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.

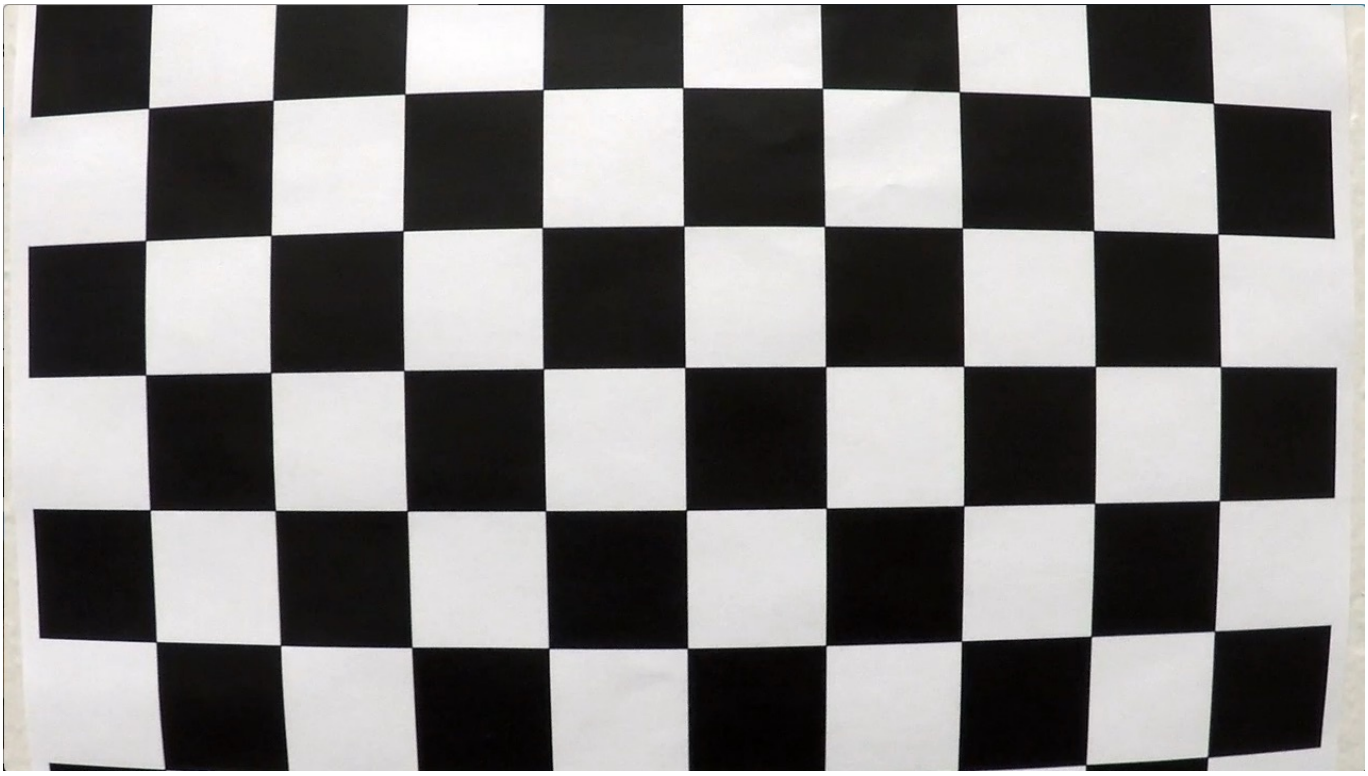
Camera Calibration

The code for this step is contained in the first code cell of the IPython notebook located in "./advanced_lane_finding.ipynb."

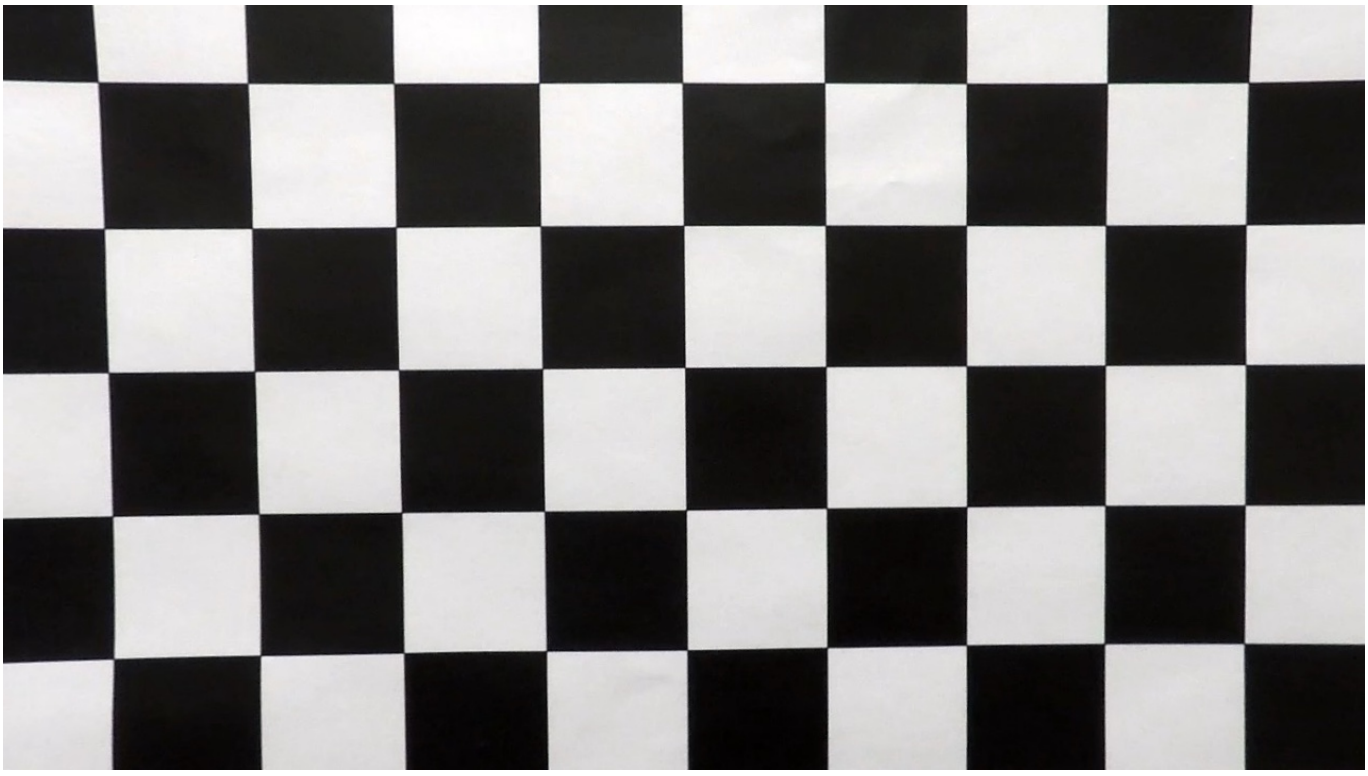
I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of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

With distortion:



Without distortion:



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:

Original image:

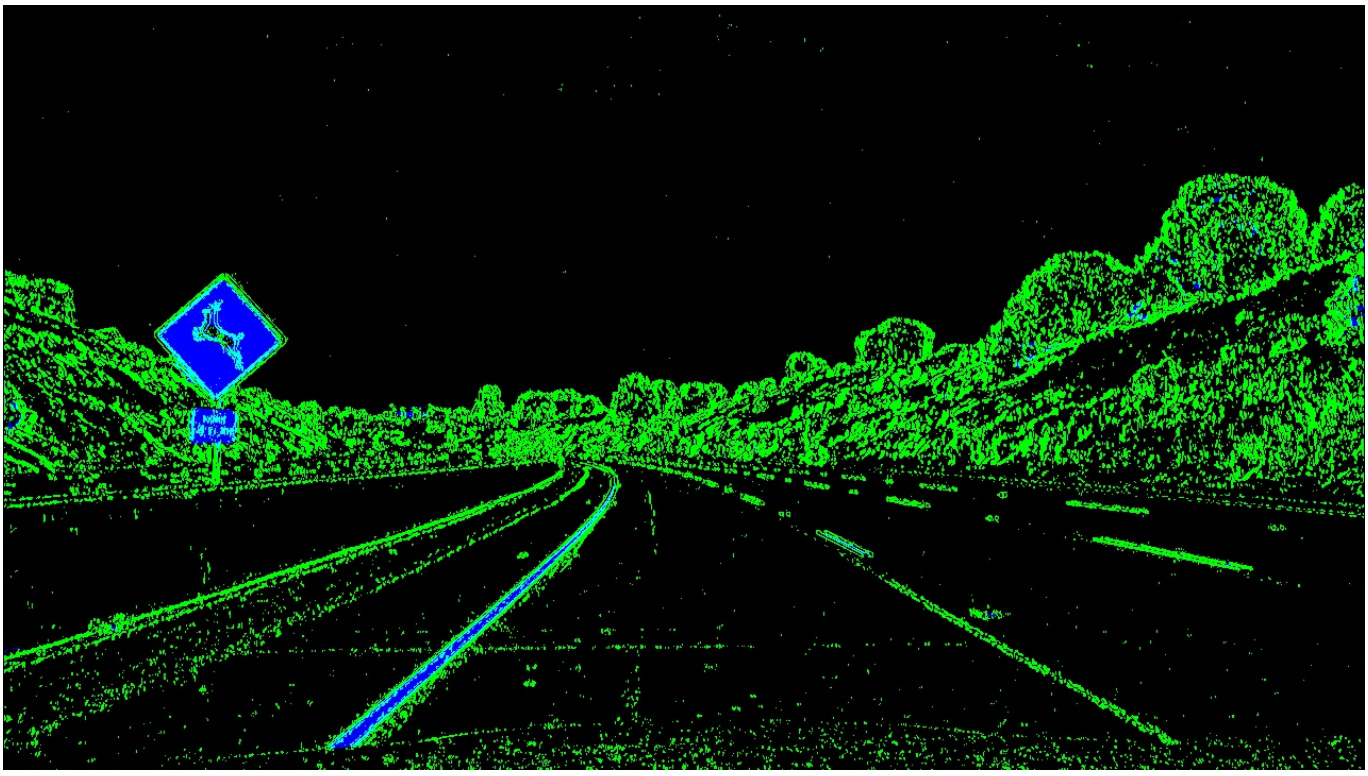
Undistorted image:

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 and gradient thresholds to generate a binary image. The function is called `threshold()` in the 2nd code cell of the IPython notebook. Here's an example of my output for this step.

Combined binary:

Color binary:



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 `warp_plus()`, which appears in the 2nd code cell of the IPython notebook. The `warp_plus()` function takes as inputs an image (`img`). The source (`src`) and destination (`dst`) points are defined inside the function. I chose the source and destination points in the following manner:

```
src = np.float32(
    [[(img_size[0] / 2) - 60, img_size[1] / 2 + 100],
    [((img_size[0] / 6) - 20), img_size[1]],
    [(img_size[0] * 5 / 6) + 60, img_size[1]],
    [(img_size[0] / 2 + 65), 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]])
```

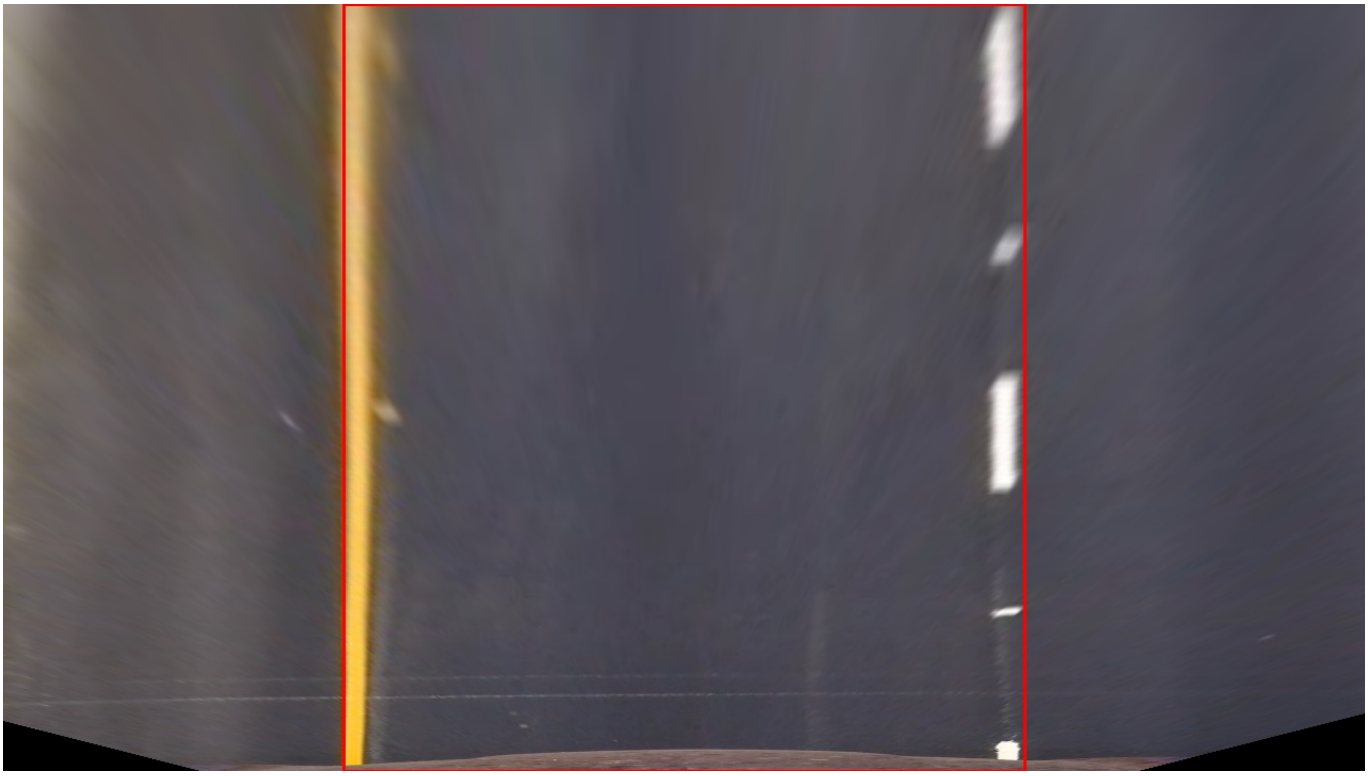
This resulted in the following source and destination points:

Source	Destination
580, 460	320, 0
193, 720	320, 720
1127, 720	960, 720
705, 460	960, 0

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image named 'straight_lines1.jpg' and its warped counterpart to verify that the lines appear parallel in the warped image.

Before perspective transform:

After persepective transform:

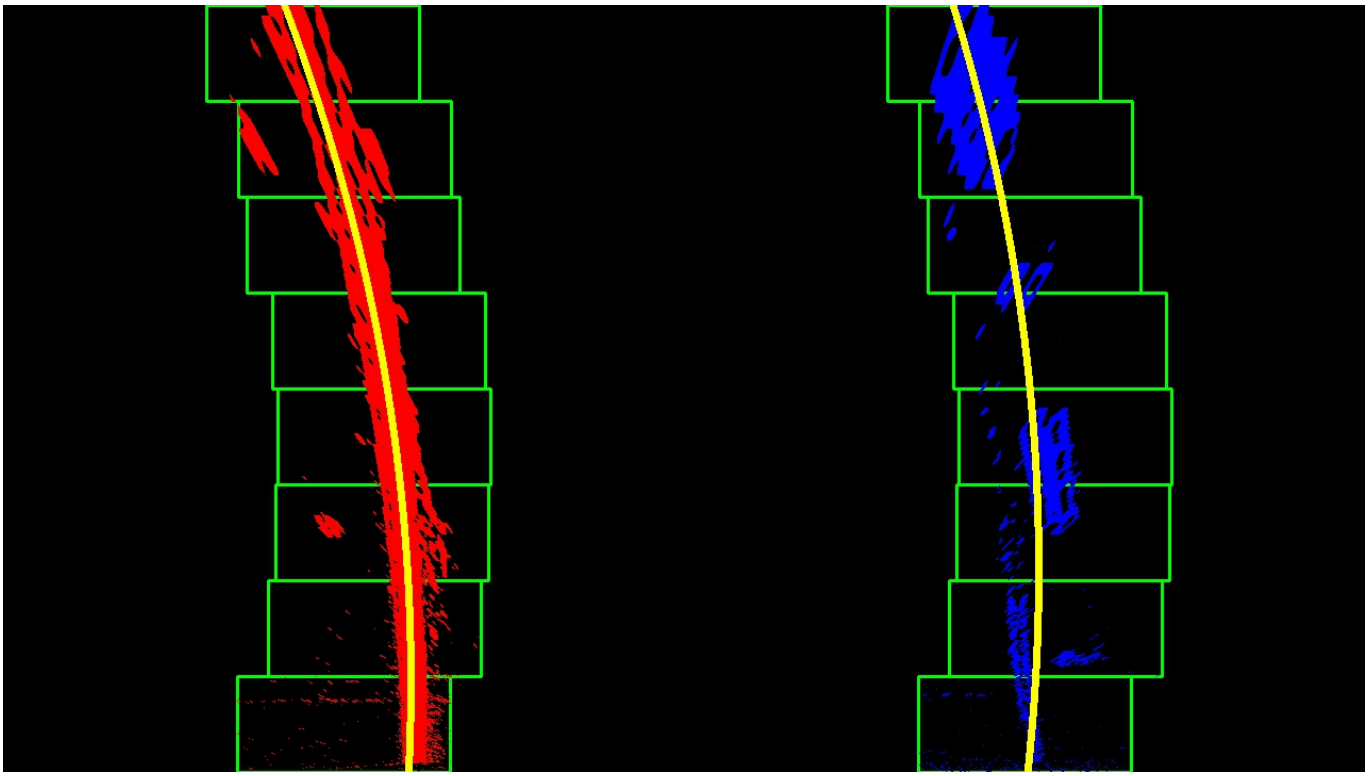


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

First I applied the perspective transform to the combined binary result of Step 2. Then I used a mask function named `region_of_interest()` in the 2nd code cell of the IPython notebook to crop out the uninterested regions. The vertices for the mask are:

```
vertices = np.array([[(200, img_size[0]),(1200, img_size[0]),(1200, 0),
(200, 0)]], dtype=np.int32)
```

Then I used the sliding window method to find the lane lines and fit my lane lines with a 2nd order polynomial. The related functions are called `find_lane_pixels()` and `fit_polynomial()` in the 2nd code cell of the IPython notebook. The result is shown below:



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 function `measure_curvature_real()` in the 2nd code cell of the IPython notebook. I defined conversions in x and y from pixels space to meters and implemented the calculation of radius of curvature and offset of car center.

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 12th code cell of the IPython notebook. 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!).

The video is located at test_videos_output/project_video_dw.mp4. Here's a link to the file.

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?

Here I'll briefly talk about the tricks that I used:

- Applying a mask to the warped binary image
- Using a Line class to track the polynomial coefficients for last N iterations (16th code cell)
- Relating the sliding windows of left lanes with those of right lanes (lines 136 to 141 in 2nd code cell)
- Adding centers of the sliding windows to the good data point lists in function `find_lane_pixels()`.

The pipeline might fail with lighting condition changes (like shadows) and obvious line marks within region of interest. To go further, I might try using deep learing method to find a more robust solution.