# Writeup

---

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.

## Rubric Points

## Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

---

## Writeup / README

**1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.**

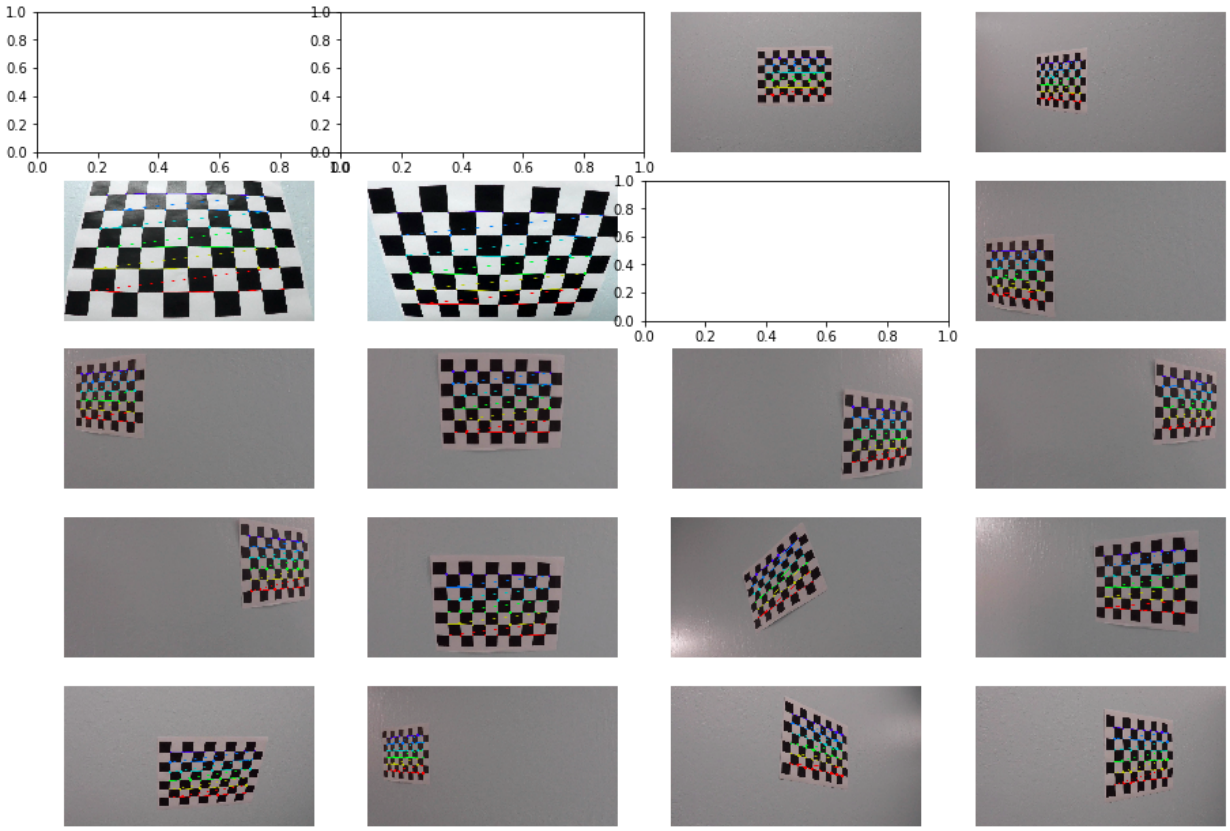
You're reading it!

## Camera Calibration

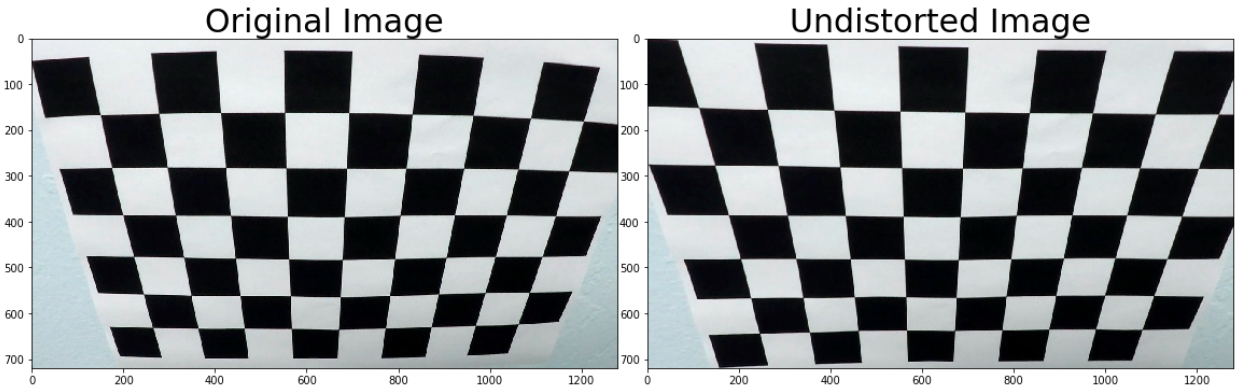
**1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.**

The code for this step is contained in the second [2] code cell of the IPython notebook located in "Project Advanced Lane Lines.ipynb"

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners 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.



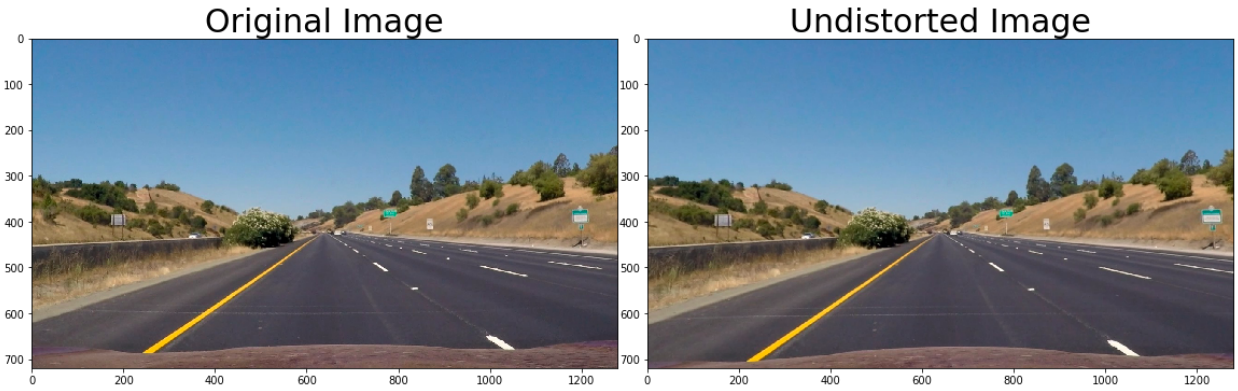
I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result.




# Pipeline (single images)

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

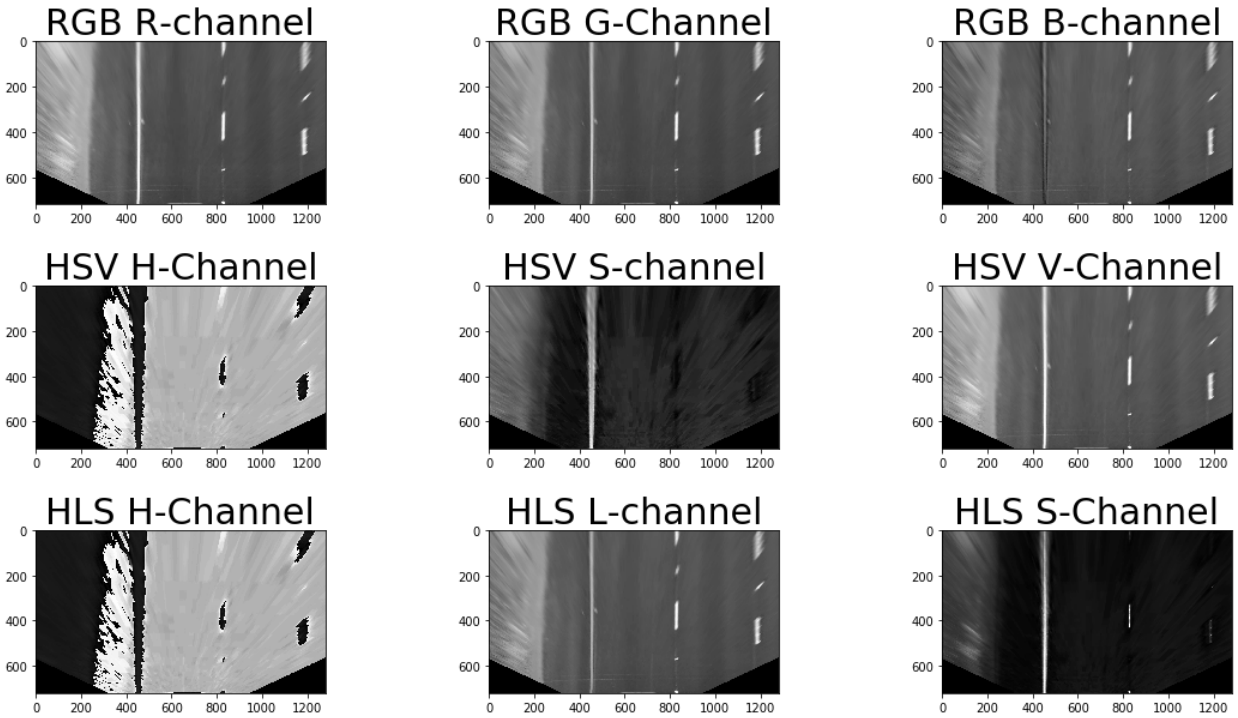
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one: (Code Block 38)




### 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.

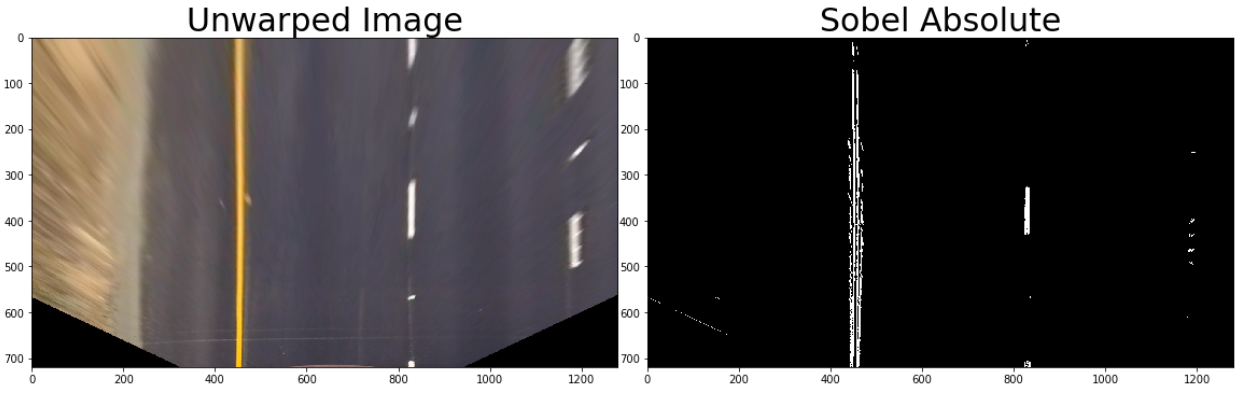
I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at code blocks ). Here's an example of my output for this step. Before this step I chose to do perspective transform and then apply the thresholds.

First I checked various color channels to identify useful channels

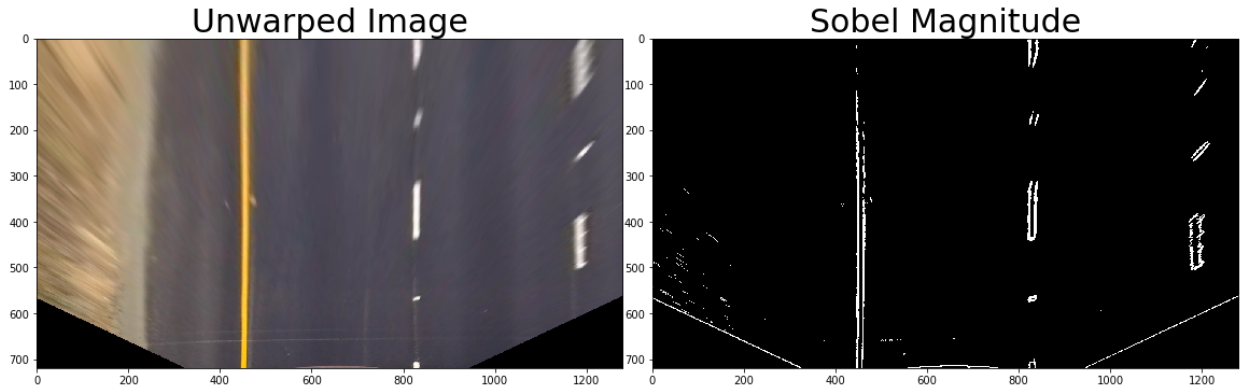



Then I tied different thresholds:

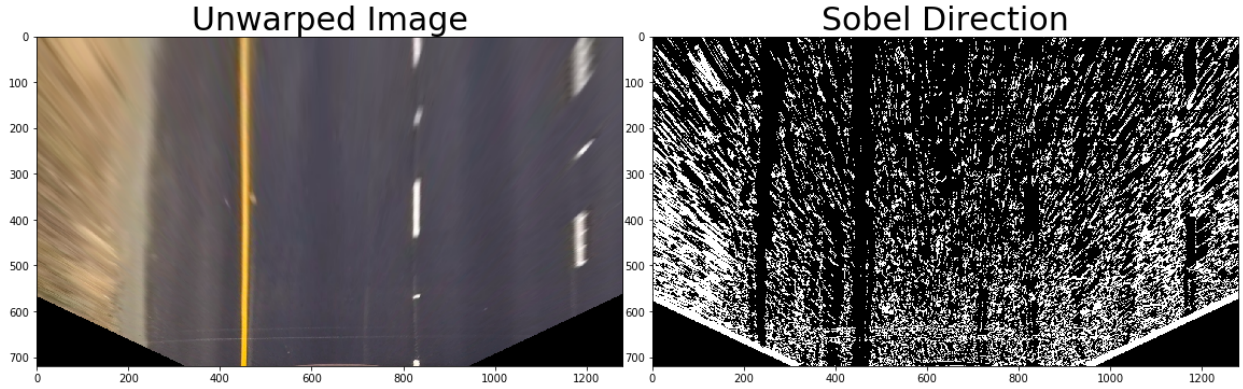
**Absolute Sobel  (Code Block 42, 43)**




**Sobel Magnitude Threshold  (Code Block 44)**

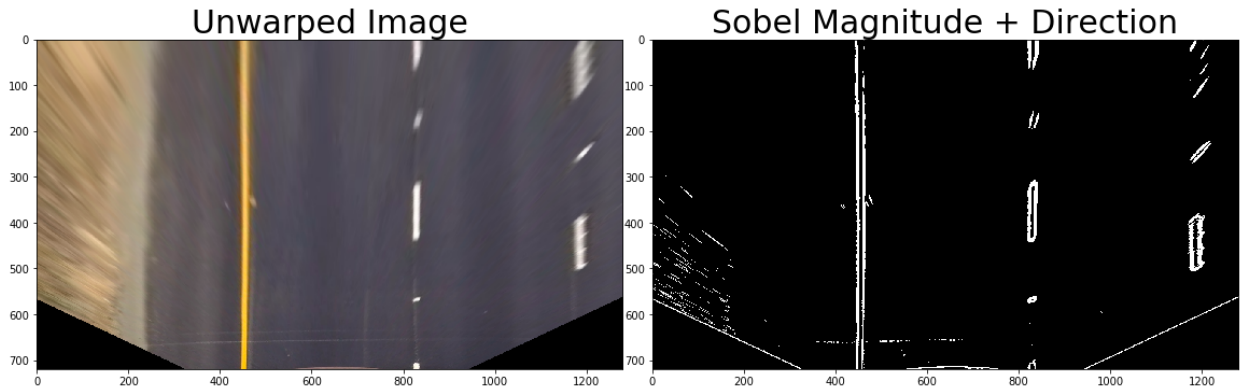



**Sobel Direction Threshold (Code Block 46,47)**

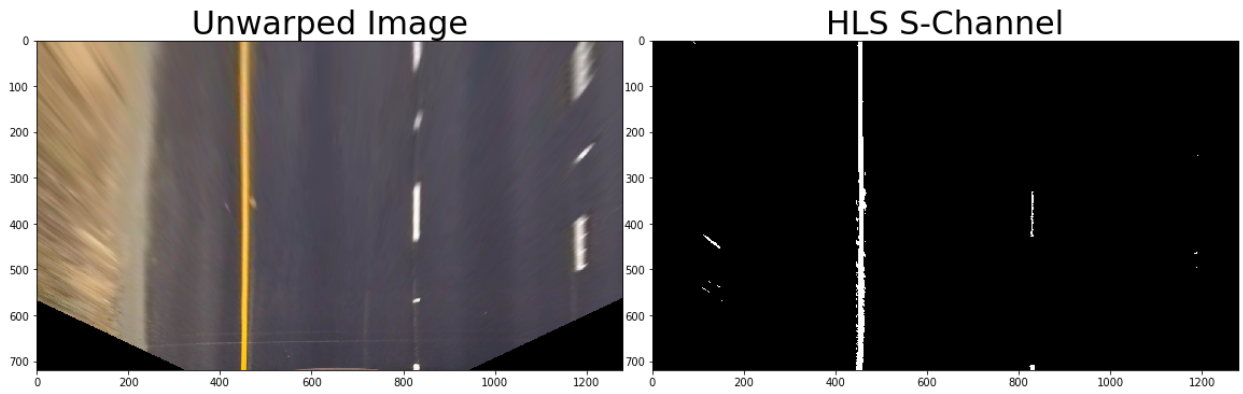



**I still have doubts on correctness of the direction threshold and so I excluded this from my final pipeline.**

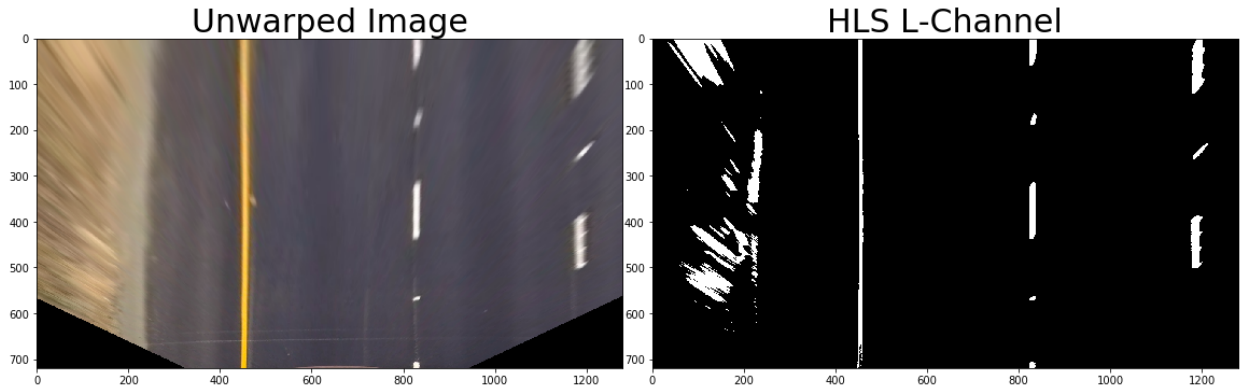
**Sobel Magnitude and Direction Threshold (Code Block 48)**



**HLS S Channel Threshold (49,50)**




**HLS L Channel Threshold (Block 51,53)**



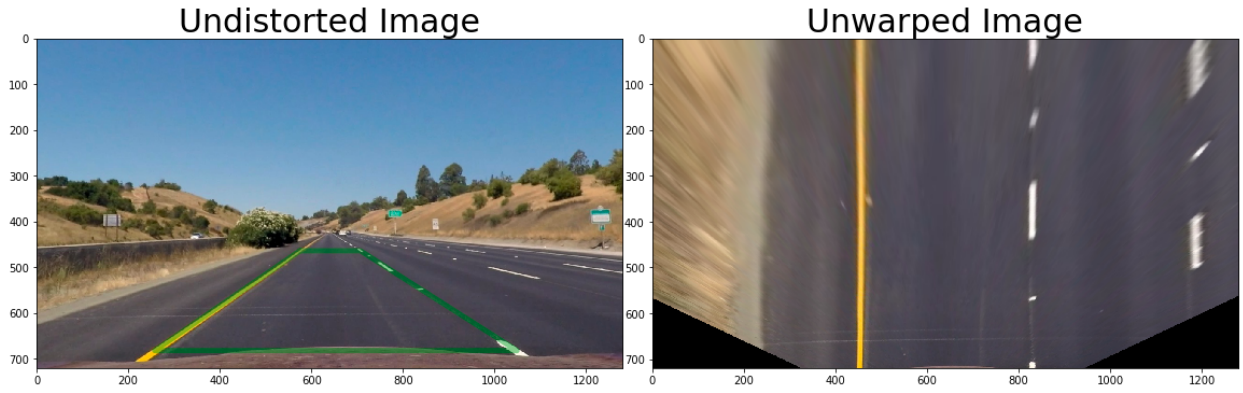

**3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.**

The code for my perspective transform includes a function called unwarp() (Check Code Block 56) . The unwarp() function takes as inputs an image (img), as well as source (src) and destination (dst) points. I chose the hardcode the source and destination points in the following manner:

```
# define source and destination points for transform
src = np.float32([(575,464),
                  (707,464),
                  (258,682),
                  (1049,682)])
dst = np.float32([(450,0),
                  (w-450,0),
```

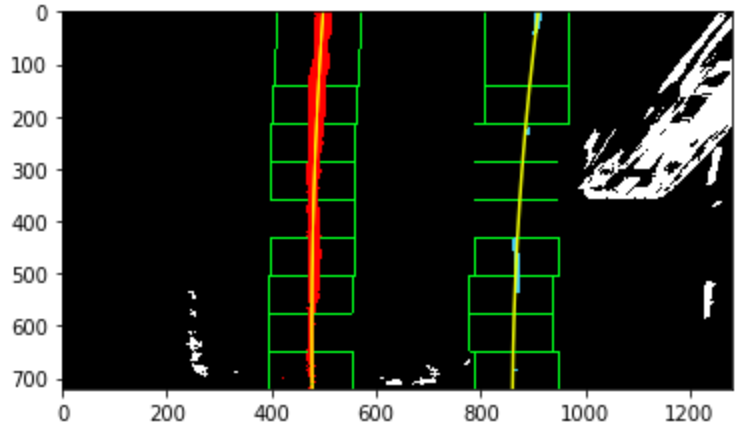
```
                    (450,h),
                    (w-450,h)])
```

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

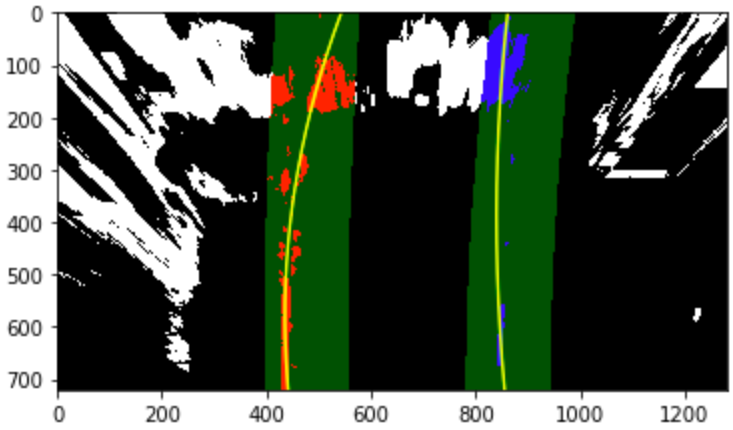



**4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?**

Then I did some other stuff and fit my lane lines with a 2nd order polynomial kinda like the one given below. I used Sliding Window Polyfit to fit a second order polynomial for lane lines (Code Block 79)



Then applied polyfit using previous frame (Code Block 77) to get this:




**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.**

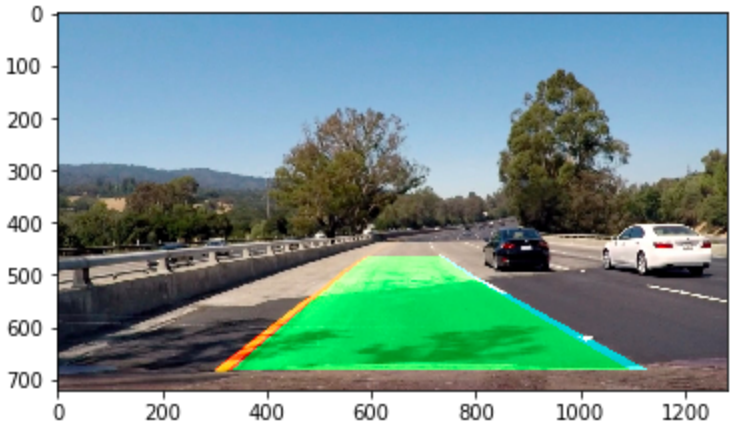
Code Block 81

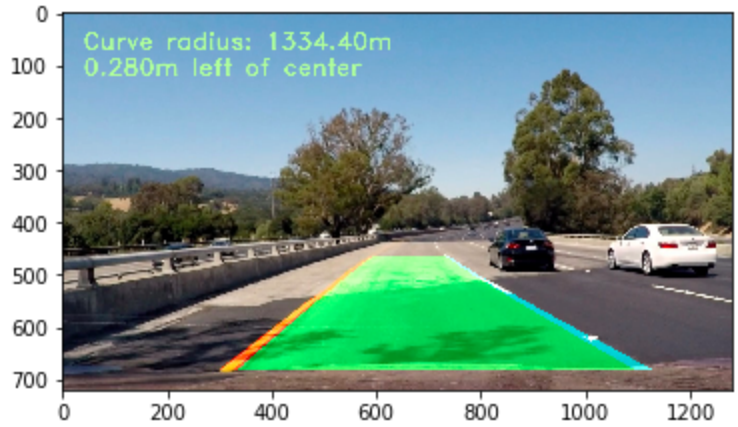
Check on test image

Radius of curvature for example: 1567.45260105 m, 1101.35467311 m
Distance from lane center for example: -0.2795713627 m

**6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.**

Code Block 89






---

## 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!).**

Here's a link to my video result

---

## 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?**

The pipeline fails on challenge video and harder challenge video. I plan to use other color channels and the gradient thresholds to create more robust approach. Here in our case since we are dependent on S and L. When the video has frames with shadow or high brightness due to sun glare, the lane finding fails as the Lightness value creates an issue. In my case the direction threshold does not seem to be working fine and I would like to come back to it later when time permits.