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

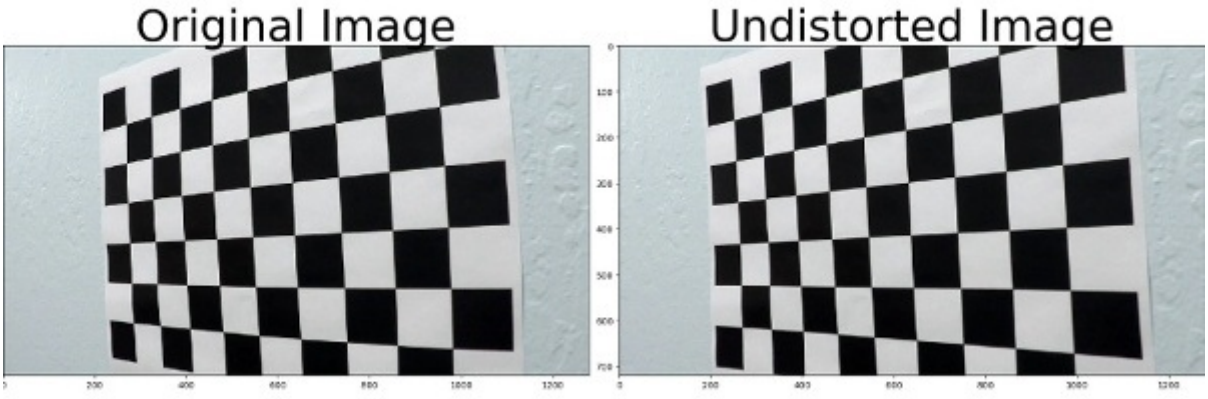
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the camera_calibration.py.

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:



Finally, I saved the distortion data in a pickle-file calib_data.p.

Pipeline (single images)

I've created following scripts to meet the requirements:

- main.py, script to run for generating outputs.
- function_lib.py all utility-functions for processing the images.
- Line.py TrackLine-class to describe lanes

For this task, the pipeline is implemented in function `process_image(img, track_line)`

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 one:

For the undistortion, I used the calibration data from calib_data.p and applied `cv2.undistort()` in the function `undistort_image(img, calib_data_file)` (function_lib.py, line 8, called in line 149)

2. 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 `warp_image(img)`, which appears in lines 115 through 134 in the file function_lib.py. The `warp_image(img)` 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:

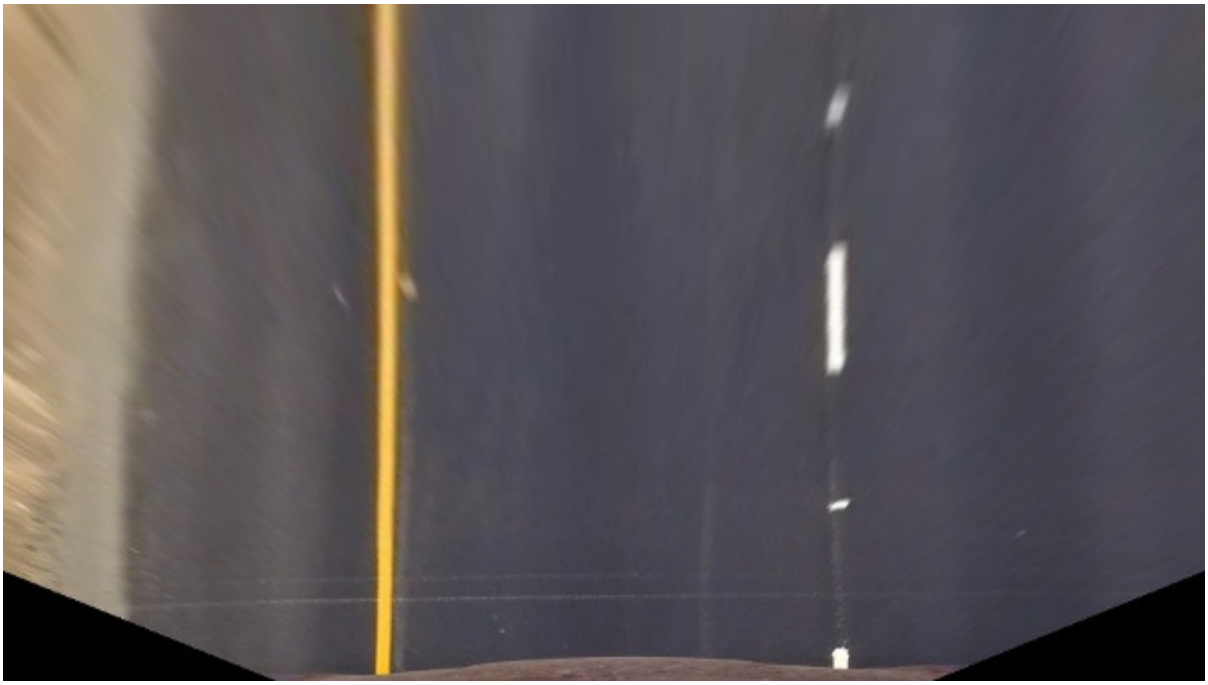
```
src = np.float32(
        [[img_size[0]*0.425, img_size[1] * 0.65],
         [img_size[0]*0.04, img_size[1]],
         [img_size[0]*0.96, img_size[1]],
         [img_size[0]*0.575, img_size[1] * 0.65]])
dst = np.float32(
        [[(img_size[0] / 4), 0],
         [(img_size[0] / 4), img_size[1]],
         [(img_size[0] * 3 / 4), img_size[1]],
         [(img_size[0] * 3 / 4), 0]])
```

This resulted in the following source and destination points:

Source	Destination
544, 468	320, 0
51.2, 720	320, 720
1228.8, 720	960, 720

Source	Destination
736, 468	960, 0

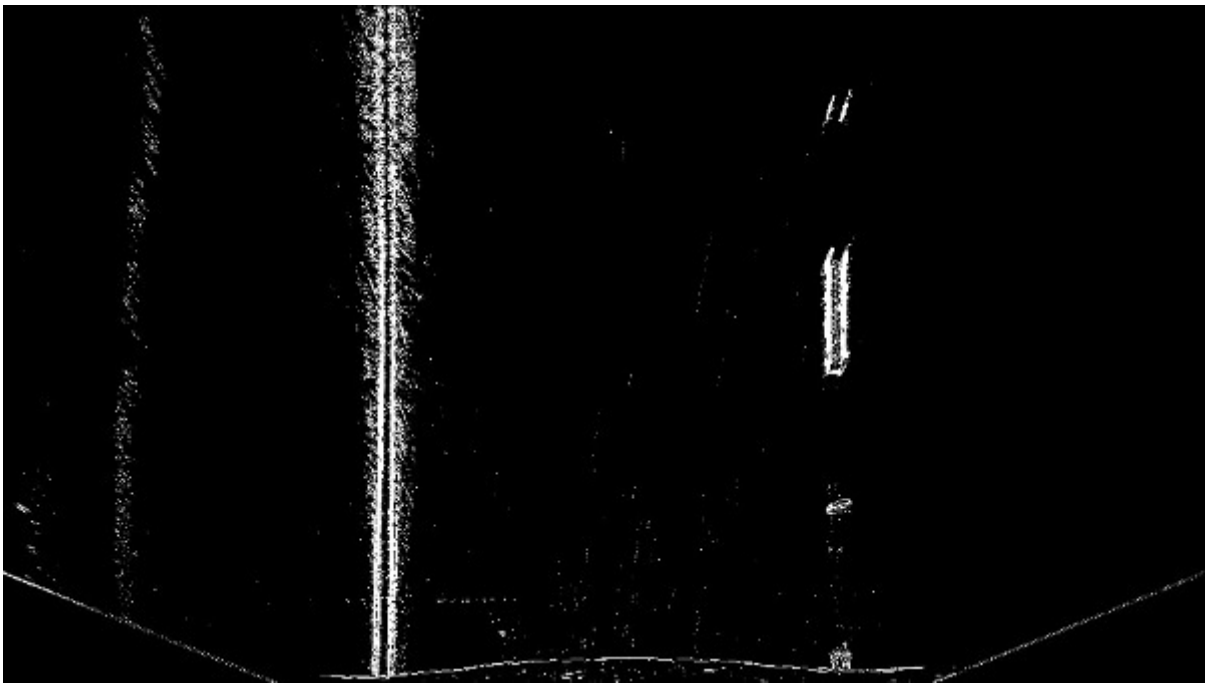
I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.



3. 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. First, I convert the warped image into HLS-space (line 155-166 in function_lib.py). L- and S-channel are given to the sobel-filter and I've combined binaries of x, y, magnitude gradient (function `combine_sobel(channel, abs_thres, dir_thres, kernel_size=3)`, line 102-112 of function_lib.py) and filtered directions (line 107) to generate binary output.

In the end, I've got following result:



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

Then I used the binary output to find the lanes and fit my lane lines with a 2nd order polynomial (line 166, function_lib.py). The required steps are implemented in the `find_lane(sobel_binary)`-method of the `TrackLine`-class.

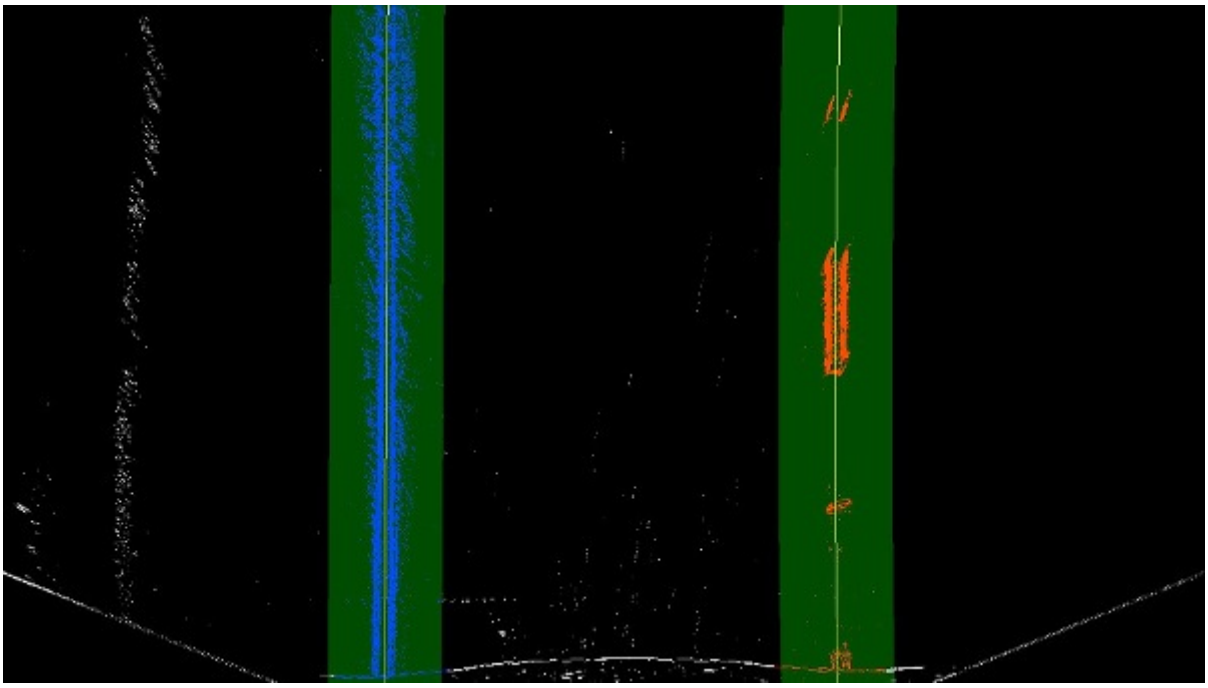
The method `find_lane()` calls `search_new_lanes()` if no lanes are found in the previous step and calls `update_lanes()` if lanes were already found and polynomial-coefficient of last step could be reused for current step.

The method `search_new_lanes()` first makes a histogram of the lower half of the binary image and based on the white pixels found (from 1/4 to the middle for left line and from middle to 3/4 for right line), it determines where the starting point should be for further search. I've used the window search method and adjust the next window position by average found pixel positions.

The method `update_lanes()` uses the polynomial coefficient of the previous step and defines the search window for current step.

After the pixels of lanes are found, the polynomial coefficients are calculated by using `numpy.polyfit()`

The final result of these steps is shown in the following image, where blue and red pixels are used to identify left and red lane lines and the yellow thin line is the fitted polynom:



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.

I did this in lines `293` through `316` in my code in `Line.py`.

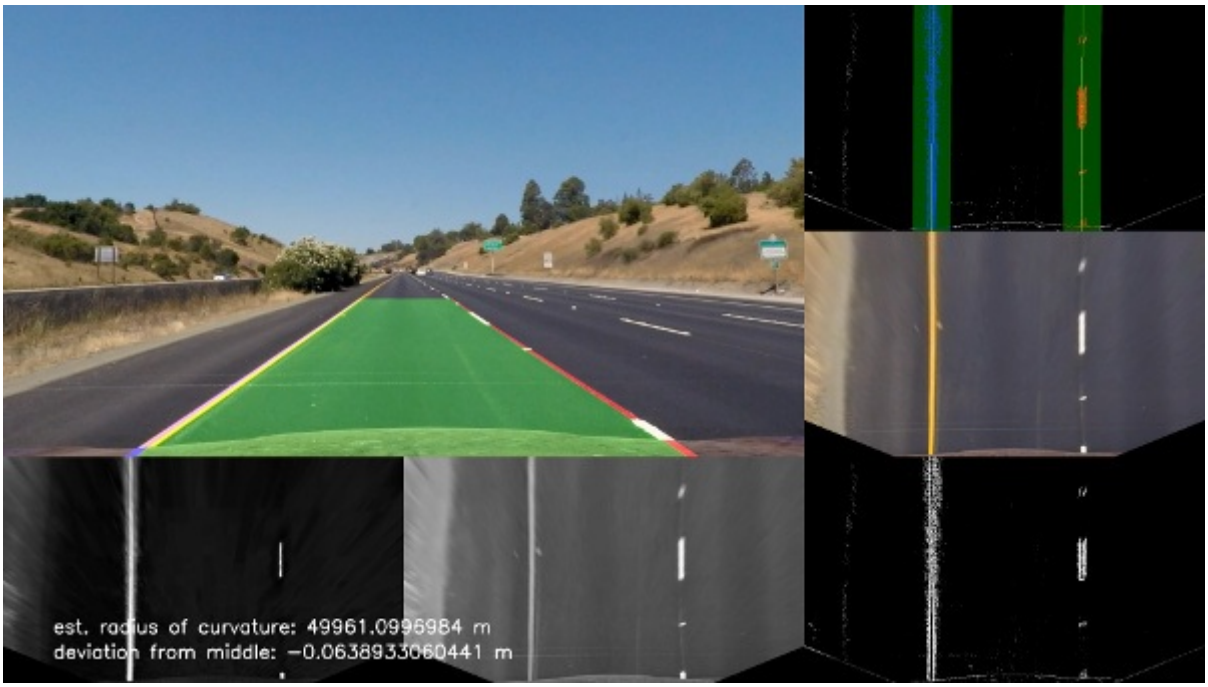
First, I defined coefficient to convert pixel-length to real world length. The coefficient I used here are estimated. After that, I calculated the real world length polynomial coefficients.

The width of the road at the bottom of the image can be estimated as difference between last polynomial coefficient of left and right polynom since that are the x-positions of the lines. The deviation from the center is calculated as difference between average of these x-positions and the middle of the image since we assume the camera is mounted at the center.

The radius of road curvature is measured at the top of the image and calculated with the curvature formula.

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 `246` through `291` in my code in Line.py in the function `draw_on_original()`. 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

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 Approach I've took did not contains following feature, which I could add in the future:

- use only color-filter to extract only yellow and white pixels, this could have better result.
- use information of previous steps
- use failure detection to identify whether the lane polynom found is valid

In order to prevent false search, I choose to search new lane every 5 steps. A better way could be that the new lane found can only change marginal compared to previous step.