

Manage topics

 22 commits

 1 branch

 0 releases

 1 contributor

Branch: master ▾
















New pull request


Create new file


Upload files

Find File

Clone or download ▾

 ameter Merge branch 'master' of github.com:ameter/CarND-AdvancedLaneLines-P4		Latest commit e189202 on Dec 15, 2017
 camera_cal	initial commit	a year ago
 examples	initial commit	a year ago
 output_images	gathered writeup images	a year ago
 test_images	gathered writeup images	a year ago
 writeup_images	cleaned up code a bit	a year ago
 README.md	Added write-up.	a year ago
 calibrate_camera.py	gathered writeup images	a year ago
 calibration.p	initial commit	a year ago
 challenge_video.mp4	initial commit	a year ago
 example_writeup.pdf	initial commit	a year ago
 find_lanes.py	cleaned up code a bit	a year ago
 harder_challenge_video.mp4	initial commit	a year ago
 project_video.mp4	initial commit	a year ago
 writeup.md	Update writeup.md	a year ago

 README.md



Advanced Lane Finding

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

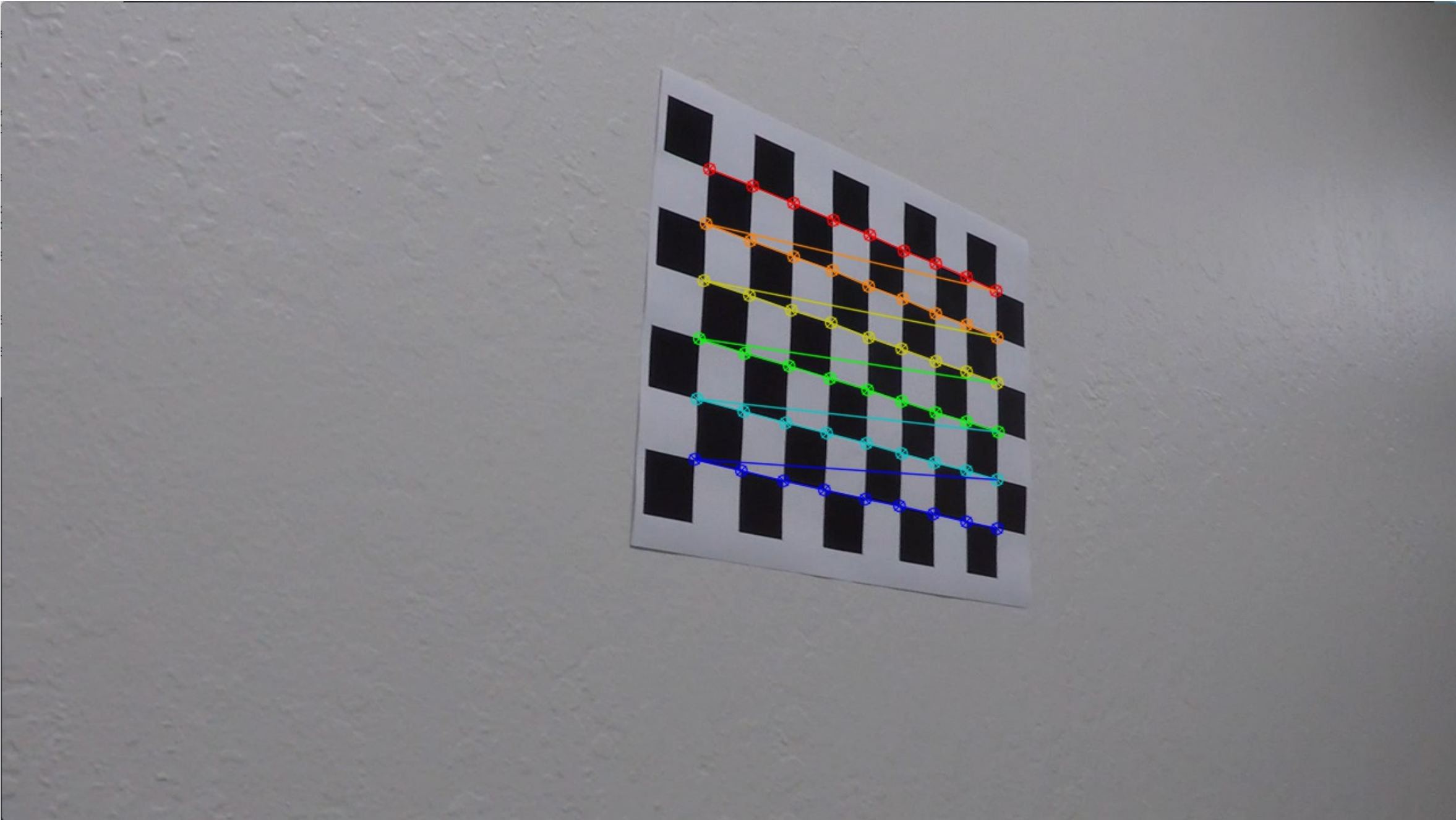
You're reading it!

Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

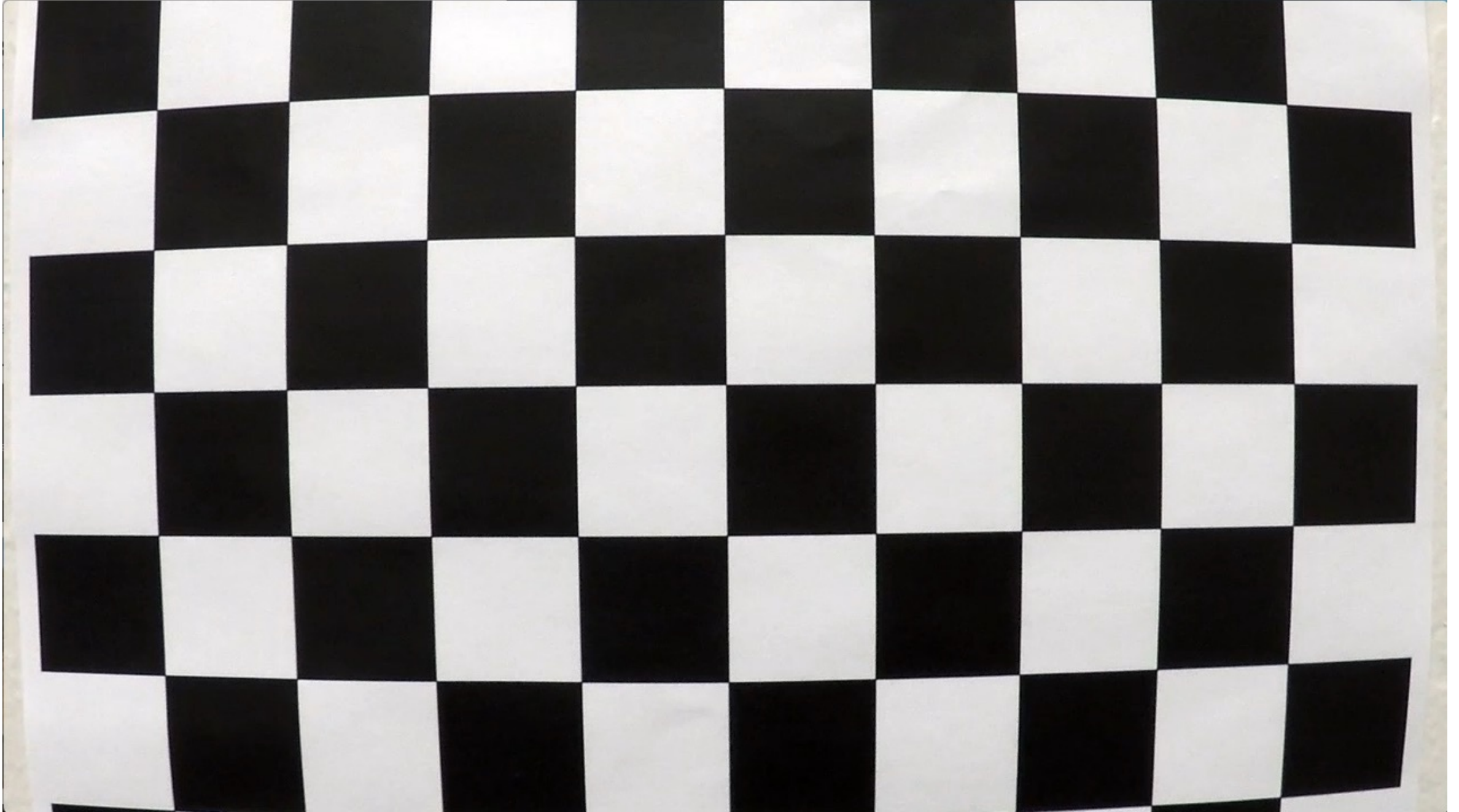
The code for this step is contained in the python file located here: `./calibrate_camera.py`.

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. The following is an example of a chessboard image with the corners detected:

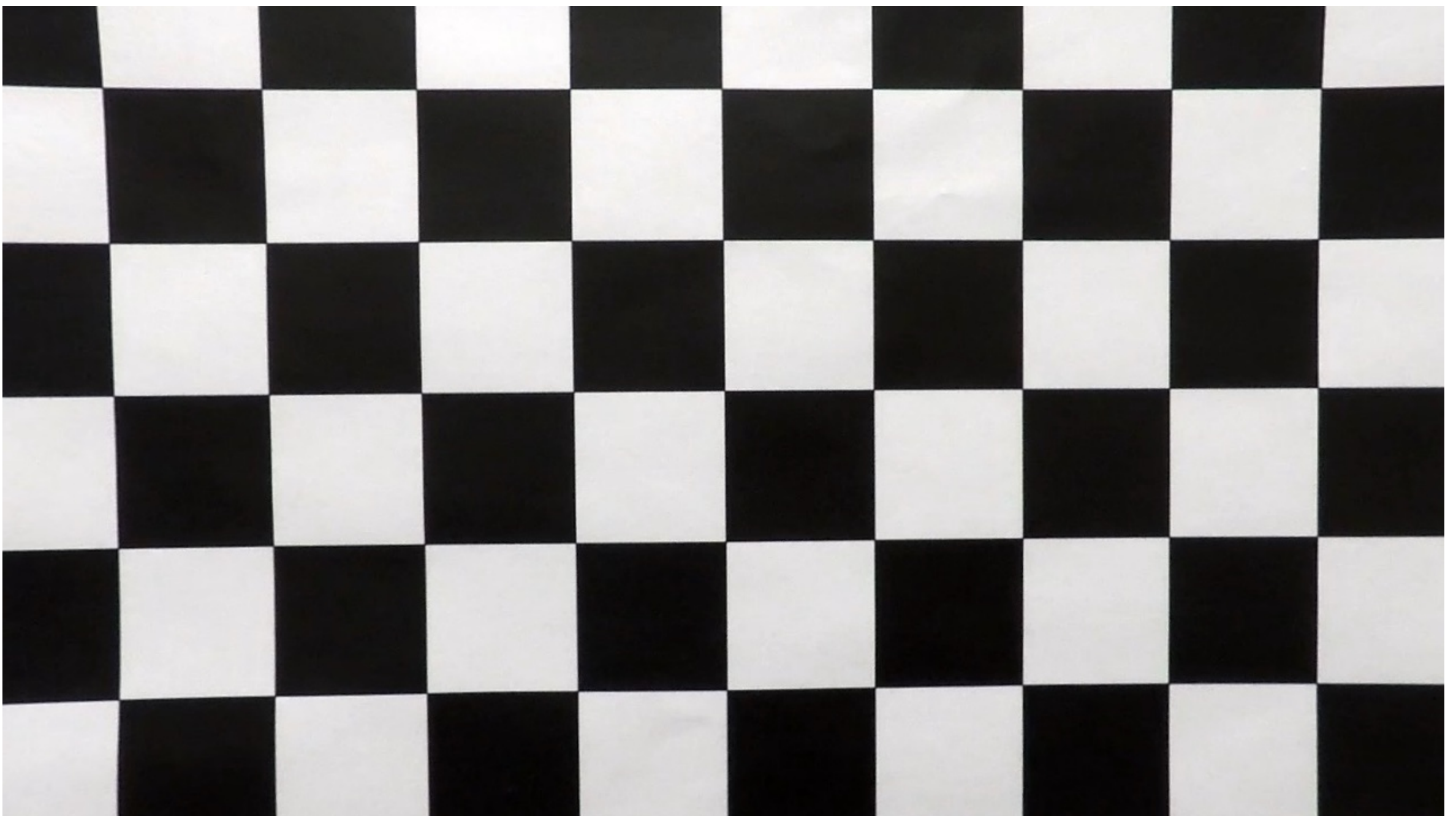


I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

Original Image:



Undistorted Image:



Pipeline (single images)

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

I used the `cv2.undistort()` function in the `./find_lanes.py` file (line 418) to apply the distortion correction to the test images like this one:

Input Image:



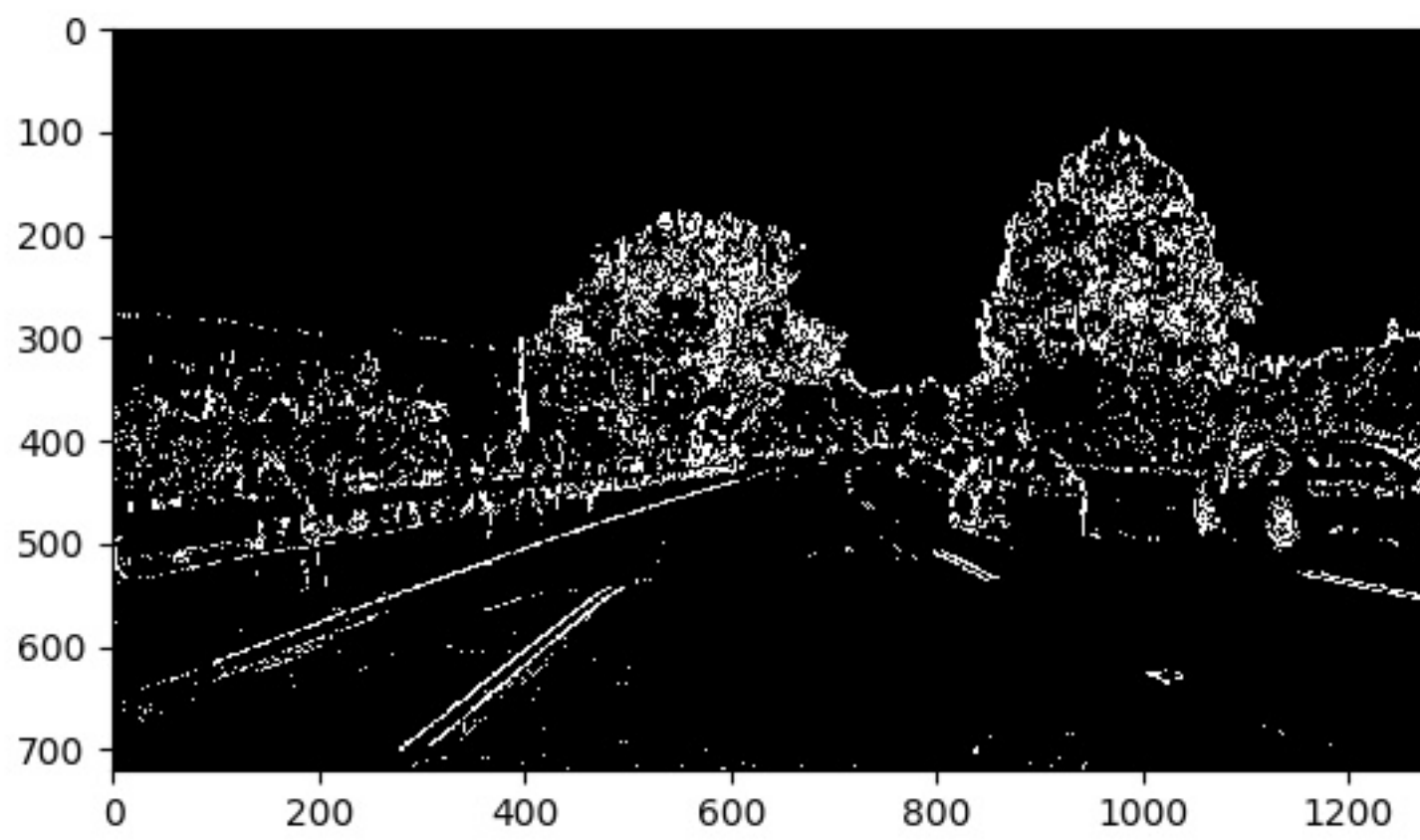
Undistorted Image:



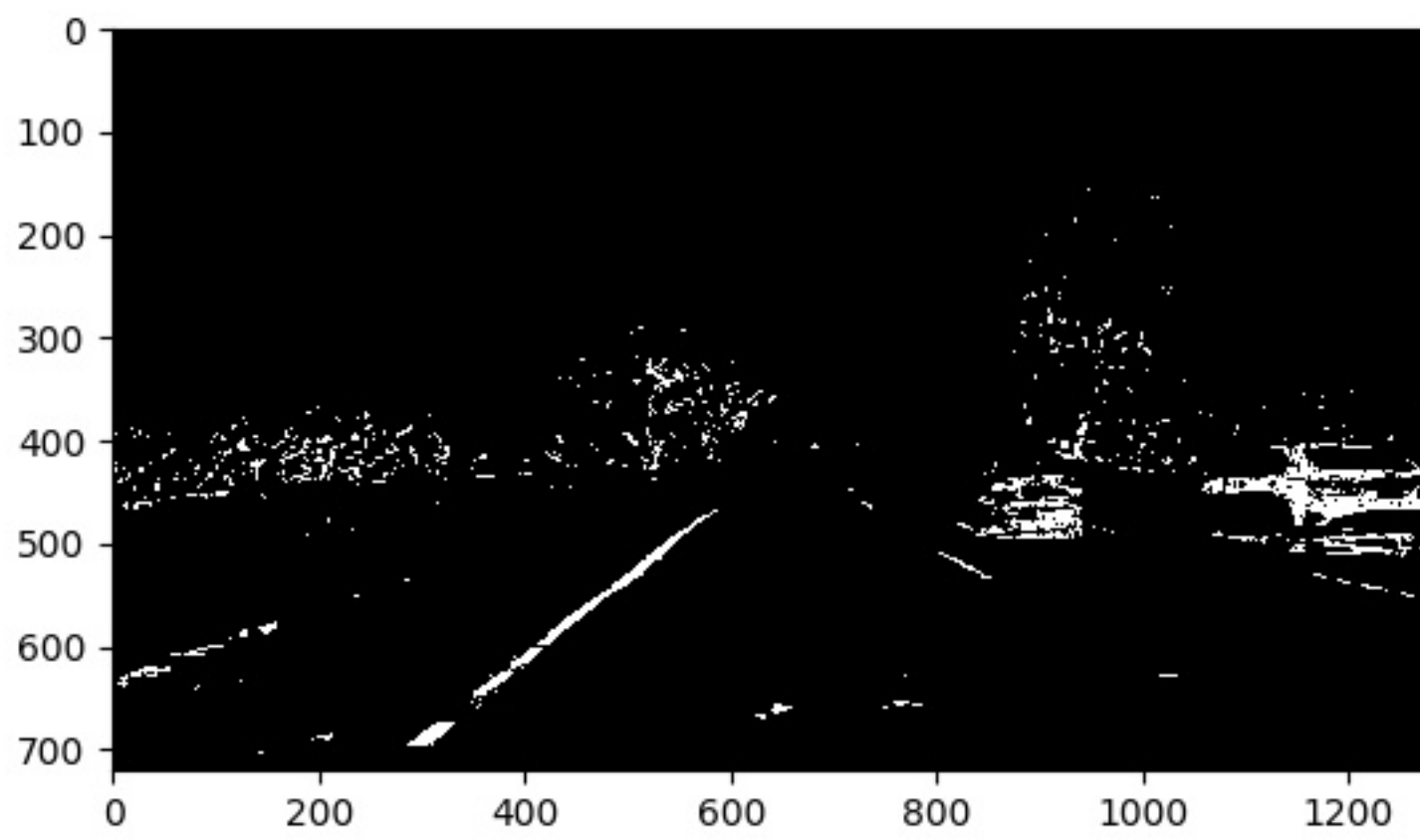
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

I used a combination of color and gradient thresholds to generate a binary image (thresholding functions at lines 70 through 91 in `./find_lanes.py`). Here's an example of my outputs for these steps.

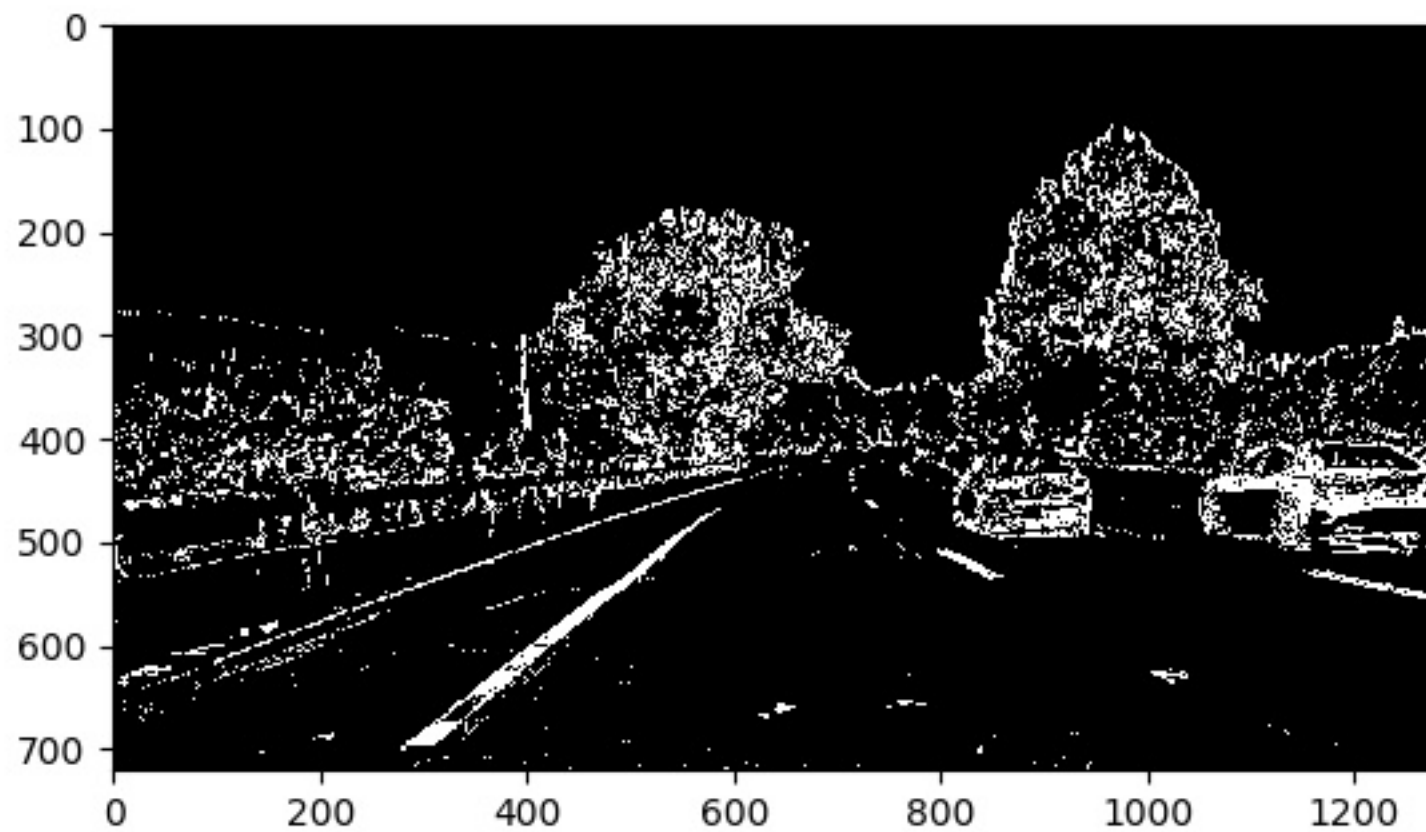
Sobel X Threshold:



HLS S-Channel Threshold:



Combined Threshold:



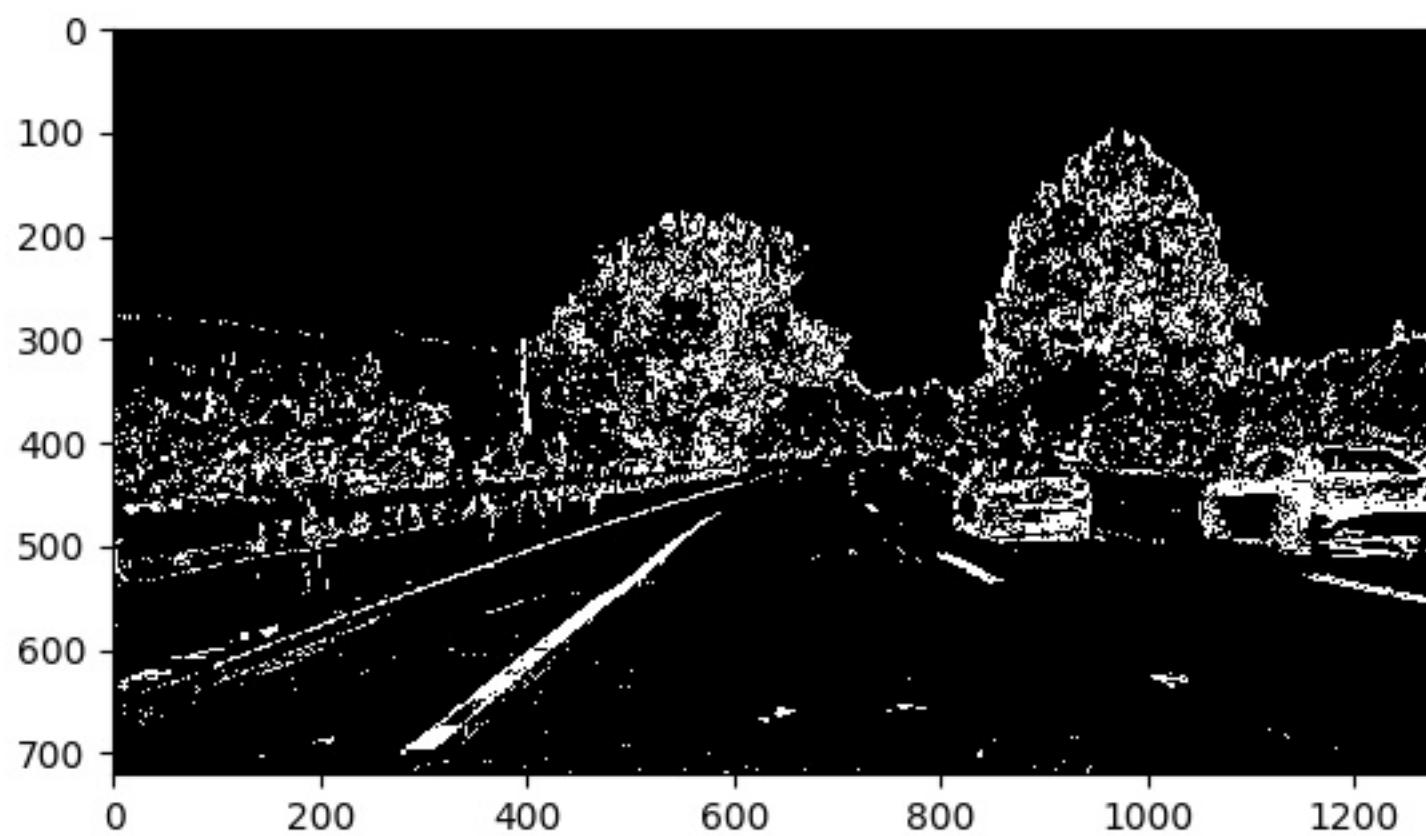
3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for my perspective transform includes a function called `perspective_transform(img)`, which appears in lines 103 through 123 in the file `./find_lanes.py`. The `perspective_transform(img)` function takes as input an image (`img`) and uses hardcoded source (`src`) and destination (`dst`) points. I chose the hardcoded the source and destination points in the following manner:

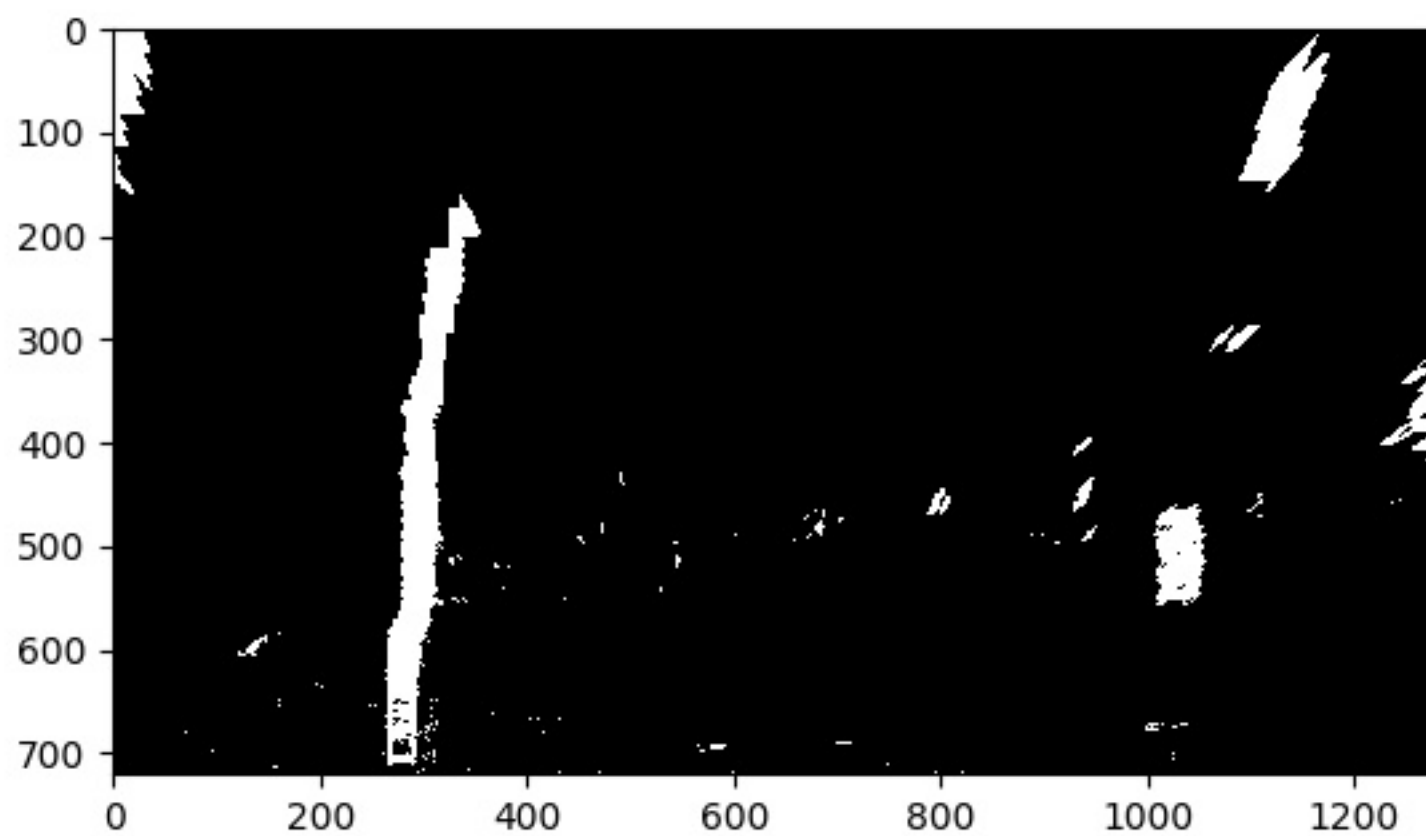
```
src = np.float32([[220, 719], [1220, 719], [750, 480], [550, 480]])
dst = np.float32([[240, 719], [1040, 719], [1040, 300], [240, 300]])
```

The following is an example binary thresholded image before and after applying the perspective transform:

Binary Thresholded Image:

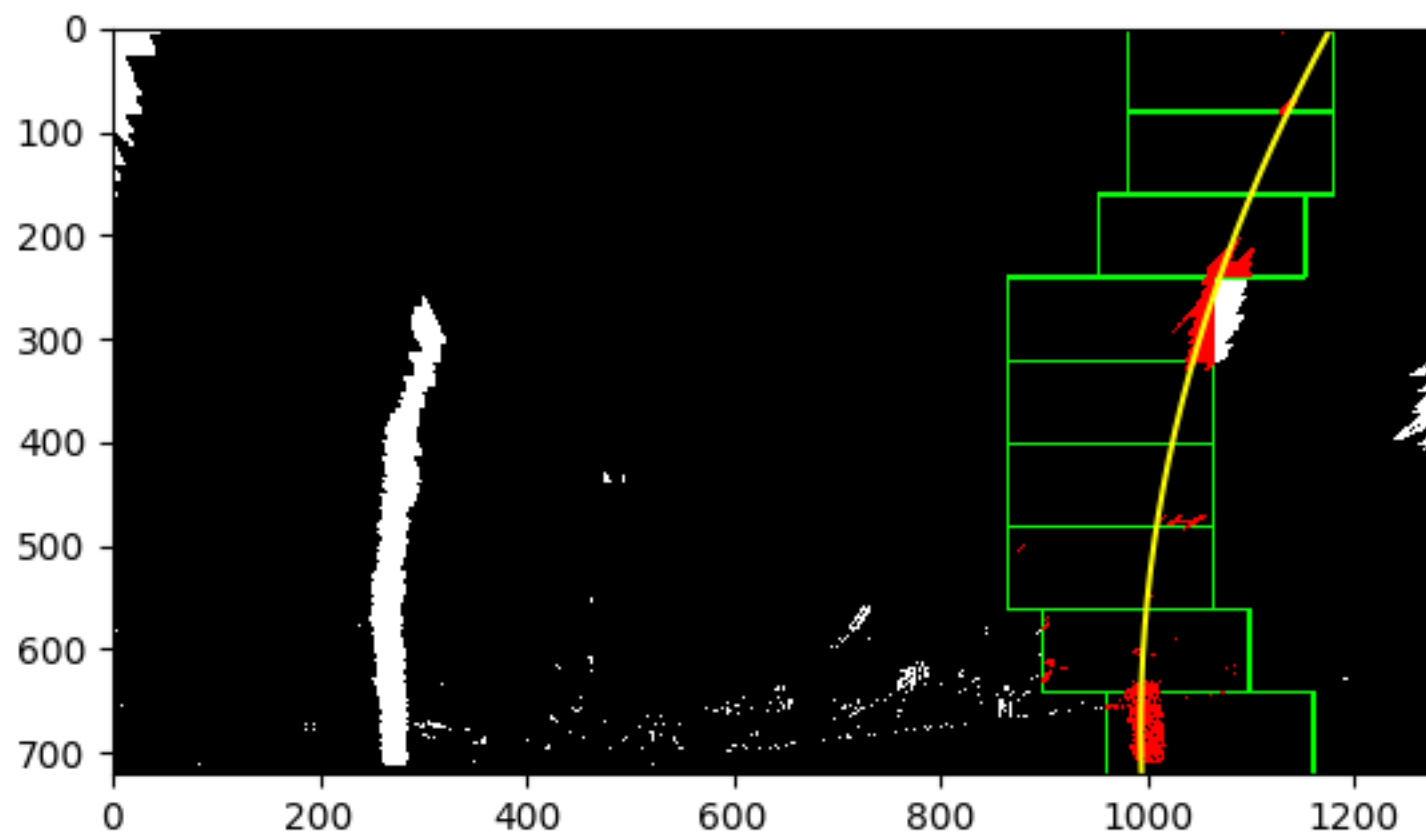


Warped Binary Thresholded Image:

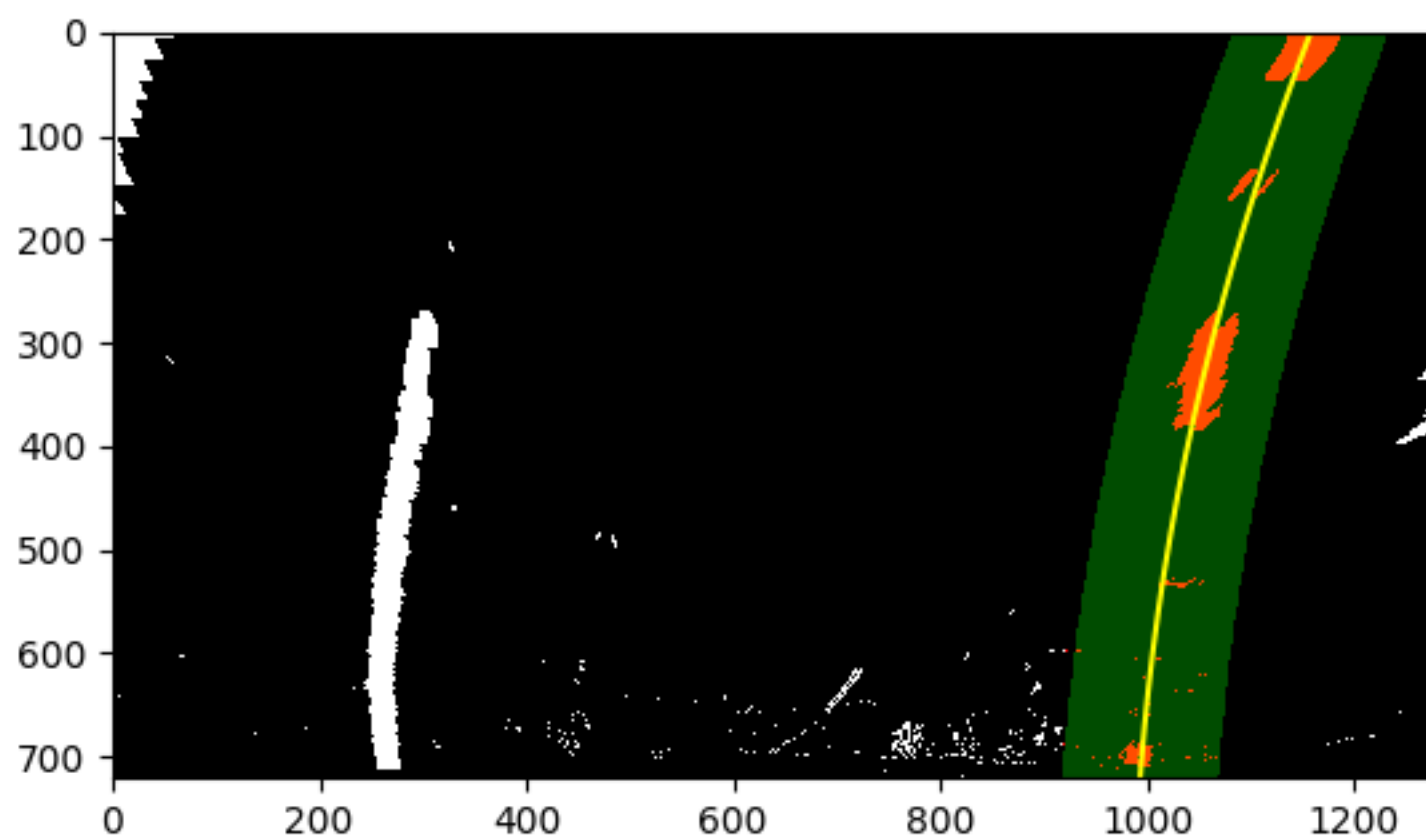


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

I created two functions in `./find_lanes.py` to find and update lane lines. If lanelines were not previously detected, I use the `find_line(binary_warped, line)` function to compute a histogram of the bottom half of the image, use that to find the bottom of the lane line, implement a sliding window approach moving upward to find pixels to include in the line (lines 126-200), and then fit a second order polynomial to the included pixels using numpy's `polyfit` function (line 249). An example of finding the right lane line with this technique follows:



If lane lines were detected within the previous 10 frames, I use the `update_line(binary_warped, line)` function in `./find_lanes.py` to detect lane lines by including pixels within a margin of 75 pixels from previously detected lines (lines 204-241) and then use the same polynomial fit function as above. An example of finding the right lane line with this technique follows:

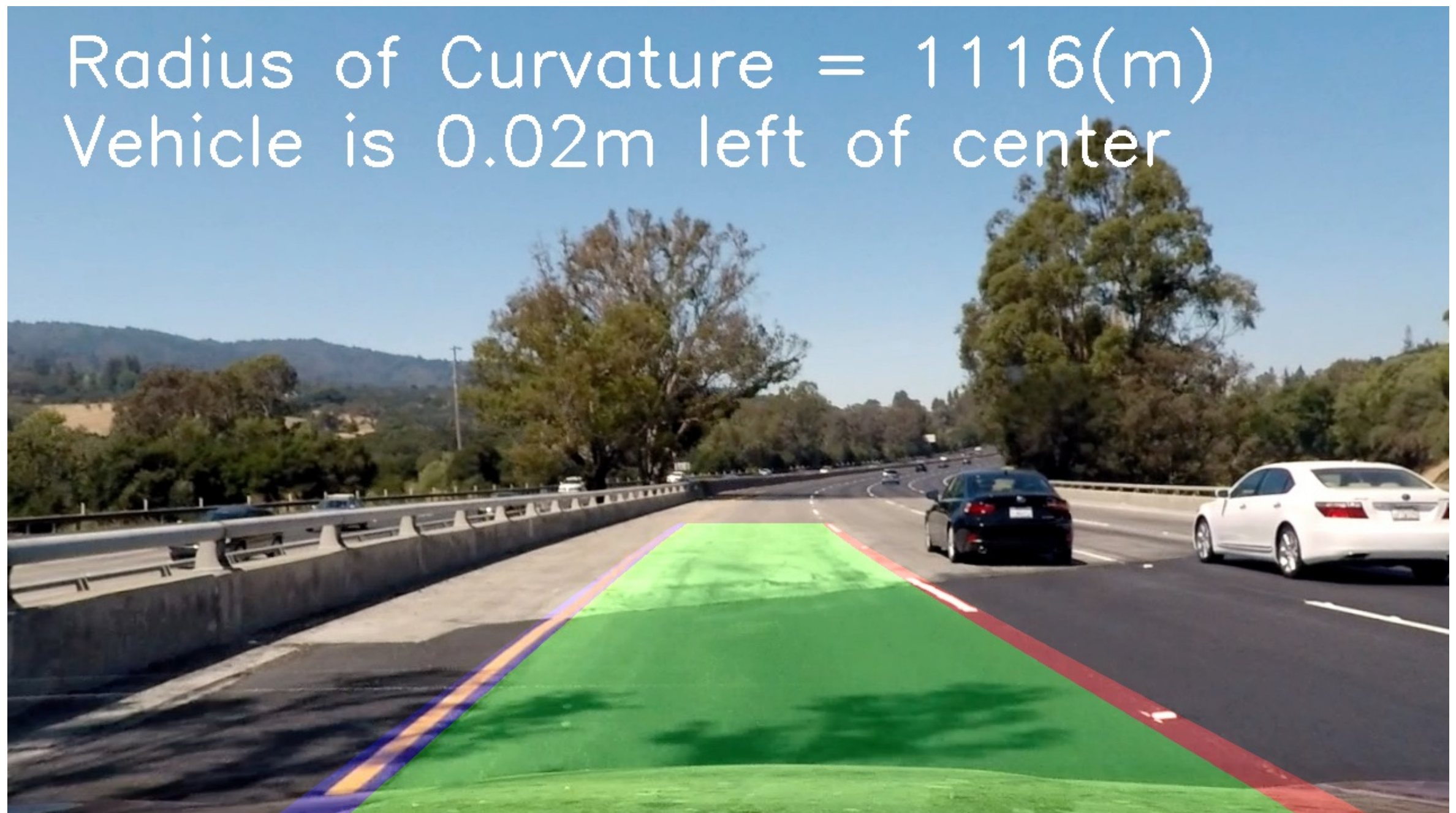


5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

I calculated the radius of curvature for each line based on the fit line described above (lines 270-275), and I calculated the vehicle position with respect to center by comparing the center of the frame (representing the camera position on the center of the vehicle) to the bottoms of the lane lines (lines 366-369) in `./find_lanes.py`.

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 function `generate_output(lines, Minv, binary_warped, img)` (lines 365-402) in `./find_lanes.py`. Here is an example of my result on a test image:



Pipeline (video)

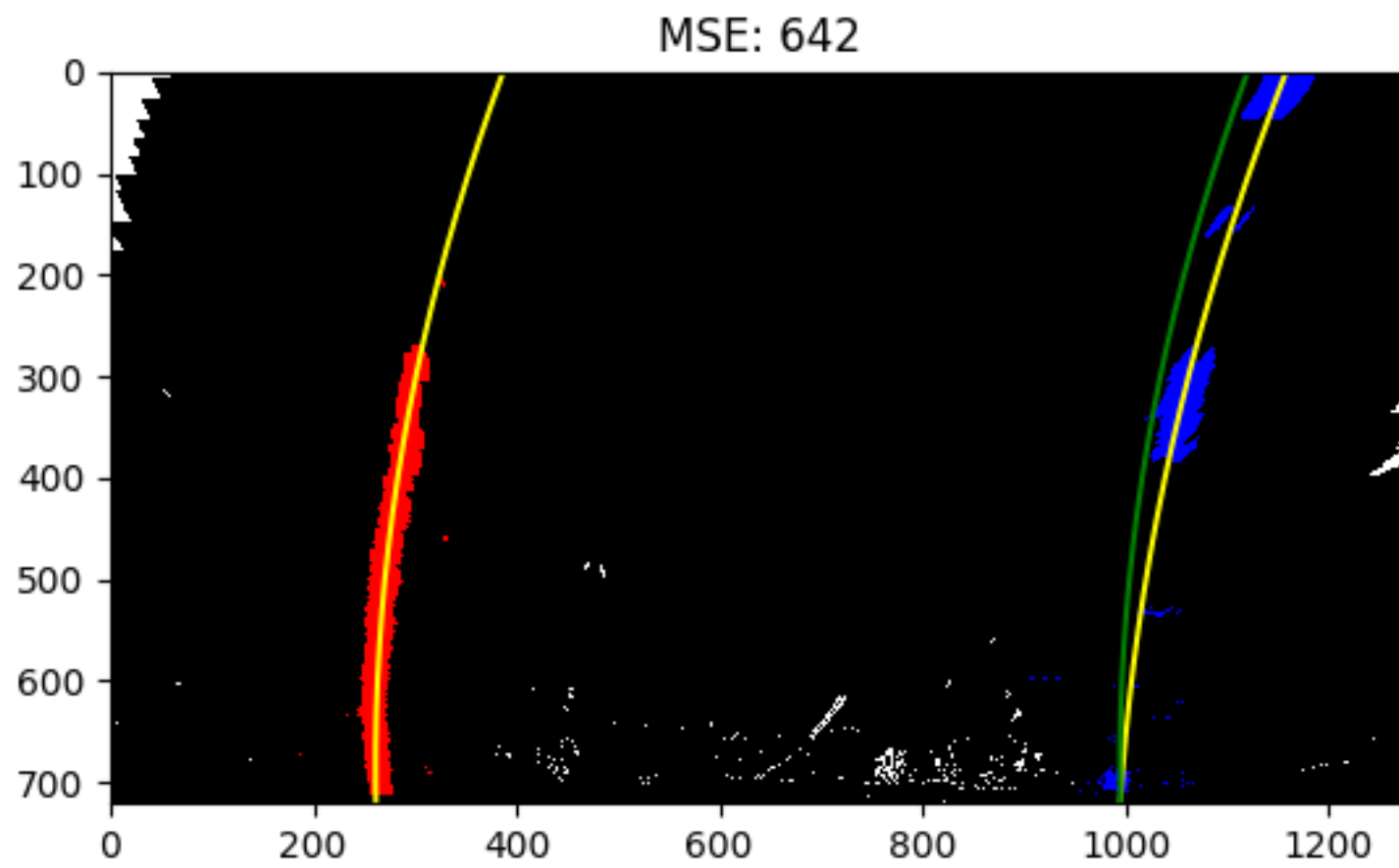
1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here'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?

My pipeline struggled with certain real-world issues such as shadows and changes in pavement, etc that led to erroneous line detections. To cope with this, I implemented a `check_sanity(lines, binary_warped)` function (lines 283-311) in `./find_lanes.py` that compared the difference in the left and right lane lines to ensure they were not too different. I experimented with a number of techniques to identify bad line detections, including assessing the change in a line's curvature from one frame to the next and comparing the curvature between the left and right lanes. Ultimately, I settled on comparing the left and right lines by overlaying the left line on the right and then computing the mean squared error (MSE) between the overlay and the right line. An example of using the technique is shown below, where the green line is the overlay and the MSE between the overlay and the right line is shown:



When bad detections are identified, the frames are dropped. If 10 frames are dropped in a row, the lane detection reverts from the margin technique to the sliding window technique, as described above. Overall, this led to good results on the project video. However, more testing would be needed on different types of images (lighting, pavement and line color differences, weather conditions, etc) to create a more robust solution.