Advanced Lane Finding Project

Feng Xiao (xiaofeng.ustc@gmail.com)

Summary

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

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

1 Writeup / README

1.1 Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You are reading it!

2. Camera Calibration

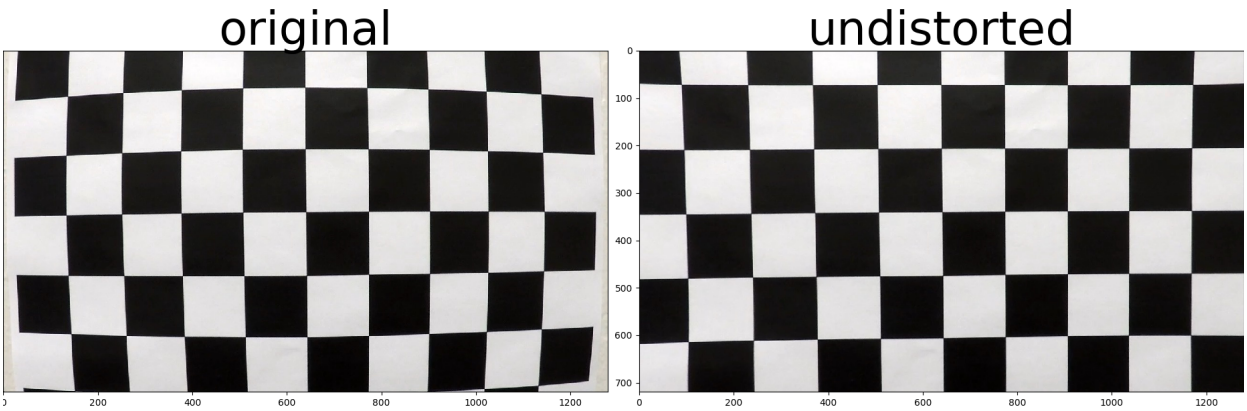
2.1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in *"src/camera_calibration.py"*. I have created a class called *CameraCalibrator* in this file. It is responsible for calibrating the camera and save the camera matrix and distortion coefficients to a pickle file to be used by the image undistortion part.

As introduced in the course, 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:



3. Pipeline (single images)

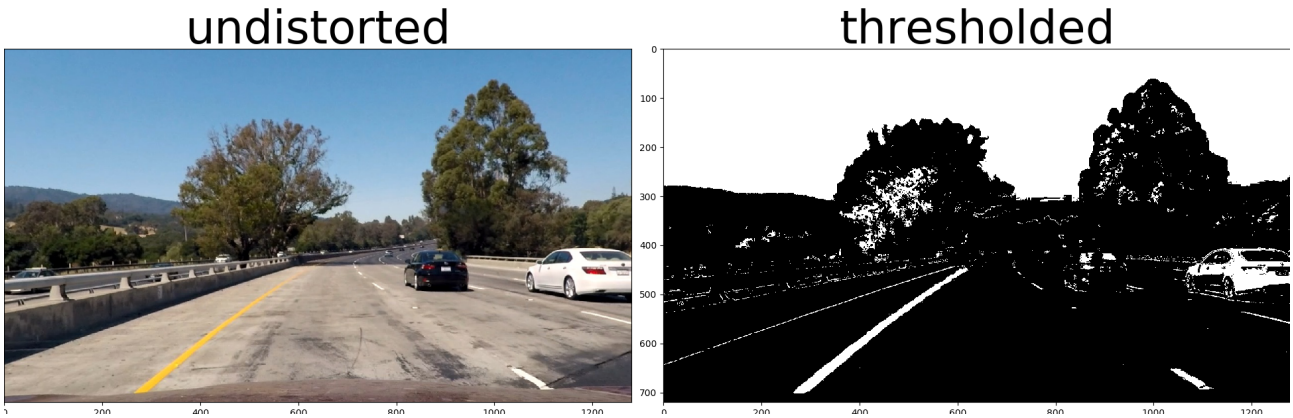
3.1 Provide an example of a distortion-corrected image.

For image undistortion, I have created a class called *ImageUndistorter* in *"src/image_undistortion.py"*, this class will load the camera calibration results generated by the CameraCalibrator, and perform the image undistortion. The following image shows the result of image undistortion applied to the test1.jpg in test_images folder.

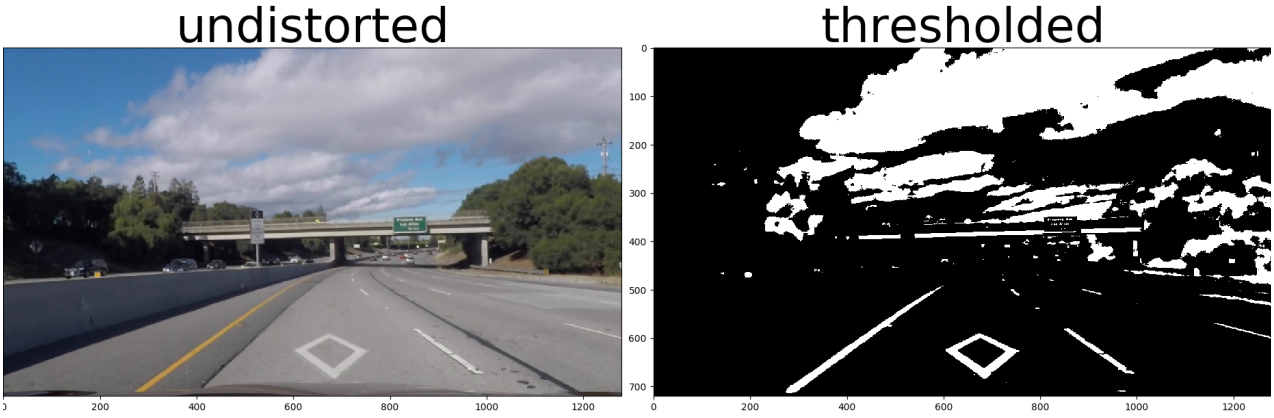
3.2 Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

Due to the different lighting conditions in the project_video and challenge_video, I used two different methods to create the thresholded binary image. You can find this from *"src/image_filter.py"*. In *"src/gradient_thresholds.py"* you can find all the gradient thresholds I have created, in *"src/color_thresholds.py"* you can find all the color thresholds I have created.

For the project video, as shown in *"src/image_filter.py"* function *image_filter_basic*, I applied both gradient and color thresholds. The gradient threshold is an "AND" combination of a sobel x filter with threshold (12, 255), a sobel y filter with threshold (25, 255) and a magnitude filter with threshold (100, 255). The color gradient threshold is also an "AND" combination of a HLS color space S channel filter with threshold (100, 255) and a HSV color space V channel filter with threshold (50, 255). And the filter result is an "OR" combination of both the gradient threshold and the color threshold. The result of applying this threshold to a project video image is shown as following:



For the challenge video, as shown in function *image_filter_challenge()* of *"src/image_filter.py"*, it only used the color threshold. It is an "OR" combination of a LAB color space B channel filter with threshold (139, 255) and a LUV color space L channel filter with threshold (174, 255). The result of applying this threshold to a challenge video image is shown as following:



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

In the code *"src/perspective_transform.py"*, I have created a class called *PerspectiveTansformer* to perform the perspective transform. The *transform* function is to perspective transform an input image and the *inverseTransform* is to inverse perspective transform an input image back.

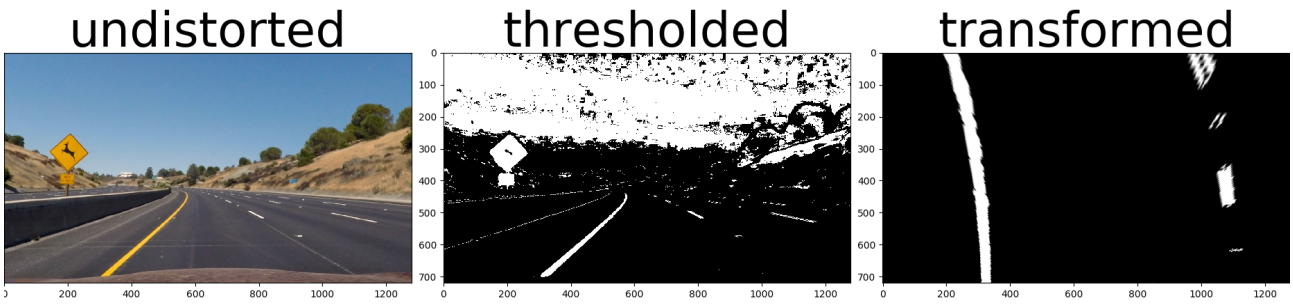
As to the source and destination points for the perspective transform, I select them manually. The details can be found in the code *"src/perspective_transform.py"* class *PerspectiveTransformer*. And the final value I use is as following in the order of [bottom_left, top_left, top_right, bottom_right]

- *SRC [(575, 464), (258, 682), (1049, 682), (707, 464)]*
- *DST [(250, 0), (250, 720), (1030, 720), (1030, 0)]*

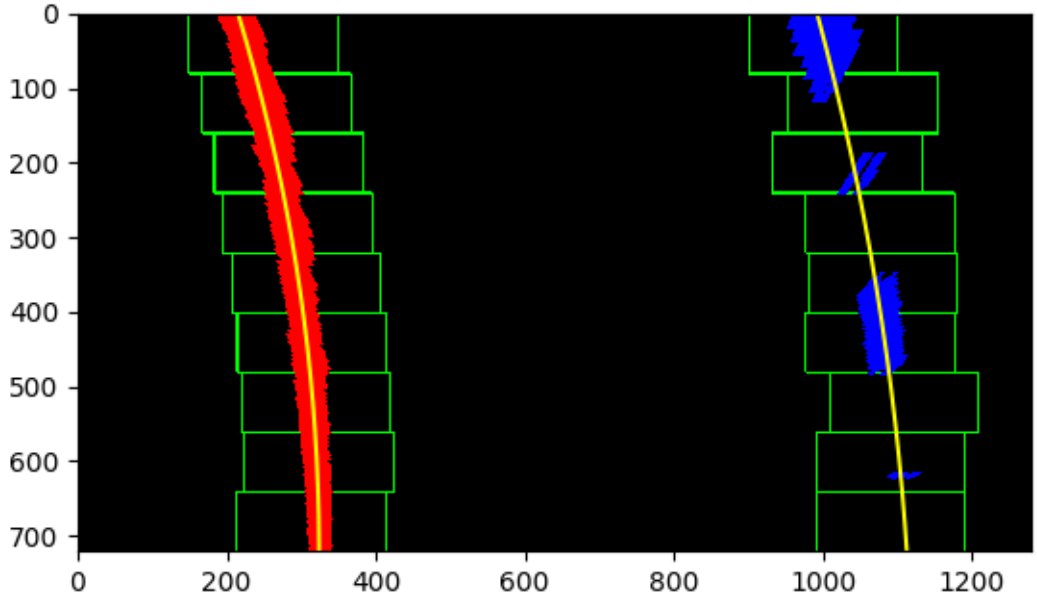
I have verified that my perspective transform was working as expected, the following shows the result of applying the perspective transform:

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

The method I have applied to identified lane-line pixels are as introduced in the course. After undistorting, thresholding and transforming the input image, I got a binary image with mainly the lane line pixels. It is as shown following:



And then, I applied the method of sliding windows to fit a polynomial for both the left and right lane. The code of this part cane be found in *"src/lane_detection.py"*. The class *LaneDetector* is mainly responsible for this work. For the lane-line pixels detected above, the *LaneDetector* provides the following fit result:



3.5 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.

The way I use to calculate the radius of curvature of the lane and the position of the vehicle with respect to the center can be found in *"src/lane_detection.py"* class *LaneDetector*.

The static method *LaneDetector.get_lane_curvature(lane_fit)* is to calculate the lane radius of curvature of the lane in meters. The input lane_fit is the fit coefficients got from the previous step.

The static method *LaneDetector.get_car_distance_to_lane(left_fit, right_fit)* is to calculate the position of the vehicle with respect to the center of the lane in meters. The input parameter left_fit and right_fit are the fit coefficients got from the previous step.

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

I have implemented in "src/pipeline.py" method Pipeline.blend_detection to plot the result back down onto the road. The following is example of the result:

On top, you can see the detected lane radius for the left and right lane, as well as an average between the two. Under that, you can see the distance of the car to the center of the lane. The detected area between the left lane and right lane is marked as green.

4. Pipeline (video)

4.1 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!).

For the project_video.mp4, my final video output can be found as the following link:

https://www.youtube.com/watch?v=vRrhHt9K-pM

For the challenge_video.mp4, my final video output can be found as the following link:

https://www.youtube.com/watch?v=L_dotRQ2_IM

5. Discussion

5.1 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?

For the whole pipeline, one of the key step is the image thresholding. And You may find in this project, I used two different thresholding functions. One for the project_video.mp4 and one for the challenge_vidoe.mp4. The reason for this is the lighting condition for these two videos are different, and with a limited time, I can't find a solution which works for both cases. Therefore I use different thresholding functions for different videos. So one thing to improve in future is to overcome the lighting difference in different weather conditions, so we can use the same thresholding for different videos.

For the challenge video, it has some part under the bridge. And my current thresholding can not detect any lane in this case since the major lane area is too dark. So another topic to make the pipeline more robust would be about detecting lanes in dark lighting conditions. In this case, I guess probably we could do some histogram analysis and change the brightness according.