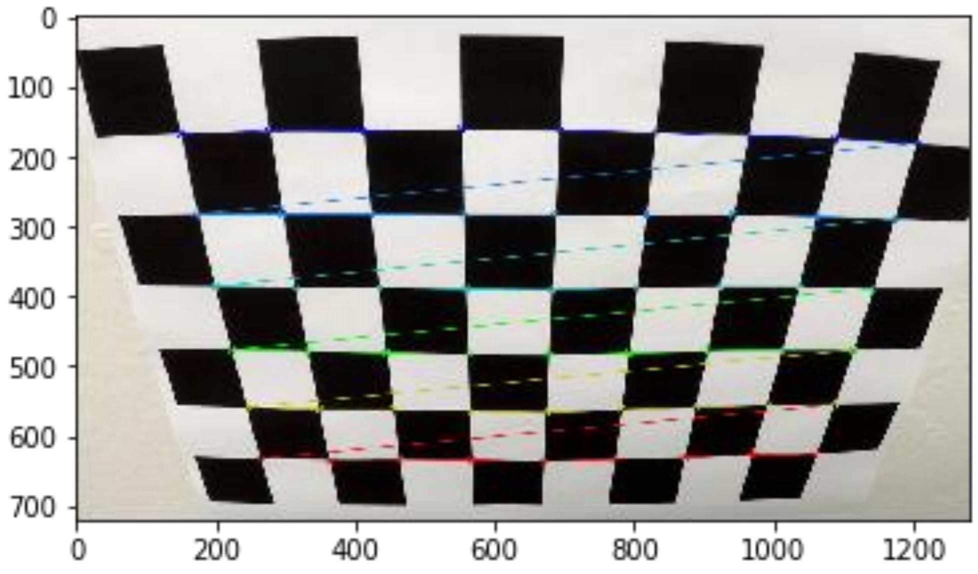
# Advanced Lane Finding

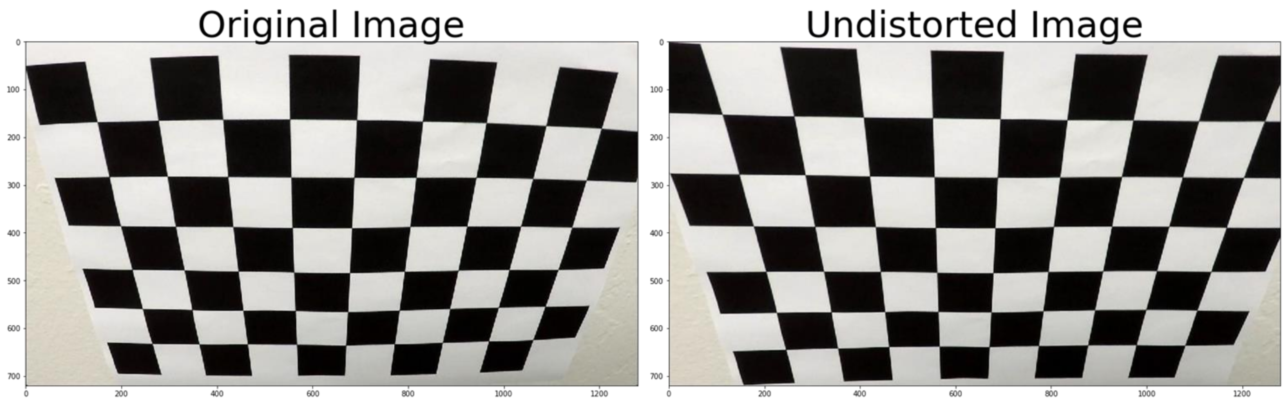
## Udacity – Self Driving Car Nanodegree

### Camera Calibration and Undistortion

The code for this step (including test functions) is contained in the 3rd - 6th code cells of the iPython notebook.
I use the example code from the lecture. I start off by preparing the object points, which will be the (x,y,z) coordinates of the chessboard corners in the world. Here the assumption is that the chessboard is fixed on the (x,y) plane at z = 0, such that these object points are the same for each calibration image. Thus ‘objp’ is just a replicated array of coordinates, and ‘objpoints’ will be appended with the copy of it each time all chessboard corners are successfully detected in a test image. ‘imgpoints’ will be appended with the (x,y) pixel position of each of the corners in the image plane with each successful chessboard detection.



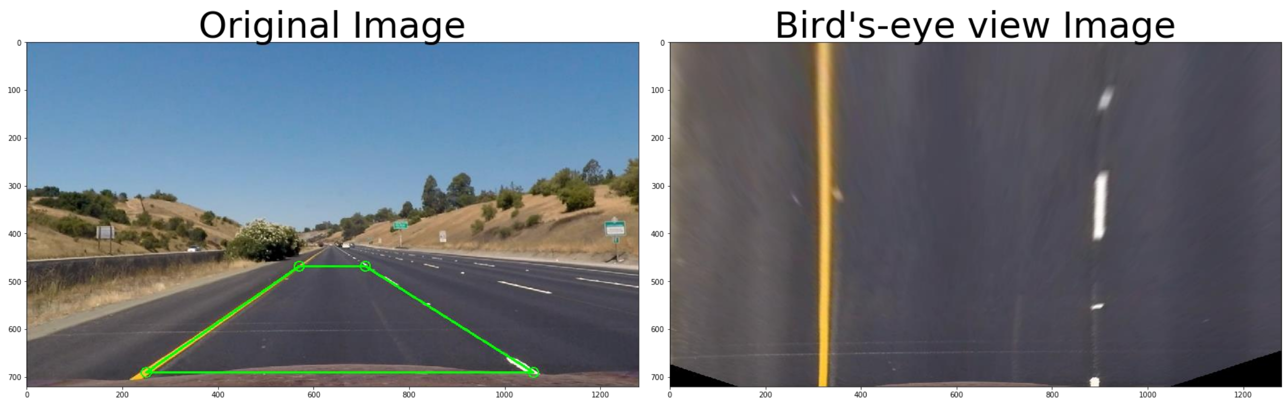
After identifying the corners I can calibrate the camera. To do that I use opencv's calibrateCamera() function, I pass the previously calculated objpoints, imgpoints and the shape of the image as arguments. The function returns the distortion coefficients and the camera matrix as well which I need to pass as arguments to the undistort() function which returns my undistorted image. Here is an example of image undistortion. On the left-hand side, the original image is shown, on the right-hand side the undistorted image.



Let’s see one example of the before/after undistortion function on real-world test images.

## Perspective Transform

The function for Perspective Transform can be found in the iPython notebook code cell Nr. 7 - 8, called warp.
Perspective transform maps the points in each image to a different, desired, image points with a new perspective. We are most interested in the bird's-eye view transform that lets us view a lane from above, this will be useful for calculating the lane curvature.
To compute the perspective transform of an image we need to define the four source coordinates and the four destination coordinates and then use the ‘getPerspectiveTransform()’ function of OpenCV. To apply the perspective transform to an image I use ‘warpPerspective()’ function. I test the function on the undistorted image shown in the previous step and obtained the following output.
The 4 point highlighting on the original image are the source points of the Perspective transformation.

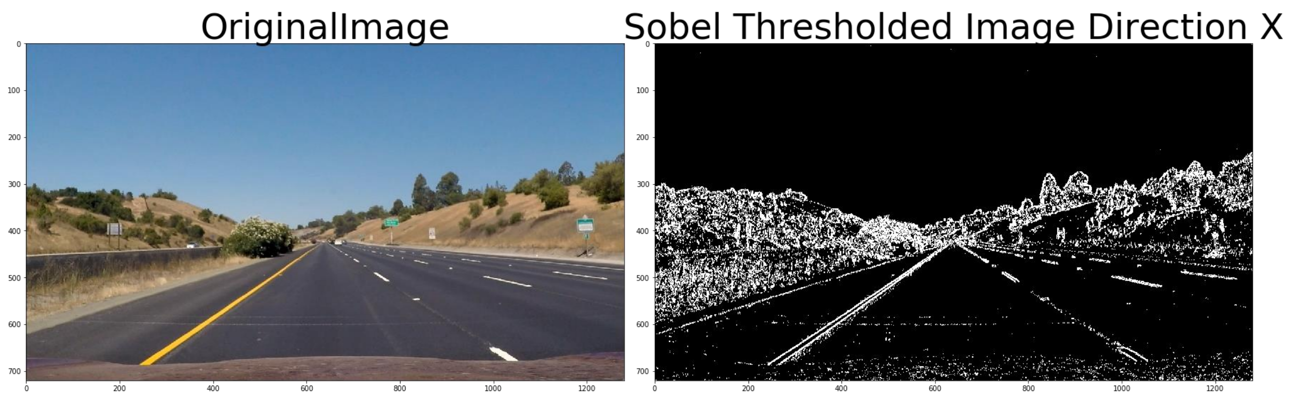


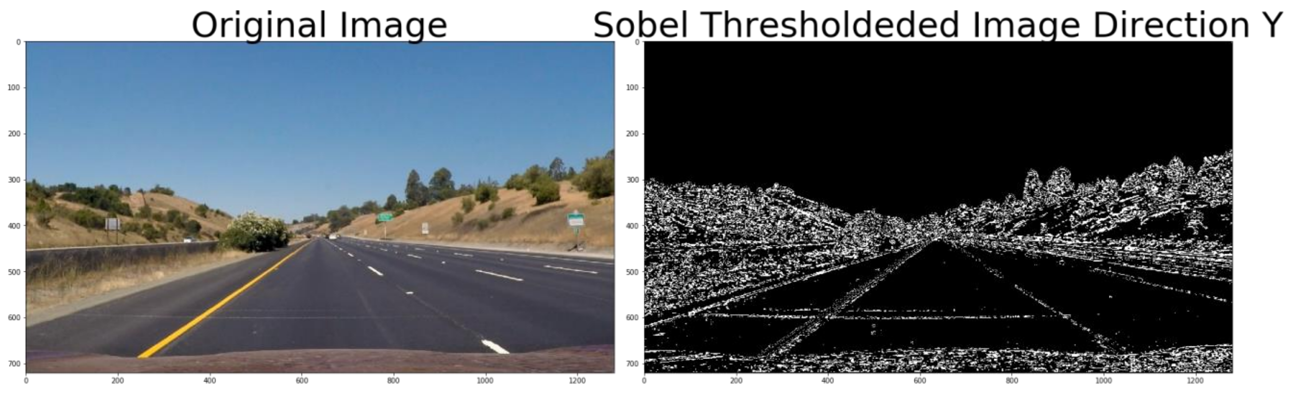
## Thresholding

The goal is the find the Lane Lines on the road, and there are numerous techniques an image can be thresholded to pick/emphasize a feature of it. Let's examine a few of these. The code is commented in the notebook, here I’m going to show only the results of different thresholding methods. The code cells containing the functions and their tests are the cell Nr. 9 – Nr. 18 in the iPython notebook.

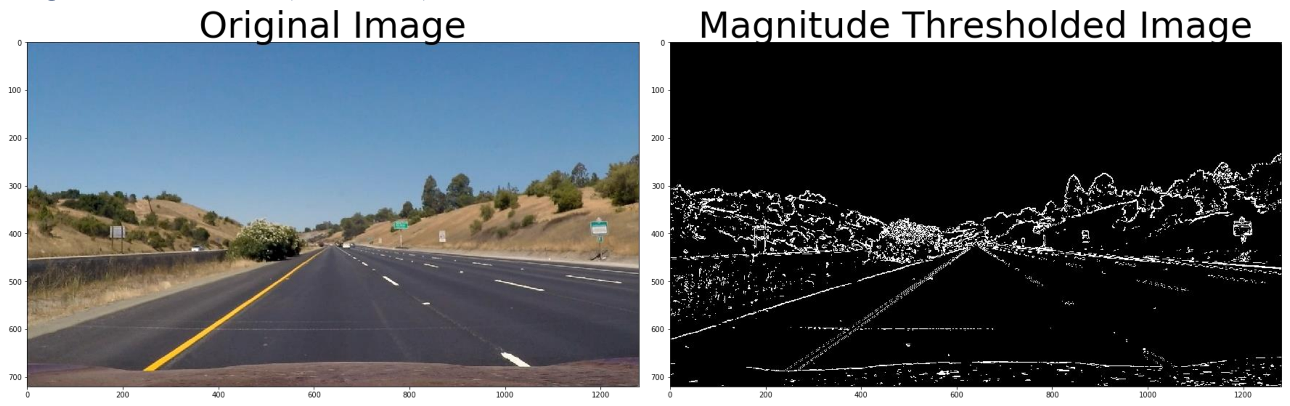
### Sobel Threshold (cell 9-10)

The Sobel operator takes the derivative of an image in either the x or y direction.





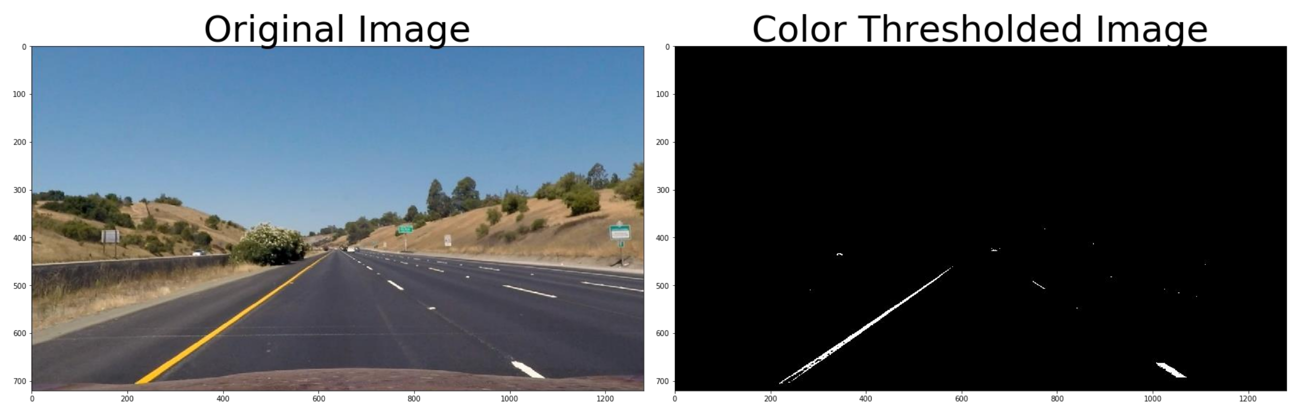
## Magnitude Threshold (cell 11-12)



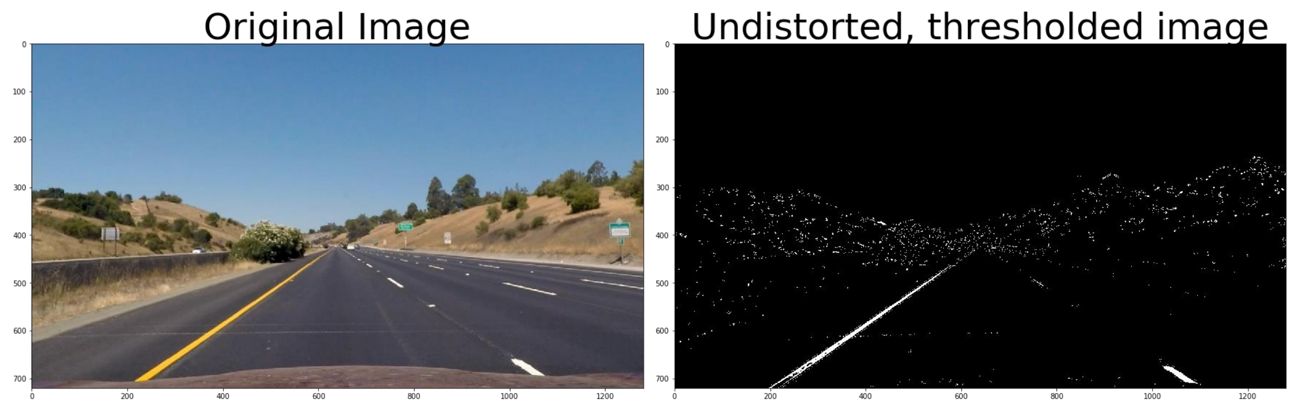
The function returns the magnitude of the gradient for a given sobel kernel size and threshold values.

## Color threshold (cell 15-16)

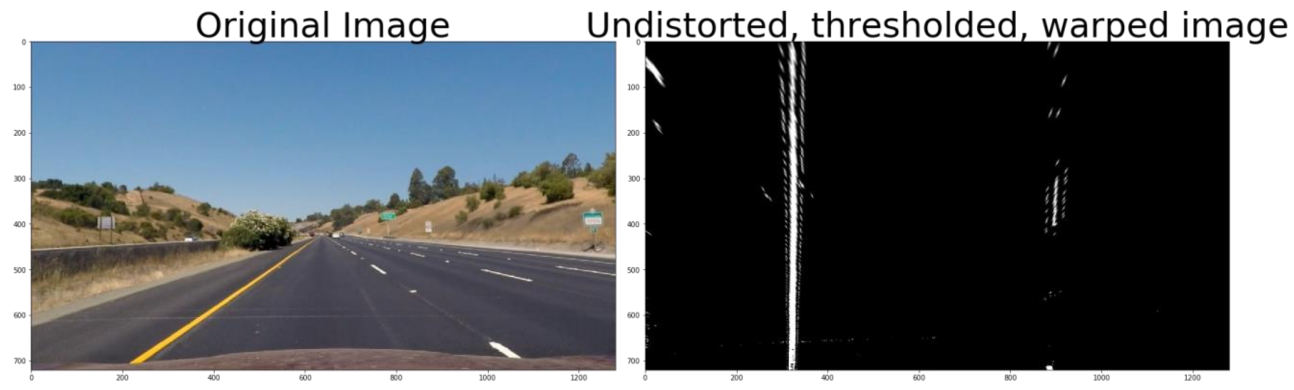
Function that thresholds combined HLS and RGB color channels to get the most out of these color spaces.



## Applying combined thresholds on the image. (cell 17-18)



## And the perspective transform on the thresholded image

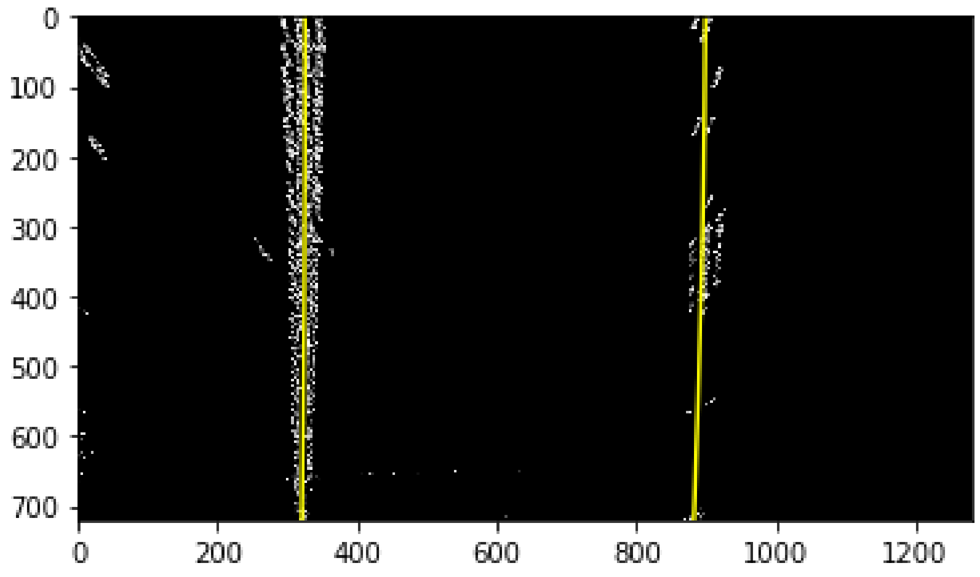


## Identifying lane lines

By now we can see the lane lines on the combined thresholded image. The next step is to fit a polynomial to both lanes lines. I apply a histogram (cell 19) first to the bottom half of the bird's-eye view of the binary threshold image. After splitting the image in halves, the points in both halves with the highest number of pixels are assumed to be the starting points of the left and right lane line respectively.

There are two possibilities we should consider. First, let’s assume we have a warped binary image and we want to find which hot pixels are associated with the lane lines. We don’t have information about the lines from previous frames. (cell Nr. 22 – search_from_scratch())
The code is based on the sample code from the lecture and exhaustively commented.

The other case is when we don't have to do a blind search again but we can just rely on the lanes found in the previous frames and search in a margin around the previous line positions. The code for that is in my final_pipeline() function in code cell Nr. 26. and was inspired by Lesson 33 – Finding the Lanes.



## Measuring curvature

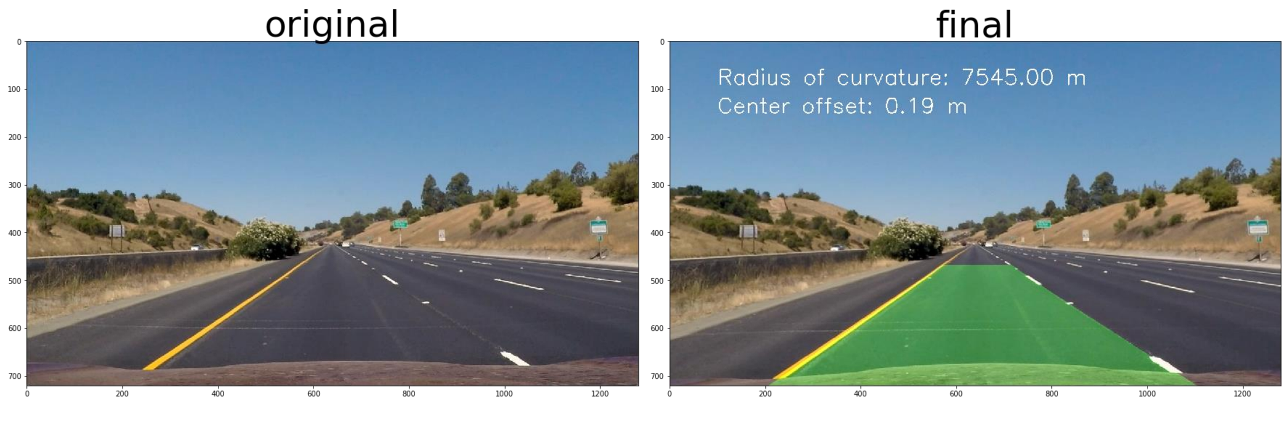
Now that we have a thresholded image with the estimated pixels that belong to the left and right lane lines, and a polynomial has been fit to both of those pixel positions, one of the last steps to perform is computing the radius of the curvature of that polynomial fit. The code for that can be found in code cell Nr. 23 and the code was inspired by Lesson 35 – Measuring the Curvature. The function takes the generated lane lines as argument and returns the curvature of it. I'll apply this curvature calculation on both lines. We should first convert from pixel space to real world space.

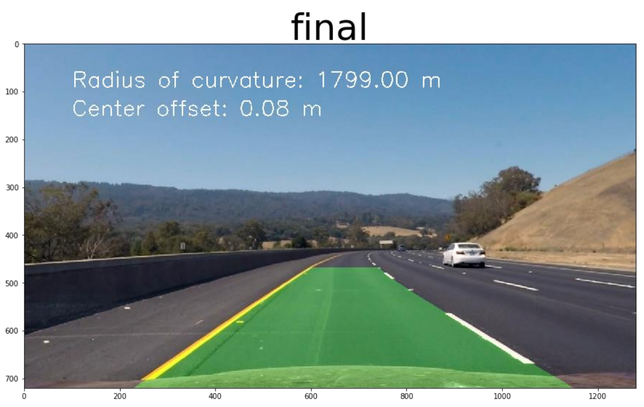
## Smoothing over frames

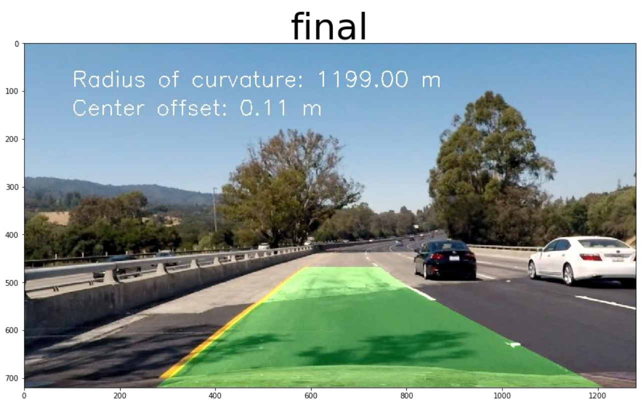
Even when everything is working, the line detections will jump around from frame to frame a bit and it can be preferable to smooth over the last n frames of video to obtain a cleaner result. Each time I get a new high-confidence measurement, I can append it to the list of recent measurements and then take an average over n past measurements to obtain the lane position I want to draw onto the image.

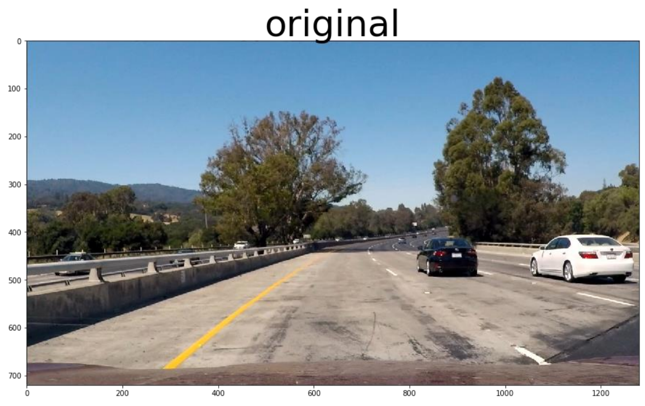
## Results

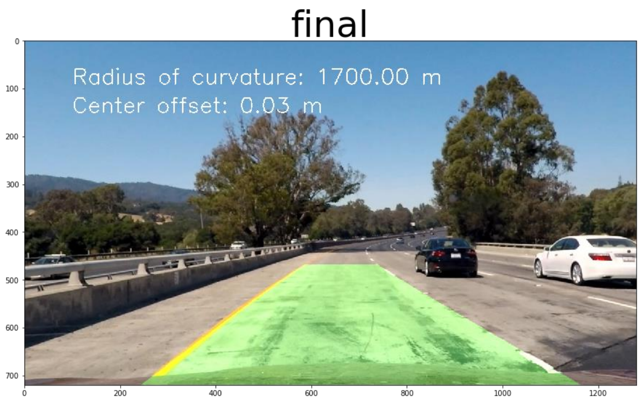
My current results are not too bad in most of the cases, but there is room for improvement. Code cells Nr. 26-31 show some of the projected lane areas.











## Discussion

The task is generally tricky. There are conditions that strongly affect the quality of the feature extraction we use. In difficult weather/lighting conditions we might not anymore rely so much on the visual clues I apply in this project to detect the lane I'm driving in.

It's been challenging the get these results. I had to play around lots of features and their parameters, and there are more ideas I didn't apply but could improve my results. The most important feature must be the color feature and to extract the lane lines – both the white and the yellow one - the best possible way in different lighting conditions I combined HLS and RGB color spaces with thresholds I found the best through my experimentation.