Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

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.

Rubric Points

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

Writeup / README

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're reading it!

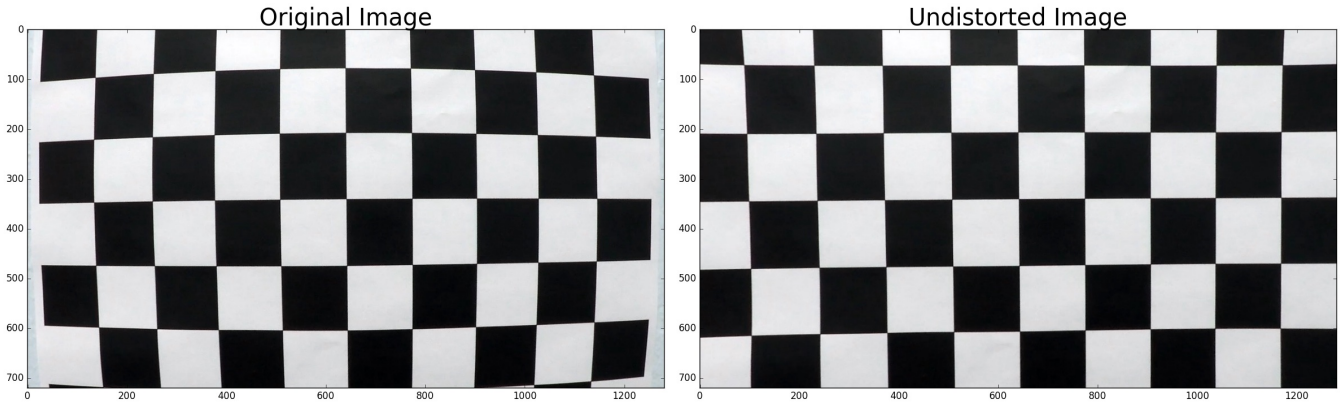
Camera Calibration

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

I defined `cameraCalibration(chass_img_list, numH, numV)` function to solve this. The code for this step located in lines 36 through 71 of the file called `testrun.py`. This function takes 3 parameters as inputs, the chassboard image list(`chass_img_list`), the number of corners along the horizontal orientation(`numH`) and the number of corners along the vertical direction(`numV`).

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:

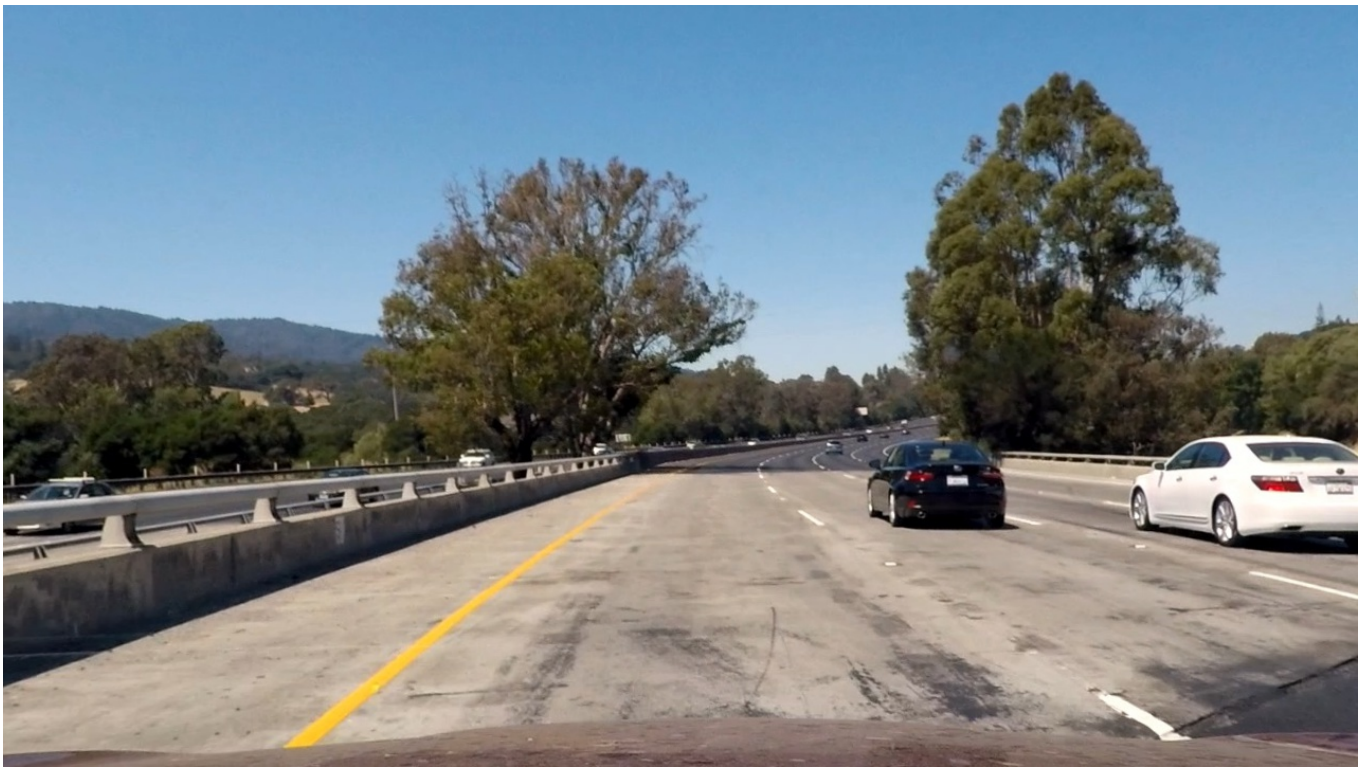


At the end of the function, I chose to store the camera calibration parameters into a pickle file for later use. Line 66 to 71 in `testrun.py` covered this.

Pipeline (single images)

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

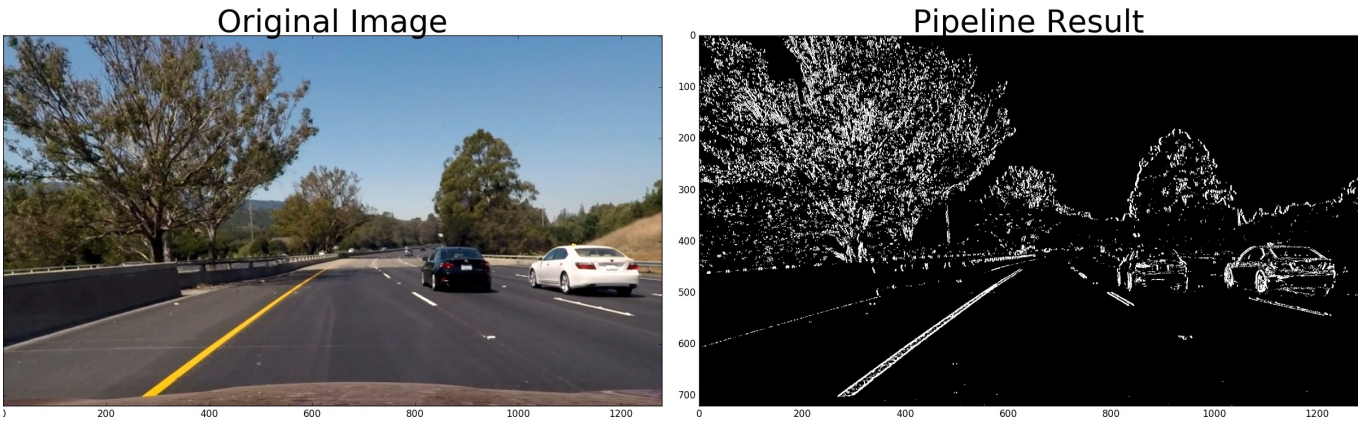
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this:



In the camera calibration section, I have derived the data set of the camera's characteristic and stored them in a pickle file called 'camera_cal.p'. It contains 'mtx' and 'dist' parameters which will be used in distortion correction process.The distortion correction was implemented by a function called `undistortion()`, shown in line 84 to 90 in `testrun.py`.

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.

I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at lines # through # in `another_file.py`). Here's an example of my output for this step. (note: this is not actually from one of the test images)



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

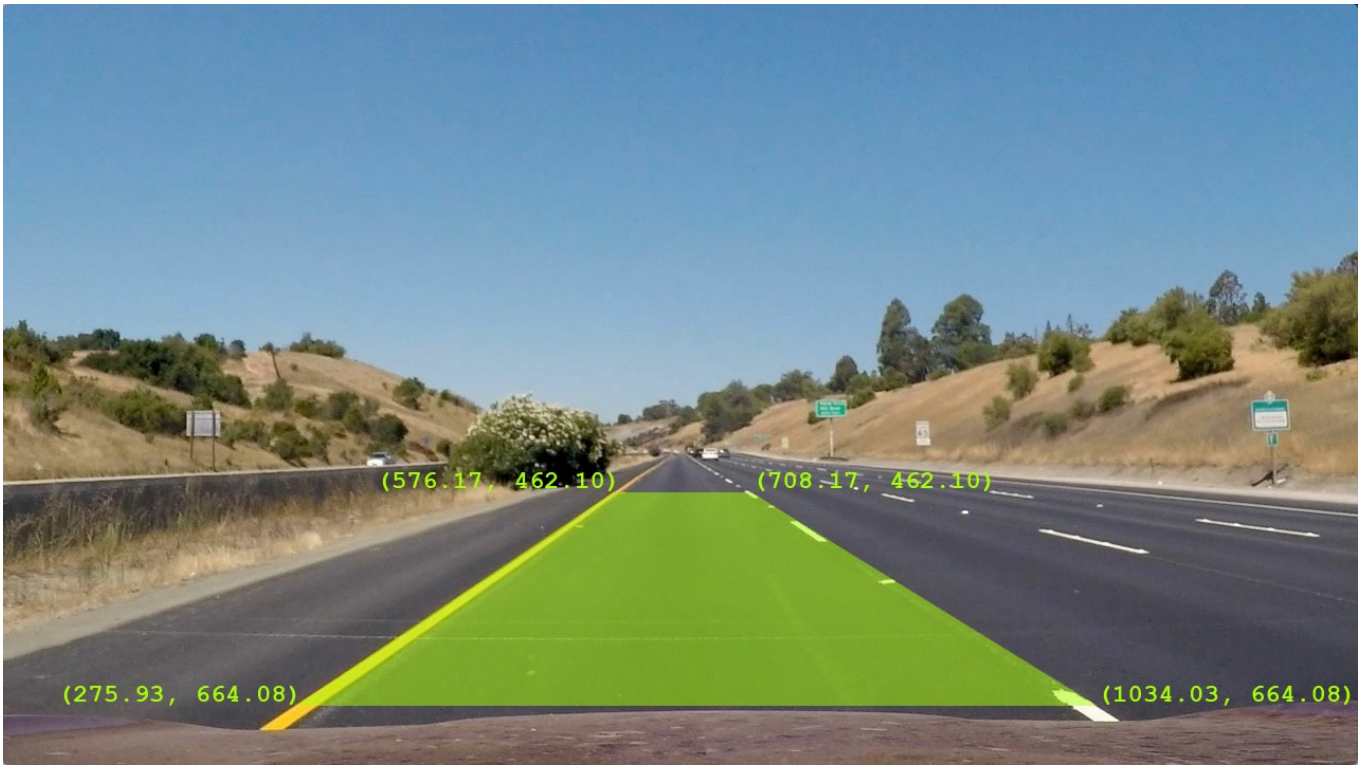
The code for my perspective transform includes a function called `perspectiveTrans(img)`, which appears in lines 191 through 213 in the file `testrun.py`. The `perspectiveTrans(img)` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. In order to precisely locate the corners of the original trapzodial zone, I chose to pick up the FOUR cordinates in the following manner:

```
    src = np.float32([[275.93, 664.08],
                      [1034.03, 664.08],
                      [708.17,  462.10],
                      [576.17, 462.10]])

```

```
    dst = np.float32([[275.93, 664.08],
                      [1034.03, 664.08],
                      [1034.03, 64.08],
                      [275.93, 64.08]])
```

This resulted zone was depiced in the following figure:

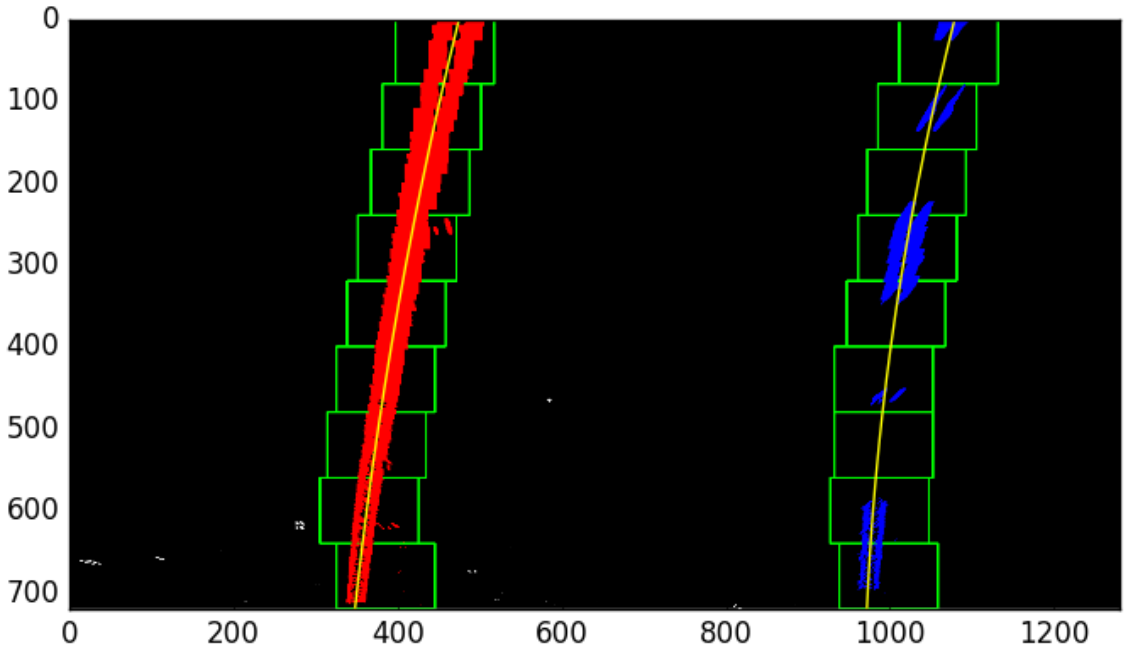


I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

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

I prepare two functions dealing with the lane line finding. First is `slidingWindows()`, the other one called `lineFinding()`. The first function just need to run once when the first frame of image is given, then the lane finding routin can be much easier by the second function. These two functions are controled by a variable 'blind_search_flag'. These two functions are shown between line 235 to 381 in `testrun.py`.

Then I fit my lane lines with a 2nd order polynomial kinda like this:



After blindly searching for the first frame using the 'sliding window' method, the following frames can be processed using `lineFinding()` method since it's reasonable to assume the lane lines' position are fixed among adjacent frames.

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.

For the curvature calculation, I designed two functions to calculate curvatures of the lane, `curvature()` for the unit of pixel and `curvature_meter()` for the unit of meter. The fundemental structure of the two functions are similar. The only difference between them is the `curvature_meter()` function utilized two coefficients to coverte pixel to meter. Lines 386 through 400 in my code in `testrun.py`.

For the deviation calculation, I defined `offset()` function. This function simply compared the difference between the image center point and lane center point. Then, the deviation can be derived based on this difference. Lines 402 through 406 in

`testrun.py`.

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

I implemented this step in lines # through # in my code in `yet_another_file.py` in the function `map_lane()`. Here is an example of my result on a test image:

Pipeline (video)

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!).

Here's a link to my video result.

Sorry, I can't put a hyperlink in this file, so just find the file named 'out_video.mp4' to play.

Discussion

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?

The foremost reason for I did not finish a excellent job in this project is kind of personal, but it indeed took away of most of mine time. The reason is,

I just become a father!

My little angel came to the world on Mar.29, 2017. And he soon occupied almost all of my daytime to look after him. So I have few time to improve my code. I feel exhausted these days since I only sleep less than 4 hours these days,and I want to say, I have tried my best to do this project.

- **Problem 1**: I feel confused of the video processing method using moviepy/imageio library, cause the only functions that I know so far are `VideoFileClip()` and `clip1.fl_image()`, which was used in the project 1. This function doesn't meet the reqirments of this implementation, since it can not produce consecutive images for each frame in the video file. Therefore, it's inconvenient to implement filters or adjustments among frames.
    - **Solution**: Alternatively, I found the 'ffmpeg' is very powerful for image processing routins. So, instead of python, I learned to use 'ffmpeg' dealing with the video file. Firstly, I converted the video into consecutive image series using built-in functions of ffmpeg, the code is like:

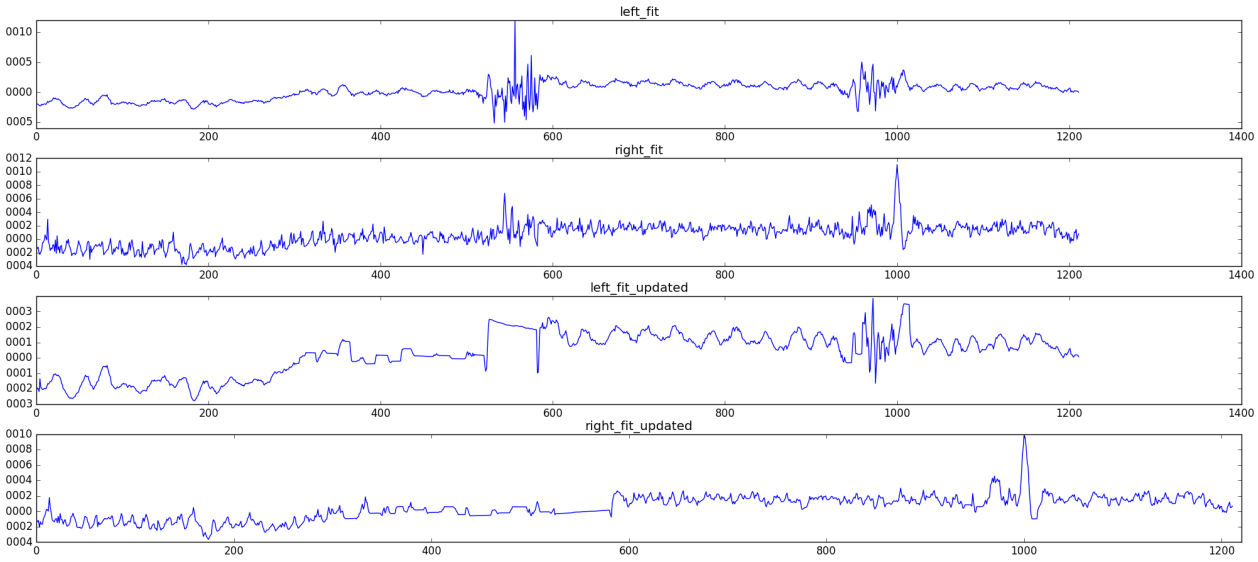
      ```
      ffmpeg.win32.exe -i project_video.mp4 -r 24 -s 1280x720 -f image2 in-%04d.jpg
      ```

      Then, I applied the lane finding pipeline on each image. After processing, output the result images sequence superposed with lane area, curvature and deviation information.
      Finaly, use the following code generating a video file based on the processed image sequence. The code looks like:

      ```
      ffmpeg.win32.exe -f image2 -i image2 out-%04d.jpg -r 24 out_video.mp4
      ```
- **Problem 2**: The vehicle always bouncing up and down during the driving, so the image recorded also unstable from one frame to another. Moreover, the errors introduced by the perspective transform section, thresholding images and polynomial fitting for lane lines are unavoidable. Besides, the lane finding algorithm

will completely failed in predicting lane areas in a few extreme conditions. Therefore, I have to design a more stable or reliable method instead of directly using each prediction result.

- **Solution**: After fine tuning perspective tranform parameters and thresholding parameters, I've designed a function called `updateFitParameters()` to optimize the curve fitting results. It serve as a buffer or a filter, which contains 3 of the most recent fit parameters. In this function, a criteria was also proposed in this function to judge whether a upcoming fit parameter is reasonable by comparing lane lines of both sides. Since the two lanes should be parallel along the road, so the quadratic term of a good fit parameter of the two lines should have the same sign, both positive or negative in other words. So if the upcoming new fit parameter meets this rule, I consider it's a confident prediction, assign a large positive weight to it:
  `current_out = 0.6*primary_in + 0.3*previous_out + 0.1*p_previous_out`.
  The final output is a weighted sum value combined with the upcoming value, previous value and second previous value.
  If a coming value fails to meet the rule, I consider it's a bad prediction and should be ignored. So the update equation becomes:
  `current_out = 0.9*previous_out + 0.095*p_previous_out + 0.005*primary_in`,
  indicating that the final output value is nearly equal to the previous value.
  This function is shown from line 503 to 531.
  The following figure shows the original fit parameters and filtered fit parameters for both side lanes.



- **Problem 3**: The calculated curvature jittered intensively due to the same reason mentioned in the above problem, while the real curvature varies slowly.
    - **Solution**: I implemented a mean value digital filter to smooth the curvature calculation result. The code implementation can be found in line 714 to 743 in `testrun.py`. I also introduced a tricky method to improve the stability of the curvature predictions. Code in line 690 to 743.

After doing that, the algorithm works well on the most part of the video. But the lane line prediction is still no robust enough. When the lane line was not clear due to the road condition, the identified lane area was unstable, no matter how to choose the parameters in the pipeline.One possible solution of this problem, in my opinion, should introduce machine learning approaches to this task, we can train a deep network identify the corresponding patterns of the lane lines in various road conditions.