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.

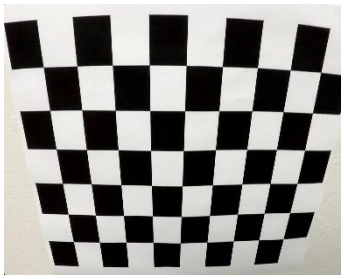
The code for this step is contained in the file cam_cal.py.

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.

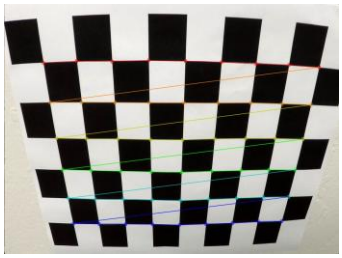
Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. Below is the original image and the image after applying the undistortion function.

Original Image



Undistorted Calibration Image

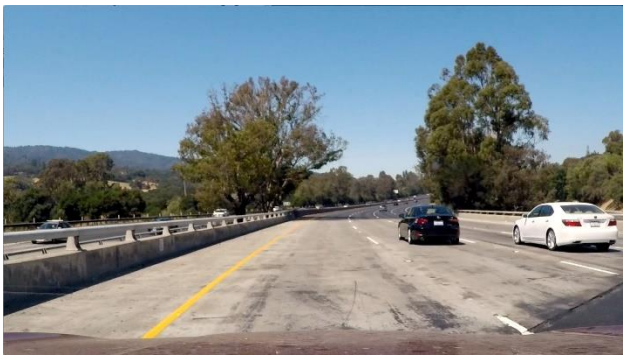


Pipeline (single images)

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

To demonstrate this step, I first saved the 'mtx' and 'dist' values from the previous step in cam_cal.py in a pickle file which I will then load in the image_gen.py file. This is the original image of test1.jpg and the result of the undistortion after applying the `cv2.undistort()` function in code lines '96 to 98' in image_gen.py.

Original Image



Undistorted 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.

I used a combination of color threshold and gradient thresholds to generate a binary image as called out in color_thresholding and abs_soble_thresh (thresholding steps at lines 100 through 110 in `image_gen.py`). For color thresholding, I used S threshold in the HLS color space and V threshold in the HSV color space. After tweaking around with the threshold values, the lane lines looked pretty decent and then I combined them to the sobel x transform to achieve good identification of both left and right lines.

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 in code lines 111 to 128 in the file image_gen.py .

To generate the transformed image, I would need the image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:

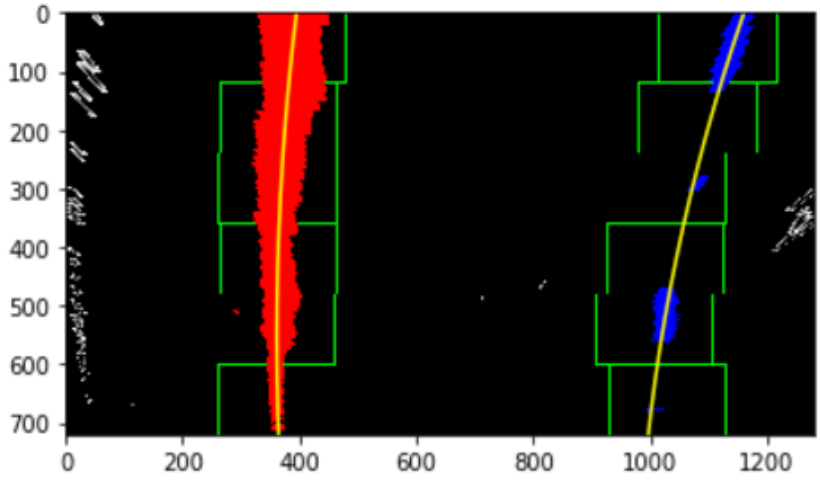
```
src = np.float32([[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],[((img_size[0] / 6)
      - 10), img_size[1]],[(img_size[0] * 5 / 6) + 60, img_size[1]],[(img_size[0] / 2
      + 55), img_size[1] / 2 + 100]])

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]])
```

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?

Next, I used the sliding window technique as taught in the Udacity classroom notes to detect lane pixels.

The first step is to compute a histogram of the bottom half of the the warped image. It will theoretically have 2 prominent peaks that correspond to the left and right lane lines. I identify the index of the peak left-lane marker by using the `np.argmax()` function on the left half of the histogram. I perform the same task for the right lane by processing the right half of the histogram. I use a search window approach to detect follow the lane line pixels. There are 6 windows, each window height is **image size/number of windows** and width with a margin of **100**. The algorithm then steps through the windows one by one to find the number of pixels inside the window boundaries. If the pixel numbers within a window is greater than the minimum number **50**, then the next window is recentred around the mean position of current pixels.
The output of all this is a set of **(x,y)** values, the windows are drawn over the image and a polynomial equation is used to fit the lines. This is depicted as a yellow line on the image.



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 238 through 265 in my code in `image_gen.py`. As we have calculated the radius of curvature based on pixel values, so the radius we are reporting is in pixel space, which is not the same as real space. Hence, the code 'ym_per_pix' and 'xm_per_pix' is used to convert the pixel values to meter. As, well I calculated the position of the car relative to the center of the road.

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 code lines 131 through 272 in image_gen.py .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 is a screencap of the final video output and the full video is in the file 'output_tracked_final.mp4'.

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 project to me was the hardest compared to the previous projects. I had to code many lines and many tweaking due to the diff options of generating the binary images or the line drawing methods. For the binary image generation, a lot of tweaking had to be done as the shadow were a feature that was hard to remove. As well, for the line finding method, I initially tried out the convolutional method in the Udacity notes but it was not able to detect a clean line so I switched to the window sliding technique which worked more better.