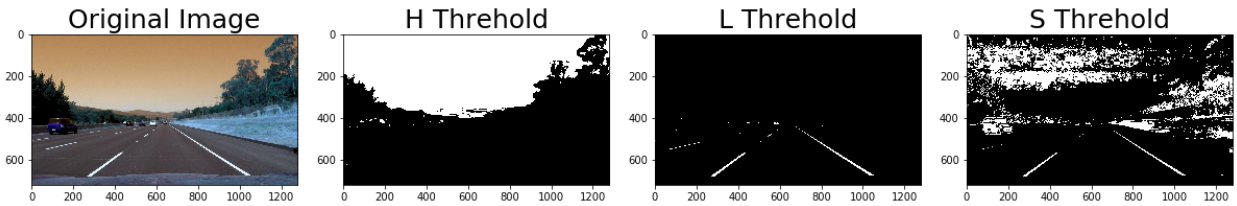
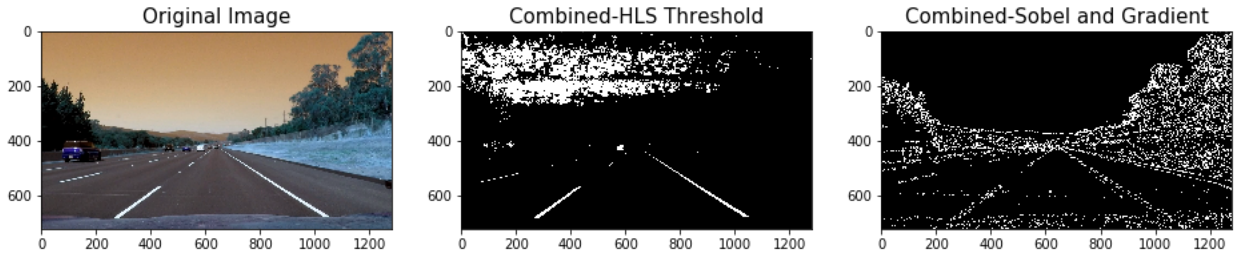
Advanced Lane Finding Project

My Experiments on the Udacity Advance Lane Detection 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

Camera Calibration

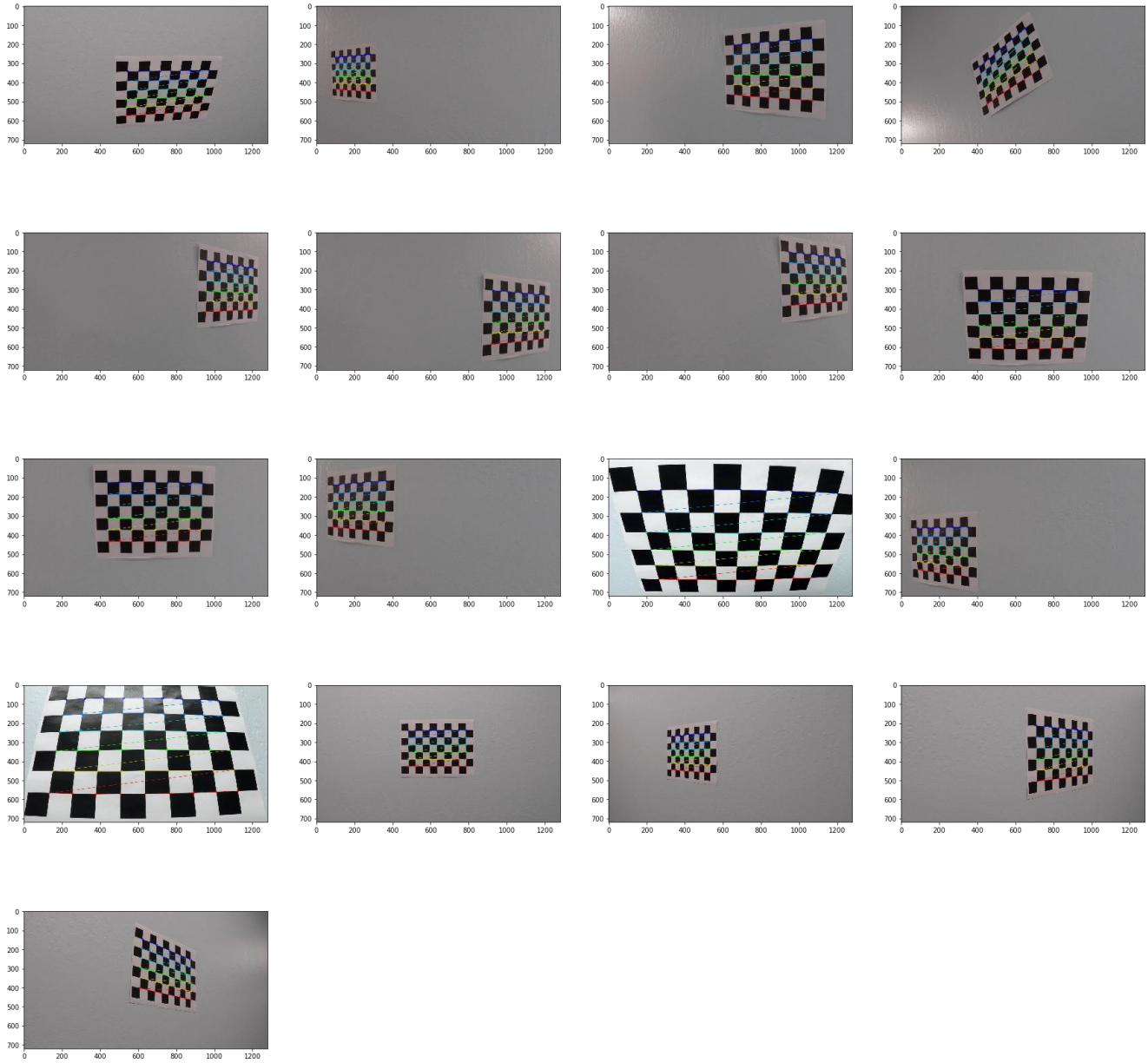
1. *Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.*

The code for this step is contained in the first code cell of the IPython notebook located in Advance Lane Detection.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:

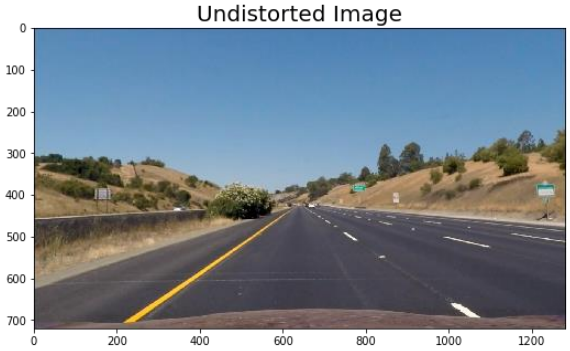


Pipeline (single images)

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

My Approach :

Calibrate Camera



1. Undistort Frame
2. Gradient Threshold
3. Color Threshold.
4. Combine Threshold
5. Perspective Transform
6. Find Lines
7. Measure Curvature
8. Measure Position.

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.*

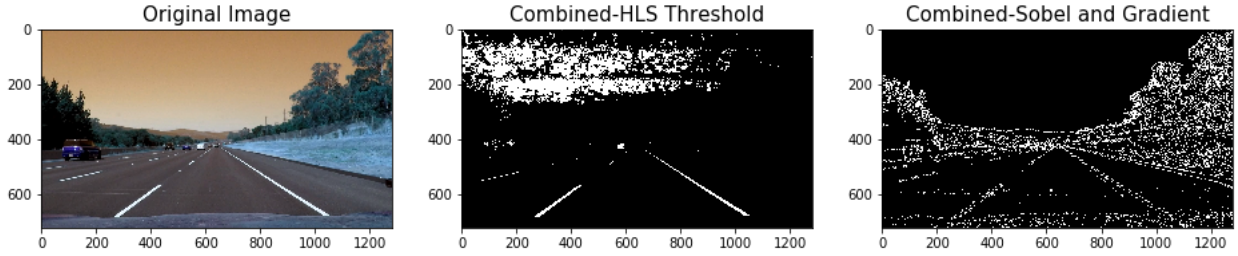
Sobel detection refers to computing the gradient magnitude of an image using 3x3 filters. Where "gradient magnitude" is, for each a pixel, a number giving the greatest rate of change in light intensity in the direction where intensity is changing fastest. Even though using Sobel, a lot of Noise had picked up, thus I tried combining the Hue, Lightness and the Saturation to detect the yellow and the white lines. I tried Using Various Color Spaces Such as YUV, HSV, LAB. All the color spaces where helpful, but I ended up choosing the HLS.

Instead of edge detection, I used adaptive thresholding to further boost the line detection since the videos, especially the challenge videos, have lighting variations. Below the overpass in the challenge video, and constant switch between tree shadows and sunlight in the hard challenge video are all examples of lighting variations. Most edge detection approaches and color scheme based filters require a static threshold, which makes line detection harder.

The Threhold Applied Where:

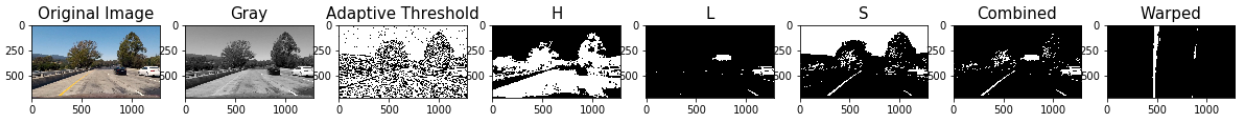
- Hue Channel : 10, 40
- Light Channel : 200, 255
- Saturation Channel : 100, 255

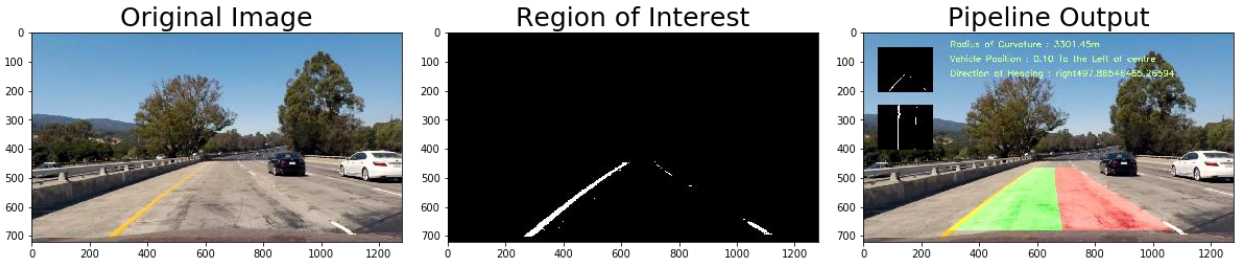
I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at lines # through # in `another_file.py`). Here's an example of my output for this step. (note: this is not actually from one of the test images)



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

```
src = np.float32([(575, 464), (707, 464), (258, 682), (1049, 682)])
dst = np.float32([(width_Offset, height_Offset), (width-width_Offset, height_Offset),
(widthOffset, height-height_Offset), (width-width_Offset, height-height_Offset)])
```





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?*

The Radius of curvature was calculated using the notes from the classroom. In addition to that, I tried Calculating the Vehicle Positon and the Heading of the Vehicle. This could help in Sensor Fusion and State Estimation.

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!).

Project Video

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?*

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

- Adaptive thresholding when combined with other image processing techniques such as histogram equalization, contrast adjustments or brightness adjustments could yield better results. I tried histogram equalization, but that had little to no effect.

- While the region of interest was both useful in general, it cut off parts of the lane in several challenging cases. One improvement would be to adjust the bird eye view to account for this It is possible for us to learn the presence of lane lines, and predict if a region is a lane or not. That might help mitigate some of the effects of lighting.

- Improved Detection of lanes Can help in exact a calculation of state estimate using a Kalman or an Extended Kalman Filter that may Update and Predict our State using Sensors and various other computer Vision technique.