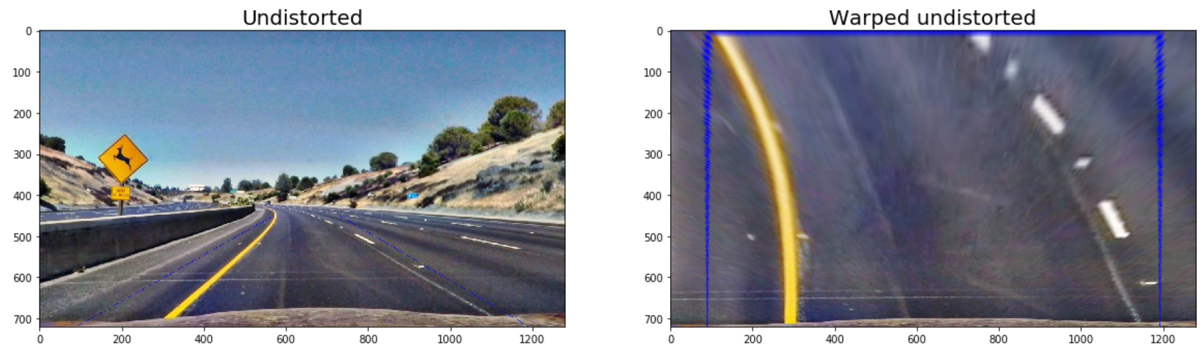
I selected ROI from the above combined-binary image which potentially contained the road lanes. These ROI coordinates were simply found experimentally. The resultant image was a crop from the original combined-binary image.



Perspective Transform for “Bird-Eye View” or “Top-Down Image”

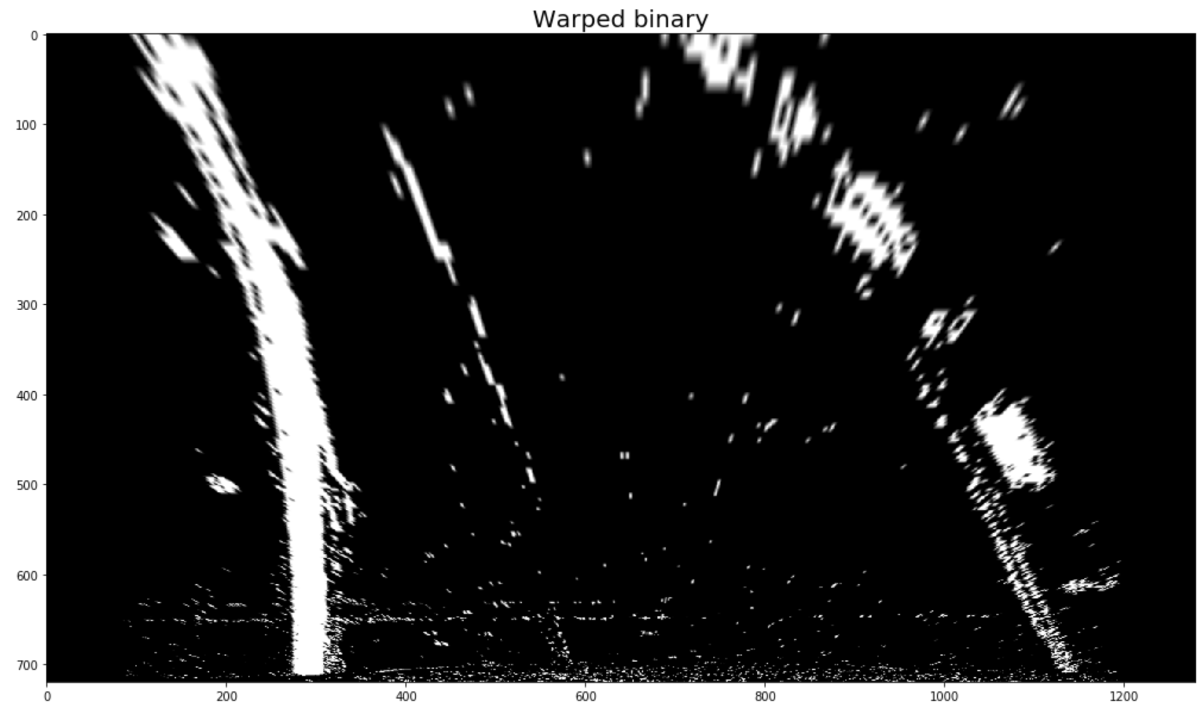
As we can see the above image, the distance between the both lanes seems to be decreasing as we see ahead in the image. We aim to apply perspective transform to negate this effect. Since, we shall compute the curvature of both lanes and we need the top-down image to compute the curvature.

We picked the 4 extreme lane points from the above shown ROI image as “source-points”. Next, we picked 4 corners of an upright rectangle as “destination-points”. Now, we used OpenCV’s function warpPerspective to warp the binary masked road image. This gives us results as shown below:



Identifying Lane Pixels

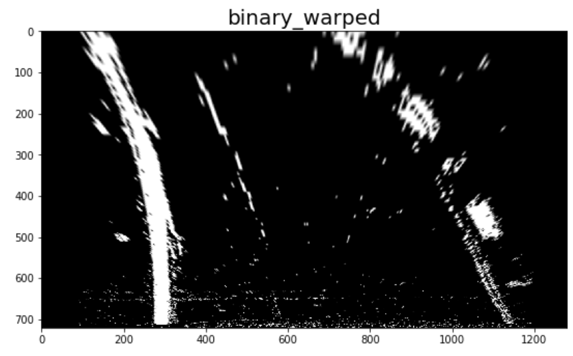
A binary warped image looks like as below:

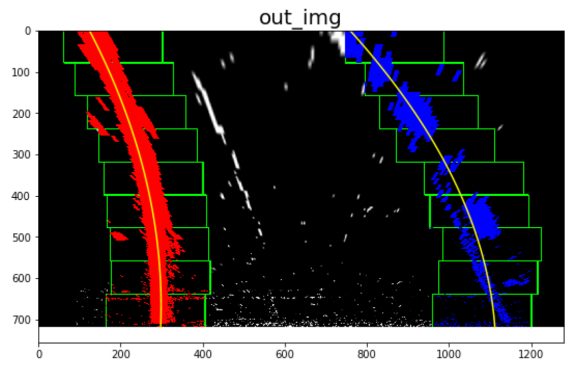


We compute histogram of X-axis projection of the above image. This given us two potential peaks along the X-axis. The first peak represent the left-lane and the next peak represents the right-lane. The 2 peaks of give us the coordinates of the two “starting-points” for both the lanes present in this image.

We divided the whole image into 9 parallel (to X-axis) windows. In each window, we take a fixed ROI around the “starting-point” computed above. We identify non-zero pixels within this ROI. These pixels are accepted as valid “lane-pixels”.

Furthermore, we fit a second-degree polynomial on the “lane-pixels” of left and right lanes independently. We can see the ROIs as green rectangles in the image below. The right-lane pixels as shown in blue and left-lane pixels are shown in red. The polynomial fit is shown as a thin yellow line.





Although, this approach of using ROIs is very costly and we do not use it for every image frame. Once we have employed this approach on an image frame, we know the lane pixels. We use this information while processing the next frame looking for lane pixels. The polynomial fit will not change much from one frame to other. We take a fixed-ROI in the current frame around the previous polynomial fit. Next, we search for non-zero pixels within this ROI. These pixels are accepted as valid lane pixels for the current frame. Now, the next step of fitting a polynomial through these lane pixels is same as before.

Finding Radius of Curvature

We find the radius of curvature at the mean Y-point in each image. Since we know the equation of the polynomial representing the each lane in the image. We know the coefficients A, B, and C.

f(y) = Ay^2 + By + C

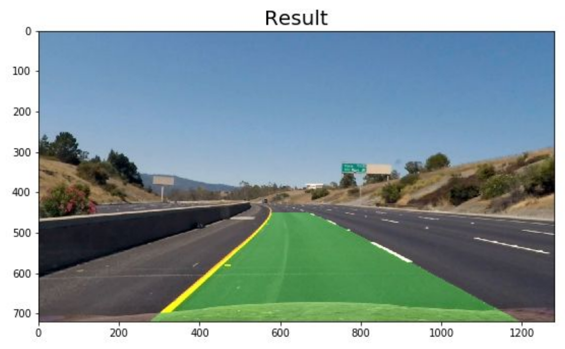
Where A gives you the curvature of the lane line, B gives you the heading or direction that the line is pointing, and C gives you the position of the line based on how far away it is from the very left of an image (y = 0).

The radius of curvature at any point x of the function above is given as follows:
Rcurve=[1+(dydx)2]3/2 / ∣d2x/dy2∣
In the case of the second order polynomial above, the first and second derivatives are:
f”(y) = dx/dy =2Ay+B
f’’(y)= d2x/dy2 =2A
So, our equation for radius of curvature becomes: Rcurve=(1+(2Ay+B)2)3/2/∣2A∣

We can simply put A,B, and C in the above equation to find out the radius of curvature for any given value of y.

Plotting the result back into the original space

We apply the inverse Perspective Transform to map the lane pixels into the original space.



Position of vehicle from the center
Since the camera is mounted at the center of the car, such that the lane center is the midpoint at the bottom of the image between the two lanes detected. The offset of the lane center from the center of the image (converted from pixels to meters) is the distance from the center of the lane.

I used the following expression to compute the offset from center:
Xoffset = (Xright - Xcenter) - (Xcenter - Xleft)

where,
Xright = x-coordinate of right lane
Xleft = x-coordinate of left lane
Xcenter = x-coordinate of center of image

Tricks and Tips
I picked up some tricks from this blog:
https://medium.com/@heratypaul/udacity-sdcnd-advanced-lane-finding-45012da5ca7d

1. I do some checks to determine if the calculated lane if 'good' or not.
    a. the radius of curvature of the both the lanes are in close proximity of the actual value, i.e. 1000 meters.
    b. the radius of curvature of the both the lanes are in close proximity of each other.
    c. the radius of curvature of the both the lanes are in close proximity of the corresponding lanes from previous frame.
    d. the radius of curvature of the both the lanes are within 100 times (larger or smaller) than the previous frames values. As the RoC can be very large for vertical lines, I found that check for bounds of 100x seemed to work pretty well

2. I see that the left_x and right_x coordinates, essentially mean x-coordinate of each lane, have not changed too much from previous frame. At 30 frames per second from a camera, the lanes should not change too much from frame to frame. I found that checking for a 25 pixel delta worked well.

3. If any of the above checks fail for a lane, I consider the lane 'lost' or undetected for that frame, and I use the values from the previous frame. Again, at 30fps this should be ok for a short number of frames as the lanes will not change too much between successive frames. If a lane has not been successfully detected for **5 frames**, then I trigger a full scan for detecting the lanes using window-histogram technique used for the very first frame.

Discussion
Few possible improvements in my implementation:

1. One useful hack would be to use moving average or exponential smoothing type of algorithm over past few frames for computing polynomial fit for a new frame. This would ensure smooth lane in successive frames.
2. The coordinates for finding ROI (potential image area containing lanes) is hard-coded and was found experimentally. This step needs to be automated.