Writeup P4

Advanced Lane Finding Project

The 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.

Compute the camera calibration parameters:

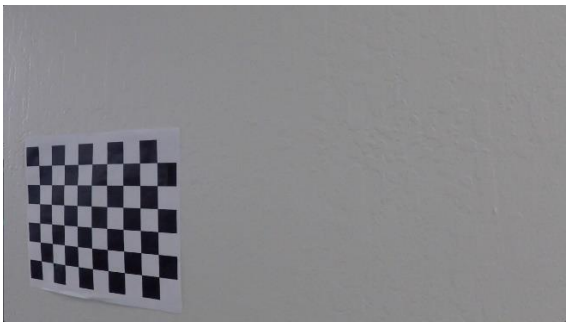
I computed camera calibration matrix and distortion coefficients given a set of chessboard images in "/camera_cal" folder.

The code for this step is contained in the first code cell of the IPython notebook located in "./example.ipynb"

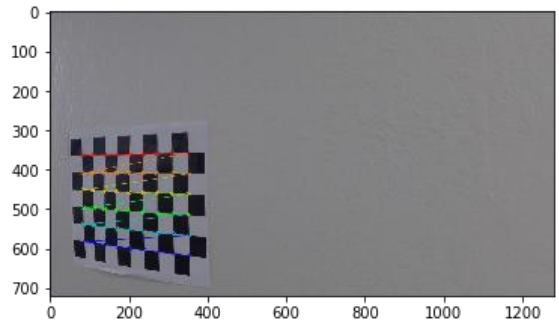
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, objp is just a replicated array of coordinates, 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:

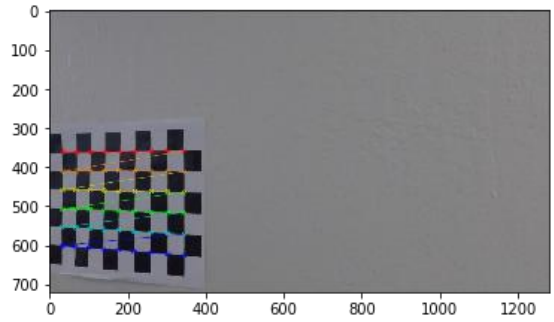
Sample Image before and after detecting corners. # Edits 10 Feb
1-original image:



2-Detecting corners:



1-Undistorted image:



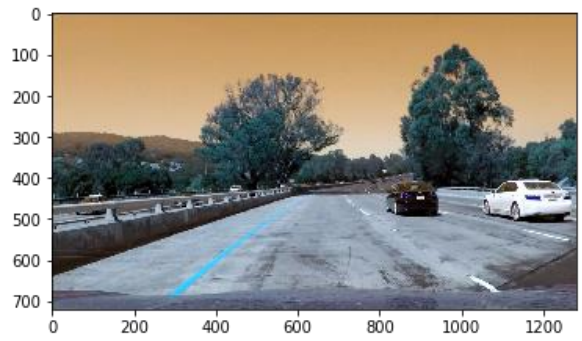
My calibration result:

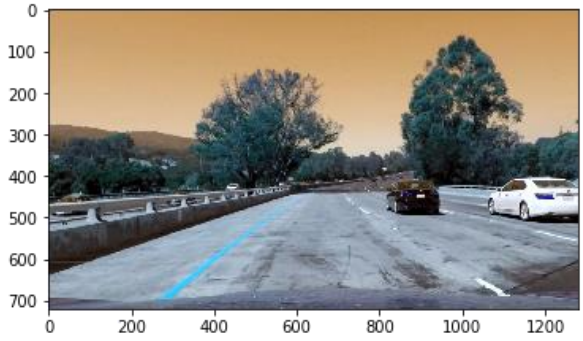
```
Ret=1.4465066740178016
mtx = [[ 1.34143703e+03   0.00000000e+00   7.06579000e+02]
 [ 0.00000000e+00   1.38865729e+03   4.07021179e+02]
 [ 0.00000000e+00   0.00000000e+00   1.00000000e+00]]
Dist= [[-0.28368669 -0.58858877 0.00265609 -0.00431255 2.09405269]]
rvecs= [array ([[-0.50713919],
       [-0.08719005],
       [-0.02730586]])]
tvecs= [array ([[ -4.75115993],
       [ -3.12931698],
       [ 12.33648673]])]
```

Apply a distortion correction to raw images:

Then I used (cv2.undistort()) function to undistort the input image.

The code for this step is contained in the third code cell of the IPython notebook located in "./example.ipynb" **"Apply a distortion correction to raw images"**





Use color transforms, gradients, etc., to create a thresholded binary image.

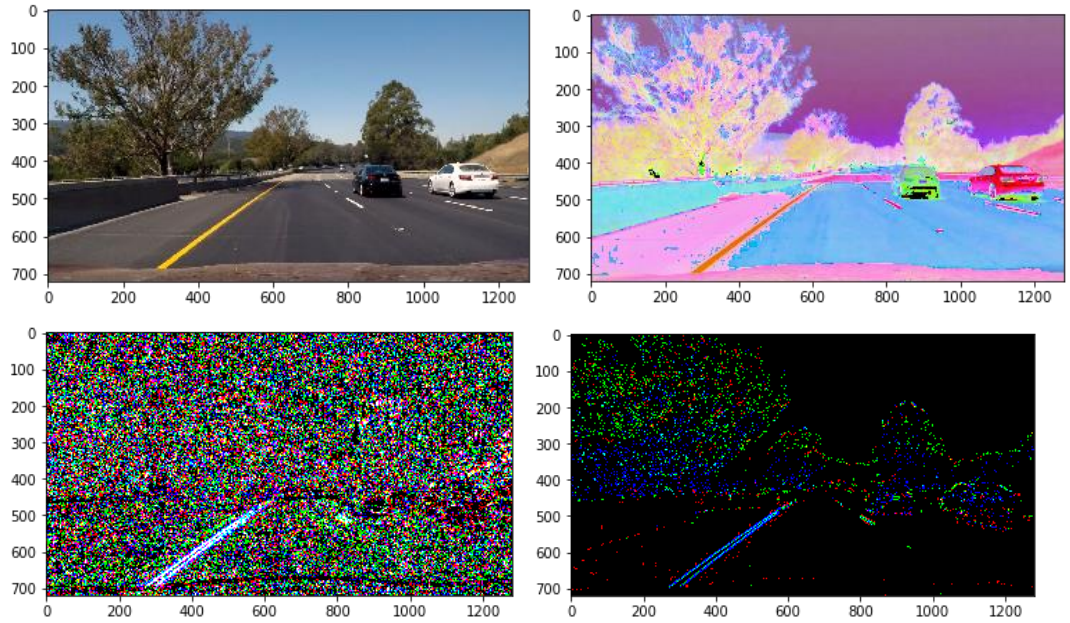
1st try: (not so good)

Then I converted the undistorted image to HLS color space using **"cv2.cvtColor(img, cv2.COLOR_RGB2HLS)"**

And using different thresholds I create a combined threshold binary image that most show the lane lines well by following technique: There is

- **abs_sobel_thresh for x**
  used min threshold =20 and max threshold = 100, The effect of x thresholding isn't very high
- **abs_sobel_thresh for y**
  used min threshold =65 and max threshold = 150, The effect of y thresholding isn't very high
- **mag_threshold**
  used min threshold = 170 and max threshold = 230
- **dir_threshold**
  used min threshold = 0.7 rad and max threshold = 1.2 rad, the orientation filter is the most effective threshold in my pipeline.
- **Combination**
  Then I combined all the last thresholds together to get the result according to the following method:
  If {input binary image pixel for **x or y** is white (1)**} and {**input binary image pixel for **mag or direction** is white (1)**} then output is white (1).**

The code for this step is contained in the fourth code cell of the IPython notebook located in "./example.ipynb" **"Create a thresholded binary image"**
Using S channel only:

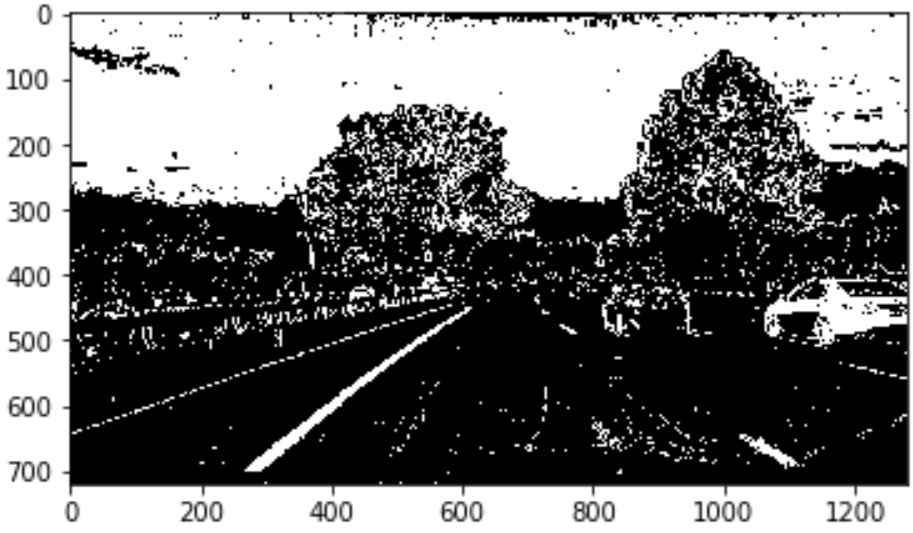


2nd try(working): Edits 10 Feb

Using S, L channels, Applying sobel x threshold at L channel and thresholding overall S, L channels.

Use combination of the three method:

> **If {**input binary image pixel for **x** is white (1)**} and {**input binary image pixel for **saturation or lightness** is white (1)**} then output is white (1).**
> (l_binary == 1) & (s_binary == 1) | (sxbinary==1)



Apply a perspective transform to rectify binary image ("birds-eye view").

Since the camera is seeing the road from its perspective (on front of the car), The distances and the shapes aren't real. For example, the lane lines aren't parallel unlike the real lane lines. So to get a prefect view with which the car can determine where exactly it is with respect to the lane lines, we should apply a transform from this view to birds-eye view.

I used { cv2.getPerspectiveTransform(src, dst)}, { cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR} functions to get this transform .

The destination set of points is some kind of a rectangle.

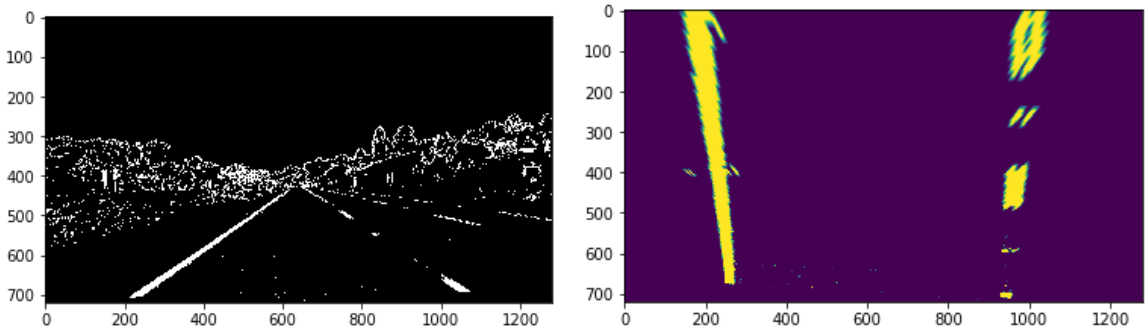
The src set of point is the same rectangle in the image but as the image shape isn't real so we should take a polygon in the picture that we think it is a rectangle in reality. I tried to do that programmatically using the pipeline of project 1that draw 2 lines using 4 points but it wasn't very good in all images. So, I did it once and set the src points manually.

These are my points.

src = [[203, 720], [585, 460], [1127, 720], [695, 460]], dst= [[260,680], [200,0], [960, 720],[960, 0]]

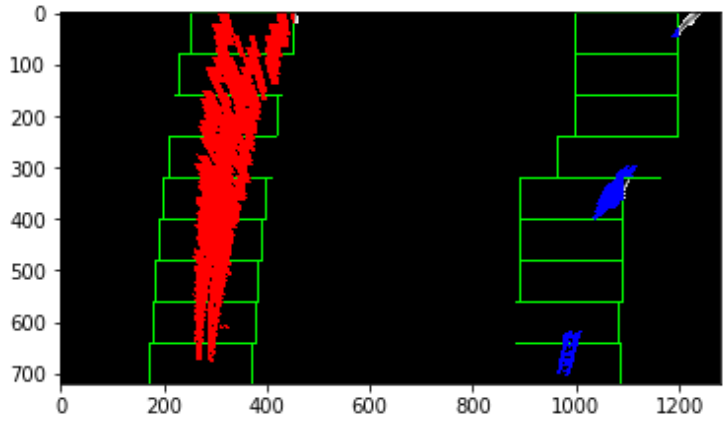
The code for this step is contained in the 5th code cell of the IPython notebook located in "./example.ipynb" **"Apply a perspective transform to rectify binary image ("birds-eye view")"**

Here are 2 pictures that shows perspective transform: **Edits 10 Feb(use straight line input)**



Detect lane pixels and fit to find the lane boundary.

The code for this step is contained in the 6th code cell of the IPython notebook located in "./example.ipynb" **"Detect lane pixels and fit to find the lane boundary"**
In this section , I first get the histogram of the image then get the maximum of right part that represent the area of right lane and the maximum of left part that represents the area of the right lane .Then looped through 9 windows to identify the nonzero pixels in x and y within the window .Then fit a second order polynomial to each of the lane lines



 # Edits 12 Feb

Determine the curvature of the lane and vehicle position with respect to center.

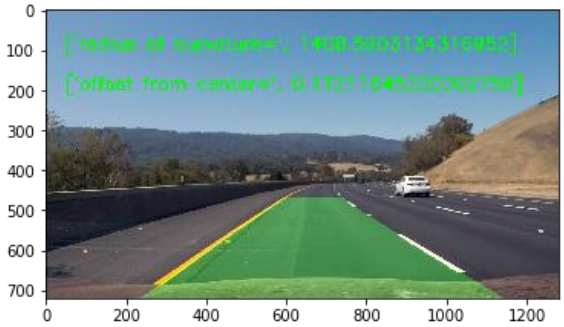
The code for this step is contained in the 7th code cell of the IPython notebook located in "./example.ipynb" **"Determine the curvature of the lane and vehicle position with respect to center."**

One sample of result is 1408.59 as radius of curvature and 0.112118 for offset.That mean that the car is towards left lane 0.112 meter.

Warp the detected lane boundaries back onto the original image.

The code for this step is contained in the 8th code cell of the IPython notebook located in "./example.ipynb" **"Determine the curvature of the lane and vehicle position with respect to center."**

 # Edits 10 Feb: Add radius of curvature and offset



Pipeline

The code for this step is contained in the 9th code cell of the IPython notebook located in "./example.ipynb" **"Main pipeline."**

In this cell, there is a sequence of the last steps done in series. There are 2 typical functions, one to test in a picture and one to use for the video.

 # Edits 10 Feb

Discussion problems/issues faced, suggested improvement.

The most important step that takes some time to develop is Thresholding binary image. I tried to depend on RGB color space as we did in project one but it was not good at shadowing and different lighting environments.

Depending on Saturation channel on HLS, it was good through most of the video but depending on color saturation isn't enough as there are different lighting environments. We can fix that by depending on both saturation and lightness.

One shortcoming of this pipeline is when there are more complicated situations, such as 2-way roads.

I think using combination of more color spaces and Laplacian transform that I recently begin to read about will handle this situation.