Writeup Template

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!

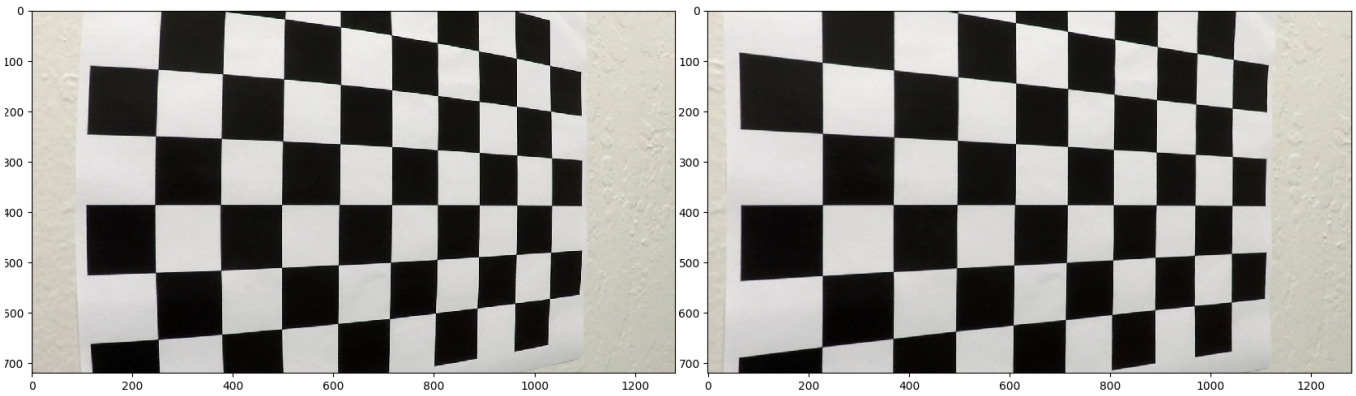
Camera Calibration

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 file `./calibration.py` in function `computeCalibMatrix()` (lines 7 through 53).

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. When applying this distortion correction to the `calibration5.jpg` image using the `cv2.undistort()` function I obtain this result:



Pipeline (single images)

General strucutre of the project. The main project file is `./advlane.py` which imports `./calibration.py` (calibration functions) and `./thresholds.py` (thresholding functions).

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

Using the camera calibration matrix and the distortion parameters computed by `computeCalibMatrix()`, I can undistort an image using the `cv2.undistort()` function (wrapped inside the `undistort()` function in `./calibration.py`, line 56)

Example using test image:

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.

All thresholding functions are located in `./thresholds.py`. For my project, I used a combination of the following thresholds:

- x-axis gradients of the red channel between 20 and 100 (on a scale from 0 to 255)
- saturation values between 80 and 255 (on a scale from 0 to 255)
- red values between X and 255 (on a scale from 0 to 255), where X is computed dynamically to be half the distance between the mean and the max red value of the frame
- gradient direction values between 0.8 and 1.2

The thresholds are applied to the image in `./advlane.py` (lines 216-232). Here's an example of my output for this step:

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 situated in the function `computePerspectiveTransform()` from the file `calibration.py` (lines 58-63). It takes as input an image (`img`) and computes a transformation matrix for the perspective transform using the following hardcoded source and destination points:

```
rect_src = np.float32(
        [[560, 475], #top-left
        [725, 475], #top-right
        [1100, 719], #bottom-right
        [216, 719]] #bottom-left
    )

    rect_dst = np.float32(
        [[0.2 * shape[0], 0.15 * shape[1]], #top-left
        [0.8 * shape[0], 0.15 * shape[1]], #top-right
        [0.8 * shape[0], 0.9 * shape[1]], #bottom-right
```

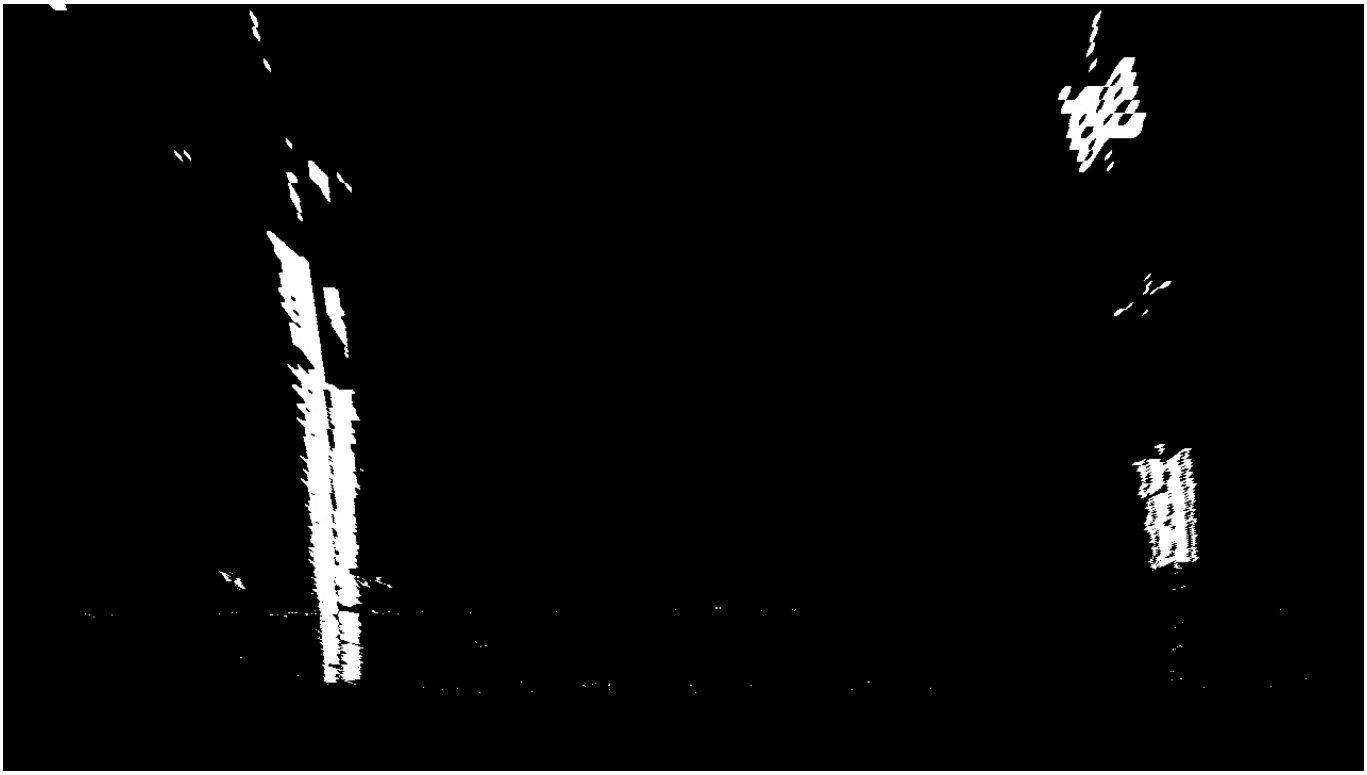
```
            [0.2 * shape[0], 0.9 * shape[1]]] #bottom-left
    )
```

This resulted in the following source and destination points:

Source	Destination
560, 475	256, 108
725, 475	1024, 108
1100, 719	1024, 648
216, 719	256, 648

The function returns two transformation matrices, a forward one (source->destination) and an inverse one (destination->source).

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image 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?

To identify lane-line pixels I generally followed the sliding window approach introduced in Lessen 7. However, I did do some alterations to make the algorithm more robust.

The algorithm is implemented in the function `find_lane_pixels()` in `./advlane.py` (lines 100-157). The function requires the image, a starting X position and another image where to draw the visualization as input parameters. Unlike the approach in lesson 7, my function only searches for pixels for a single line. Thus, the starting position for the sliding window has to be passed via parameters. To compute it, I use a histogram over

the bottom half of the image to find the X-value with the most active pixels (lines 245, 246, 257 and 269 of the function `process_image()`). Thus, the function is called twice, once for the left line, once for the right.

The first step of the algorithm is to compute the indices of the pixels which have not been identified as potential line pixels by the binary mask using numpy's nonzero function (lines 105-107).

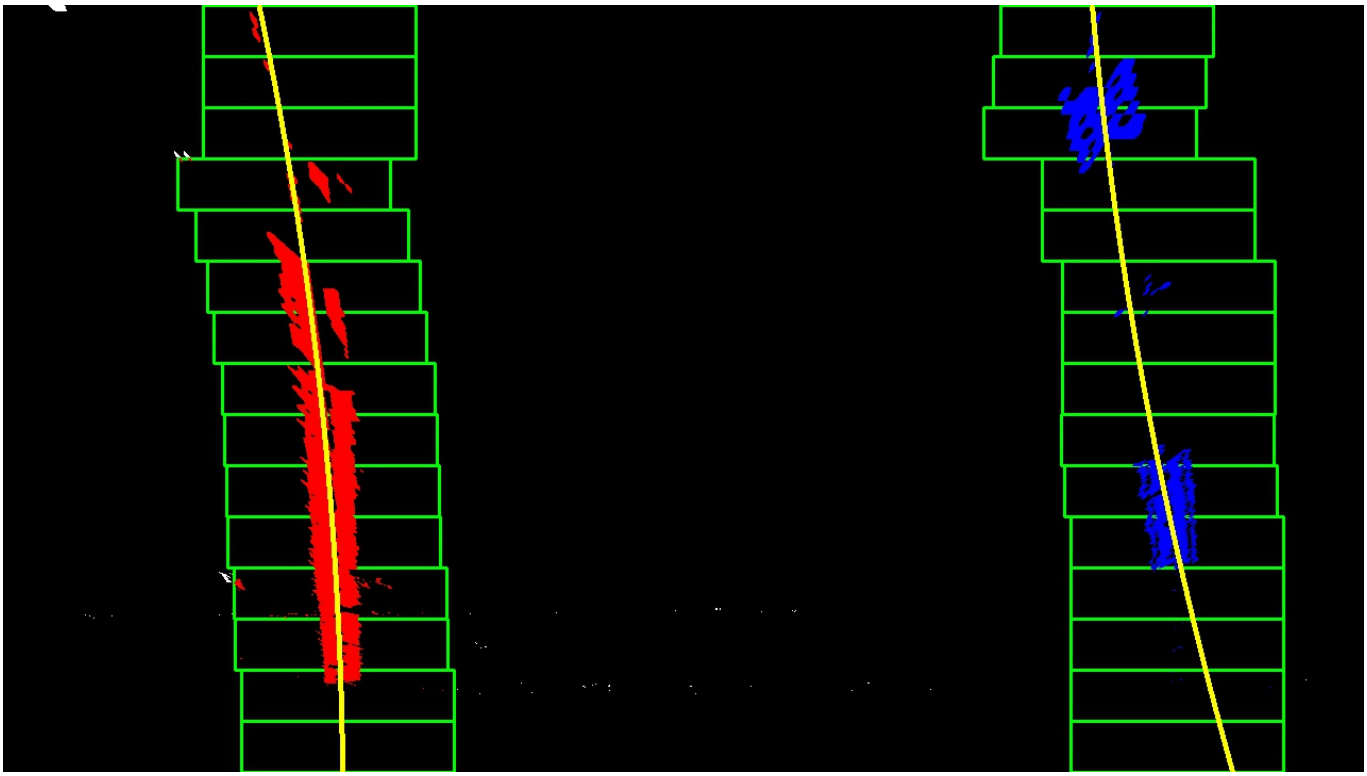
After initializing the sliding window parameters, we enter the main loop. The hyperparameters have been set as follows:

```
nwindows_ = 15
window_width_ = 200
minpix_ = 80
```

The first step in the main loop is to count the number of pixels in the current window. If this number exceeds the `minpix_` threshold, the algorithm accepts the new window and the found pixels are appended to the total set of pixels.

After we found all the pixels for the left and right lines, a 2nd degree polynomial is fitted through each set of points using numpy's `polyfit()` function. If a valid polynomial already exists (from the previous frame of the video), we call the `update_lane_pixels()` function (lines 162-179) instead of `find_lane_pixels()`. This function simply follows the fitted polynomial and searches left and right of it (within a window) for new points.

The following image shows the result of the `find_lane_pixels()` function.



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 computation of the radius of the curvature is implemented in lines 181 through 195 in my code in `./advlane.py` in the function `rcurveabs()`. The function is implemented using the equation from lesson 7.

The function accepts as parameters a 2nd degree polynomial and a y value.

The function `measure_curvature()` calls `rcurveabs()` for both left and right line polynomials. Additionally, a scale parameter allows one to specify a specific scale of the y-parameter, e.g. for converting pixels to meter.

As a variation, the function `rcurve()` ditches the `abs()` to provide a signed radius of the curvature, thus allowing one to differentiate between left and right curves.

The vehicle position is computed in `process_image()` (lines 442-444) by simply calculating the difference between the lane and image middle points. For the lane middle point I compute the mean of the left and right lines at the bottom of the image (y = 720).

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 line 435 in my code in `./advlane.py` in the function `process_image()`. Here, I call the `warp()` function of the `calibration.py` file using the warped image and the inverse transformation matrix computed before.

Here is an example of the result of the entire pipeline on the first frame of the project video:

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?

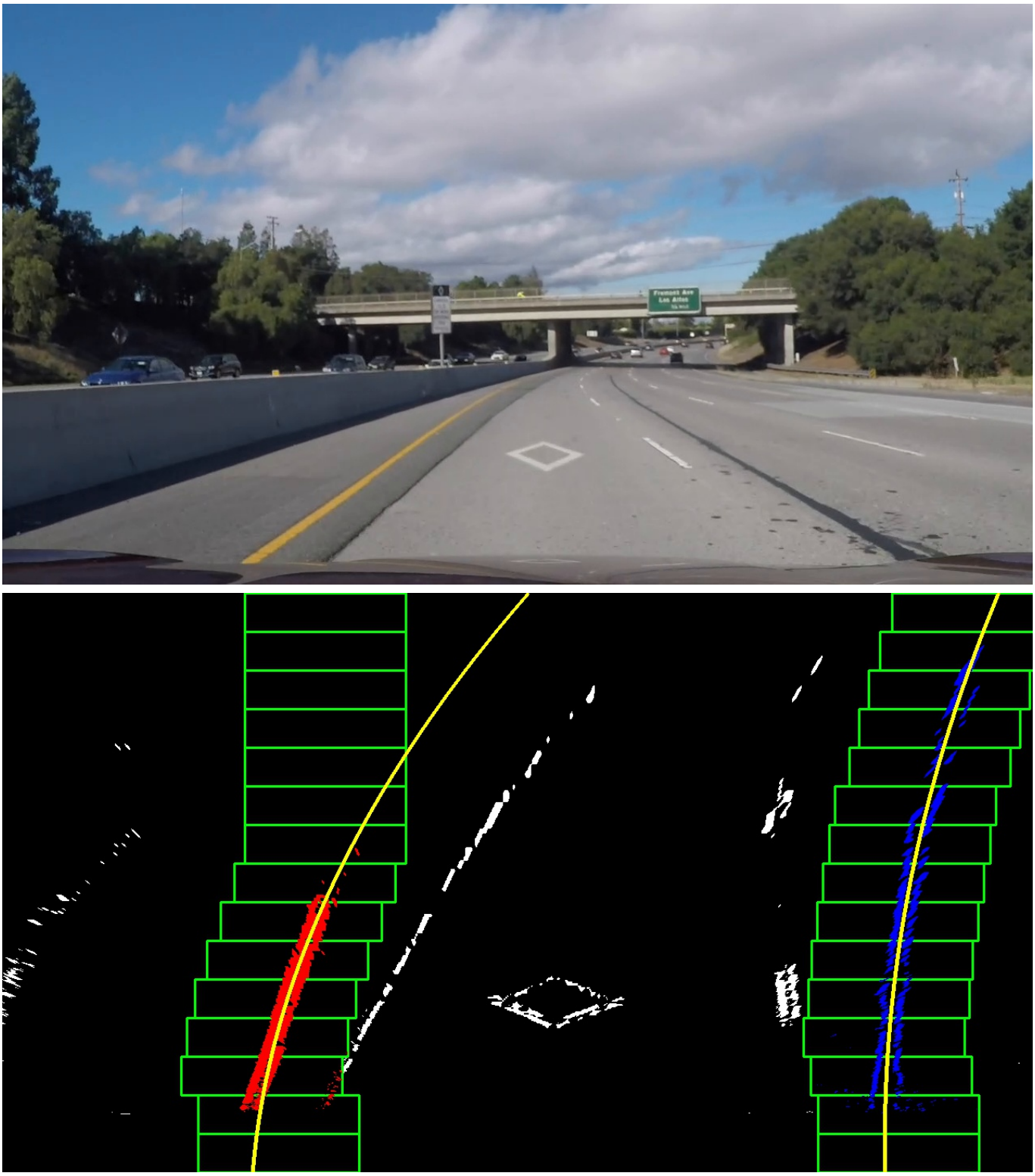
The approach performed reasonably well on the `project_video.mp4`, yet had some real difficulties on thr two challenge videos. To tackle these I implemented multiple checks.

Firstly, the algorithm verifies the validity of a polynomial by checking for plausible curvatures (>500 px) for each line. The inverse curvatures of the two lines (lines 297-301) are also compared between each other. If the difference is too great, both polynomials are ignored.

Secondly, the distance between to two lines is also used as a sanity check. If the mean distance between the points on the lines is below 3m or above 4.5m, the fitted lines are ignored.

Despite these checks, the results on the first challenge video are modest at best. One issue is the shadow area under the bridge which effectively renders the binary masking useless.

Another problem is the cracks in the tarmac which are very hard to distinguish for normal lines. Despite trying multiple combinations of gradients and color spaces, I have not found any truly satisfactory solution to the problems.



A check on the direction in which the sliding window moves within the `find_lane_pixels()` function could also mitigate the chance we jump onto non-line pixels (e.g. barrier, cracks, shadows). This could be done by fitting a polynomial after every window and checking if the next window falls roughly in the path of this polynomial.

The implemented algorithm also falls short when applied on the second challenge video. The reason for this is mainly the complex (and three-dimensional) road geometry which cannot be approximated using a 2nd degree polynomial. Solving this would probably require a more complex road model which can handle extreme curvature on all three axis.