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.

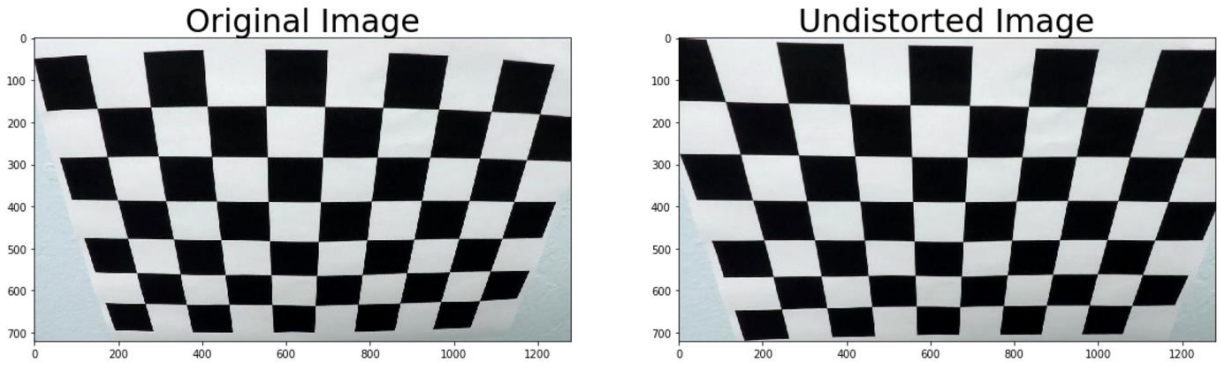
You're reading it!

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 first few code cells of the IPython notebook located in "./Notebook.ipynb".

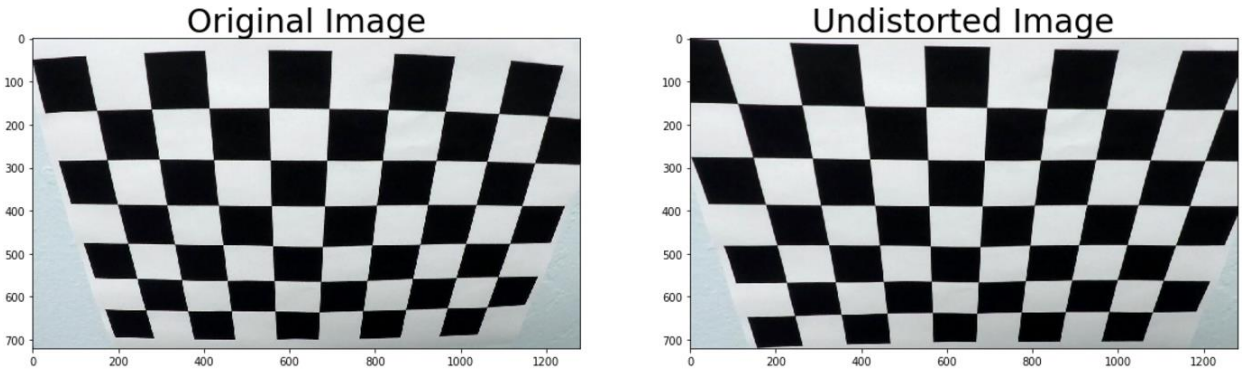
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of 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:



Pipeline (single images)

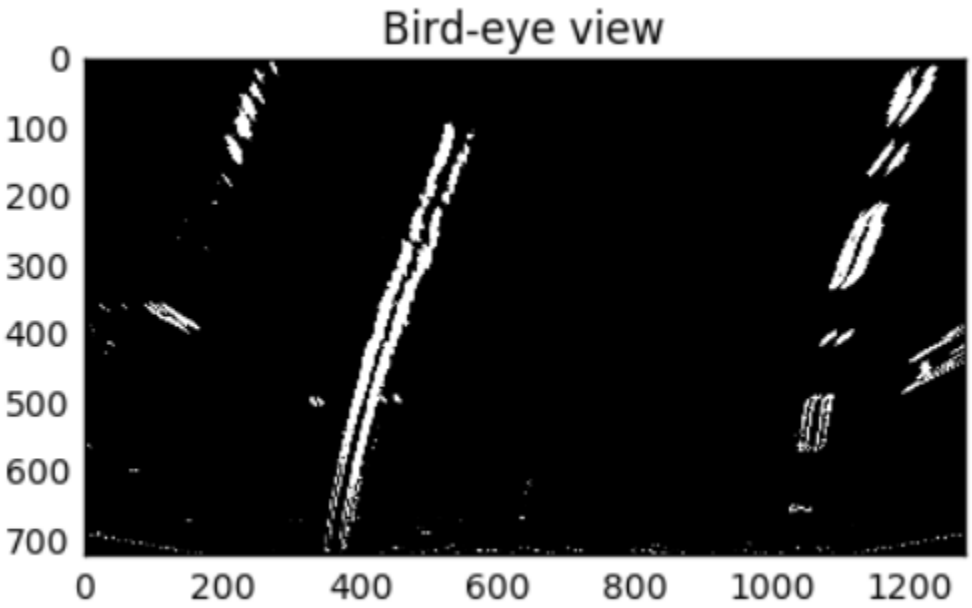
1. Provide an example of a distortion-corrected image.

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



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



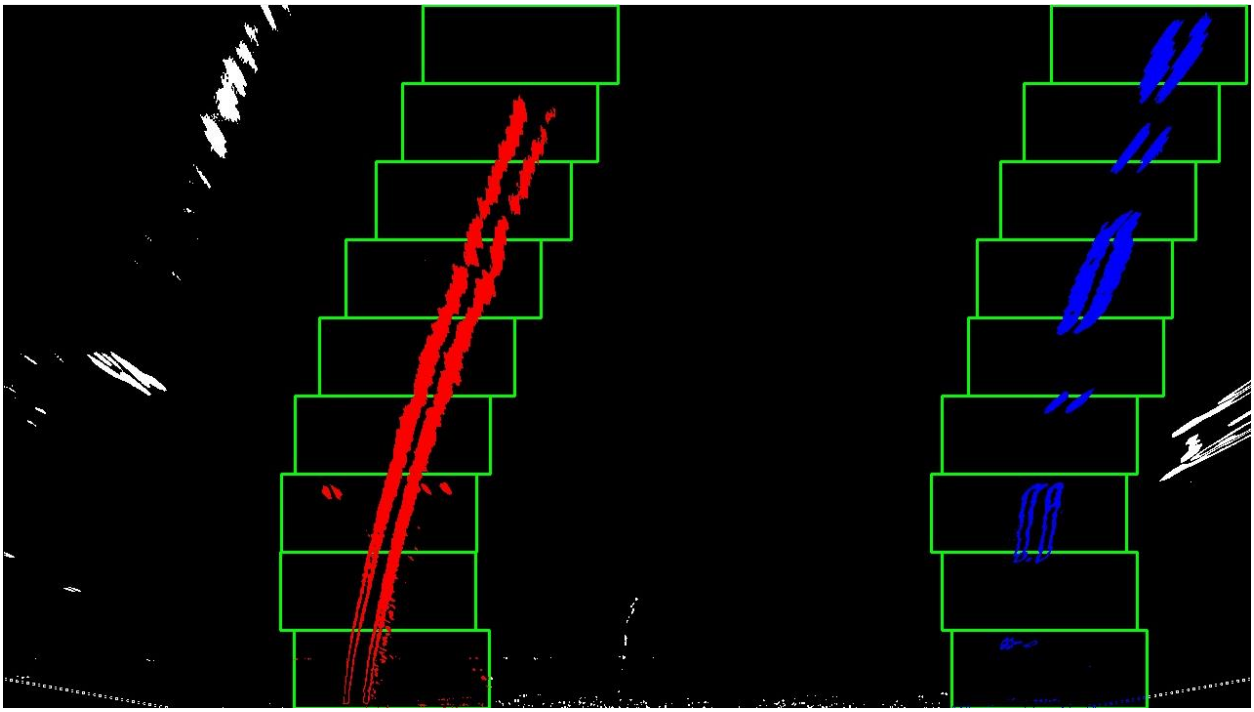
3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

I implemented this step in lines # through # in my code in `Finding_Lanes.py` and transformation.py. The code for my perspective transform includes a function called `warp_prespective()`, which appears in "transformation.py". The `warp_prespective()` function takes as inputs an image (`image`), 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([
                 [h // 2 - 76, w * .625],
                 [h // 2 + 76, w * .625],
                 [-100, w],
                 [h + 100, w]
                 ])
dst = np.float32([
                 [100, 0],
                 [h - 100, 0],
```

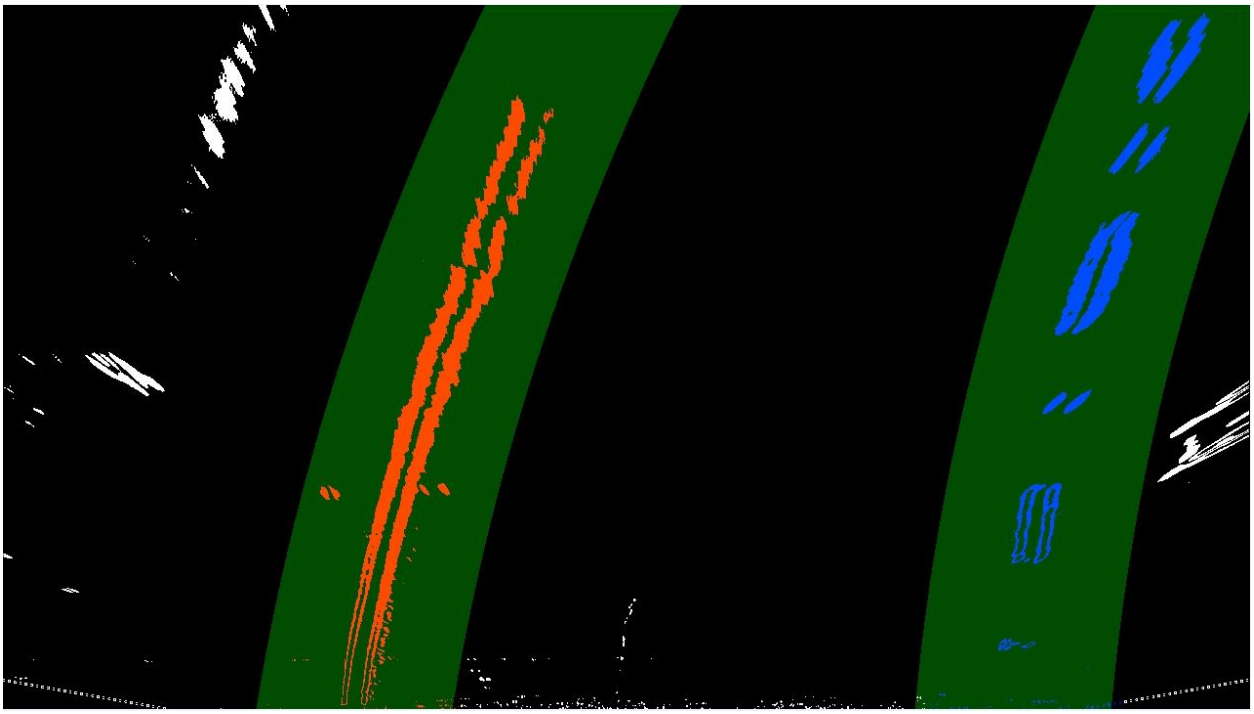
```
                [100, w],
                [h - 100, w]
])
```

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 implemented this step in lines # through # in my code in `Finding_Lanes.py`.We first check the histogram of the lower of image and find the two peaks for the left and right lines. Then we use the sliding window method to work our way upwards and find the relevant points in the image which mark the lane. Next, we use the `np.polyfit()` method to fit a second degree polynomial to these points. I did this in the `fit_polynomials()` 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.

I implemented this step in lines # through # in my code in `Finding_Lanes.py`.Given the detected points, we determine a second degree polynomial that fits these points, and then we calculate the radius of curvature of this polynomial. Also, we convert from the pixel space to real space in meters. I did this in `get_curvature()` function.

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 `Finding_Lanes.py`. Here is an example of my result on a test image:

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?

Selecting optimum parameters for the gradient thresholds is a challenge - because we want to do it in a way that works for a wide number of scenarios. This was achieved by testing it on the 6 test images that are given.

The pipeline currently fails on the test images, mostly because we don't keep state of previous video frames as of now. We can make the processing more efficient by making it remember the last detected lane line and then using that as a hint for detecting the lane in the next video frame.

Also, to make it more robust, we should add checks like checking that the radius of curvature is similar for both the left and right lane lines. Also, it should continue to work if just a single lane line is visible.