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!

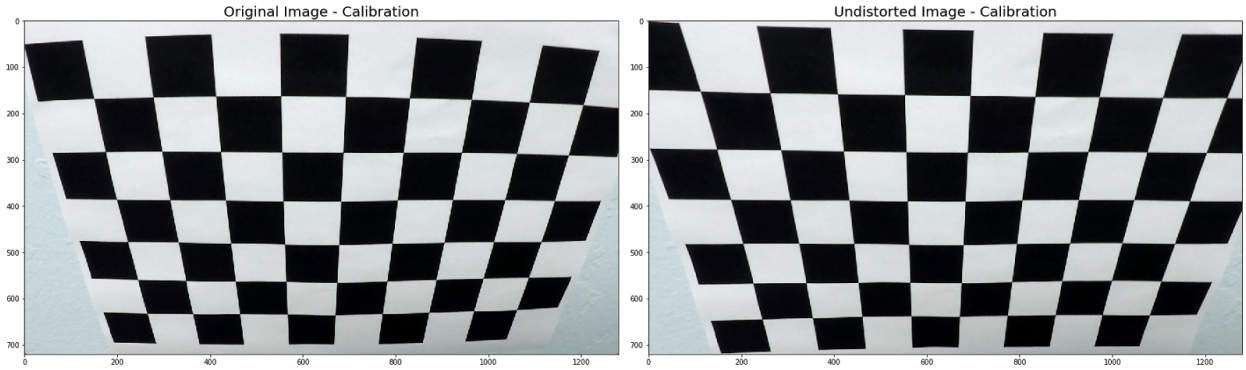
Camera Calibration

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 2-5 code cells of the IPython notebook located in "core.ipynb". All code is in this one IPython notebook

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 used a combination of color and gradient thresholds to generate a binary image (thresholding code cell 6 and process(image function) in the notebook). I tried magnitude and direction thresholding but did not necessarily get better results (at times worse) so I removed it. For color threshold I use both HSL (S component) and HSV (V component) and then combine their result. 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 includes a function called `warper()`, which is in the code cell 7 of the IPython notebook. The `warper()` function takes as inputs an image (`image`) and calculates the src and dest points based on a region of interest defined in the function as offset from image edges. I define the region for the source and destination points in the following manner:

```
src = np.float32(
    [[img_size[0]*(.5-midw/2), img_size[1]*height],
    [[img_size[0]*(.5+midw/2), img_size[1]*height],
    [img_size[0]*(.5-botw/2), img_size[1]*bot_crop],
```

```
    [(img_size[0]*(.5+botw/2), img_size[1]*bot_crop])
dst = np.float32(
    [[offset,0],
    [img_size[0]-offset, 0],
    [offset, img_size[1]],
    [img_size[0]-offset, img_size[1]]])
```

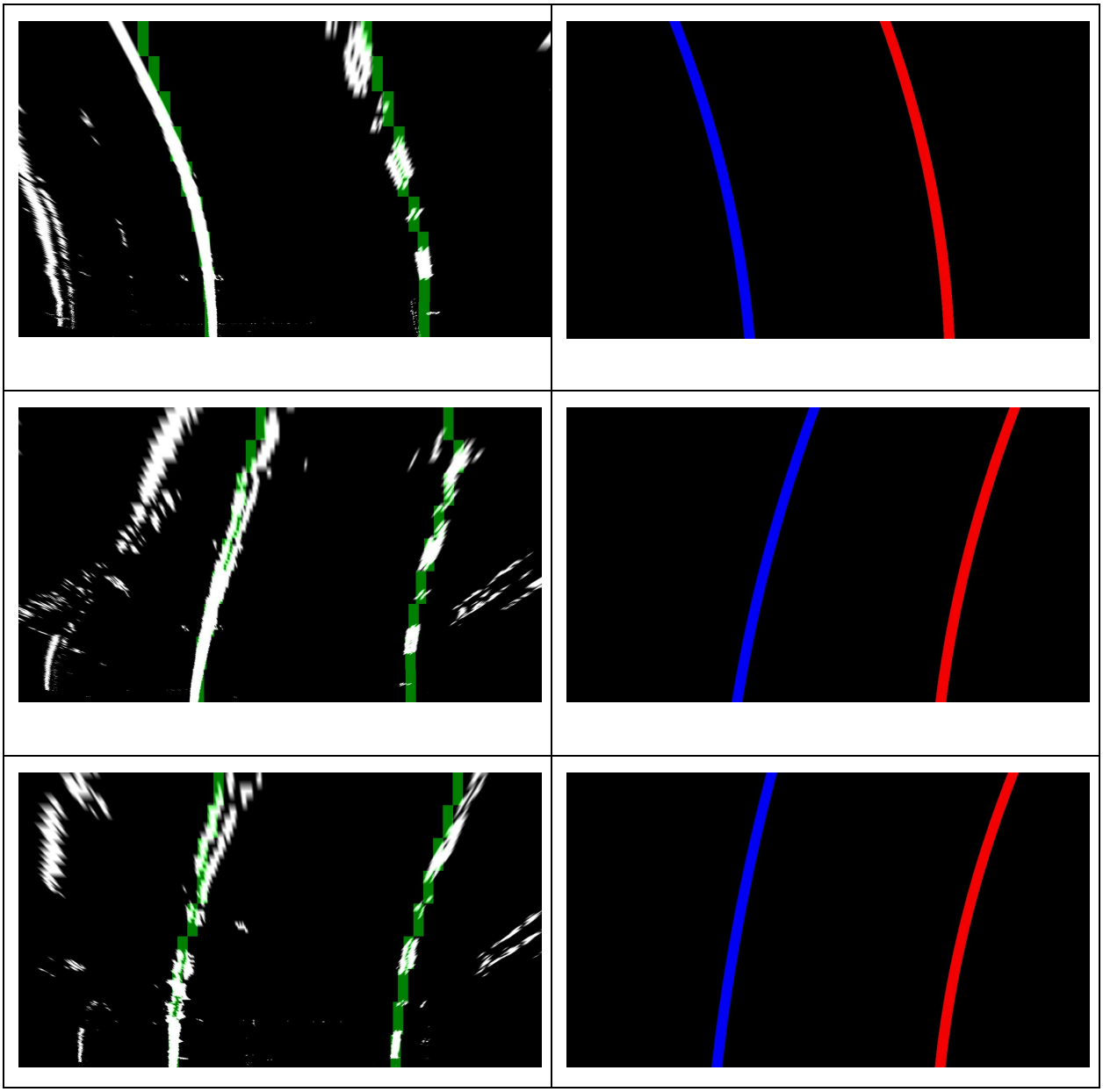
Where,

```
img_size = (image.shape[1], image.shape[0])

botw = 0.76 #width of the trapeziodal region at the bottom

midw = 0.08 #width of the trapeziodal region at the mid

height = 0.62 #height of the trapeziodal region

bot_crop = 0.935 #crop the bottom part of the image where the car hood is

offset = img_size[0]*0.25 #for the rectangular region in the transform
```

Here is an example of the transformed binary:

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

Then I did a convolution to find window centroids (code cell 8 and #4 and #5 in cell 9) where pixels were found. In this approach we slide a window template across the image from left to right and any overlapping values are summed together, creating the convolved signal. The peak of the convolved signal is where there was the highest overlap of pixels and the most likely position for the lane marker. Then I fit my lane lines with a 2nd order polynomial with the result (in code cell 9):



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 code cell 9 and point #6 using the formula

Rcurve=((1+(2Ay+B)2)3/2)/ |2A|

I found the curvature for the left lane and used that. Another approach could have been find the curvatures for for left and right and then average them or to find the curvature of the mid line

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 the code cell 9 spread between points #5 and #6. Here are examples of my result on test images:

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?

This pipeline works well for the video provided in the project but it does not do well for the challenge video. The improvements would likely be in the thresholding part where more techniques and experimentation with thresholding values could result in a cleaner binary image which forms the basis for all computation after that.

I also tried using the sliding window approach and due implementation difficulties I then followed the convolutions approach for finding the left and right lane in the transformed image.