Advanced Lane Detection Project

Brief steps of this project are as follows:

- 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.

The Zipped file submitted consists of the following:

1. README in pdf format.
2. Jupyter notebook (Source code)
3. Final output images
4. Warped Images
5. Camera_cal_pickle.p: file contains camera calibration matrix and distortion coefficients.
6. Video Output

Overall Code Implementation:

A - Helper Functions:

1. Calculate_Sobel : Function for calculating Sobel gradient.
2. Hls_threshold : For extracting channels H/L/S out of a provided image.
3. region_of_interest : Marking a region of interest from an image and masking any other pixels outside the marked region.
4. find_lane_pixels_blind : Search blindly for lane lines in the given warped image. It is always used in images/ First frame of video when we have no clue/history of where we may find the lane lines in an image/frame.
5. search_around_poly : If we have history where we can find the lane lines e.g. next frames of a video, we can just search around the previously known lane line by a +/- margin (Hyperparameter)
6. fit_polynomial : It fits in a polynomial function for the discovered lane lines from find_lane_pixels_blind, search_around_poly.
7. draw_lanes : This function
8. measure_curvature_pixels : This function calculates the average curvature of detected lane lines, in addition to the car offset from center.
9. print_statistics_to_image : This function takes input of statistics (curvature and car offset) and prints it at the top of the output image.

B - Main Functions / Logic :

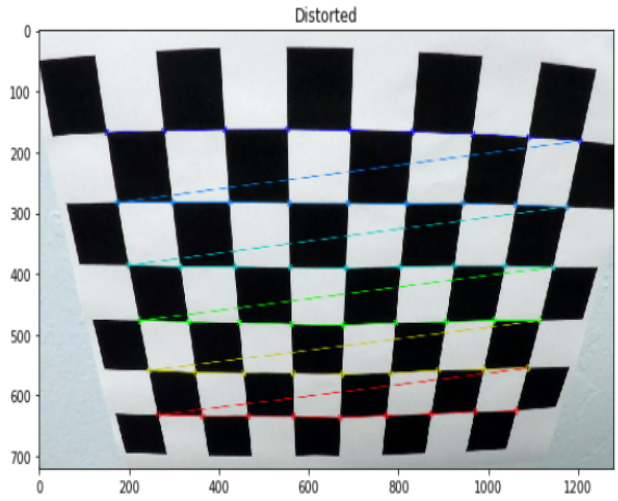
1. Camera Calibration: Code that runs through the calibration images (chess board images) and calculates the camera calibration matrix and distortion coefficients and saves them in a file called camera_cal_pickle.p. These values will be used later to undistort the input images from the car's captured images/videos.
2. Image_Pipeline: It takes the input images and draw lane lines on top of them along with statistics like curvature and car offset.
3. Video_Pipeline: It takes the input video and processes it frame by frame and outputs a video after detecting and drawing the lane lines for it.

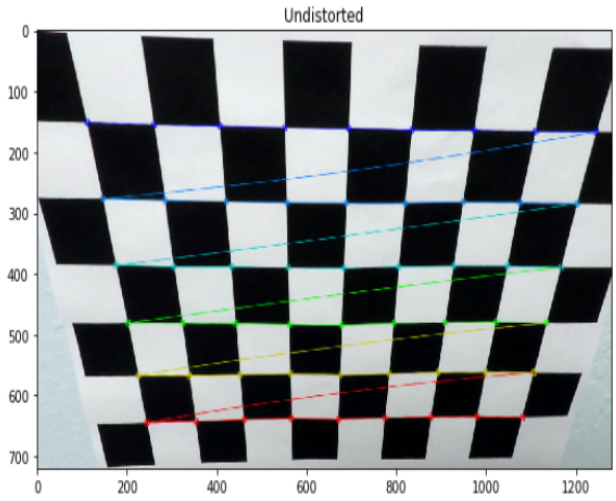
Detailed Code Implementation:

1 - Camera Calibration :

The camera takes the chess board calibration images and calculates the camera calibration matrix and distortion coefficients, it takes distorted images and un-distort them as shown below:

The calculated camera calibration matrix and distortion coefficients are saved to a file named “camera_cal_pickle.p”.
These values will be later used to undistort the input images from the car’s captured images/videos.





2 - Image Pipeline:

1 - Apply a distortion correction to raw images using camera calibration matrix and distortion coefficients previously calculated. This was done using the “cv2.undistort” , “pickle.load” functions.

2 - Calculate Adaptive threshold/Sobel threshold, H, L, S and generate combined image. I found the adaptive threshold to be quite productive on the images. This part was done using the “cv2.adaptiveThreshold” , “hls_threshold” functions.

Reference:
https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

3- Mark the region of interest for the combined image. This was done using the “region_of_interest” function.

4 - Apply a perspective transform to rectify binary image ("birds-eye view") using the “cv2.getPerspectiveTransform” function.

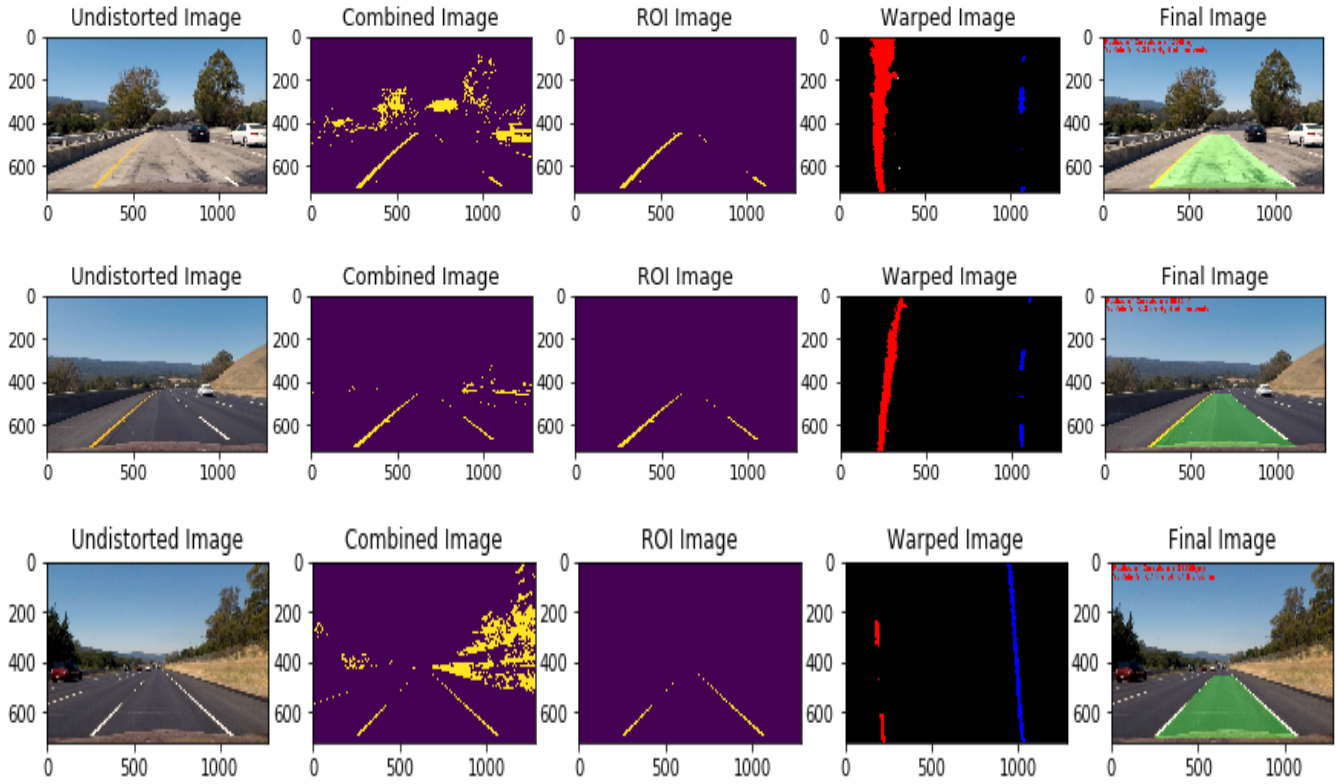
5 - Detect lane pixels and fit to find the lane boundary using functions: “fit_polynomial” , “find_lane_pixels_blind”.

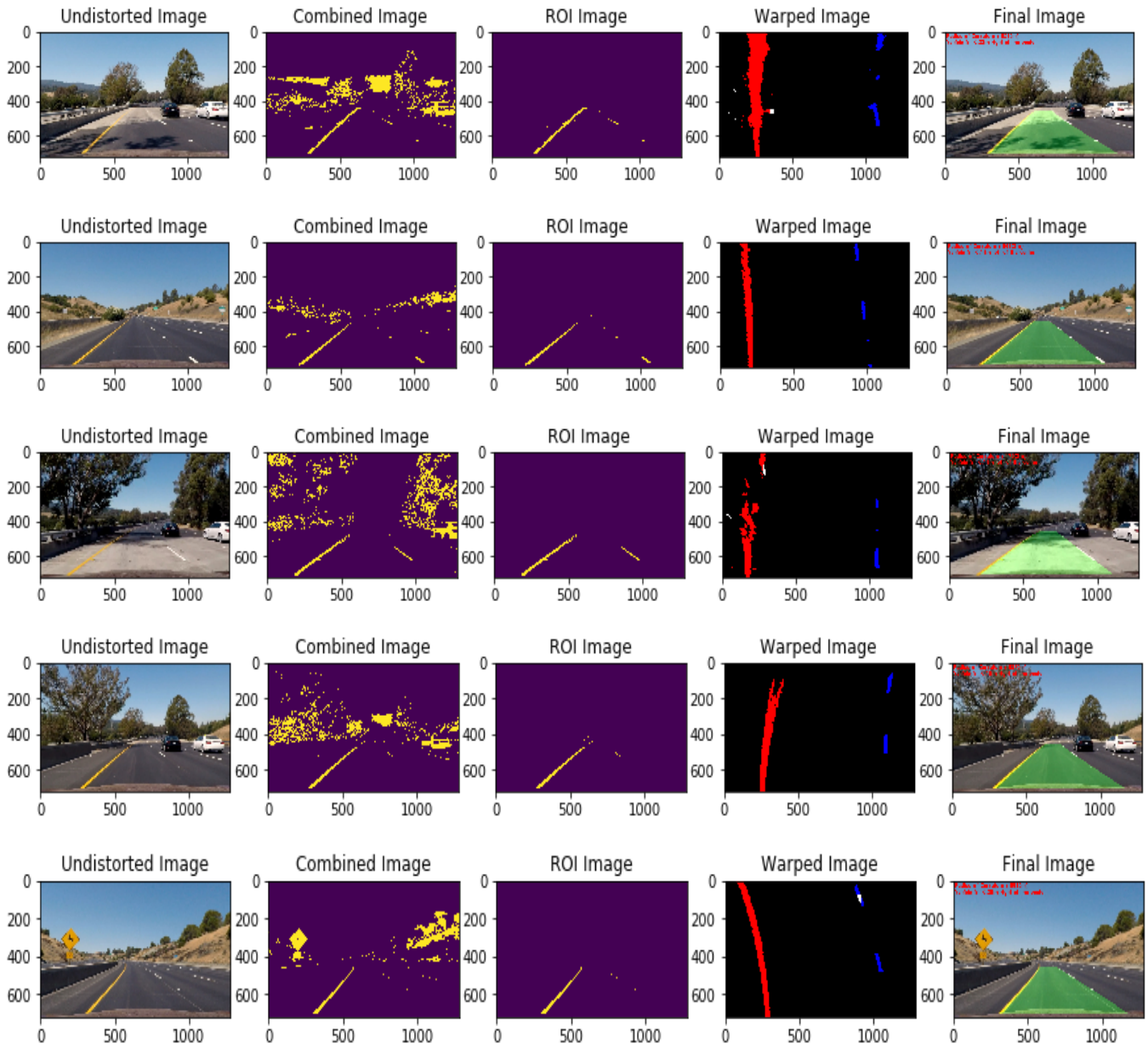
6 - Determine the curvature of the lane and vehicle position with respect to center using the “measure_curvature_pixels” function.

7 - Warp the detected lane boundaries back onto the original image using the “draw_lanes” function.

8 - Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position using “print_statistics_to_image” function.

Output Samples from Image pipeline:
I have attached the final output images in the zipped file.
Here are some screenshots of various stages during the code development.





3 - Video Pipeline:

The video pipeline followed exactly the same steps, the only difference was in step number 5. Finding lane lines were done by the “find_lane_pixels_blind” function at the first frame only. After getting a history/clue where we can search for the lane lines in the next frames, I searched around the previously detected lanes by the “search_around_poly” function.

Output of Video Pipeline:

I have attached the video output in the zipped file.

Discussion:

Although this architecture worked well for the provided video, honestly I found issues in the challenge videos.

Sometimes, it was very difficult to qualify/fit point or lane lines. Perhaps the lane lines did not appear in the region of interest or the road had some pavement color issues or very sudden curvatures. Some countries may not have the proper infrastructure.

So I am expecting in the next chapters to learn more techniques. I think it can get more complex and depend on other input sources like radar for example along with the cameras.