# Advanced Lane Finding Project - Write-up

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.

---

1) Provide a write-up that includes all the rubric points and how you addressed each one.

You're reding it.

# Camera Calibration

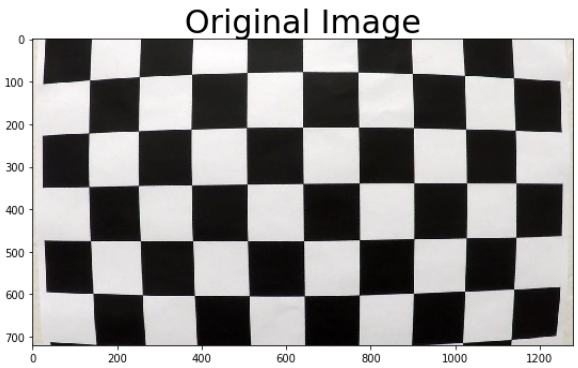
1) Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

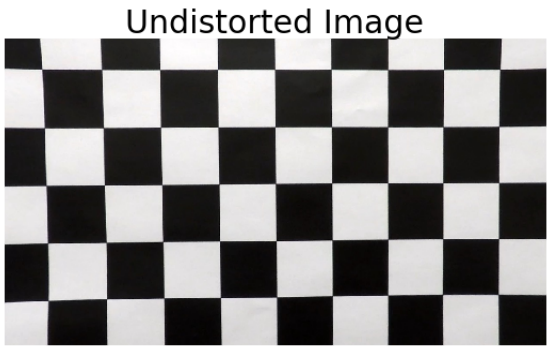
The code for this step is contained in the second code cell of the Jupyter Notebook located in "P4_Advanced_Lane_Lines4.ipynb."

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.

In [4]:

```
Done
```




## 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 the one below. By obtaining correct camera matrix and distortion coefficients from above, we can apply them on new images taken by the same camera and correct their distortion using the cv2.undistort function (fifth and sixth code cells). Here is the result:

In [6]:

```
Done
```



(This is slightly out of order because I jumped right into testing binary options on warped images.)

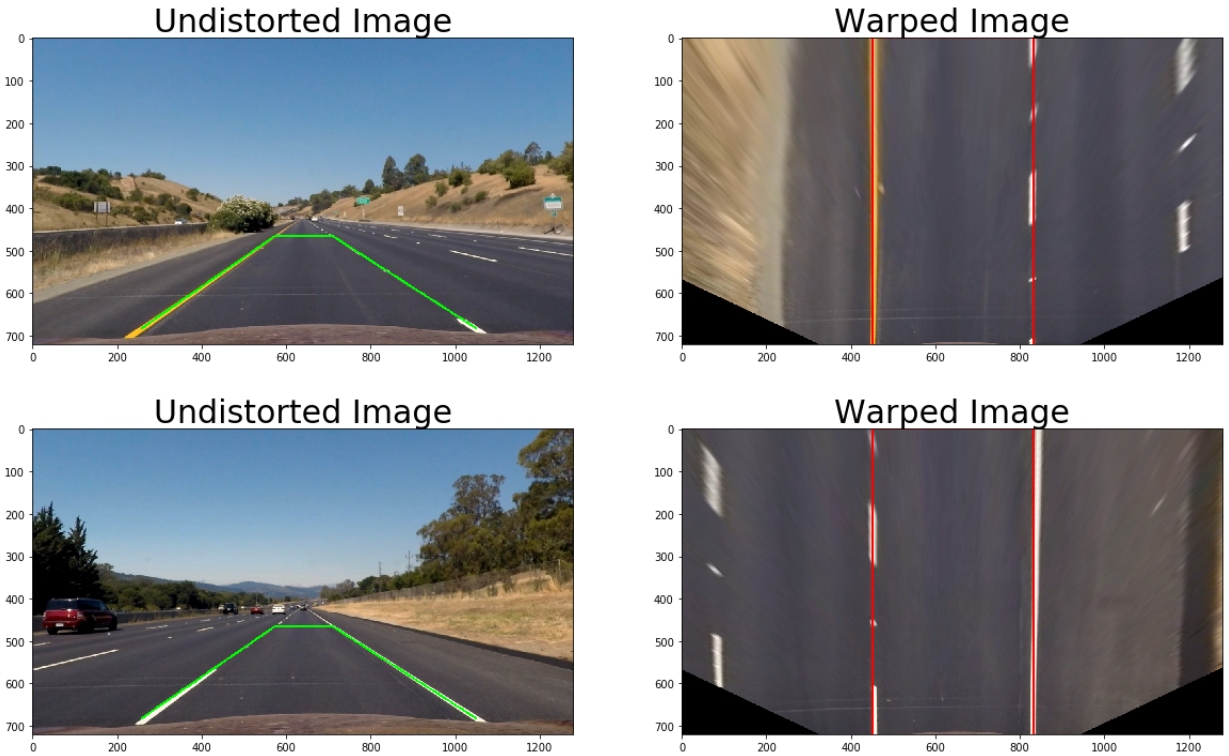
3) Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for my perspective transform appears in the eighth code cell in the Jupyter Notebook. First, the cv2.getPerspectiveTransform() function takes as input source (src) and destination (dst) points. Then the cv2.warpPerspectiveTransform() takes that output, along with an image and image size information.

- src: (575,464) (707,464) (258,682) (1049,682)
- dst: (450,0) (830,0), (450,720), (830, 720)

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.

In [10]:




(Again, these are slightly out of order because I jumped right into testing binary options on warped images.)

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.

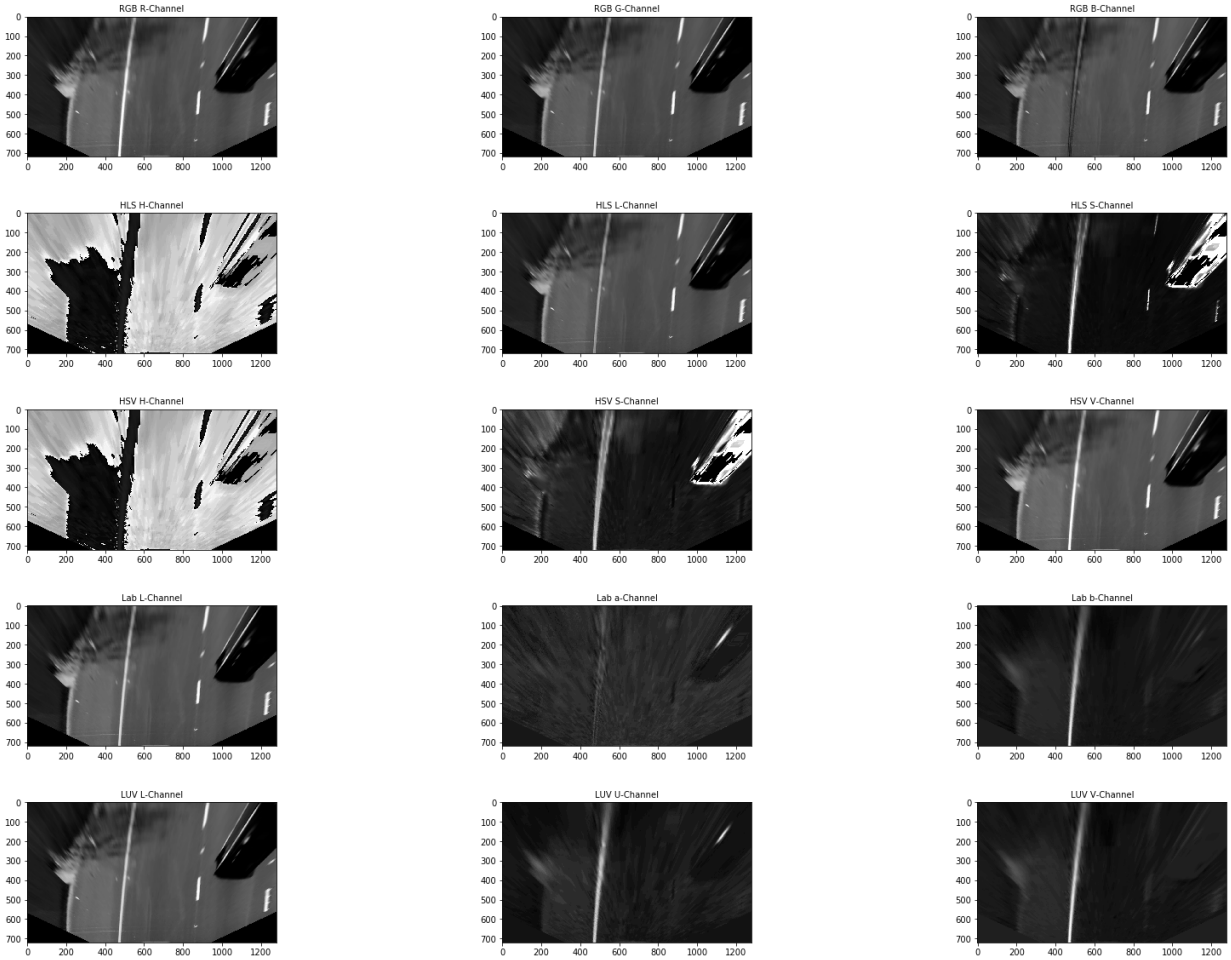
My code for this section is in the 19th code cell of the Jupyter Notebook.

I opted to focus on color to generate a binary image, based on the favorable lighting conditons for the project video. I tinkered with the abs_sobel operator for my first submission and looked at it again for this second version of my project but I ultimately felt that isolating yellow and white using the L channel from the LUV colorspace and the b channel from the Lab colorspace would suffice for this project. I realize I will need more robust thresholding methods for more challenging conditions.

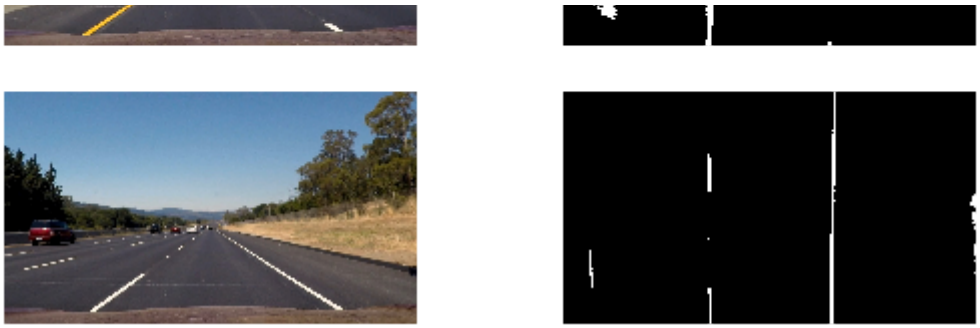
Here are examples of my colorspace testing, as well as examples of the thresholding I went with:

In [12]:

Out[12]: <matplotlib.figure.Figure at 0x11664d390>



In [21]:

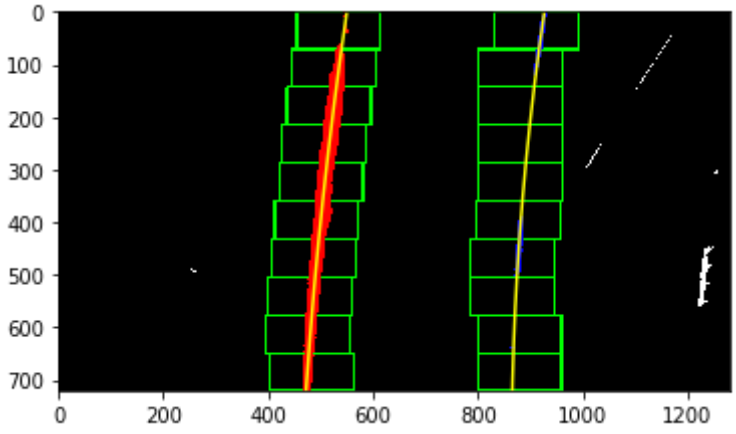


4) Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The code for my lane finding includes a function called sliding_window(), which appears in the 21st code cell of the Jupyter Notebook. The main idea of this algorithm is using histogram peaks in the lane lines in many sliding windows on the image. After calibration and thresholding and performing a perspective transform on an image, we should obtain a binary image on which lane lines are pretty clear. By taking a histogram of the image along small window sizes we should see some peaks in pixel values, which are indicative of lane lines, at least at their starting point. Once we find out where a lane line starts in the first sliding window at the bottom of our image, we can use that to find out how the lane line continues as we move to the second sliding window. So in summary, my sliding_window() function takes a binary_warped image. After choosing a certain number of sliding windows and their height, I begin with the very first one at the bottom of the image and we take a look at the histogram of the image through that sliding window. As soon as we find a 'hot' pixel in the window, we record them to be updated for each window. We can assume a margin around the found positions to be considered as our lane lines. We keep track of the lane line pixel indices until we slide through the whole image and concatenate the arrays of indices.

In [23]:

Out[23]: (720, 0)

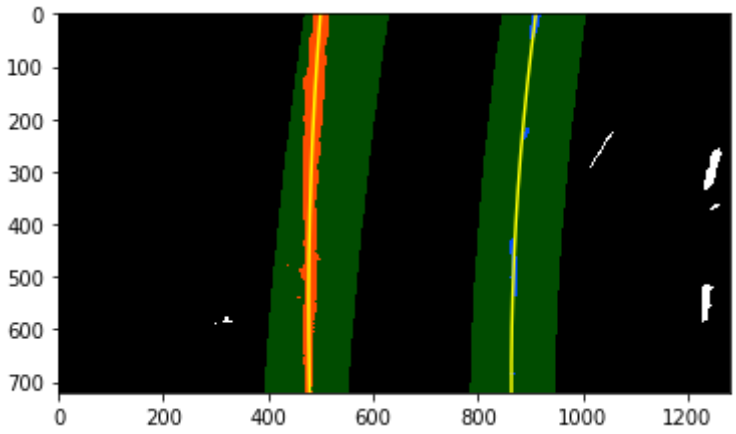



4) continued...

We can fit an equation to the found pixels of each lane. This is done in secondary_polyfit() which takes the outputs of sliding_window() and gives us x and y values for plotting the lane lines. Here is an example:

In [26]:

Out[26]: (720, 0)




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.

This can be found in the 26th code cell of the Jupyter Notebook in the lane_curve_n_center_dist() function. The labeling for the formula is:

curverad and it takes the x and y pixel coordinates of the lane lines and converts them to real world space coordinates based on US road regulations and given information about our images. By knowing that a lane marker is around 3 meters long and a standard lane is 3.7 meters wide, along with an image's height and width, we can compute meters per pixel in both x and y dimensions. By fitting a polynomial to our computed metric coordinate values, we can determine the equation of each lane line in real world space coordinates.

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

I implemented this step in the 29th and 32nd code cells in the Jupyter Notebook. Here are examples from what seem to be sequential test images:

In [34]:

Out[34]: <matplotlib.image.AxesImage at 0x11d81b940>


In [35]:

Out[35]: <matplotlib.image.AxesImage at 0x11d885198>


## 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: https://www.dropbox.com/s/b132c4sqb8znqnu/output_project_vid.mp4?dl=0 (https://www.dropbox.com/s/b132c4sqb8znqnu/output_project_vid.mp4?dl=0)

## 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?

This is my second submission. Upon trying to make a number of what could be described as "quick fixes," which only seemed to make my output video even more shaky, I decided that starting over from nearly scratch would be a painful, yet valuable learning experience. I did a lot more testing and

ran a lot more "sanity checks" along the way to be sure I had not strayed from the goal for this project. I knew all along that color thresholding would have its limitations (darker lighting, less structured lanes and windier roads would probably all pose a problem for this implementation). This proved to be accurate in the challenge videos. But I'm happy with the much smoother outcome that I have achieved with this attempt and I assume there will be more opportunities to improve upon these methods for situations resembling the challenge videos as we move into Terms 2 and 3.