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

Camera Calibration

The code for this step is contained in the code cell under Step 1 of the IPython notebook.

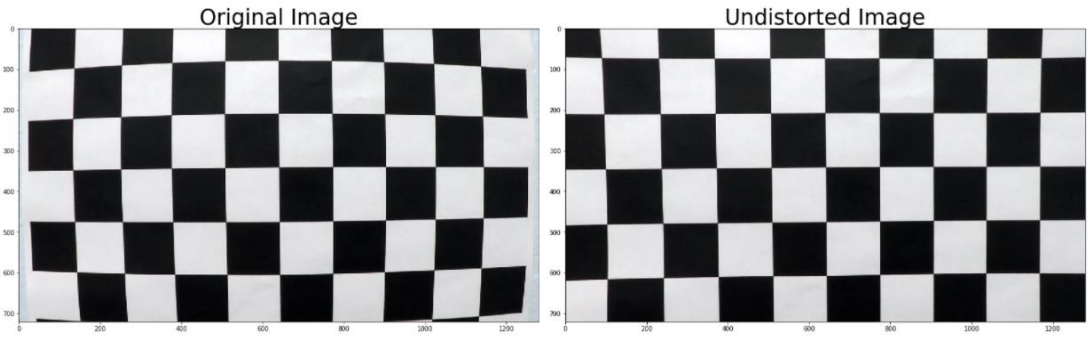
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, 'objp' is just a replicated array of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

Pipeline (single image)

1. Apply a distortion correction to the raw image

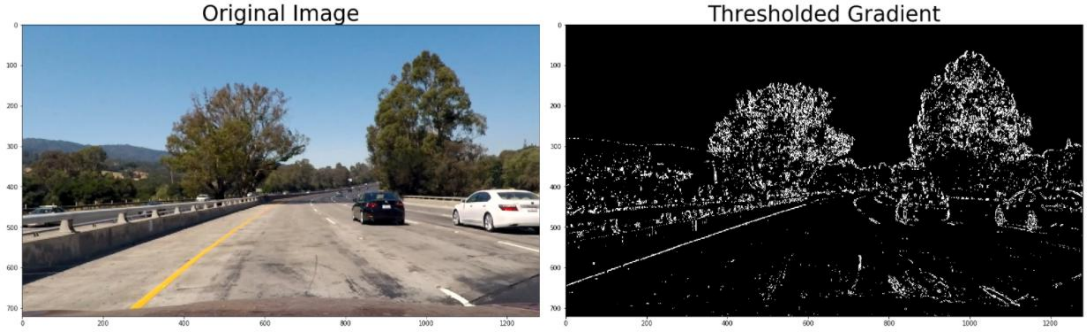
To undistort the raw image, I used the results calculated by camera calibration and appied them into the function cv2.undistort(). The corrected image of the chessboard is shown below:



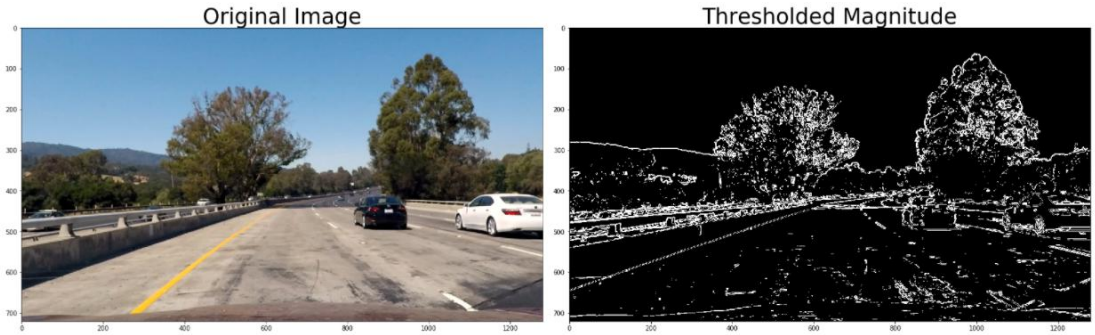
2. Use color transforms, gradients, etc., to create a thresholded binary image

This step is to detect the lane lines in the binary image. By combining different thresholds, it makes the detection more accurate.

First, I obtained the binary output of the gradient in “x” (horizontal) direction. I ignored the gradient at “y” direction because the lane lines will be detected by gradient in “x” direction and including “y” directional gradient will add unnecessary boundaries of the environment. To get this, I converted the image to gray scale and compute the soble gradient along the “x” direction and only picked the points whose value is within the predefined threshold. The binary output of the “x” directional gradient is shown below:

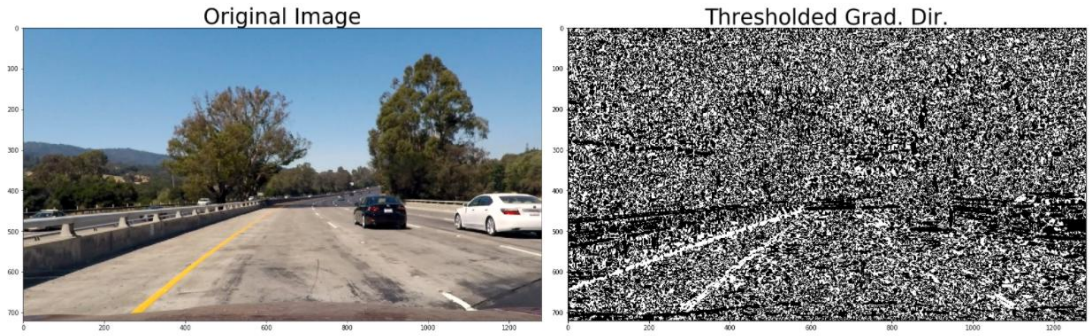


Second, I obtained the binary output of the thresholded magnitude of the gradient. The magnitude is defined as the square root of the sum of the squares in both x and y directions. This binary image is shown below:

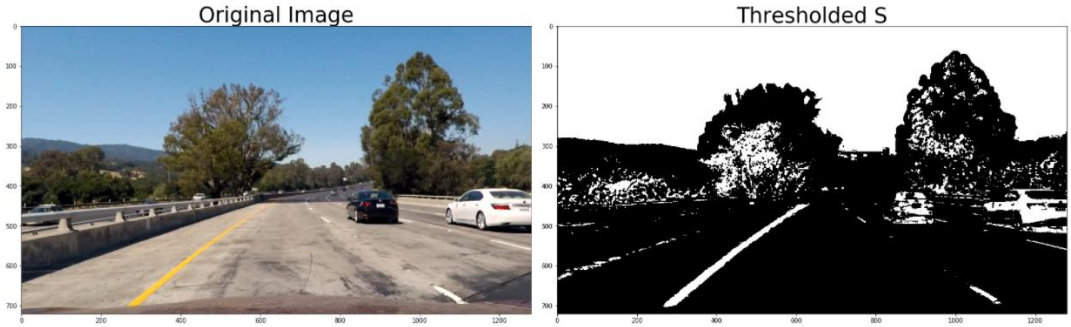


Third, I obtained the binary output of the thresholded direction of the gradient. The direction of the gradient is simply the inverse tangent (arctangent) of the y gradient divided by the x gradient.

Points that satisfying both magnitude and direction threshold will be good candidates for the lane lines. The binary image of direction of gradient is shown below:



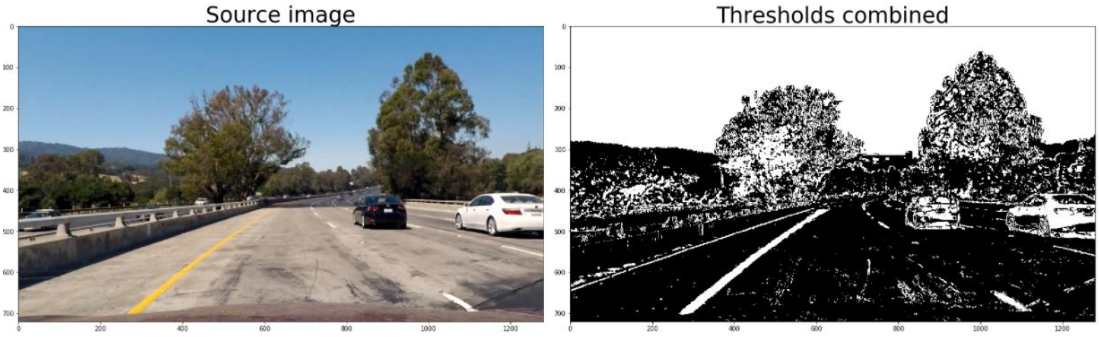
Then, I converted the image into HLS space and got the binary output for the S channel because it does a good job in detecting yellow and white lane lines. The binary image is :



Al last, I combined all those binary output using a “or” logic：

combined[(grad_binary == 1) | ((mag_binary == 1) & (dir_binary == 1)) | (hls_binary == 1)] = 1

And the combined binary output image is:

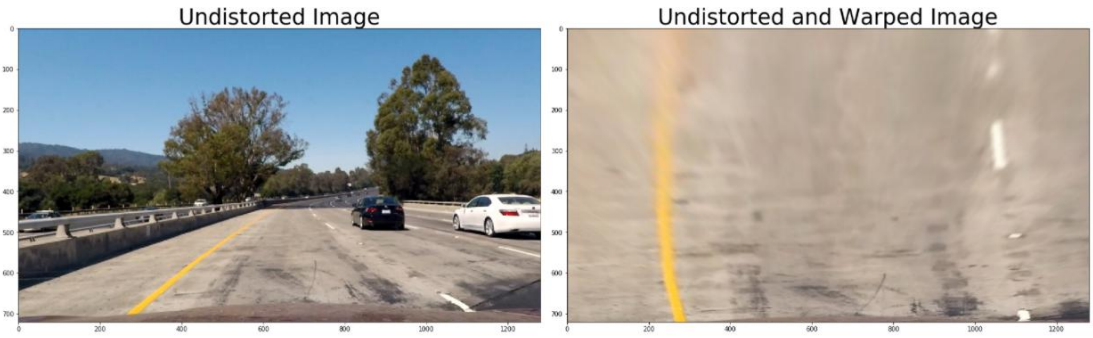


3. Apply a perspective transform to rectify binary image ("birds-eye view").

The code for my perspective transform is includes a function called corners_unwarp(). It takes in region of interest (src) and transform it into a top view (dst). I chose the hardcode the source and destination points in the following

```
src = np.float32([[220, 700],  #bottom left
                  [1180, 700], #bottom right
                  [750, 460],          #top right
                  [590, 460]])         #top left
# For destination points, I'm arbitrarily choosing some points to be
# a nice fit for displaying our warped result
# again, not exact, but close enough for our purposes
dst = np.float32([[220, 720],   #bottom left
                  [1180, 720],  #bottom right
                  [1180, 0],    #top right
                  [200, 0]])    #top left
```

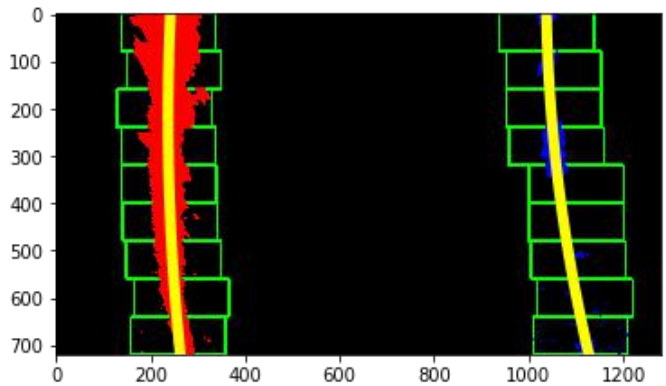
I verified that my perspective transform was working as expected by drawing the src and dst points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



4. Detect lane pixels and fit to find the lane boundary.

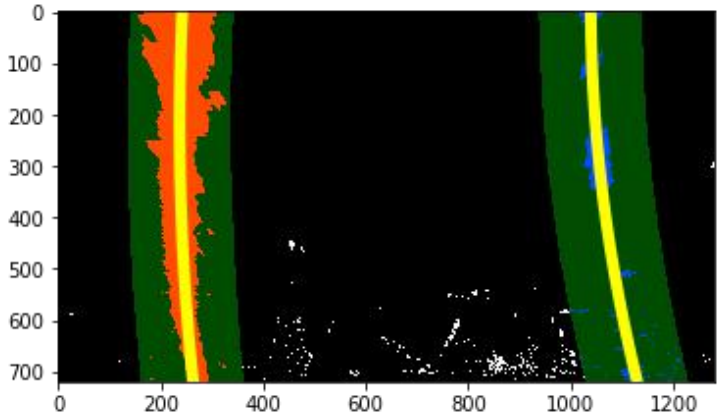
First I plotted a histogram of where the binary activation occur across the image. With this histogram I added up the pixel values along each column in the image. In our thresholded binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines.Those two points were set to be starting points for where to search for the lines. Then I used a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

In each of the sliding window, we only search the points in the horizontal margin and fit the points into a quadratic polynomial. As the sliding window moving top, we can draw a curve through polynomial curve fitting. This was achieved by a customized function find_lane_pixels().



Once the lane lines were found, we don't have to repeat the same steps for every frame. Instead,

we can just search around the lanes from the previous frame. The green shaded area shows where we searched for the lines.



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

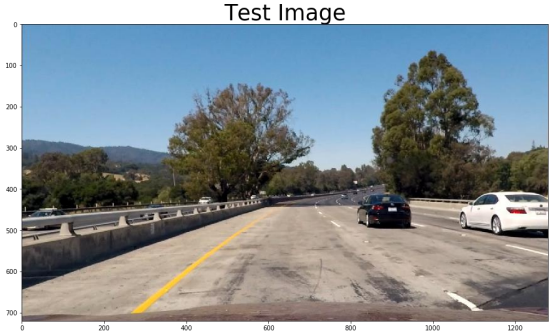
The curvature of both lane lines can be calculated by the coordinates of points on the lane curves. One thing needs to be noted is that the values in the figures should be converted to the values in real-world. And the offset is the difference between the center of the lane lines and the center of the image, which is considered as the position of the car. The results are:

Left Lane Line Curvature: 1588.98635002 m
Left Lane Line Curvature: 990.897161181 m
The car offset from center is: -0.296561943298 m

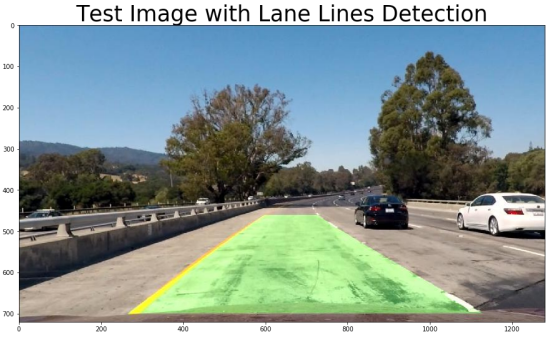
6. Warp the detected lane boundaries back onto the original image

After the lane lines are detected, it should be transferred back to the original graph. This uses function cv2.warpPerspective() and takes the inverse perspective matrix of step 3. The result is shown as:

Test Image



Test Image with Lane Lines Detection



7. Run the pipeline in the video:

Here's a [link to my video result]

Discussion

This is really challenging project for me since I'm not quite familiar with Python programming and it stuck me a long time writing everything into a pipeline. Thus, I am quite happy with the result I got.

There is still something that needs to be improved:
This pipeline does not perform well for the complicated lane line shapes. For example, if the curvature varies too frequently and there was continuous sharp turns, it won't detect them well. That's because the second-order curve may not able to express the lane lines. And the lane lines on the next frame does not lie in those on the previous frame.

I am not sure how to deal with this challenge. Maybe more threshold criteria is needed or we should apply a higher-order curve fitting and increase the number of sliding windows to make it more accurate.