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.

Approach

I started using a jupyter notebook and took samples from the lesson and started playing with them. Some of the write up just grabs code and images from this notebook.
I included these notebooks in the github repo.

Once I had a fairly good understanding of what was going on I started developing a python class lane_lines.py to encapsulate this knowledge. The class is a little large, and there are definitely further refinements I would make if I took more time. I did encapsulate each detected lane line in a separate class line.py

I spent a fair bit of time with the color and gradients portion , after processing a video I found places where it lost the lanes, or had difficulty with shade, changing road surfaces etc. I surmised that the color transforms and gradients were probably going to be pretty important. So I put together a test app that allowed me to quickly slide the threshold values for the gradient x,y , magnitude and direction, this helped me to pick a good set of thresholds. I also tried out different color conversion, and I settled on the s_channel from HLS colorspace as the best performing for the project video. I did observe that this did not do particularly well on the other challenge videos. I have plenty of ideas how to improve this part to help deal with those challenges but I've put enough time into the project for now.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

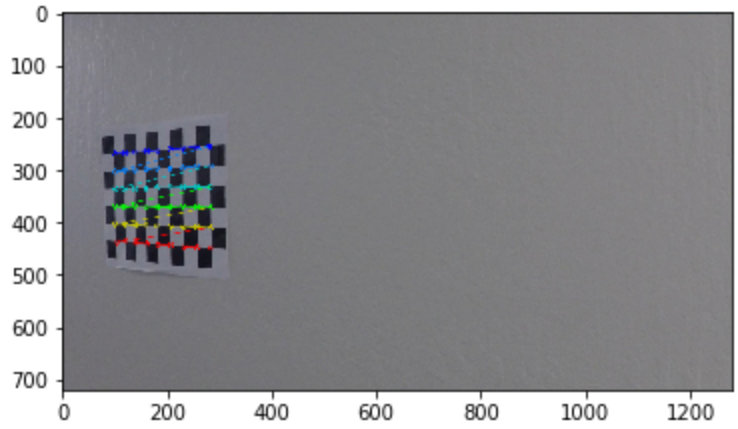
Camera Calibration

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

The camera calibration is abstracted into the camera module.
I have a method called **_calibrate_camera_from_image** that

1. Built an expected set of object points in 3d space depending on the expected rows and cols of chessboard corners
2. Converted the incoming image to grayscale
3. Used findChessBoardCorners to locate the actual points in the image if successful
    a. appended any detected chessboard corners from the provided image into an imgpoints array.
    b. Appended the object points to an objpoints array
4. It lastly captured the points visually as pictured below.

Here is an example of the chessboard detection for the camera.

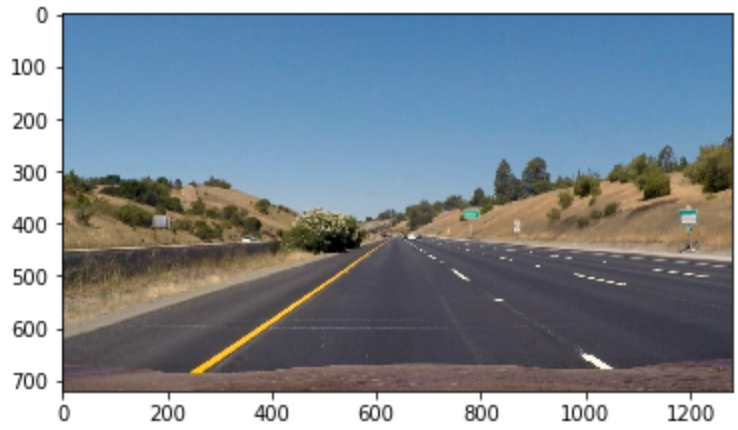


I wrote a higher level method **_init_calibrate_camera** that used the **glob.glob** method to return all calibration images in a folder and repeatedly called **_calibrate_camera_from_image** with those images building up a large set of image points and corresponding object points. This was passed to **cv2.calibrateCamera** to process. If this was successful, then I saved the resulting distortion camera matrix, distortion coefficients, rotation and translation vectors into a python pickle file. This allowed quickly restoring these values when testing and processing images later on.

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 this one:



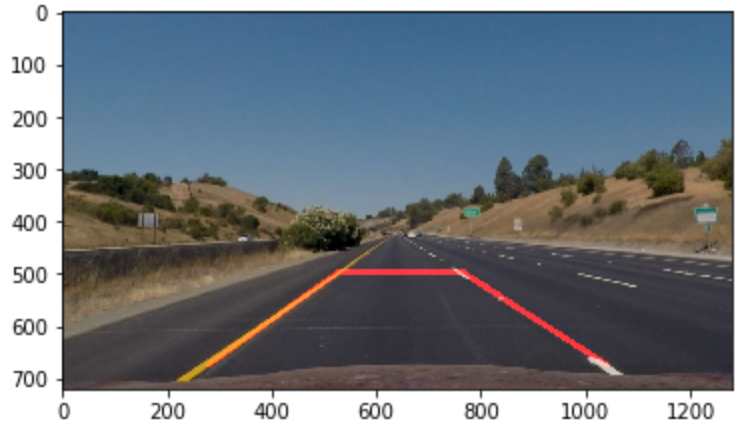
2. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

I used my scratch jyputer notebook to undistort one of the images. I then opened this image in gimp and measured out a trapezoid from the lane lines.

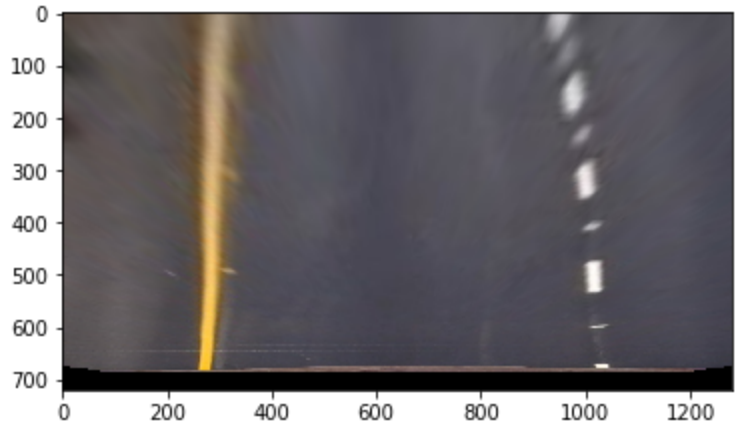
I then used this code to verify that this was correct.

```
for file in glob.glob('output_images/*straight_lines*.jpg'):
    print (file)
    img = mpimg.imread(file)
    img_size = (img.shape[1], img.shape[0])
     src = np.float32([[230.0, 700.0], [531.0, 495.0],
                       [762.5, 495.0], [1080.0, 700.0]])
       dst = np.float32([[230.0, 700.0], [230.0, 495.0], [
                       1080.0, 495.0], [1080.0, 700.0]])

    plt.figure()
    plt.imshow(weighted_img(draw_lines_between_points(np.shape(img), src),
img))

    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)
    plt.figure()
    plt.imshow(warped)
```

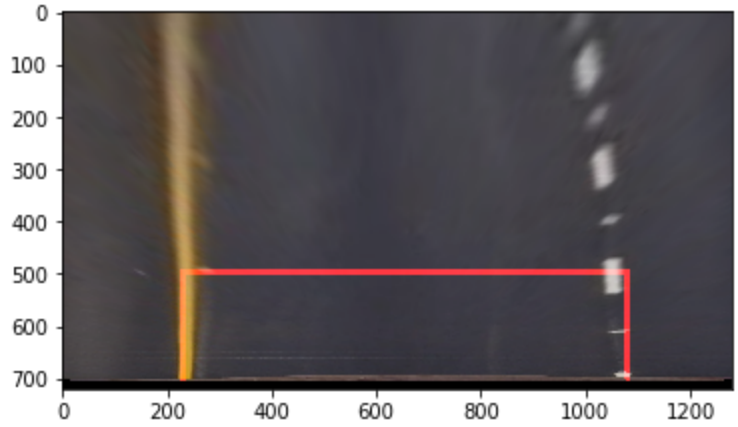
In my final code submission, I just initialize the transform and inverse transform matrix during the initialization of the Perspective class that manages transforming the images and maintains the inverse for use later to map the lane back to the original image.



Which when transformed becomes



I used this to measure the lane width in pixel space.



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.

This turned out to be a fairly lengthy process to get right. I choose to do this step after the perspective transform was applied. It doesn't really matter which order these two steps are run in, but for me I felt it was easier to focus on the line detection via the different gradient and color steps after the perspective transform.

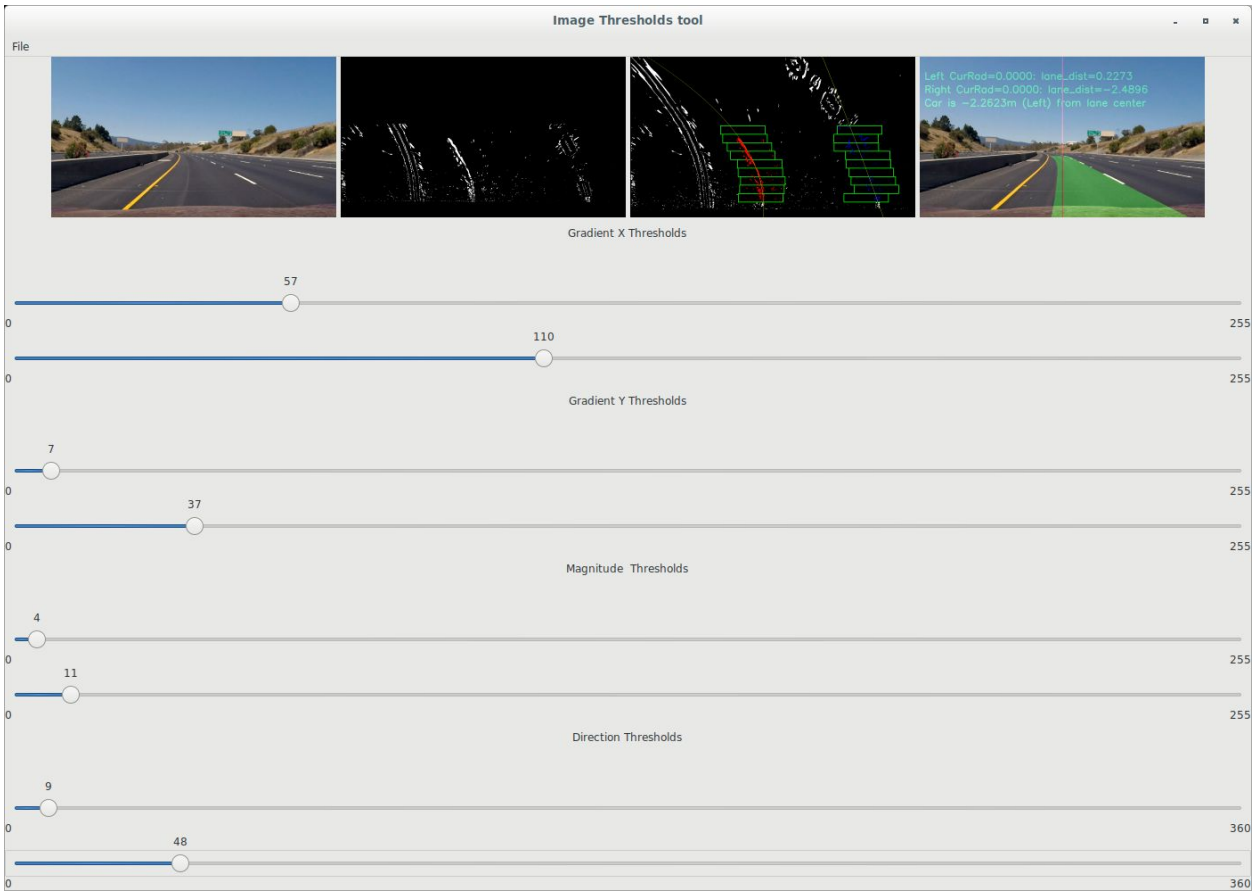
I put together the various techniques from the lesson , gradient x axes, gradient y axes , gradient magnitude and direction, using different kernel sizes for the sobel transform and then using different thresholds , finally combining them into a binary image for line detection.

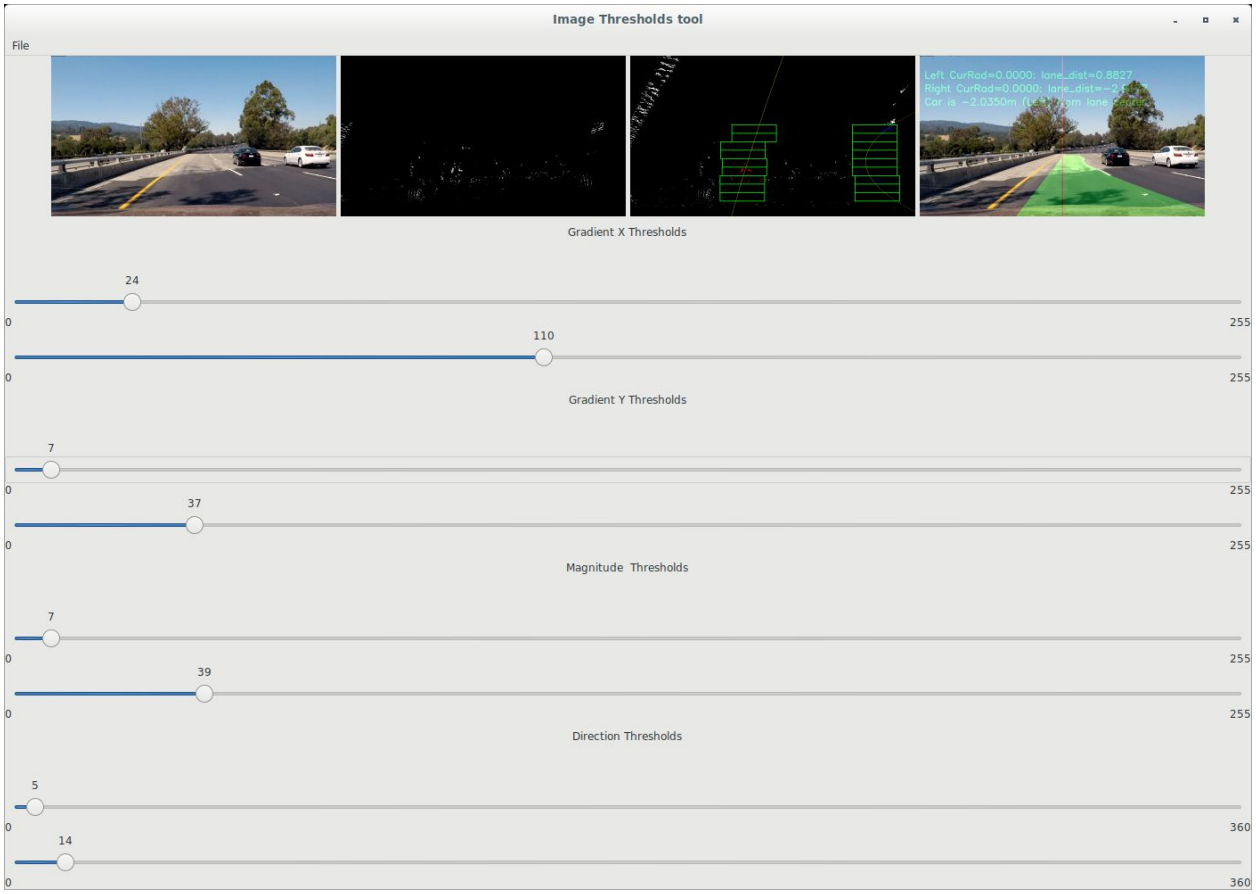
Initially I did try to experiment with a jupyter notebook I found it to be a very slow process to experiment. So wrote a visualization tool in wxPython to help. This is in the github repo as app.py

There are some screenshots below to show it in action. These screenshots are from an earlier version where I used the full pipeline, later on I just focused on the binary gradient portion of the pipeline and used diagnostics output within the video to better debug issues in the state machine portion of the lane detector.

This is an example of using grayscale colorspace with various gradient settings. In this case it's detecting a difference in the road color in the middle of the lane and getting a false positive in the search.

This tool was invaluable during the project.





Af first I choose to use the s_channel from an HLS colorspace conversion of the image, plus a sobel kernel of 9 and the thresholds you will see in the code. This code is encapsulated in the BinaryImage class in file binary_image.py.

This class can be initialized with a pickled file of threshold values - created by the app.py tool. My first attempts used various different thresholds on the s_channel only and it was somewhat successful. However even when smoothing and outlier rejection was implemented I found that it wasn't great for every image and frame and did very poorly on the harder challenge videos.
So I combined the s_channel from HLS with the red channel from the RGB version of the image with a different set of thresholds. This was somewhat better but there were false positives in these images too.
Eventually after the first project reviewer suggested YUV. And I had some experience with using YUV to extract lines in the first project in this term. I setteled on a Y and V channels as the final implementation.

The actual application of the gradient occurs in the BinaryImage class in binary_image.py

- **_build_gradients_for_source** , which calls

- **_build_grad_x**
- **_build_grad_y**
- **_build_grad_mag**
- **_build_grad_dir**

The results of these methods are combined to produce the binary warped image used for lane finding.

The LaneLines class utilizes a logical OR of the Y and V channels binary images to then search for lane pixels.

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

The code for detecting the line pixels is in the main LaneLines class.
This class maintains the state of the line search , assumes that there is a left and right line and when detects lines, sanity checks them and smooths them.

In **_find_line_full_search** I used a slightly modified lane search from the technique in the lesson. I mostly used the same idea of a sliding window , but I reduced the search space as I noticed that both the bottom of the image and the top were noisy and less indicative of lane lines.

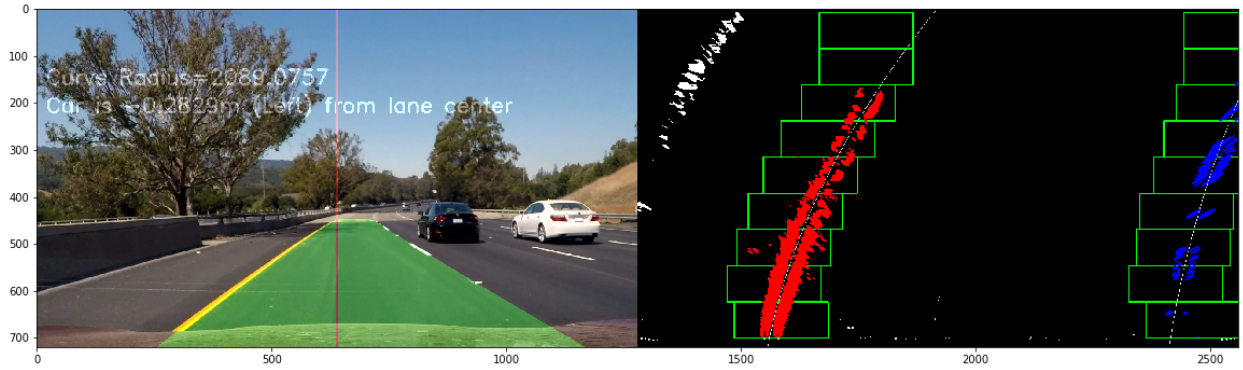
When the first few images are processed I used the histogram + sliding window search. I provide a simple sanity check on the two lines detected and if it passes the polynomial coefficients are stored in a sample array and a weighted average is used to create a smoothed curve. One a minimum threshold of smooth samples have been detected the search switches to the fast search.

Once a successful fit was achieved I then used the polynomial fit from the previous frame + a margin to locate the lines in the next image. This is in the method **_find_lines_from_smooth**. After each detection, a sanity check is implemented by calculating the average lane width between the 2 polynomial curves and if the mean is within a range of pixels widths and below a certain standard deviation pixels. These are configured as hyper parameters in the Line class. I experimented with different values to get the best results. The final values are here.

When I first implemented this I found that some of the video images would completely lose the lane, and even though I had some rudimentary checks for when there were no pixels detected this wasn't robust enough.

So I implemented a couple of approaches to smooth the input data. First on the incoming video frames , where basically I keep the last 3 images in a buffer and merged these together before running the line search. Secondly I implemented the weighted average smoothing on the polynomial coefficients.

This is an example of the lane line detection, the detected curve is plotted in yellow.



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.

I captured each line in a separate class - (line.py) This class provides a function to calculate the curvature **_calculate_curvature.**

This code assumes the following constants to convert from pixel space to meters.
I'm pretty confident in the x meters per pixel - as I measured this from the perspective transform. However it's more difficult to get a good reading in the y dimension. So I used the suggested values from the lesson.

ym_per_pix = 30 / 720 # meters per pixel in y dimension
xm_per_pix = 3.7 / 1080 - 230 # meters per pixel in x dimension

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

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

I also created an image with an extra panel of diagnostics per frame to visualize performance during the processing of the full video.

Here's a link to my video with diagnostics

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?

I had many issues with this project. The actual code for lane finding was relatively straightforward. The first main issue was trying to find good values for the thresholds. This drove me to spend a fair bit of time on my visualization tool. I would have liked to spend more time on the tool, extending it to select different color spaces, cropping regions, maybe even import the video and allow more interactive debugging of the images. I think with this it would be easier to find the best combination of color spaces, gradients etc to handle more situations.

My initial submission did not sanity check the curves - for this I choose to use the average and standard deviation of the xl difference (essentially the lane width in pixels) at each y. This gave a pretty good estimation of how parallel the detected lines were. This could be further improved by checking how much each curve has deviated from the previous curve and filter out large changes. Lastly , my current sanity checking only covers comparing the lines from the current image, the same technique could be used for the newly detected line against the other line's smoothed curve - this would maintain the accuracy of the good lane when one line is badly detected.

Another approach that occured to me when looking at some of the harder examples, would be to process the different color space images separately and from the set of lines produced, compare the different combinations of lines detected on the left and the right for probably pairs of lane lines. This helps in cases where the red channel picks up false positives due to differences in the color of the surfaces (tar lines etc)

Lastly ,the hyperparameters for detection could be tuned more systematically than my rudimentary testing. I'm pretty sure these could be improved upon to get better detections.