Self-Driving Car Engineer Nanodegree

Advanced Lane Finding on the Road

Completed by Sitaram Ryali

Overview

In this project, I have written a software video pipeline to identify the lane boundaries in a video from a front-facing camera on a car. The camera calibration images, test road images, and project videos are available in the project repository.

GitHub repo: https://github.com/SitaramRyali/Self_Driving_Car-NanoDegree/tree/master/Project-LaneLines_Finder/CarND-Advanced-Lane-Lines-Project

Data/Files Explanation:

Project Source Code: Project_Code/Advanced_Lane_Detector.ipynb

Camera Calibration p file: Project_Code/wide_dist_pickle.p

Source Video: project_video.mp4

Output Video: project_video_lanes_detected.mp4

Goals/Steps

My pipeline consisted of 10 steps:

1. Import and initialize the packages needed in the project,
2. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
3. Apply a distortion correction to raw images.
4. Use colour transforms, gradients, etc., to create a threshold binary image.
5. Apply a perspective transform to rectify binary image ("birds-eye view").
6. Detect lane pixels and fit to find the lane boundary.
7. Determine the curvature of the lane and vehicle position with respect to centre.
8. Warp the detected lane boundaries back onto the original image.
9. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.
10. Create a pipeline for the above steps to process the image.

Import and initialize the packages needed in the project

I have chosen to use some well-known libraries:

- OpenCV - an open source computer vision library,
- Matplotbib - a python 2D plotting libray,
- Numpy - a package for scientific computing with Python,
- MoviePy - a Python module for video editing.

Compute the camera calibration using chessboard images

The next step is to perform a camera calibration. A set of chessboard images will be used for this purpose.

For each image path:

- reads the image by using the OpenCV cv2.imread function,
- converts it to grayscale using cv2.cvtColor,
- find the chessboard corners using cv2.findChessboardCorners

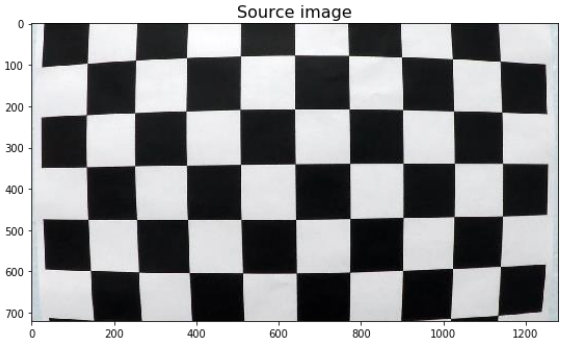
Finally, the function uses all the chessboard corners to calibrate the camera by invoking cv2.calibrateCamera.

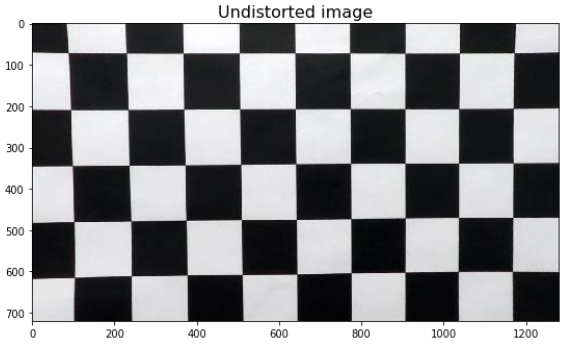
The values returned by cv2.calibrateCamera will be used later to undistort our video images.

Apply a distortion correction to raw images

Another OpenCv funtion, cv2.undistort, will be used to undistort images.

Below, it can be observed the result of undistorting one of the chessboard images:





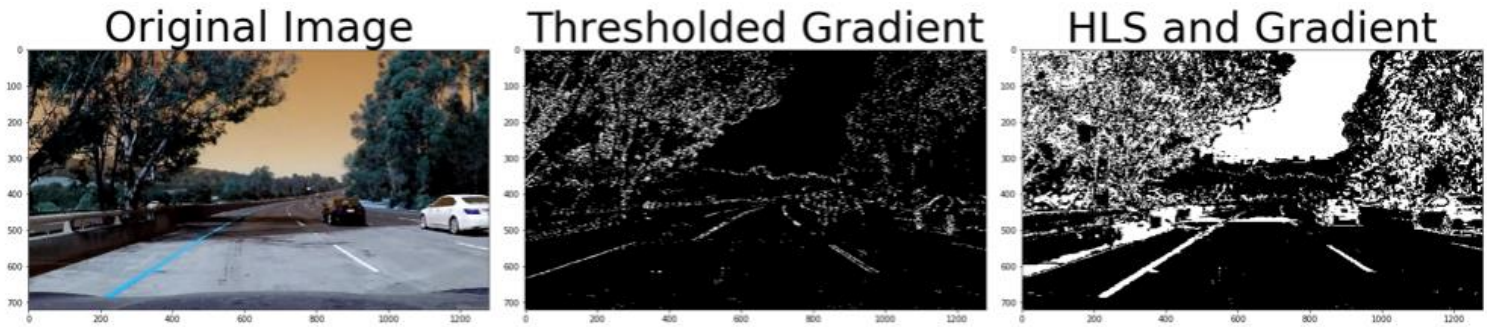
Use colour transforms, gradients, etc., to create a thresholded binary image.

In this step, define the following functions to calculate several gradient measurements (x, y, magnitude, direction and color).

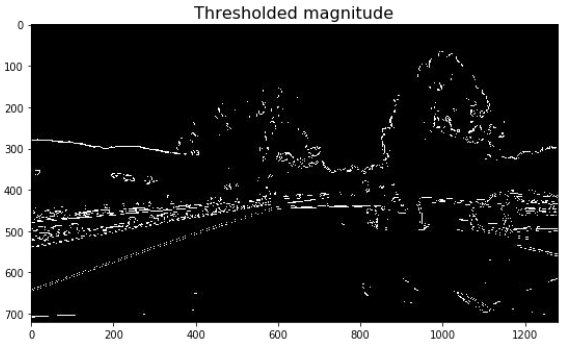
- Calculate directional gradient: abs_sobel_thresh().
- Calculate gradient magnitude: mag_thres().
- Calculate gradient direction: dir_thresh().
- Calculate color threshold: hls_select().

Then, combined_thresh() will be used to combine these thresholds, and produce the image which will be used to identify lane lines in later steps.
Below, I have copied the result of applying each function to a sample image:

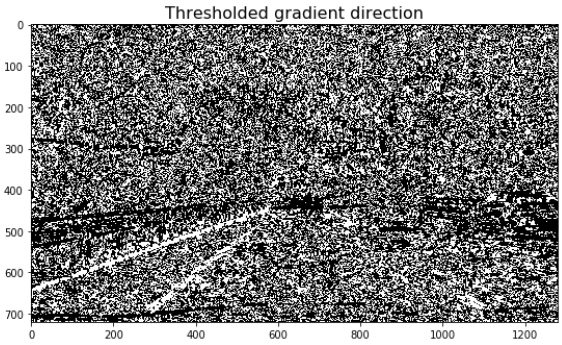
- Calculate directional gradient for *x* and *y* orients:



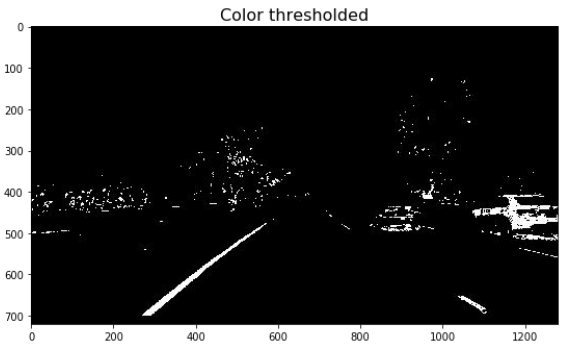
- Calculate gradient magnitude



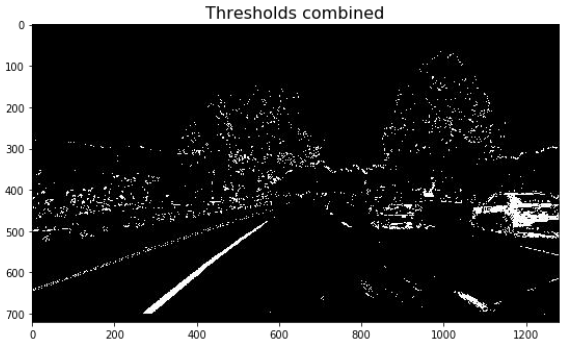
- Calculate gradient direction



- Calculate colour threshold



The output image resulting of combining each thresh can be observed below:



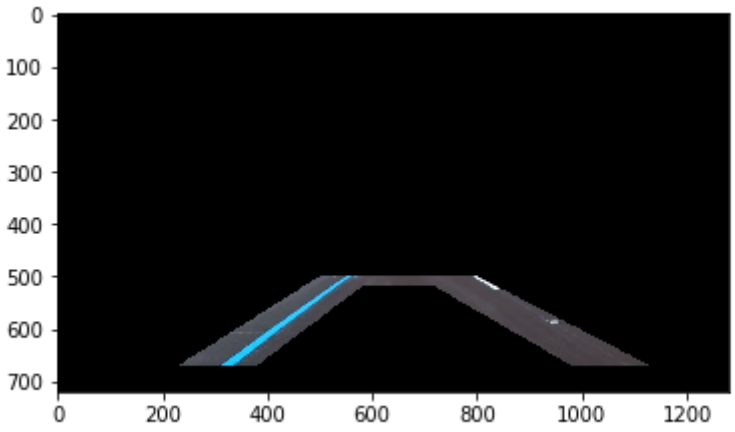
Apply a perspective transform to rectify binary image ("birds-eye view").

The next step in our pipeline is to transform our sample image to *birds-eye* view.

select the region of Interest in image view

Here region of interest-based image selection is getting performed to process and make the image available for bird's eye view of road lines

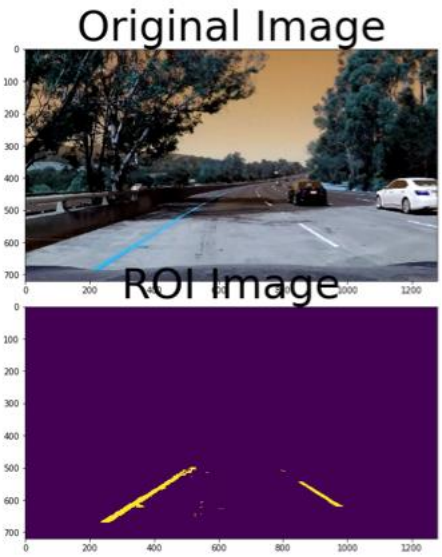
- Step 1: Made the selection for the road lines region interested based on the array of **Vertices.**
- Step 2: Removed the road noise occurring the within the ROI image using the array **noisy vertices**

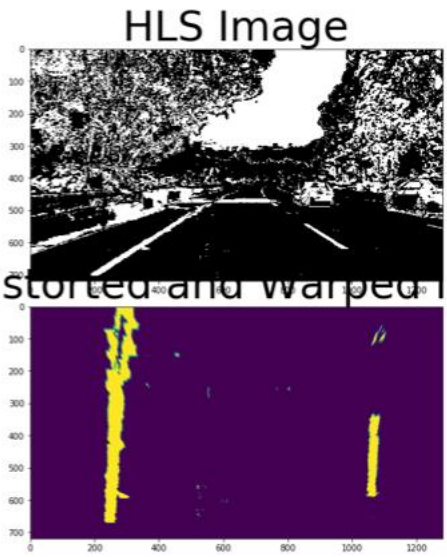


Perform the perspective transform for the ROI frame

- Then, you have to define the destination coordinates, or how that trapezoid would look from *bird's-eye* view.
- Finally, Opencv function cv2.getPerspectiveTransform will be used to calculate both, the perspective transform *M* and the inverse perspective transform Minv.
- *M* and *Minv* will be used respectively to warp and unwarp the video images.

Please find below the result of warping an image after transforming its perspective to birds-eye view:

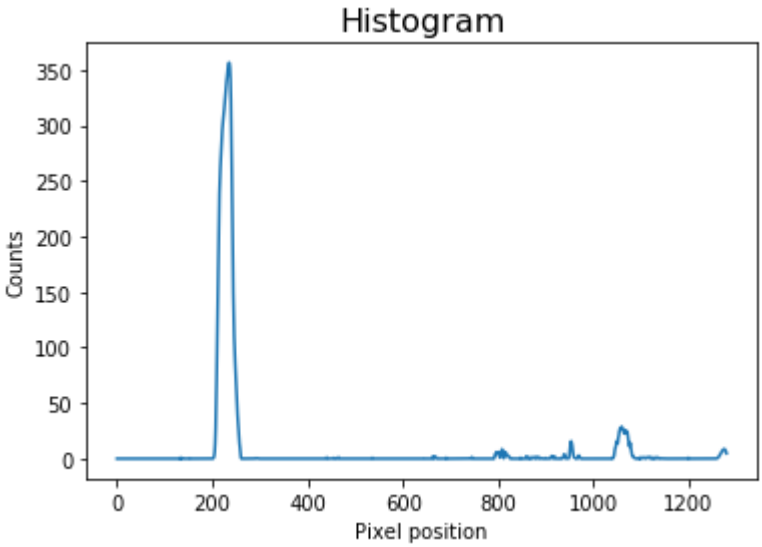




Detect lane pixels and fit to find the lane boundary.

In order to detect the lane pixels from the warped image, the following steps are performed.

- First, a histogram of the lower half of the warped image is created. Below it can be seen the histogram and the code used to produce it.

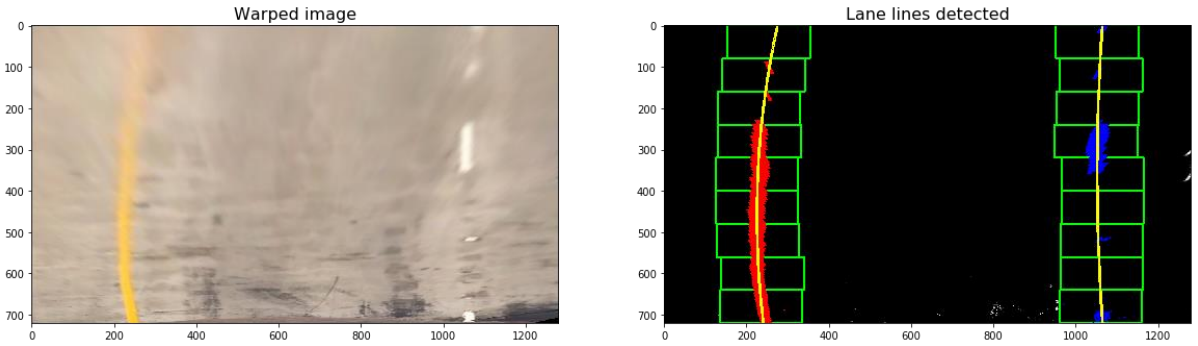


Then, the starting left and right lanes positions are selected by looking to the max value of the histogram to the left and the right of the histogram's mid position.

- A technique known as *Sliding Window* is used to identify the most likely coordinates of the lane lines in a window, which slides vertically through the image for both the left and right line.

- Finally, using the coordinates previously calculated, a second order polynomial is calculated for both the left and right lane line. NumPy's function np.polyfit will be used to calculate the polynomials.

Please find below the result of applying the detect_lines() function to the warped image:



The complete code for this step can be found in the nineteenth code cell of this Jupyter notebook.

Once you have selected the lines, it is reasonable to assume that the lines will remain there in future video frames. detect_similar_lines() uses the previosly calculated *line_fits* to try to identify the lane lines in a consecutive image. If it fails to calculate it, it invokes detect_lines() function to perform a full search.

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

At this moment, some metrics will be calculated: the radius of curvature and the car offset. The code is quite self-explicative, so I'll leave to the reader its lecture.

Processing video Frame pipeline:

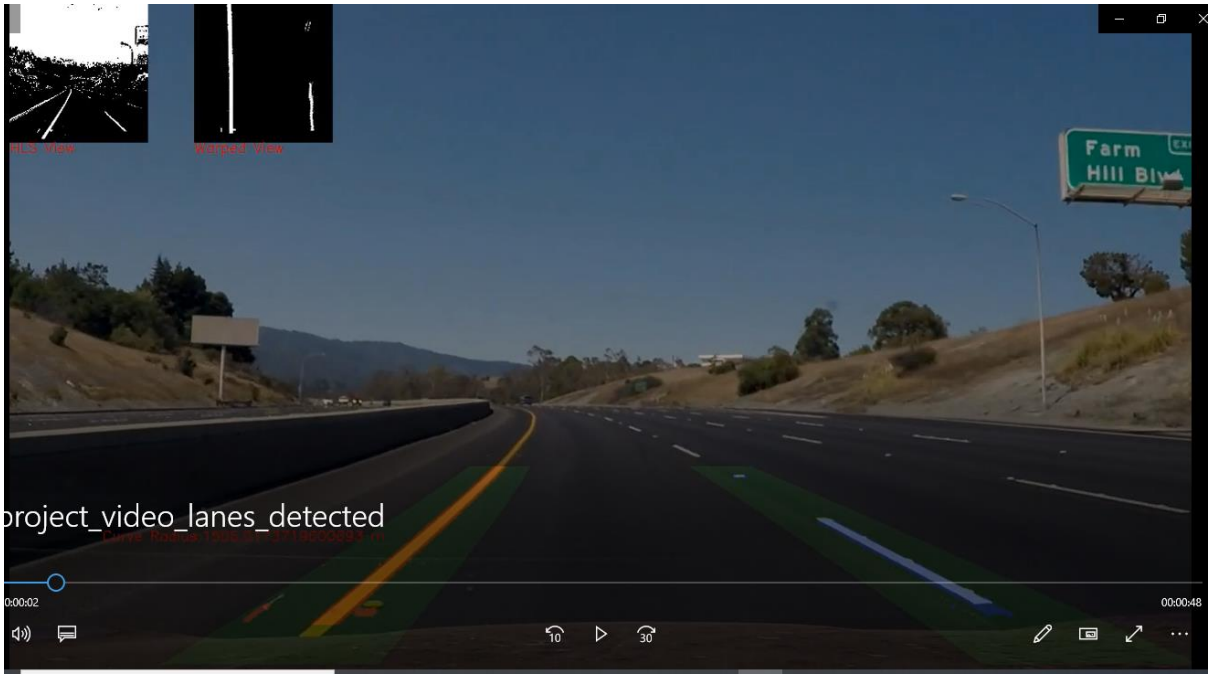
This is the crucial step of the algorithm.

Here above all steps are induced in pipeline to process each frae of the project video and then evaluate to find the lane lines.

Additionally, I have included two views to help the reader about what is happening inside.

HLS view of the image after colour thresholding

Warp view of the ROI image of current frame. These two are added to top left corner of the video.



Conclusion:

This has been a really challenging project and I am quite happy with the results.

Nevertheless, there is still some room for improvements:

- Keep track of the last several detections of the lane lines were and what the curvature was, so it can properly be treated new detections:
- Perform some sanity checks to confirm that the detected lane lines are real:
    - Checking that they have similar curvature,
    - Checking that they are separated by approximately the right distance horizontally,
    - Checking that they are roughly parallel.
- Smoothing:
    - Even when everything is working, line detections will jump around from frame to frame a bit and it can be preferable to smooth over the last n frames of video to obtain a cleaner result. Each time a new high-confidence measurement is calculated, it can be appended to the list of recent measurements and then take an average over n past measurements to obtain the lane position to draw onto the image.

I will implement these changes in the future, but I am running out of time to complete term 1, so I have decided to leave them for the next weeks.

Finally, the pipeline might fall if a white car is too close to the left or right lanes, and also when a white car is in front of our car and quite close to us.