Advanced Lane Finding Project

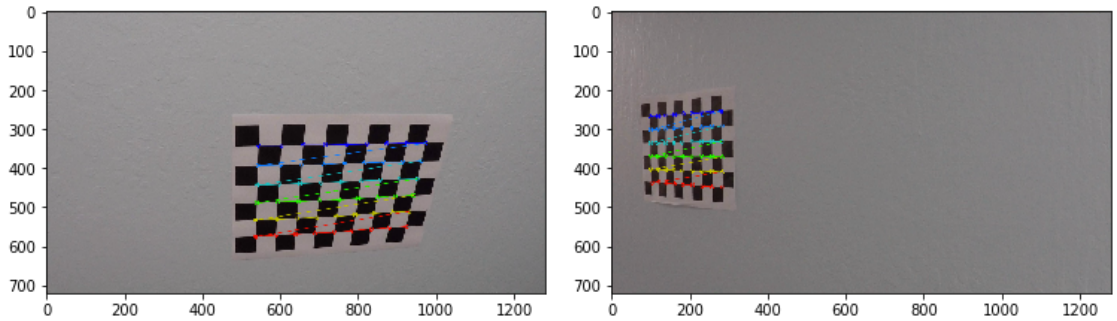
The goals / steps of this project are the following:

- 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.

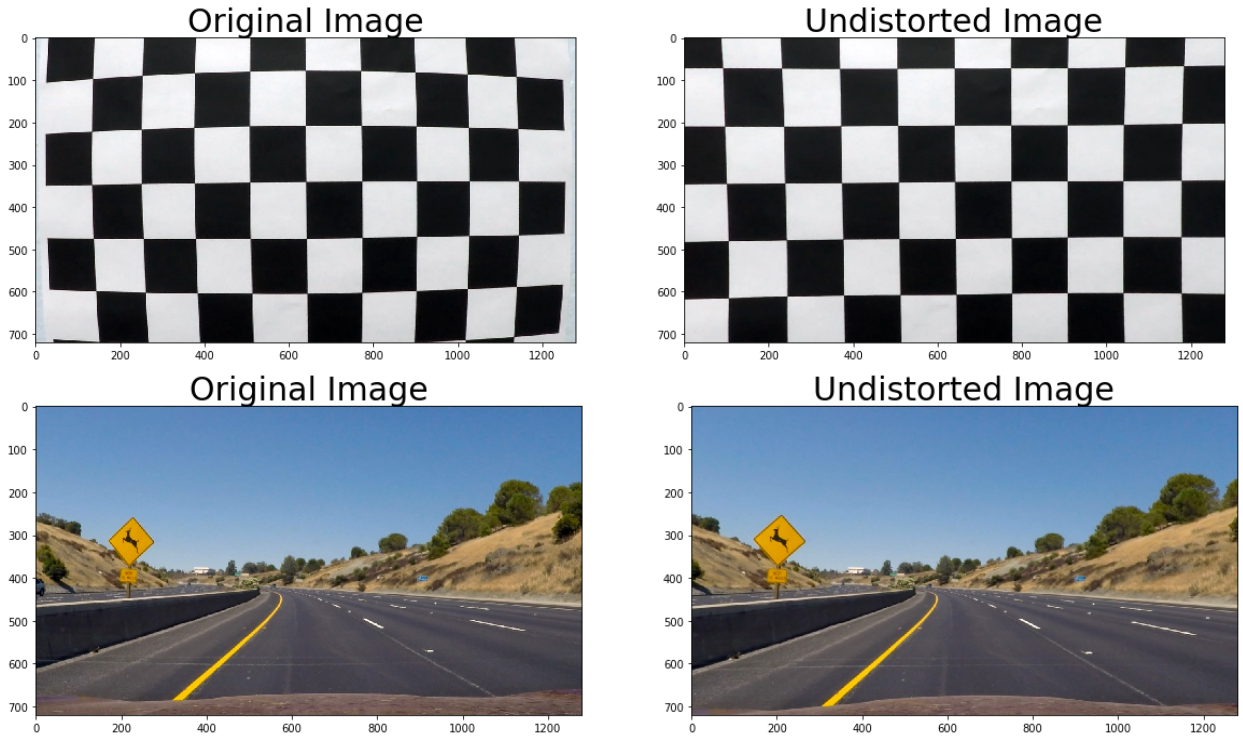
Camera Calibration

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 IPython notebook located in “Advanced_Lanes.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:



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

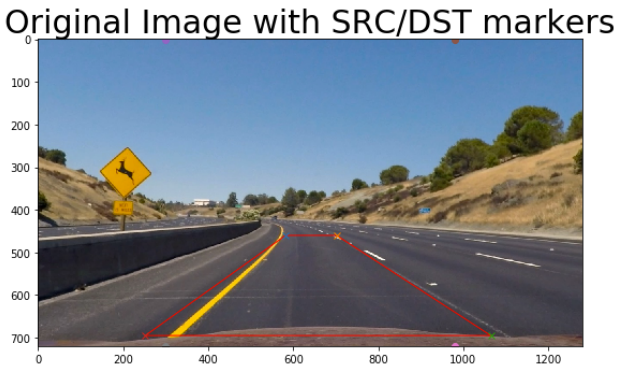
The code for my perspective transform includes a function called `warp()`, which appears in 4th code cell of the IPython notebook. The `warp ()` function takes as inputs an image (`img`). I chose to hardcode the source and destination points in the following manner:

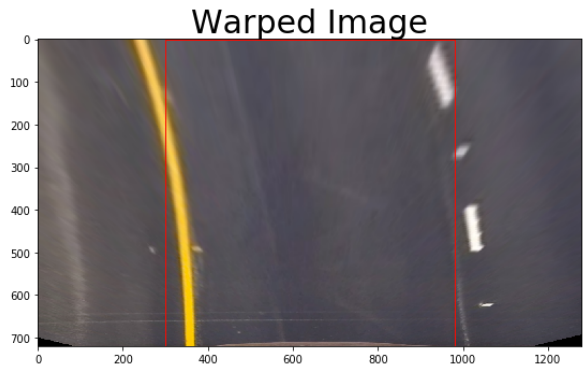
```
    c1 = (582, 460)
    c2 = (702, 460)
    c3 = (1065, 695)
    c4 = (252, 695)
    src = np.float32([c1, c2, c3, c4])
    d1 = [300, 0]
    d2 = [x - 300, 0]
    d3 = [x - 300, y]
    d4 = [300, y]
    dst = np.float32([d1, d2, d3, d4])
```

This resulted in the following source and destination points:

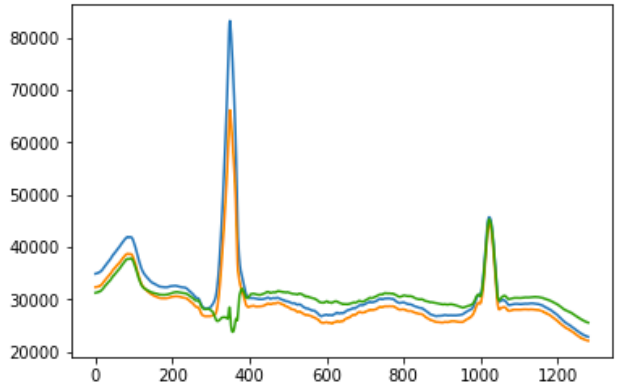
Source	Destination
582, 460	300, 0
702, 460	x-300, 0
1065, 695	x-300, y
252, 695	300, y

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.



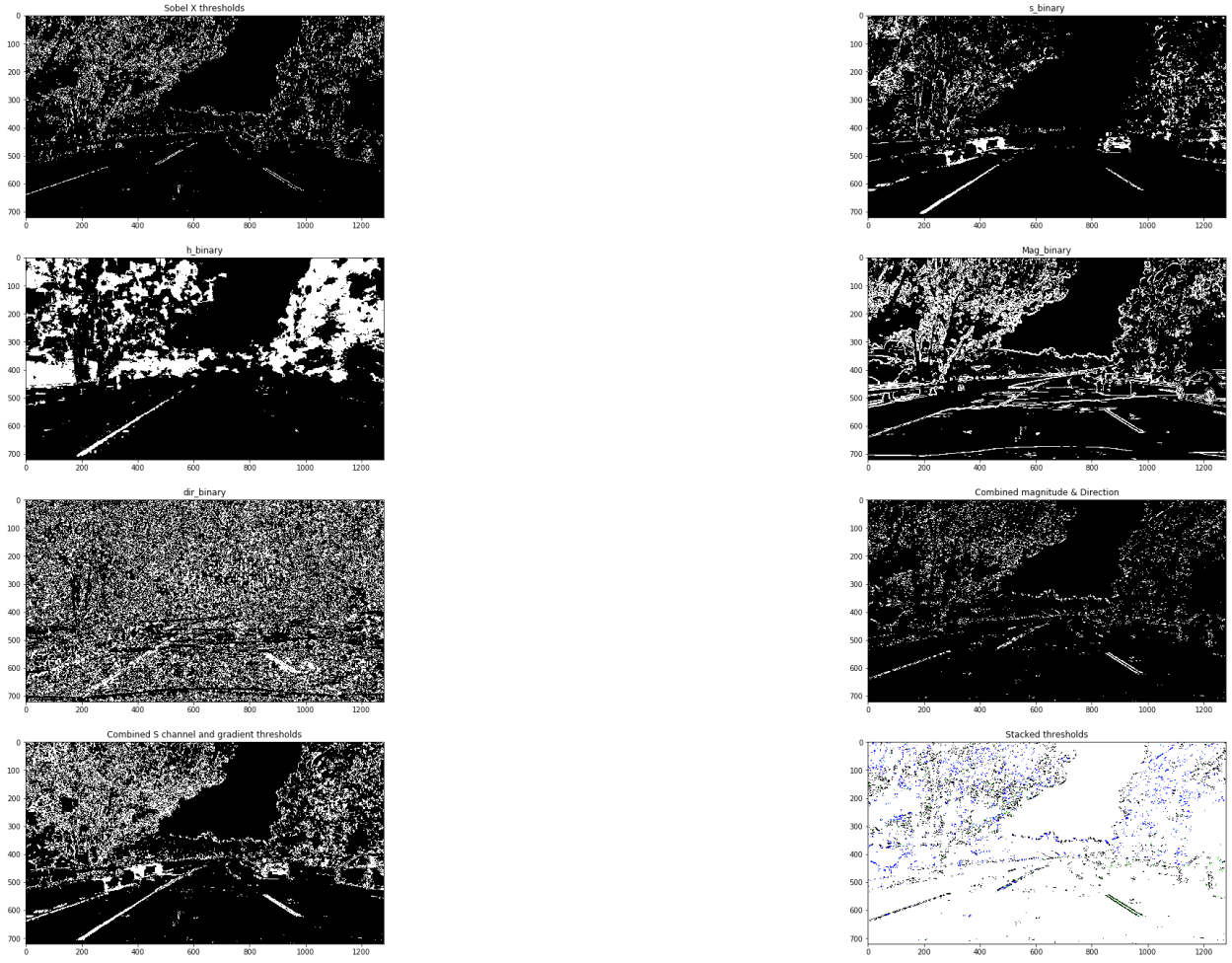


I also used this opportunity to look at the histogram of the bottom half to validate that we can find the start of each lane line. The histogram bins the color data for each column in the warped image. The peaks represent the lines which can be seen below:



3. 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.

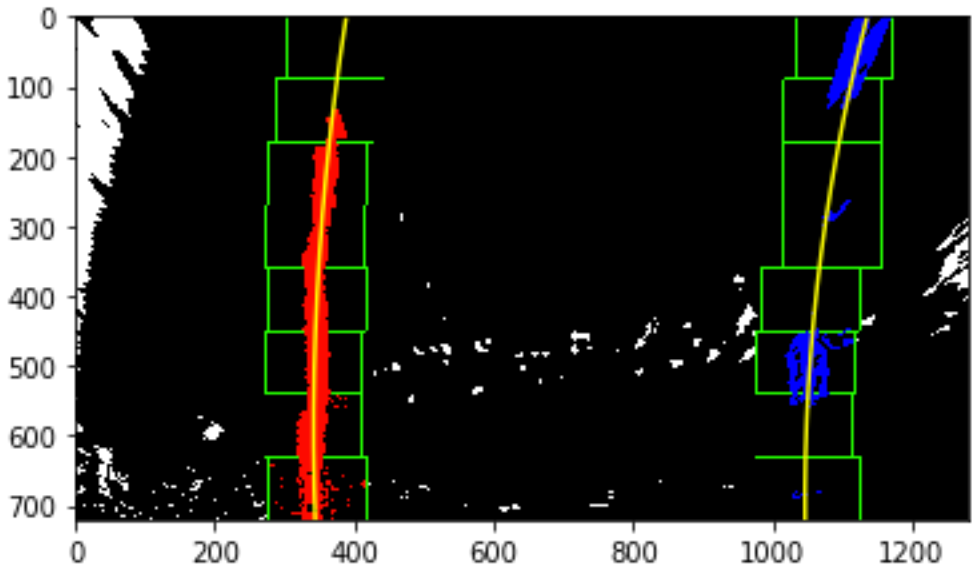
I used a combination of color and gradient thresholds to generate a binary image (thresholding steps starts on line 73 of the 6th code cell of IPython notebook). Combining all of these enabled me to get as much of the lines as possible. Although there is some noise from some of the thresholding, I am able to filter it out during the moving window and look ahead lane finding functions discussed in later sections. Here's an example of my output for this step.



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

To identify and calculate the lane line positions and curve, I took the output of the binary combination and transformed to the bird's eye view. There were two methods (In code cell 8) for finding the lines with this view: 1) moving window (line#38) and 2) look ahead (line#103). The moving window method was used when no lanes have been found yet.

We start with the histogram telling us where the start of each lane might be as seen in section 2 above. With these starting points, I create a window to look for pixels in, dividing the image into 8 regions vertically. Each lane has its own window and we re-center the search window where the majority of the pixels in the previous search were. The second method is the look ahead method where we search around the previous lane. This saves time and computation power because we're expecting the lane to be within a certain area of margin. Once we have found the pixels we consider our lanes, we find a best fit 2nd order polynomial over it.



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.

After we identify the x positions of each of the lines and the fitted curve, I calculated the scaled radius by converting pixels to meters. Once I have the curve, I calculate the radius using the following equation:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

This is done using the function *get_radius()* in Code Cell 8 line#151, which takes in the binary image and two arrays, left/right x values and left/right y values of the lanes.

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 line#179 of code cell 8. Here is an example of my result on a test image:

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?

Some of the challenges I came across were with getting the thresholding right. There was a lot of noise introduced using some methods, so I would spend more time tweaking this section. I think the better the image I get to work with, the better every

other result will turn out. I would also spend more time making the code more efficient, because we're doing a lot of line fitting when we could probably fit the line once and calculate what we needed to from that.