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.

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

Camera Calibration

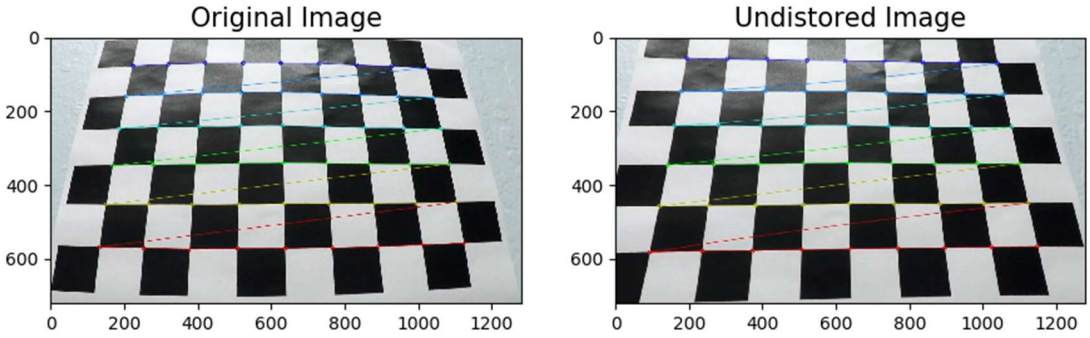
1. **Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.**

The code for this step in the camera_calibration.py which located in **camera_cal** folder. run the `camera_calibration.py` to process the calibration. In the function `camera_cal()` , the process as below:

- prepare the objp[], which is same matrix as chessboard 9X6
- create list objepiont[] and imgpionts[] to hold the 3D and 2D pioints.
- go through all the calibration image to search for conerner. use opencv findChessboardCorners function.
- use opencv calibratteCamera() to get mtx and dist
- write the calibration paramter to a pickle file *camera_cal.pk*

To test if the pickle file which contain the 'mtx' and 'dist' work. use the *test()* function to test and visualize the result.

- read the pickle and load the mtx and dist
- read test image
- use `cv2.undistort()` to undistort the test image. the result as below



Pipeline (single images)

`pipeline.py` include the class Pipeline, the function `pipeline()` which used to handle the image.The process described below:

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

use the cv2.undistort and the parameter from pickle file `camera_cal.pk` , to get a undistort image, I had made a distortion correction for test1.jpg , the result as following:

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.

In order to identify lane lines, we need to focus on the yellow and white lines, and also need to eliminate the influence of environmental noises. The code is in the `image_process.py`, the process described below:

- ✓ Firstly, threshold filtering of yellow and white lane lines was carried out by using the fuction of `yellow_white_thresh (img, y_low=(10,50,0), y_high=(30,255,255), w_low=(180,180,180), w_high=(255,255,255))`;
- ✓ Second, convert the image from RGB to HLS color space, and filter the x-direction gradient and color threshold of the s channel to eliminate

noise，the function saturation_threshold(img,s_thresh=(170,255), sx_thresh=(20, 100));

- ✓ last，combined the first two results to get a binary image by using function binary_combined(img);

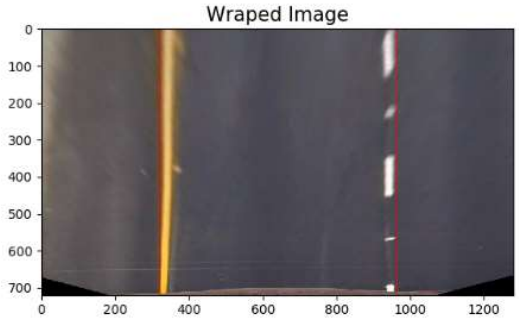
Here's an example of a binary image of my output for these steps.

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

The code for my perspective transform is includes a function called warper().The warper() function takes as inputs an image (img), as well as source (src) and destination (dst) points. I chose the hardcode the source and destination points in the following manner:

```
src = np.float32([[(img_size[0] / 2) - 63, img_size[1] / 2 + 100],
                  [((img_size[0] / 6) - 20), img_size[1]],
                  [(img_size[0] * 5 / 6) + 60, img_size[1]],
                  [(img_size[0] / 2 + 65), img_size[1] / 2 + 100]])
dst = np.float32([[(img_size[0] / 4), 0],
                  [(img_size[0] / 4), img_size[1]],
                  [(img_size[0] * 3 / 4), img_size[1]],
                  [(img_size[0] * 3 / 4), 0]])
```

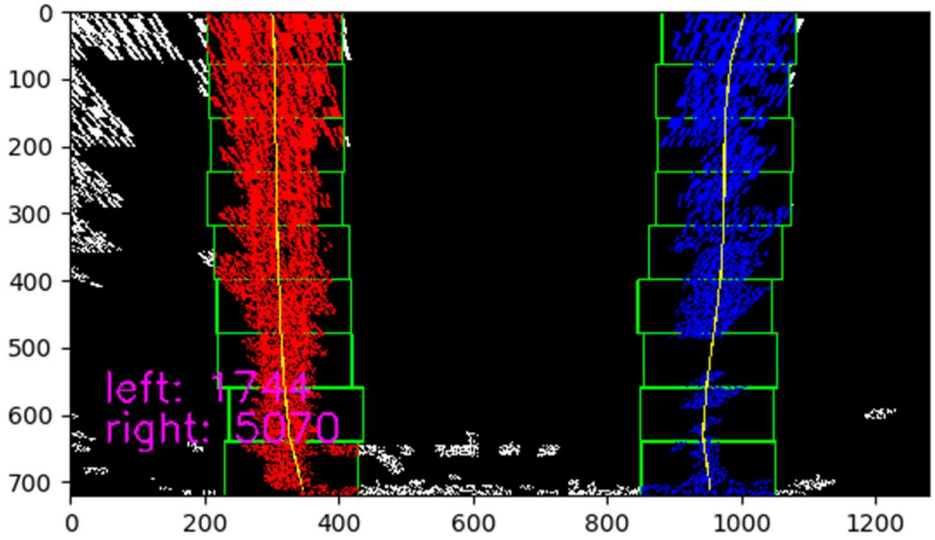
I verified that my perspective transform was working as expected by drawing the src and dst points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



4. Lane-line pixels detection and fit their positions with a polynomial

The lane_detection.py implement the lane-line pixels detection. the function find_lane_pixels() use a slide window to find the lane-line pixels, after get the lane_pixels, use np.polyfit() to get polynomial paratmeters, this is done in get_polynomial() .

To visualize the search result and fit polynomial, use fit_polynomial() function. the visualized result as below. show the search window/lane pixels/fit polynimial.



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

The radius of curvature calculation was done by the measure_curve.py, the function measure_curv() used to calculate radius.
Firstly, transform the lane piont from pixels to meters,

```
# Transform pixel to meters
leftx = leftx * xm_per_pix
lefty = lefty * ym_per_pix
rightx = rightx * xm_per_pix
righty = righty * ym_per_pix
```

then fit these data use np.polyfit()
After get the polynomial parameter, use the function R = (1+(2Ay+B)^2)^3/2 / (|2A|)

For the offset, it is similar, transfer pixel to meter, compare the lane center with picture center to get offset. these are in the function measure_offset().

6.Plot lane area and display the radius and offset.

In the Pipeline class, use below code to visualize the lane and calculation. to avoid number quick jump in the screen, display the last 15 frame average data.

```
def project_fit_lane_info(self, image, color=(0,255,255)):
    """
    project the fited lane information to the image
    use last 15 frame average data to avoid the number quick jump on screen.
    """
    offset = np.mean(self.offset[-15:-1]) if len(self.offset) > self.smooth_number else np.mean(self.offset)
    curverad = np.mean(self.radius[-15:-1]) if len(self.radius) > self.smooth_number else np.mean(self.radius)
    direction = "right" if offset < 0 else "left"
    str_cur = "Radius of Curvature = {}(m)".format(int(curverad))
    str_offset = "Vehicle is {0:.2f}m ".format(abs(offset)) + "{} of center".format(direction)
    cv2.putText(image, str_cur, (50,60), cv2.FONT_HERSHEY_SIMPLEX,2,color,2)
    cv2.putText(image, str_offset, (50,120), cv2.FONT_HERSHEY_SIMPLEX,2,color,2)
```

In the pipeline.py, you could go to line299, choice test on images to see how the pipeline work.

```
if __name__ == '__main__':
    """
    #image_test_tracker(), test pipeline on one image and show the image on screen
    images_test_tracker(), test pipeline on images and write the result to related folder
    """
    images_test_tracker("test_images/", "output_images/Lane_fit/", "project", debug_window=False)
```

One example as bellow showed:

Pipeline (video)

To apply the pipeline on the video, you could run the video_rec.py to generate video. the option is explained in the document description. the code is in line 44-61

```
if __name__ == "__main__":
    """
    choise one line to uncoment and run the file, get the video.
    the video will be output to ./outpu_videos/temp/
    option: subclip = True, just use (0-5) second video, False, use total long video.
    option: debug_window = True, project the debug window on the up-right corner of the screen to visualize the image handle process
                                                and write the fit lane failure/search lane failure image to ./output_videos/temp/images
    """


    #get_video_tracker("project_video.mp4", subclip=False, debug_window=True)
    #get_video_tracker("project_video.mp4", subclip=False, debug_window=False)

    #get_video_tracker("challenge_video.mp4", subclip=False, debug_window=True)
    #get_video_tracker("challenge_video.mp4", subclip=False, debug_window=False)

    #get_video_tracker("harder_challenge_video.mp4", subclip=False, debug_window=True)
    get_video_tracker("harder_challenge_video.mp4", subclip=False, debug_window=False)
```

1. Pipeline issue

I've been trying to find a general approach that works for all situations.

With the image process established on single image. For the project video , lane line detection is ok, but for situations with light and shadow, such as challenge video, although some lane lines detection is implemented, it is still not good, and for harder challenge video has no effect.

3 Video processing results

You can find the video processing results in the \output_video\temp\

Discussion

1. the time/efficiency issue

find_lane_pixels() (helpers/lane_detection.py) is used to search the whole warped image to find the lane pionts.

The pipeline handle the image off-line, so not consider the efficiency issue. In real application, the pipeline must handle the image before the next image arrive. a quick search method should be applied.

2. lane_sanity check

The lane_sanity_check() (helpers/lane_detection.py) function is very simple. To check if the fitted polynomial lines, just compare the fitted lines three y postions x distence to check if the fitted lines is OK. this is not work very well when the lane curve changed dramticlly just like the case in the harder_challenge video.

3. Improved algorithm

My algorithm is not very good for the roads with lighting and shading or more curves. My idea is to adjust the threshold and lane pixel detection methods,.

But I lack some confidence in using cameras to identify lane lines. The road environment is too complicated.

This project is frustrating for me, but it does recognize lane lines. It's interesting.

Folders and Files

- **camera_cal** the code which calibration the camera
- **helper** all the functions which used in **pipeline.py** and **video_rec.py**
- **output_images** the images which processed by different functions.
- **output_video** the video has finished
- **test_images** the images used to test the pipeline
- **test_video** three video for testing the lane detection
- **pipeline.py** the code which use to hande the images. the actual lane dection happend.
- **video_rec.py** the code with use "pipeline" to handle the vidoe and generate the output video.