Advanced Lane Finding Project

The goals / steps of this project are the following:

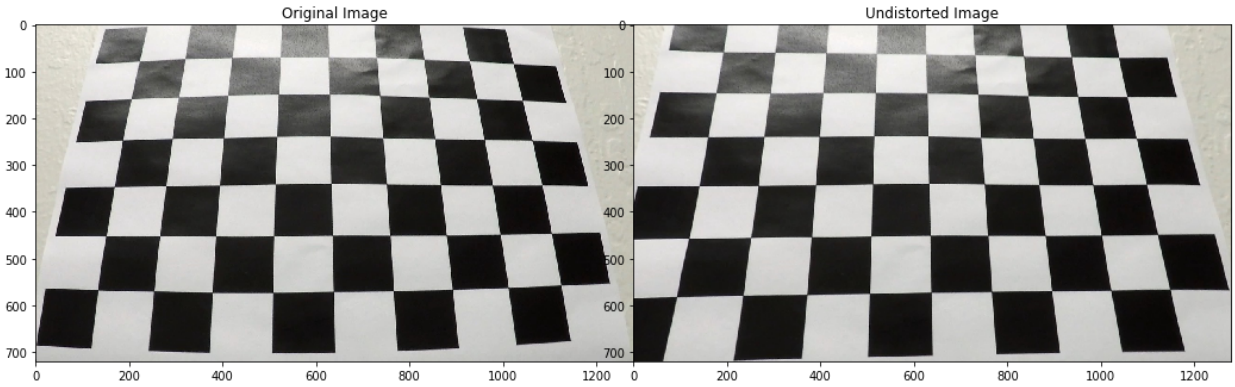
- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- 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.

Camera Calibration

The code for this step is contained in the first code cell of the IPython notebook located in "./P4.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:



Pipeline (single images)

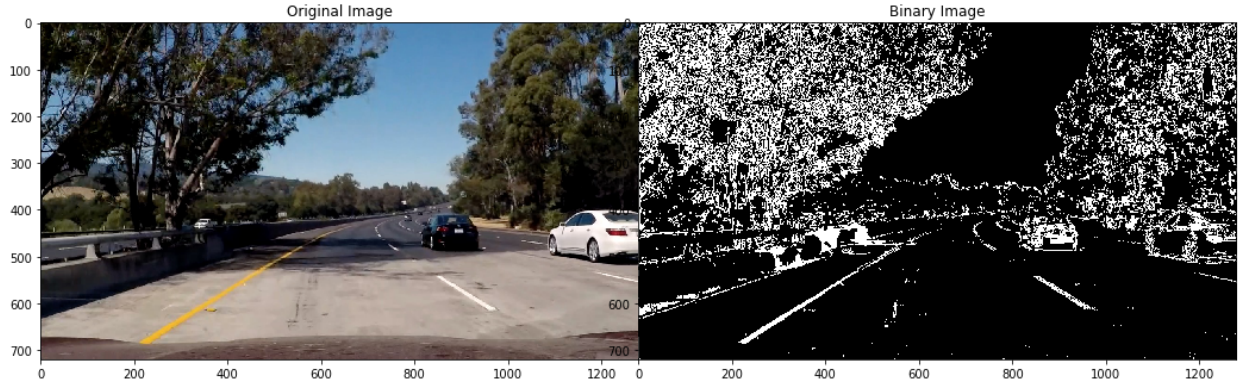
1. Provide an example of a distortion-corrected image.

The function undistort_image uses the mtx, dist computed by cv2.calibrateCamera and undistort the image. I use this function to undistort real world lane image

2. Create thresholded binary image.

I used a combination of color and gradient thresholds to generate a binary image (in the 3rd code cell of the IPython notebook located in "./P4.ipynb".). I converted original image to HLS space and apply sobel to the image on X direction. Here's an example of

my output for this step.



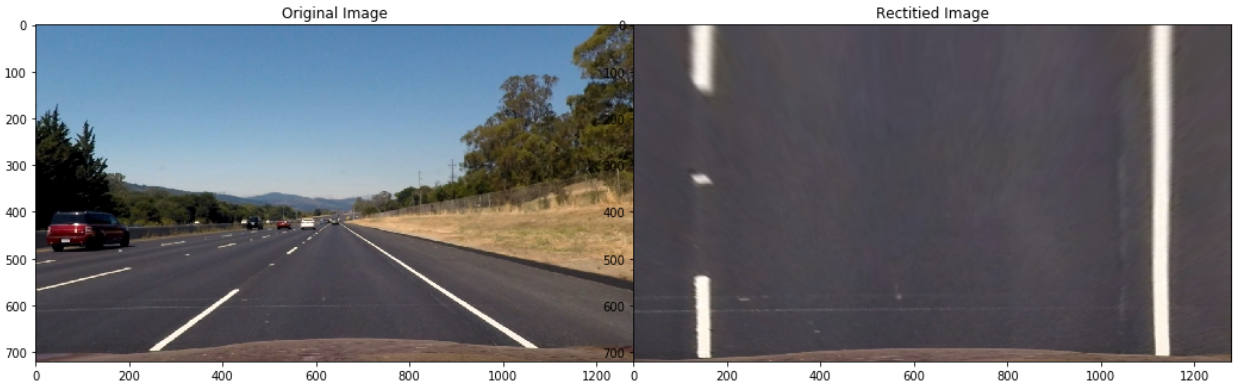
3. Rectify image

The code for my perspective transform includes a function called `rectify_image()`, which appears in the 4th code cell of the IPython notebook located in "./P4.ipynb". The rectify_image() function takes an image (img), and an optional parameter show_image as input. In this function I defined source (src) and destination (dst) points. I chose the hardcode the source and destination points in the following manner:

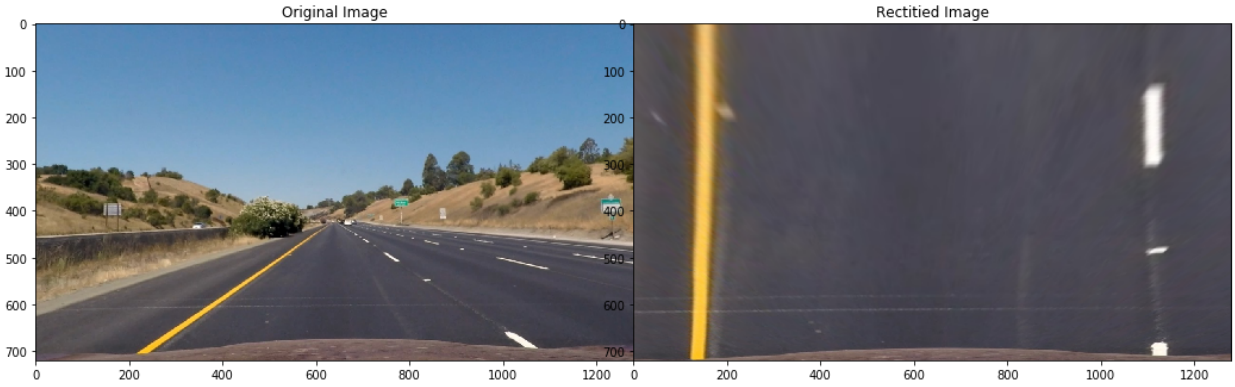
```
src = np.float32([[739, 482], [550, 482],
                  [318, 648], [996, 648]])

dst = np.float32([[img_size[0]-offsetx, offsety], [offsetx, offsety],
                  [offsetx, img_size[1]-offsety],
                  [img_size[0]-offsetx, img_size[1]-offsety]])
```

These data are decided by experiment I ran on test_images/straight_lines2.jpg.

Here are the examples:

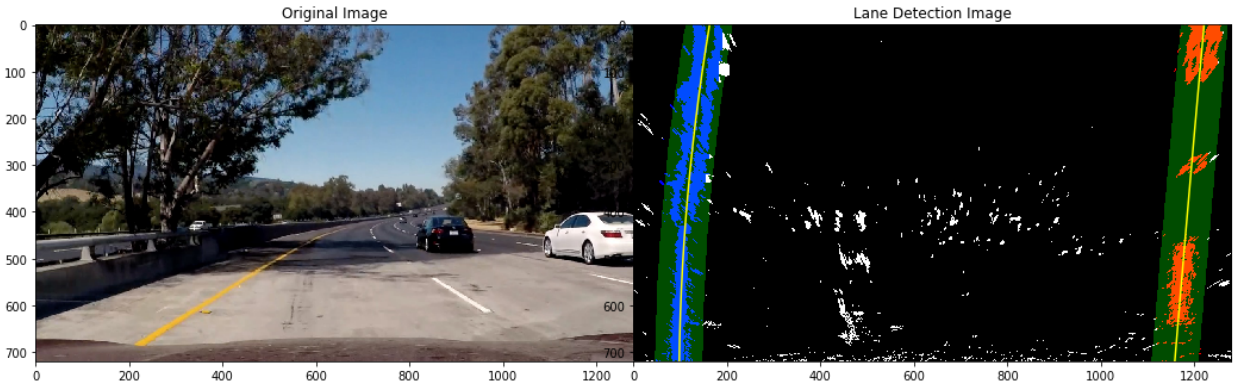


I verified that my perspective transform was working as expected by drawing the src and dst points onto another test image and its warped counterpart to verify that the lines appear parallel in the warped image. I use cv2.getPerspectiveTransform to get transform matrix and cv2.warpPerspective doing the perspective transform.



4. Identify lane-line pixels and fit their positions with a polynomial.

Corresponding part coding located in 6th cell of ./P4.ipynb. Before process, I created a function convert_to_rectified_binary_image to wrap previous functions can return a rectified binary image. Then I create a function calls find_lane. Except img and show_image option, it also takes optional input left_fit and right_fit, which are ploynomial fit data for left and right lines. If these two input are set, we will search based on these two data and image. Otherwise we will slide window to find lines info. It will return left_fit, right_fit and ploty.



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

I do this in the 7th cell of ./ipynb. I calculate ploty first and using left_fit, right_fit to compute left_fitx and right_fitx. Also I defined ym_per_pix, xn_per_pix to help me convert distance in pixels to meters. I used the algorithm introduced in the class to compute radius of curvature, here I used y_eval = 360, means the curvature I computed is the middle point of image. Also I used left_fitx[-1] and right_fitx[-1] to compute offset. The function is (1280 / 2 - (left_fitx[-1] + right_fitx[-1])/2) * xm_per_pix. If the number is negative means car is on the left side, otherwise it's on the right side of road.

The result returned by get_curvature are like below:

(2272.2624135806336, 46066.55730034459, 0.042876864954098669)

6. Visualization.

I implemented this step in 8th cell of ./P4.ipynb. Visualize_result function will take img, wraped, left_fit and right)fit as input. Here wraped is the rectified binary image. I used src, dst point to compute Minv. Using these data I visualize result on the original iamge.

Pipeline (video)

Here's a [link to my video result]

I created a Lan class to process images and detect lanes. It's in the 9th cell of P4.ipynb. The main idea is to store several recent info and ustalize them to compute image lane. I select the memory_size as 5, means only will take no more than 5 recent images. Also, any coefficient in the left_fit and right_fit need to be less than 20. Otherwise, we will consider we fail to capture the lines for this image.

Discussion

My function works well on project_video. For challenge and harder challenge it exposes some problems.For challenge video, it wrongly detected black line as boundary line. For harder challenge video, it does not perform well since the video is affected by sunlight, also the road condition is complex. For the challenge video, we can fix it by using new binary threshold. For example, only keep white and yellow pixels will help us to avoid such noise. But notice that, in real world some lanes are divided by black line, maybe it's better to use lane width to help us filter out suspicious line. For harder challenge, we can set larger memory_size to smooth the result. Also, if we detect the image is too light we may skip it. Finally, some complex algorithm can introduced here. For example, we can compute each line info based on another one. (We need to keep lines are parallel and their distance as the same value)We do several round iteration for each image, make lines are parallel and close to previous results. Or, maybe we can connect to GPS and using GPS data to some correction.