Advanced Lane Finding

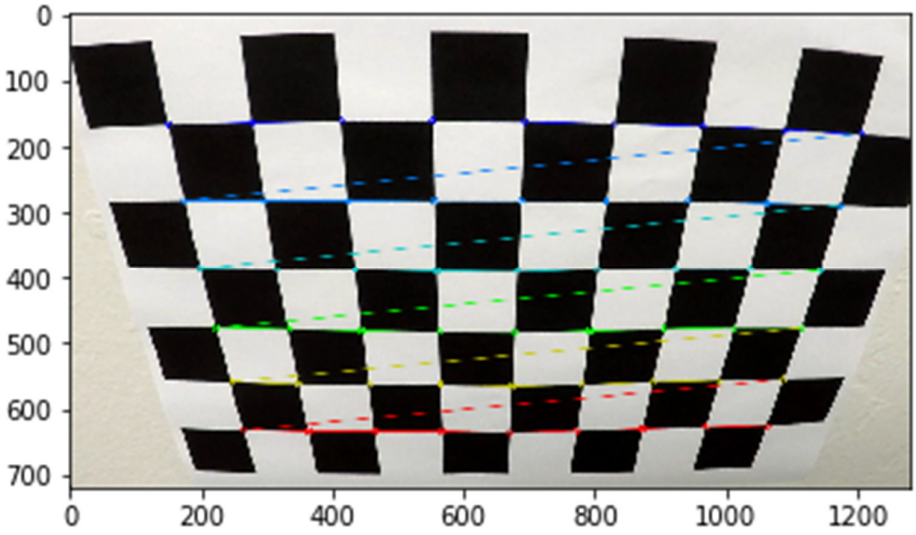
The goals / steps of this project are the following:

- 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 and Distortion Coefficients

The code for this step is in the 3rd and 4th cells in the Jupyter notebook AdvancedLaneFinding.ipynb.

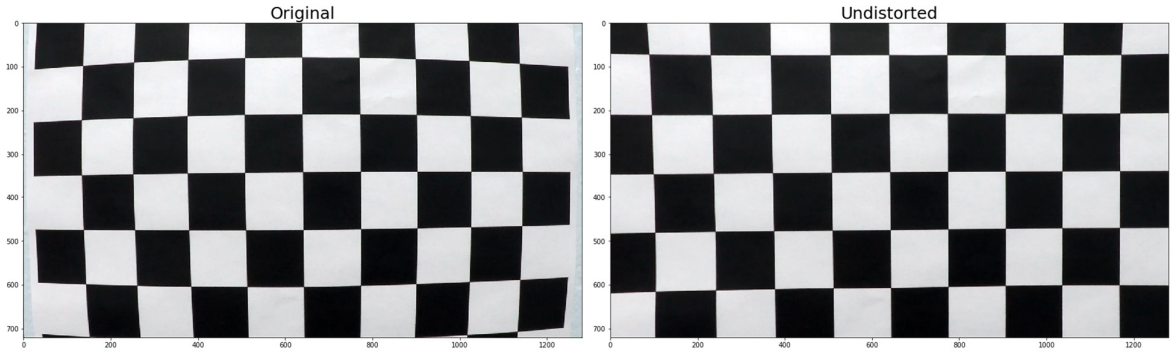
This code loads calibration images of chessboards taken at various angles. First the image is converted to grayscale and then the chessboard corners are found using cv2.findChessboardCorners(). This results in object points which are the x, y, z coordinates of the chessboard corners.



Next the corner points and object points are used to calibrate the camera using cv2.calibrateCamera().

Distortion correction is applied using the function cv2.undistort() using the calibrated image from the previous step.

Here is the undistorted output:



Applying distortion correction to raw images

This is cell #5 in AdvancedLaneFinding.ipynb, titled “Distortion correction and warped image”.

Here I used the corners_unwarp() function similar to lesson 6 chapter 18.

Using the camera matrix and distortion coefficients from the previous calibration step, here is the undistorted and warped result:

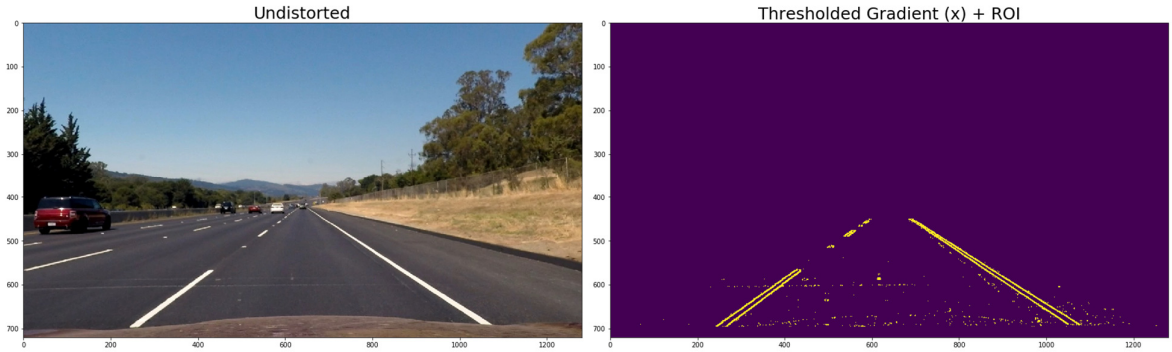
Using color transforms, gradients, etc. to create thresholded binary images

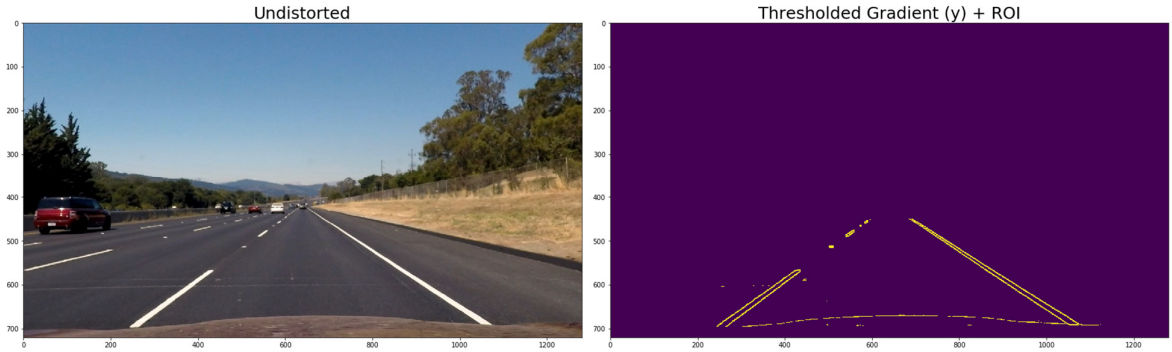
I created the binary image using a combination of color and gradient thresholds. The functions are in the 9th cell titled Edge Detection functions – gradients and color transforms.

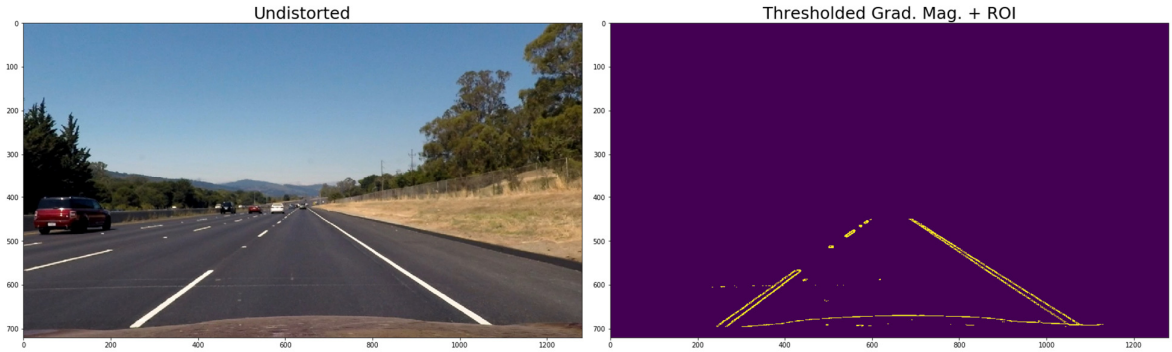
These functions find the absolute sobel thresholds, magnitude of the gradient, direction of the gradient, and color transform applying Saturation thresholds in the HLS space, similar to what we learned in lesson 7 chapter 10.

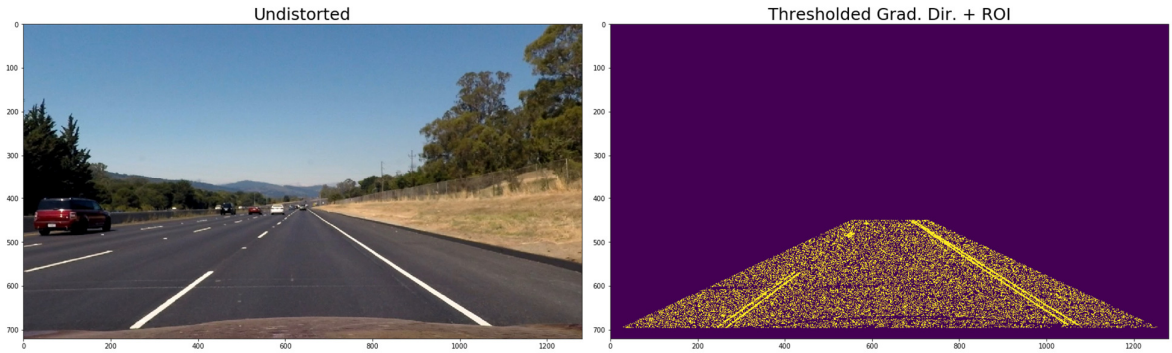
I defined a function region_of_interest() in cell #7. This function defines a region of interest where we think the lane lines will be, considering that the camera is mounted in the center of view.

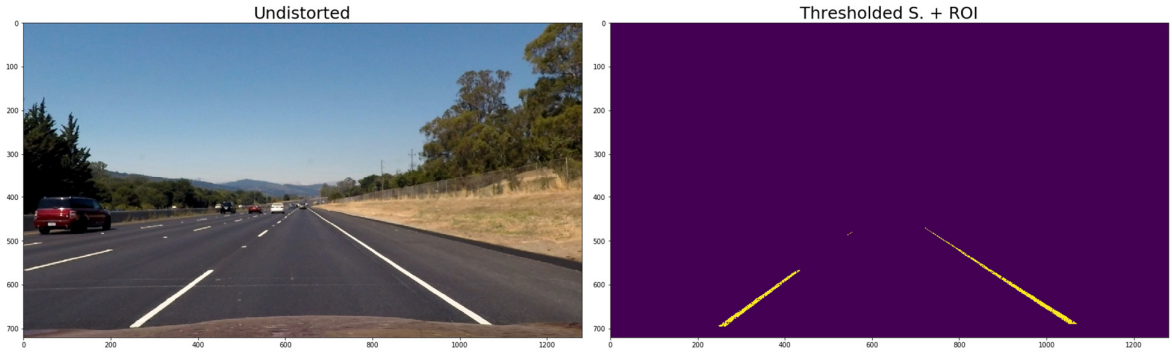
Outputs of this step are as below. Although all the functions to calculate gradients convert the image to grayscale, I am not sure why the result image does not show in grayscale but shows as purple and yellow instead. I spent a lot of time trying to understand why but eventually decided to just proceed with the rest of the project.


Undistorted
Thresholded Gradient (x) + ROI


Undistorted
Thresholded Gradient (y) + ROI


Undistorted
Thresholded Grad. Mag. + ROI


Undistorted
Thresholded Grad. Dir. + ROI

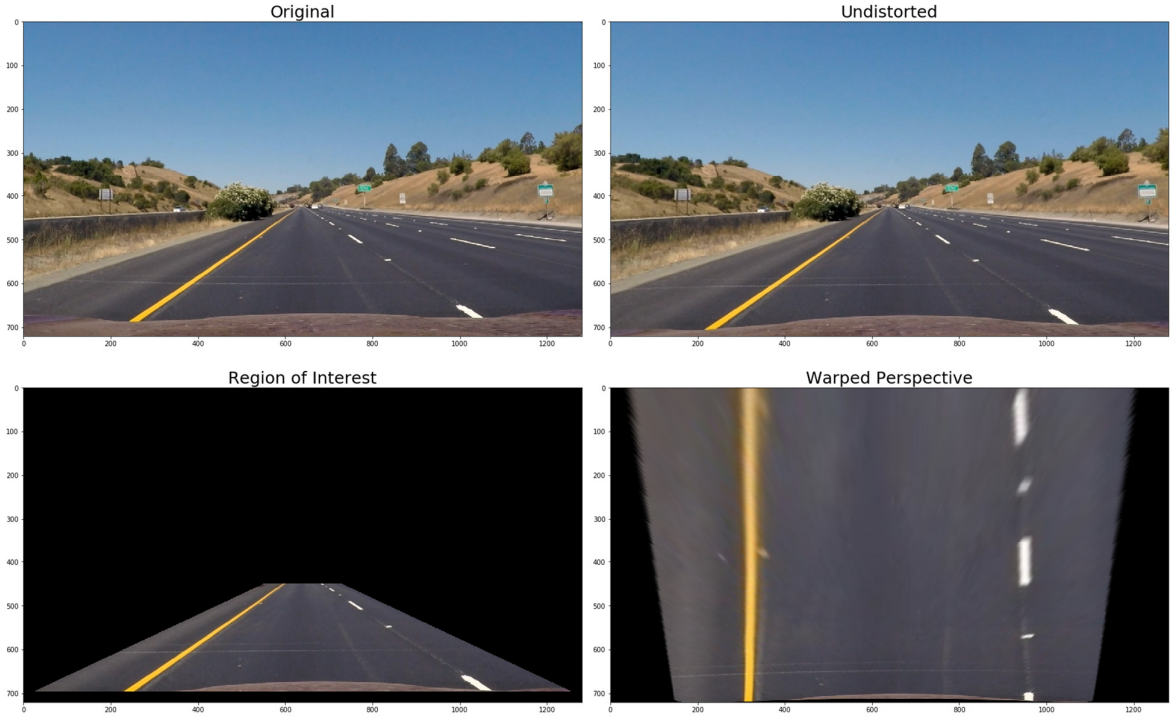


Perspective Transform and birds eye view

This is in cell #8 titled Perspective transform.

Here, the get_perspective() first applies distortion correction using cv2.undistort(), then applies a perspective transform using cv2.getPerspectiveTransform() to get the transform matrix M, and finally uses cv2.warpPerspective() to apply M and warps the image to birds eye view.

The image is read, and region of interest is applied to only consider the area where lane lines can be present in the field of view of the camera. Then the get_perspective() function is called to create the birds eye view as shown below:

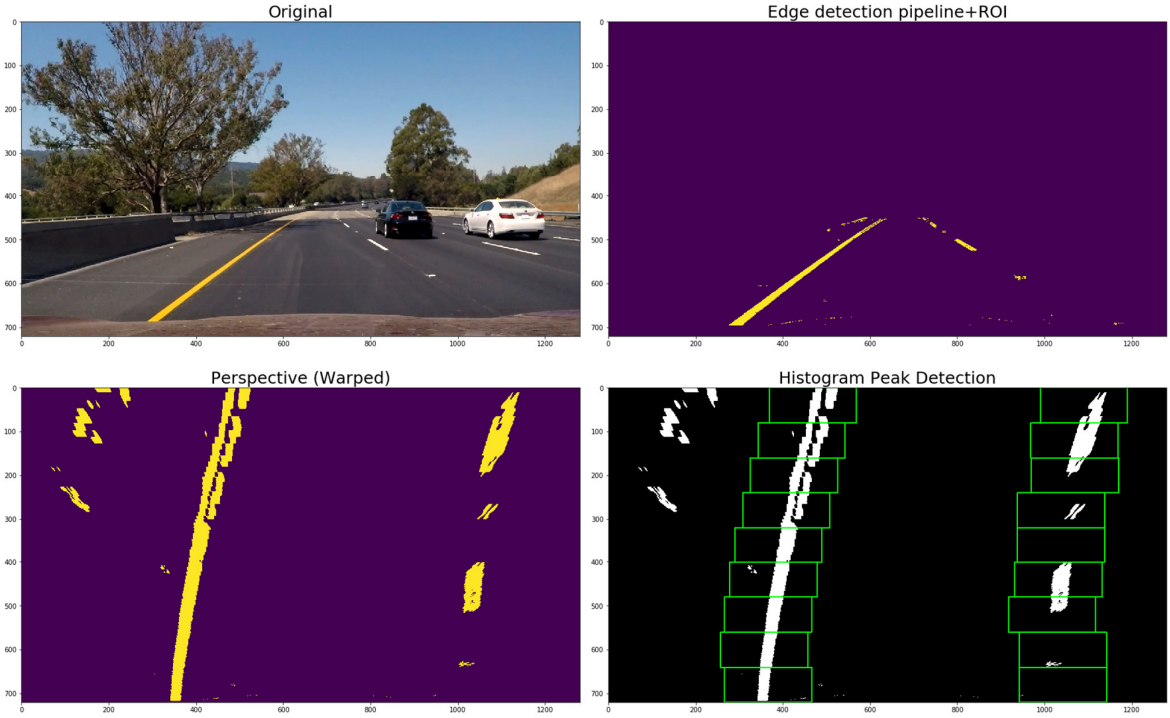


Detect lane pixels and fit to lane boundary

I created a function edge_detection() that combines the above edge detection functions to create a pipeline. This returns the thresholded binary image.

Next, the hist() function reads in the binary thresholded warped image to find the peaks of the left and right halves of the histogram to be used as starting points for the left and right lane lines. This function uses 9 sliding windows, as done in lesson 8 chapter 4.

The output is as below:



Complete pipeline

The complete pipeline is in cell #14. This cell finds radius of curvature, fits polynomials and finds center position.

Here are the steps in this pipeline:

1. Read in an image and correct distortion using cv2.undistort()
2. Apply the edge detection pipeline
3. Apply the region of interest over the lane lines
4. Get the birds eye view, i.e. convert to image space
5. Fit a 2nd order polynomial to pixel positions using np.polyfit() with conversions from pixel space to meters. The polynomial used is from lesson 8 chapter 6.
   $f(y)=Ay2+By+C$
6. Radius of curvature is calculated using the derivatives of above equation
   $Rcurve=|2A|(1+(2Ay+B)2)3/2$
7. To determine the position of the vehicle with respect to center, I first estimated mean x values of left and right lane points to get the lane center. Then, using half the x value of warped image shape as vehicle center, and subtract the previous lane center to get the position of vehicle with respect to center. This part is in the lower half of cell #14.
8. Finally, draw the lane onto the warped blank image.

Output of complete pipeline is as below:

Project video

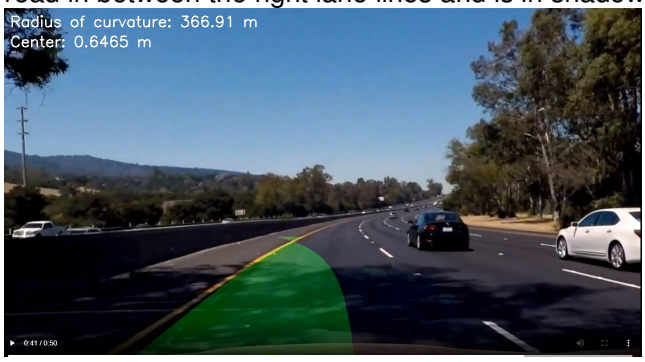
Video is titled project_video_out.mp4

Limitations and potential improvements

1. As seen from the video, the lane lines on the left are jittery at seconds 21 to 24. This is because the left lane line is not visible to the camera due to the lane line being faded.

In this case, it may be possible to improve detection using more than 9 rolling windows, however this will increase the amount of processing required.

2. At the 41 seconds mark, the pipeline seems to have a major glitch. This is the part of the road in between the right lane lines and is in shadow.



This can possibly be improved by fine tuning parameters according to the image, or use a longer buffer to keep historical positions and check for plausibility.