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.

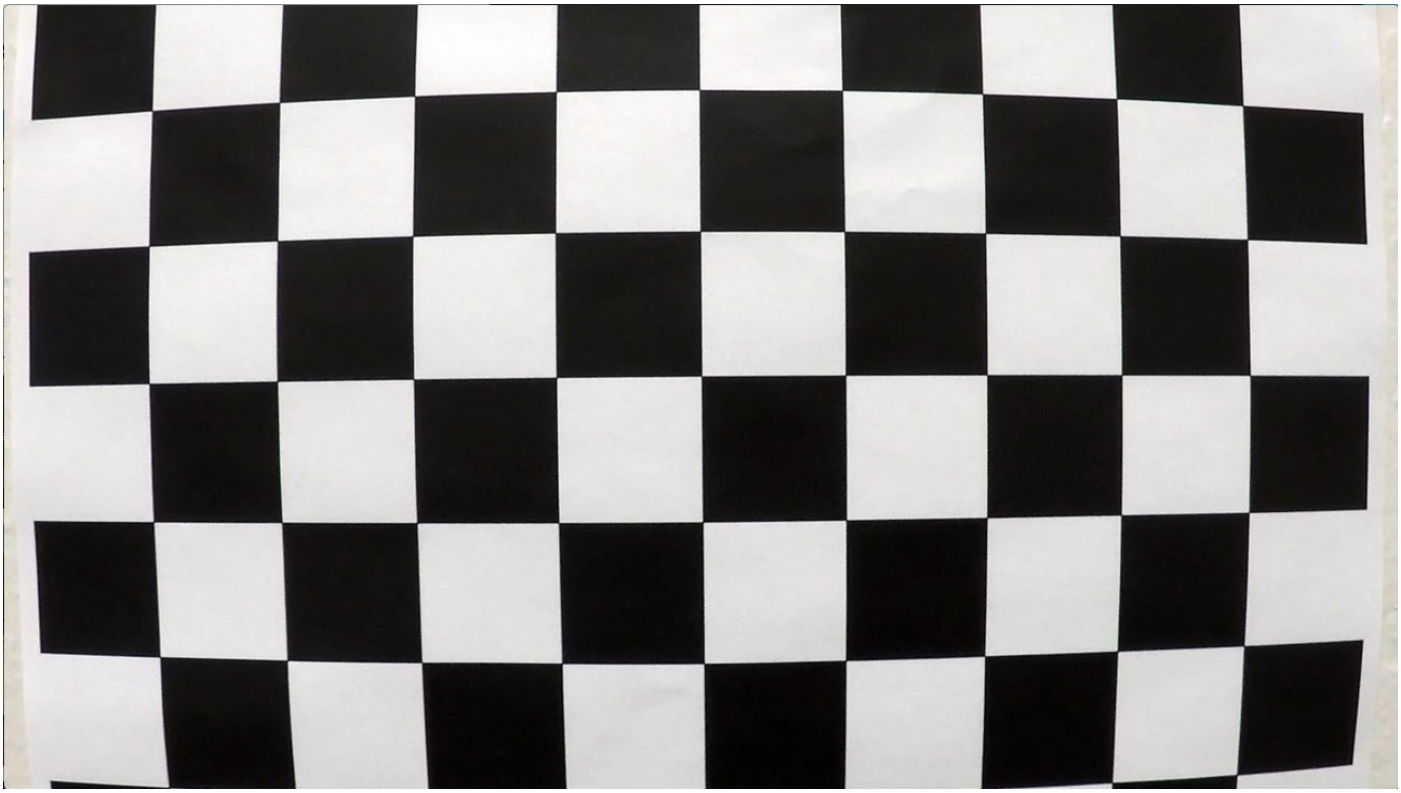
Camera Calibration

Code reference: `calibrator.py`

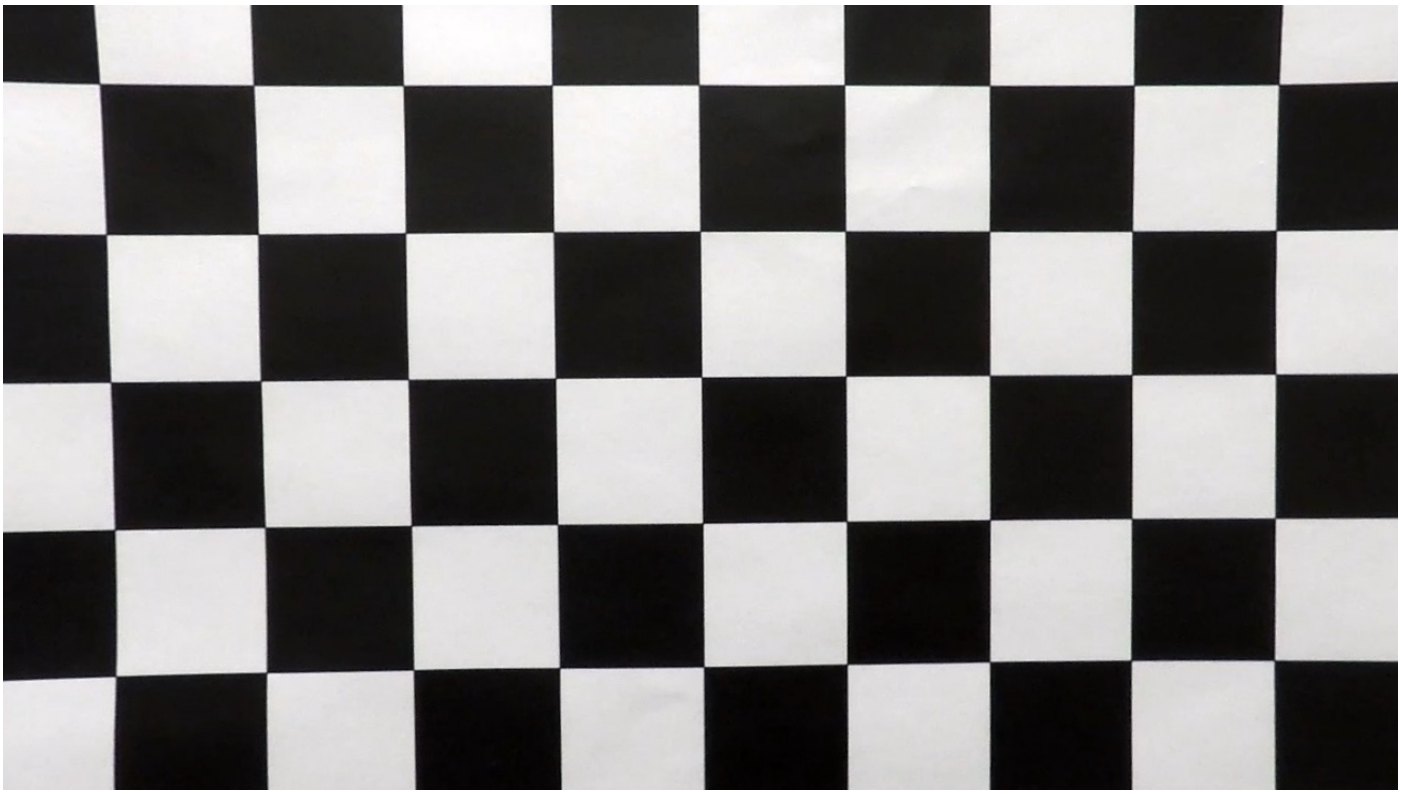
I followed what the course showed me: using `cv2.findChessboardCorners()` to calculate the `imgpoints`. Then by using `cv2.calibrateCamera()` with `imgpoints` and `objpoints`, we can get the final camera calibration parameters. As a result, we can use `cv2.undistort` with these parameters to undistort an image.

Here’s the difference between original and undistorted image.

Original



Undistorted



Pipeline

I have implemented a pipeline which processes each image from the incoming video.

Step 1: Undisort image

Code reference: `calibrator.py`

I use the calibrated `Calibrator` to undistort input image. (See function `calibrate_with_chessboard_images()`, `undistort()`)

Before:

After

Step 2: Perspective transform

Code reference: `image_processor.py`

I hardcoded two reigons: original and transformed, to represent the perspective transformation:

```
src = np.float32(
    [[260, 691],
     [602, 446],
     [683, 446],
     [1049, 682]])

dst = np.float32(
    [[260, 691],
     [260, 0],
```

```
    [1049, 0],
    [1049, 682]])
```

Then I use `cv2.getPerspectiveTransform()` to calculate the transform and invert transform matrix, which is later used to perspective transform an image. (See function `warp()`). Here's the result:

Before:

After:

After this step, we get a bird eye view of the lane.

Step 3: Extract the interesting pixels and generate binary image

Code reference: `image_processor.py`

I use Sobel algorithm to calculate the gradient on x-axis. I choose x-axis because of our lane line orientation. Only pixels with gradient that lies between my threshold will remain, and other pixels will be removed. As a result, we will only leave the area that has great change on RGB value.

I aslo filter image using the saturation channel and lightness channel in HLS model. Using saturation channel can effectively help me remove the ground, and lightness channel can effectively help me remove the shadows. (See function `to_binary_edge()`)

The above filter results are mixed by:

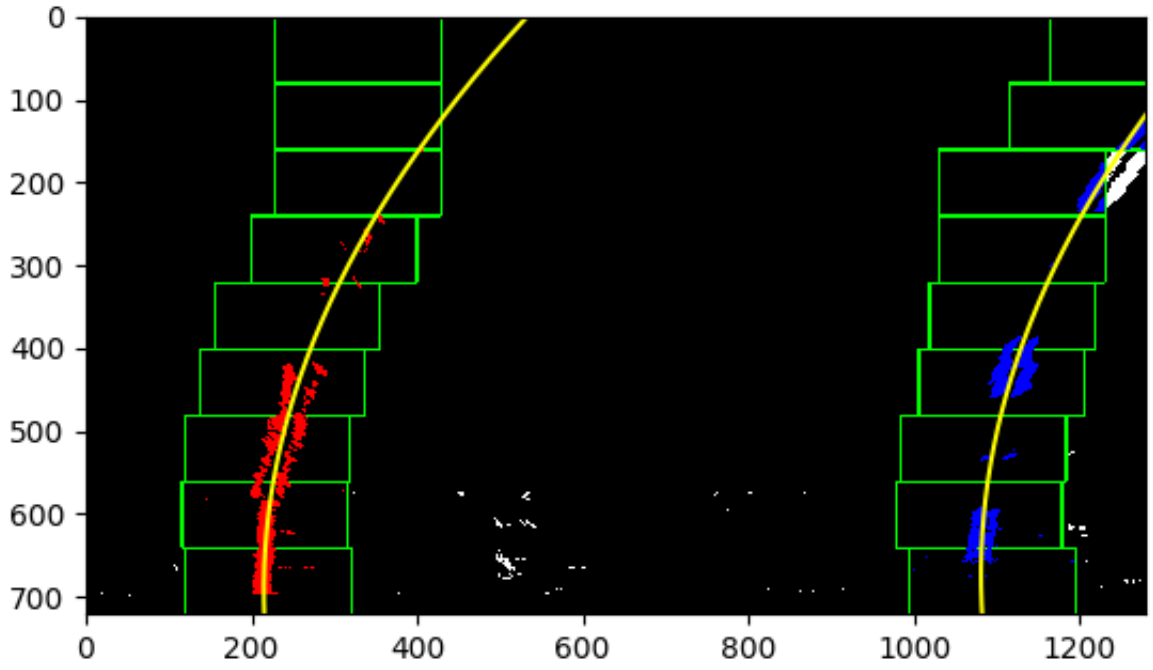
```
combined_binary[((s_binary == 1) | (sx_binary == 1)) & (l_b
inary == 1)] = 1
```

And here's the result:

Step 4: Polyfit lane pixels

Code reference: `lane_finder.py`

I followed the example given in the course to use a sliding window apporach to find the pixels for left line and right line, then fit them separately into a Ax^2 + Bx + C polynomial, and then draw the result on screen. (See function `calc_lane()`). Here's what I get:



Step 5: Draw lane onto original image

Code reference: `lane_finder.py`

I also followed the example givin in the course to fill the area between two polynomial lines first, then unwarp the image into its original perspective. (See function `draw_lane_on_image()`). And here's the result:

Step 6: Calculate lane curvature and car position

Code reference: `lane_finder.py`

Since I have no idea about the real world size each pixel represents, I have to make some assumption and give very rough estimate about lane curvature and car position. On y-axis, I assume 300 pixels correspond to 30 meters; On x-axis, I assume 900 pixels correspond to 3.7 meters. Then I can calculate the lane curvature using the formula provided in the lecture. (See function `calc_lane()`)

To calculate the car's offset to lane center, I simply calculate the pixel differences between image center and lane's center, then multiply by the length each pixel represents on x-axis. (See function `calc_lane()`). The I draw these info back to road image (See function `draw_info_on_image()`).

Here's the result:

Process video using pipeline

Using the pipeline above, I was able to process the video. Please see the video in my submission (`project_video.mp4`).

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?

First I found that shadows are very annoying as they cuts the lane line pixels and they will also generate noises. I then try to use lightness channel thresholds to only include the pixels that are not very dark, and reduce the effect of shadow a little. But strong shadow is still a big issue.

Also I found that the fitted polynomial to be instable. I then did a moving average by looking at current frame and 6 past frames to get an smoothed result. This does stablize the lane but also make it less senstive to rapid lane curve changes.

I guess there’s a lot more to be fine tuned as well.