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 (https://review.udacity.com/#!/rubrics/571/view) 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 (https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/writeup_template.md) is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

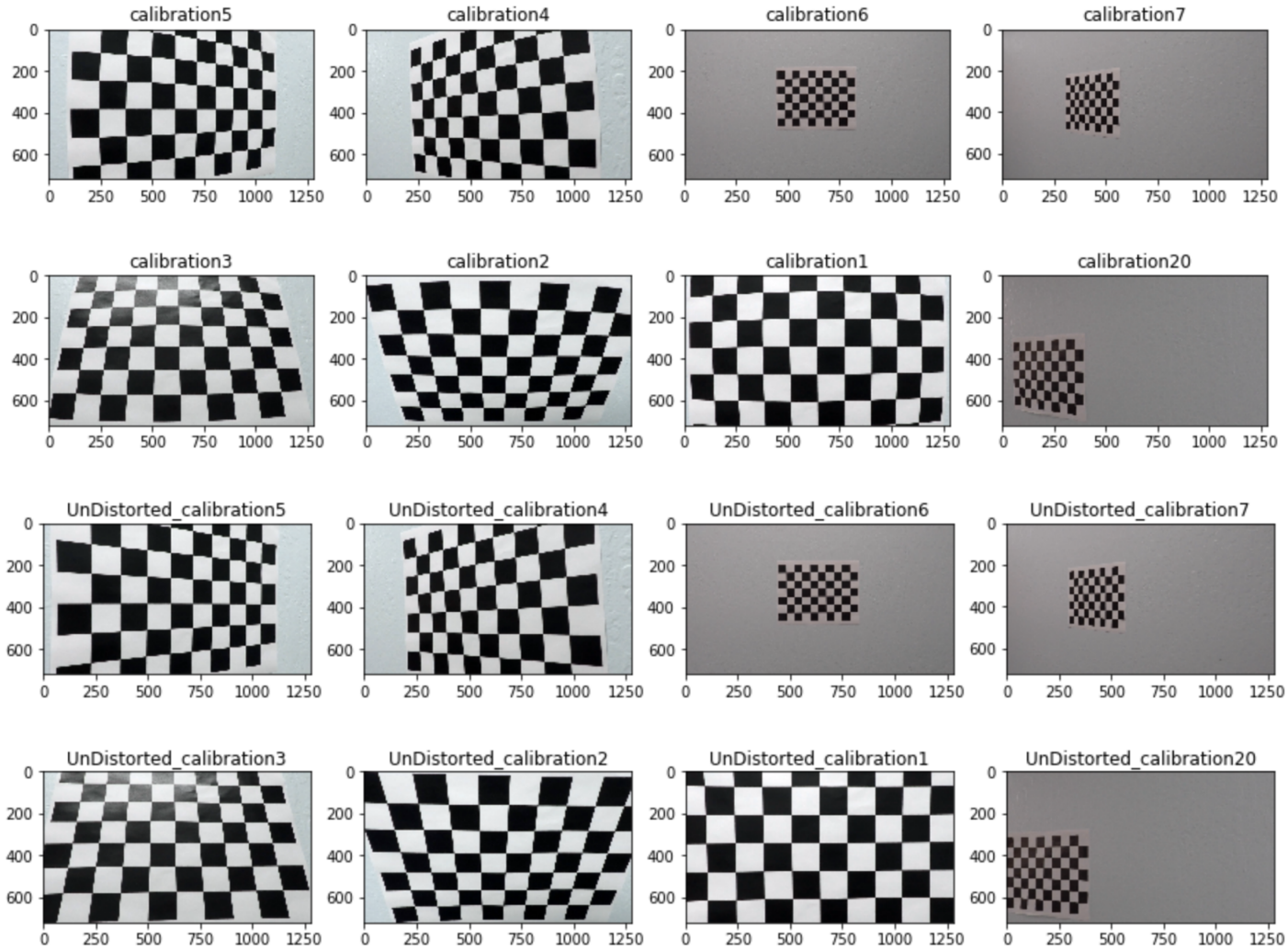
Camera Calibration

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 2nd-8th code cells of the IPython notebook located in "p4.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:



Pipeline (single images)

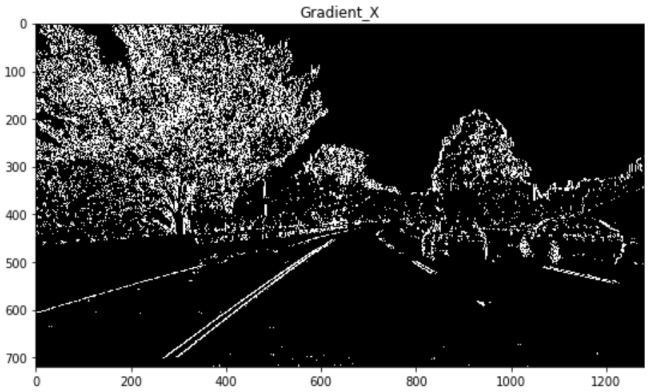
1. Provide an example of a distortion-corrected image.

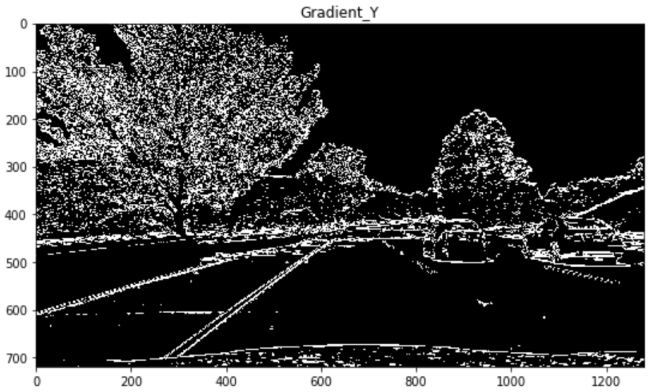
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

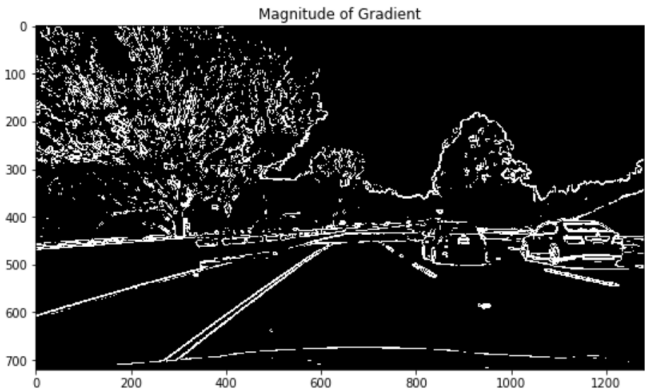
I used the `mtx` and `dist` I got using chessboard, then I applied the `cv2.undistort` function: `dst = cv2.undistort(img, mtx, dist, None, mtx)`. Then I saved all 8 undistorted images in a numpy matrix `undistorted`.

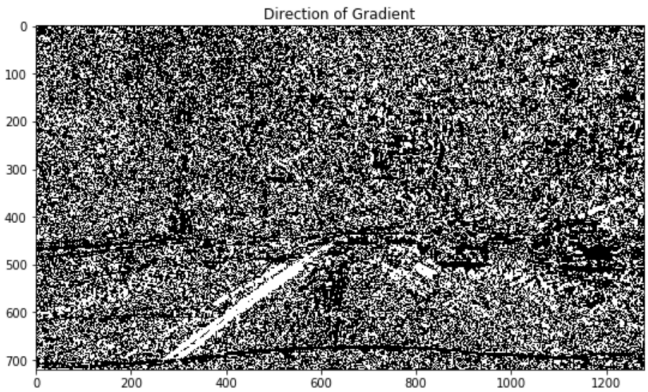
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

I first tried to use gradients and test parameters on one of the undistorted test images:





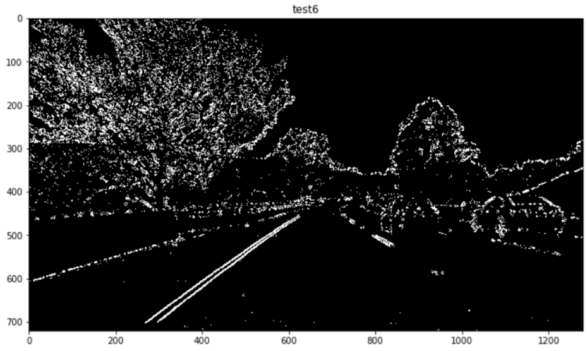


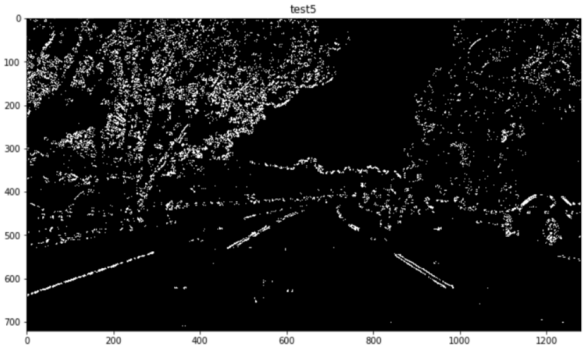


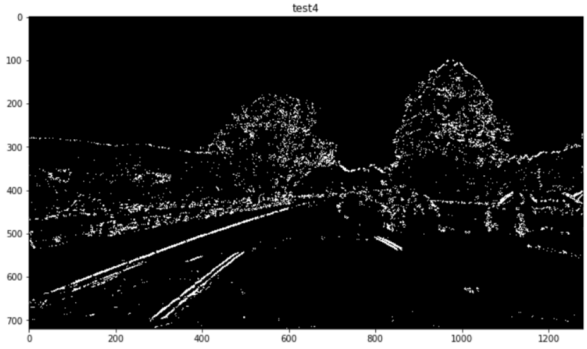
My final gradient pipeline look like this:

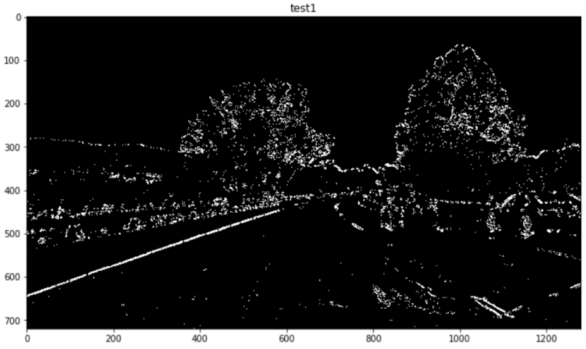
```
gradx = abs_sobel_thresh(img, orient='x', sobel_kernel=3, thresh=(20, 100))
grady = abs_sobel_thresh(img, orient='y', sobel_kernel=3, thresh=(20, 100))
mag_binary = mag_thresh(img, sobel_kernel=9, mag_thresh=(50, 200))
dir_binary = dir_threshold(img, sobel_kernel=15, thresh=(0.7, 1.3))
```

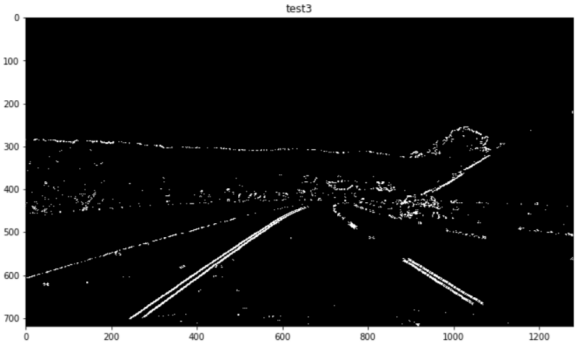
Then I combined all four gradient methods `grad_combined[((gradx == 1) & (grady == 1)) | ((mag_binary == 1) & (dir_binary == 1))] = 1` and obtained the following binary image:

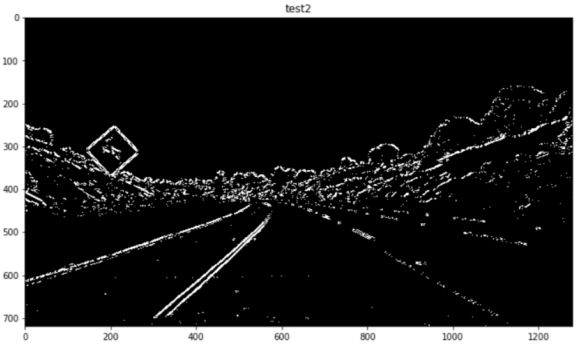


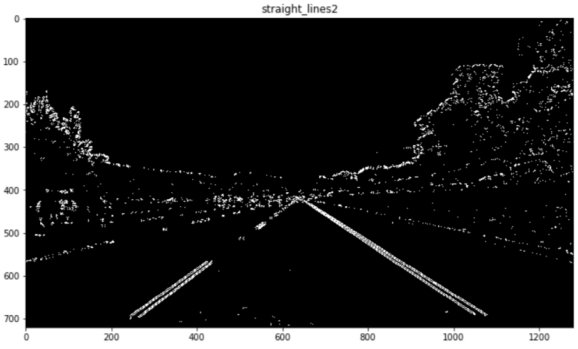


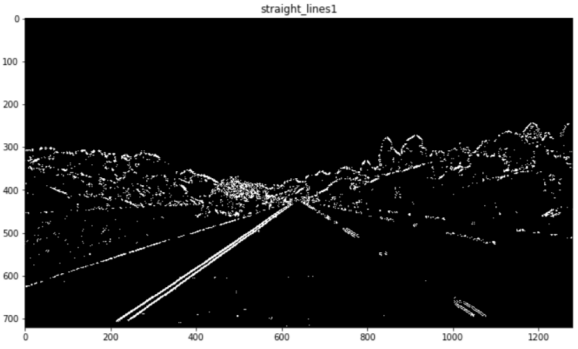




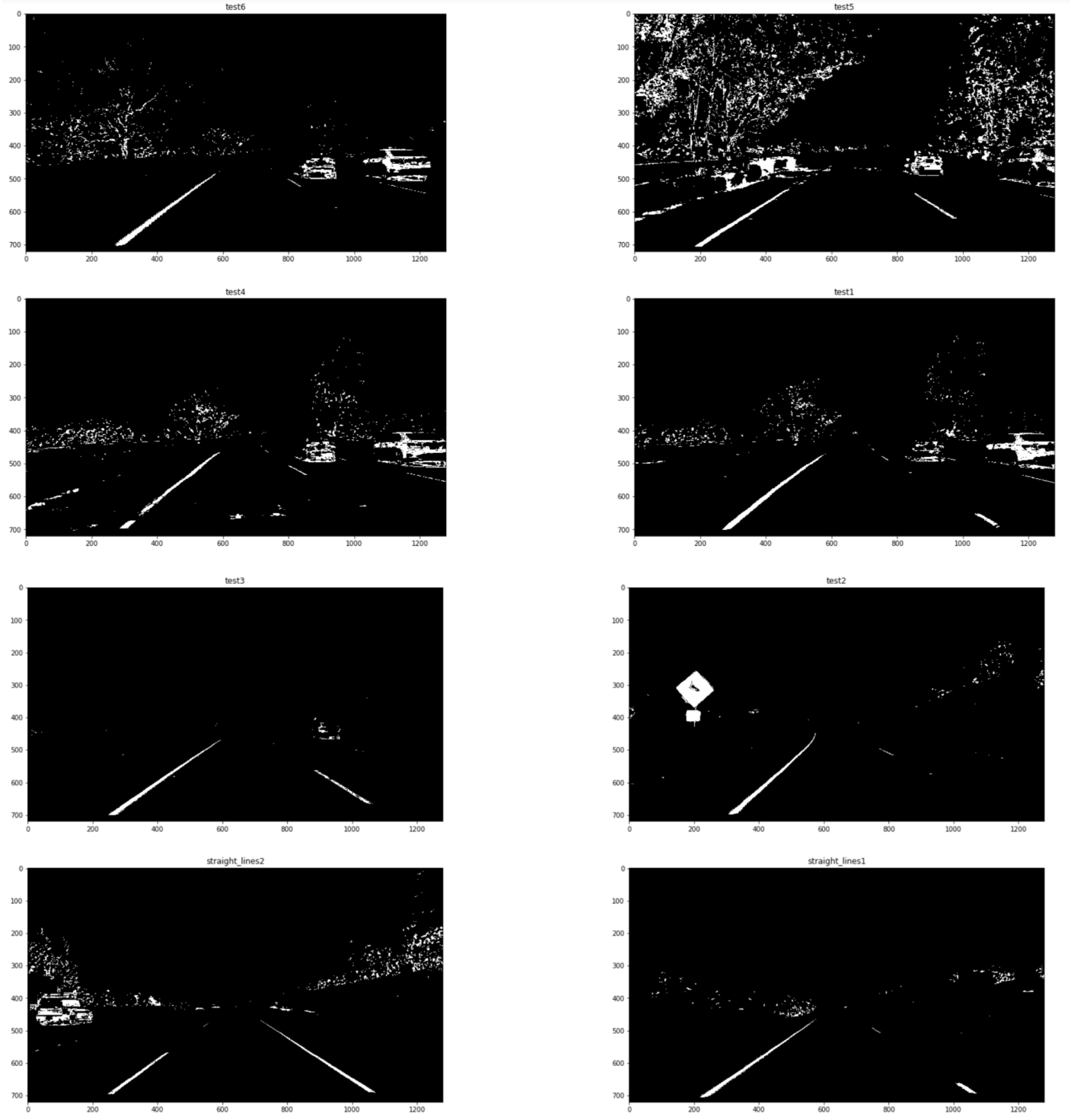




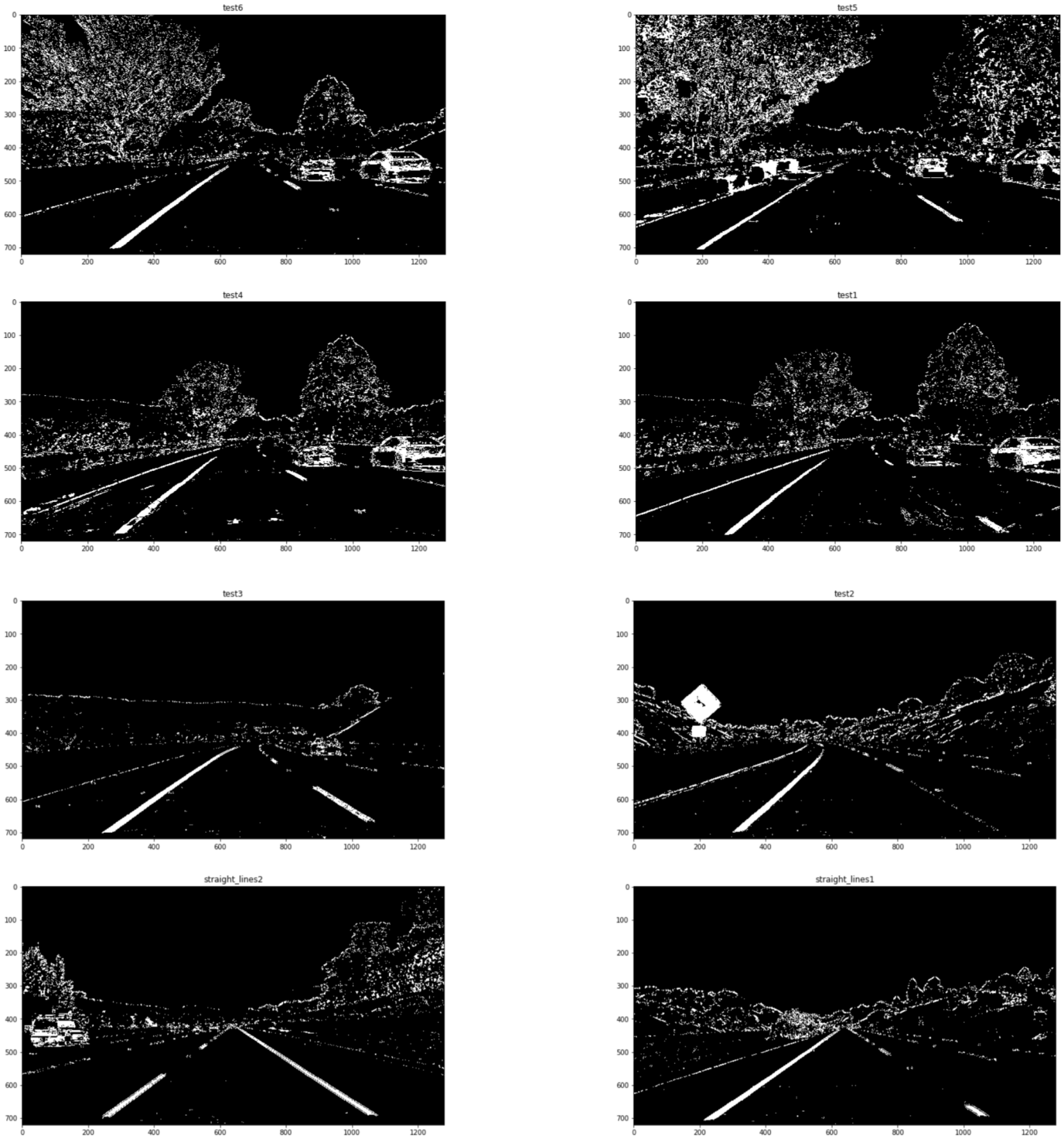




Then I used HLS color space to detect lane lines, which is also a very effective method. I first obtained HLS channels using `cv2.cvtColor(img, cv2.COLOR_RGB2HLS)`. Then I kept only the pixels where the `s` channel is within the threshold: `binary_output[(s_channel > thresh[0]) & (s_channel <= thresh[1])] = 1`. My final threshold is set to be (160,255). The result looked like this:



Finally, I combined gradients and color space methods together using the | operator: `combined[(grad_combined == 1) | (hls_binary == 1)] = 1`. The results look like this:



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

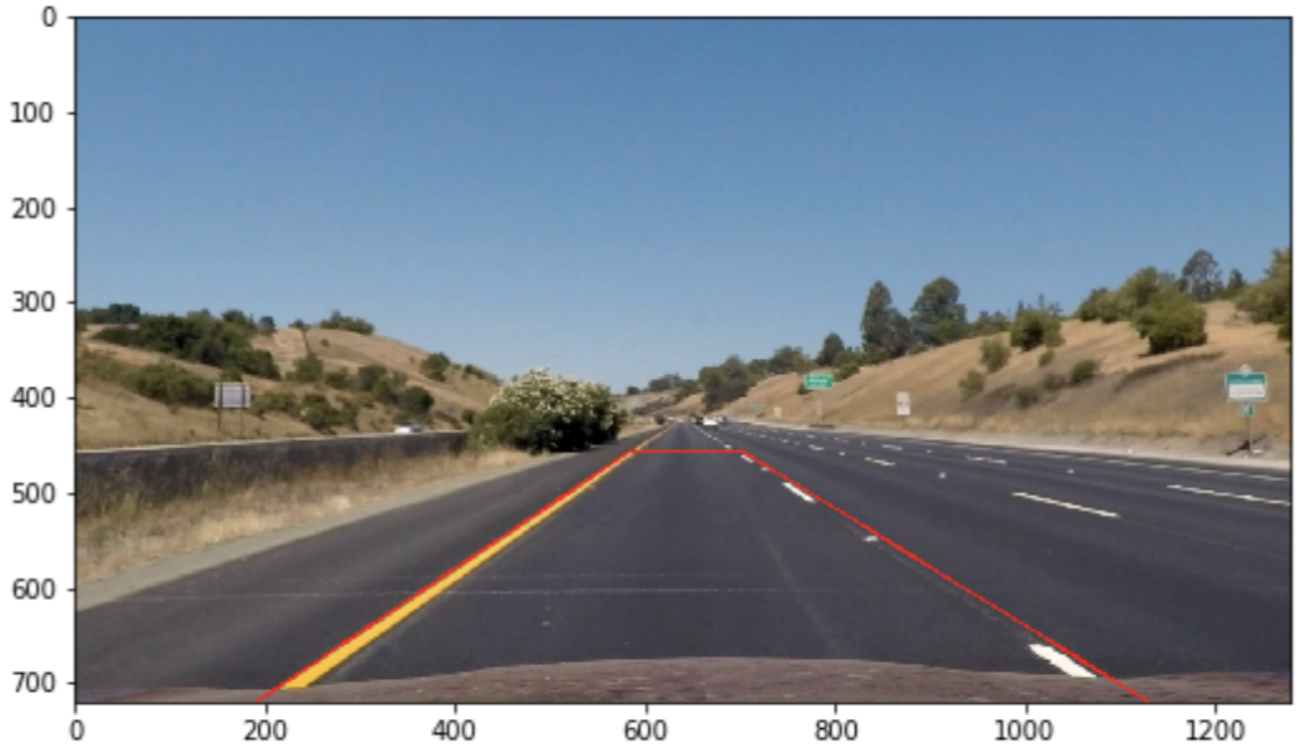
The code for my perspective transform appear in 25th to 28th cells in `p4.ipnb`.

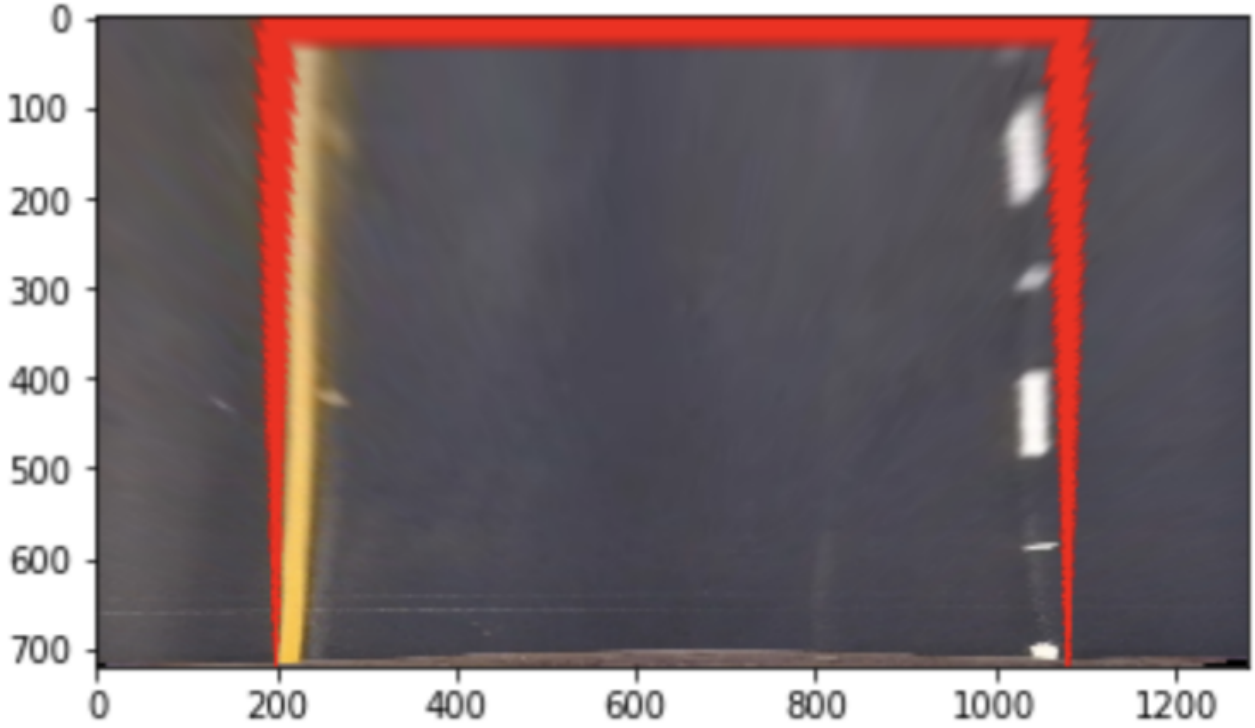
```
lb = [190, 720]
lt = [585, 455]
rt = [700, 455]
rb = [1130, 720]
src = np.float32([lt,rt,rb,lb])

gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
nx = gray.shape[1]
ny = gray.shape[0]
offset = 200
dst = np.float32([
    [offset, 0],
    [nx-offset, 0],
    [nx-offset, ny],
    [offset, ny]
])
dst = np.float32([
    [320, 0],
    [960, 0],
    [960, 720],
    [320, 720]
])
```

1280 720 This resulted in the following source and destination points:

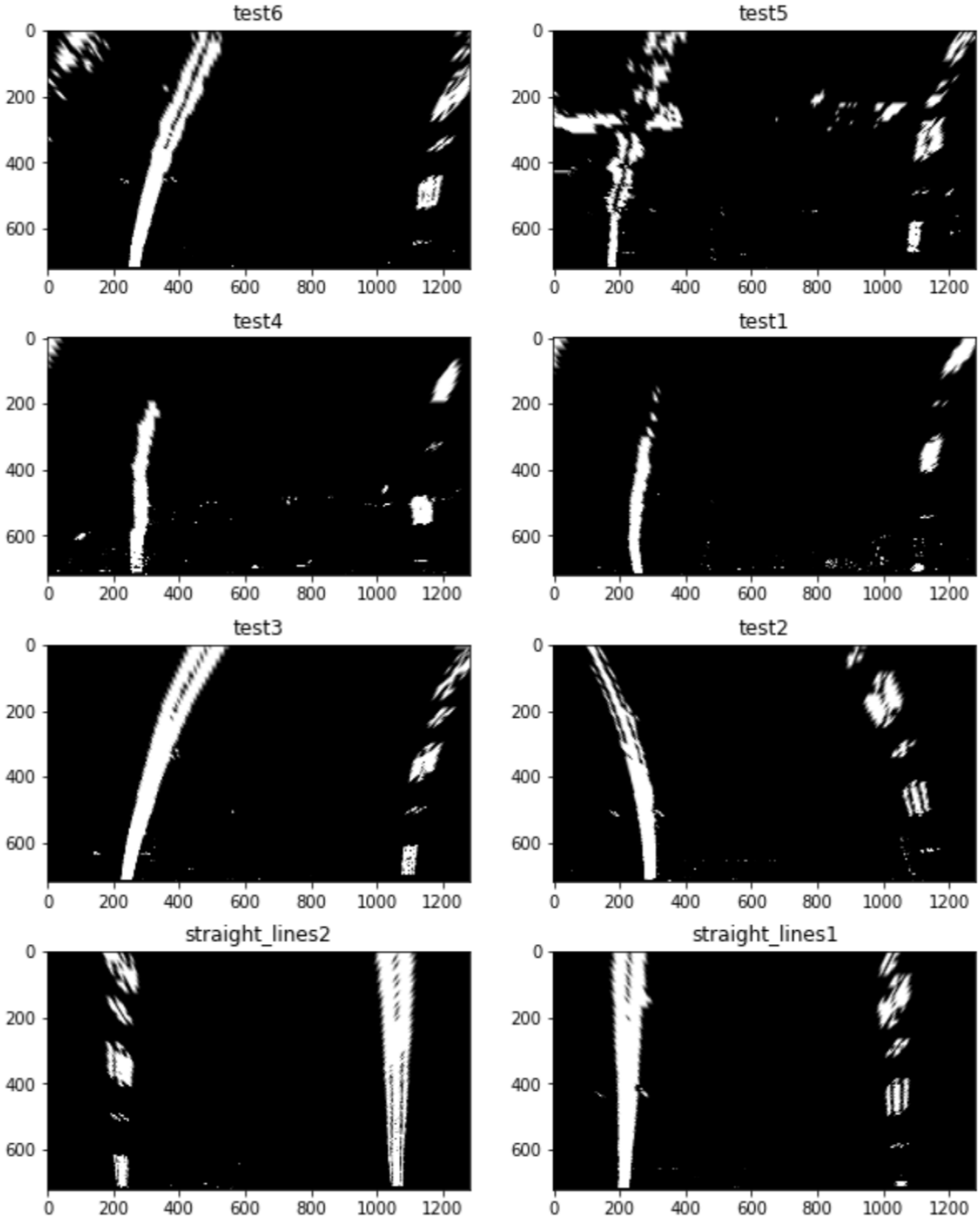
Source	Destination
585, 455	200, 0
190, 720	200, 720
1130, 720	1080, 720
585, 455	1080, 0

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. Also, this perspective transform obtains resonable radius of curvature values later.



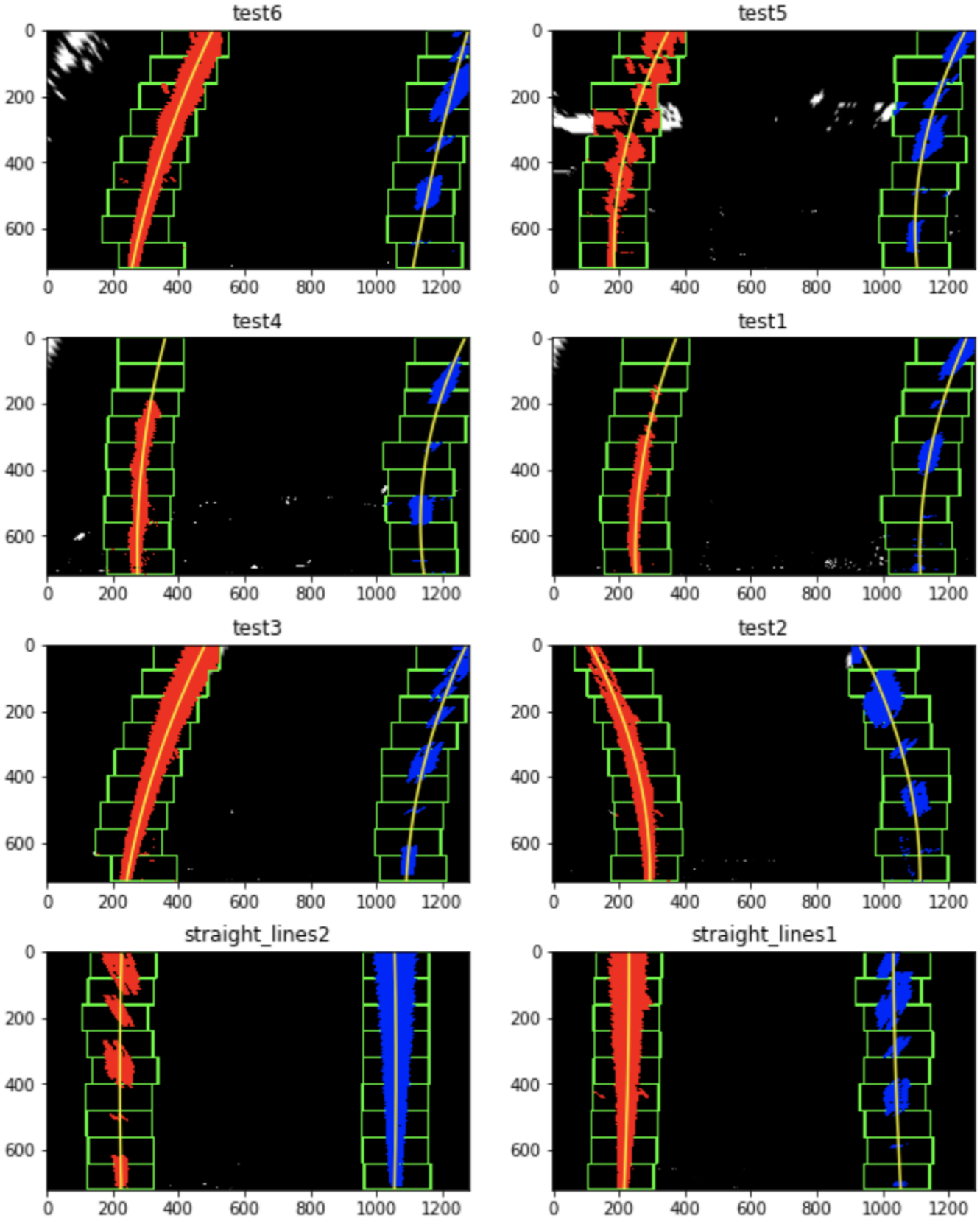


Then I applied perspective transform on all test images:

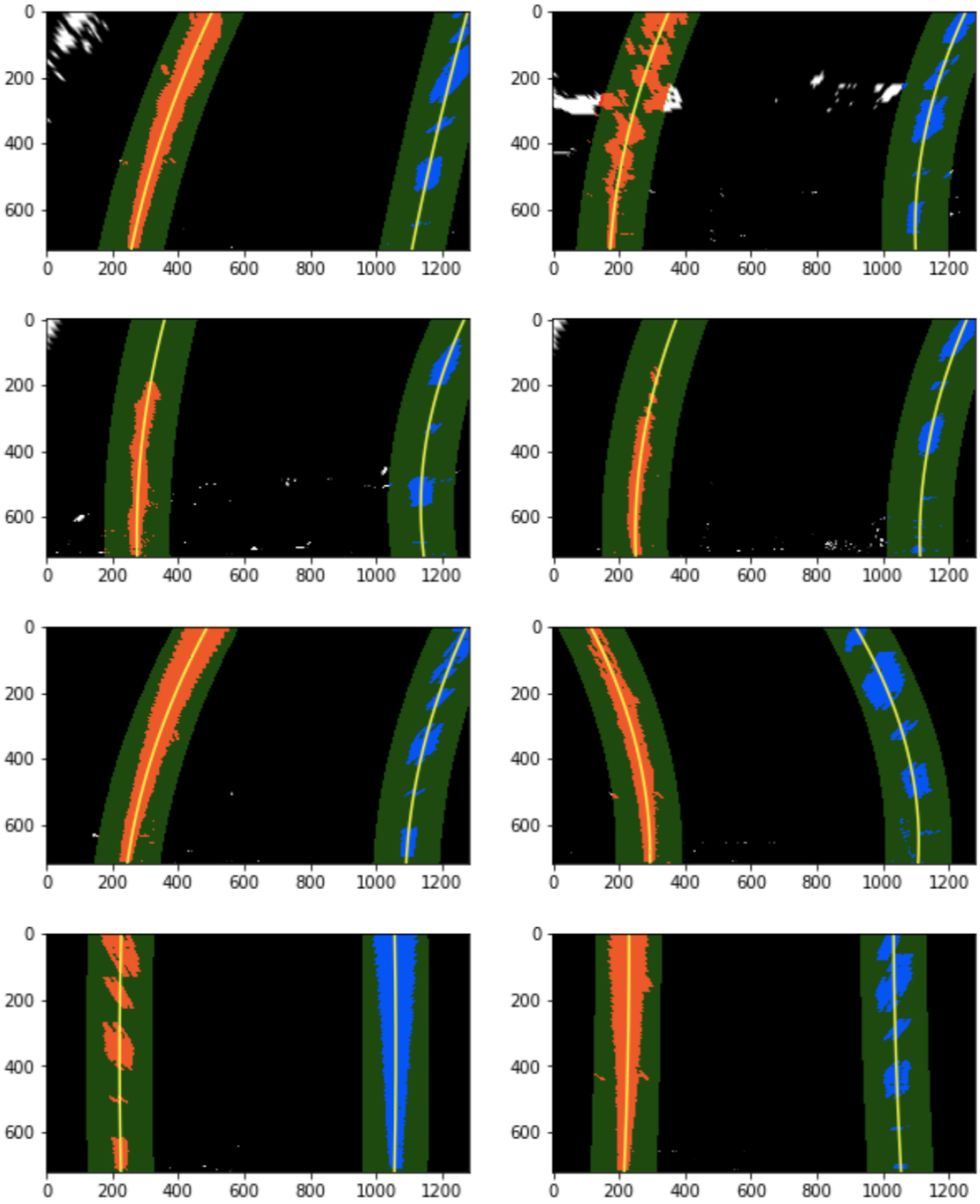


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

Then I applied the code in the lecture `Finding the Lines` to find lines in the warped binary images. I first took the lower half of the images, summed pixels values vertically to obtained left and right cluster points as the searching bases. Then I searched pixels with value equals 1 (white pixels) upward, both from the left base and the right base. I used 9 windows and set the width of the windows +/- `margin = 100` and the minimum number of pixels found to recenter window `minpix = 50`. Then I fit a 2nd order polynomial both for the left and right white pixels. The precess can be illustrated by the following image:



I also applied the method in the case I know where previous lane lines are, so I only need to search white pixels within a margin around the previous lane lines, illustrated in the follwing image:

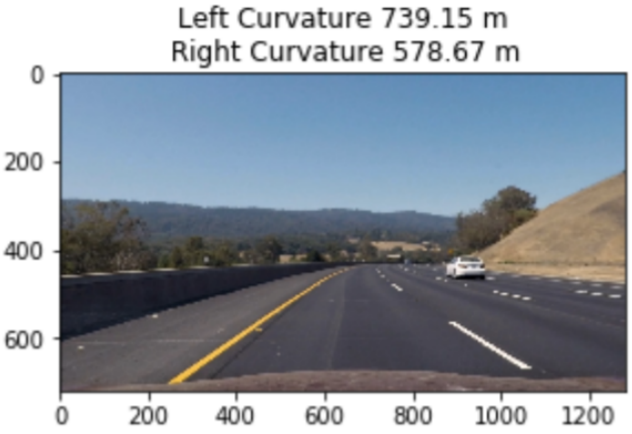

0
200
400
600
800
1000
1200

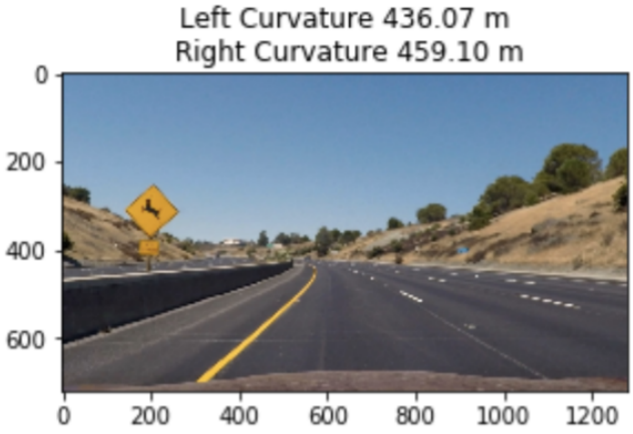
5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

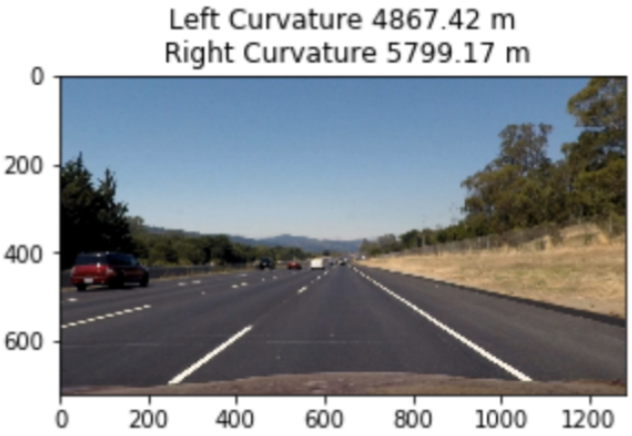
I did this in lines # through # in my 50th code cell in `p4.ipnb`, by applied just a math formula using the fit parameter I got from the previous step, where `ym_per_pix` is meters per pixel in y dimension.

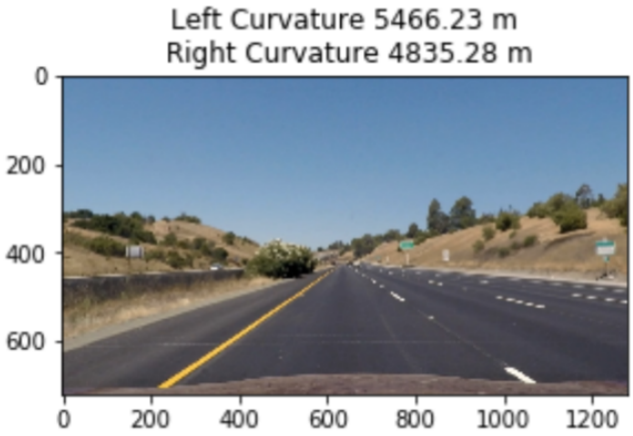
```
((1 + (2*fit[0]*719*ym_per_pix + fit[1])**2)**1.5) / np.absolute(2*fit[0])
```

Then I computed the left and write radius of curvature of all 8 test images:


Left Curvature 739.15 m
Right Curvature 578.67 m
0
200
400
600
0
200
400
600
800
1000
1200

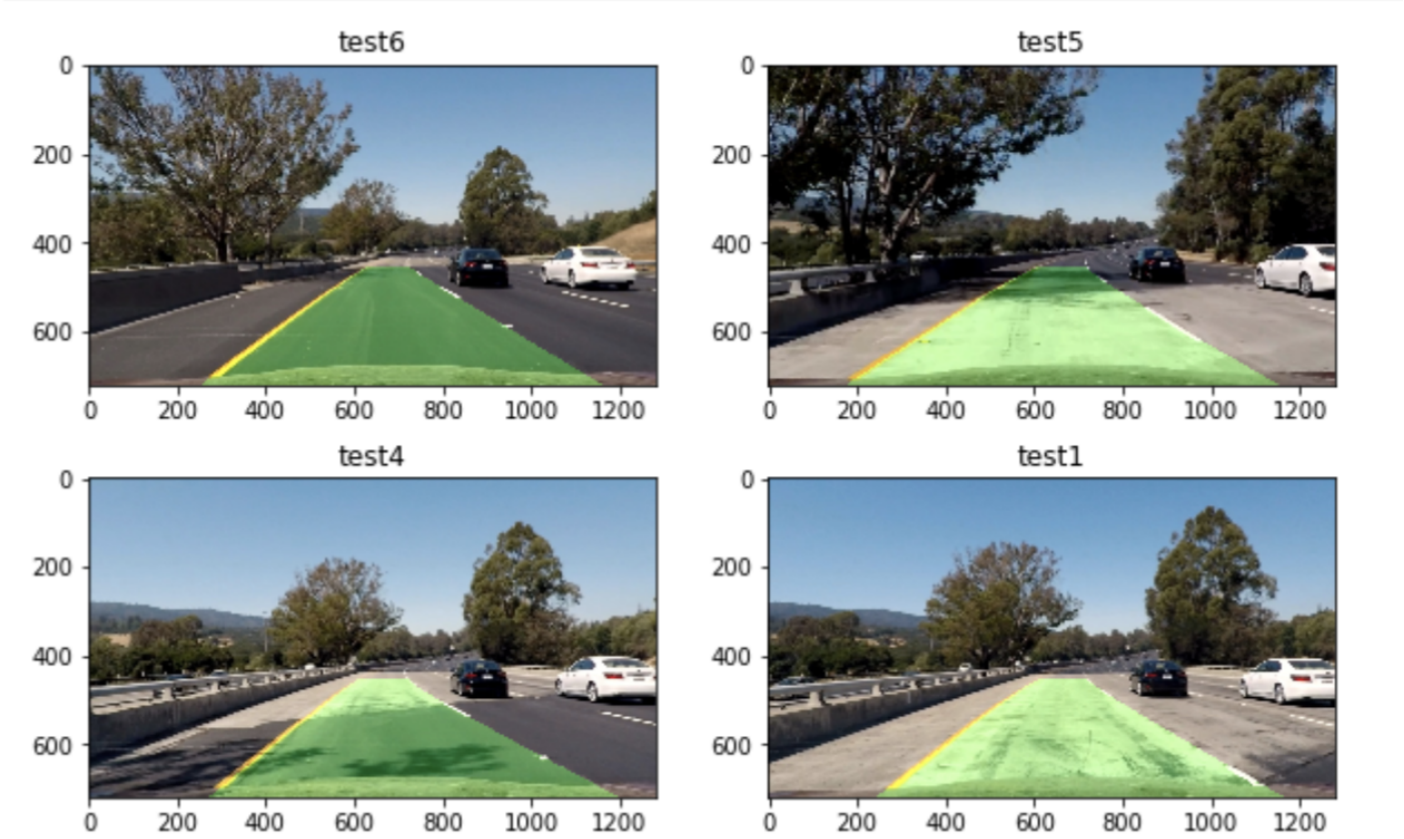

Left Curvature 436.07 m
Right Curvature 459.10 m
0
200
400
600
0
200
400
600
800
1000
1200

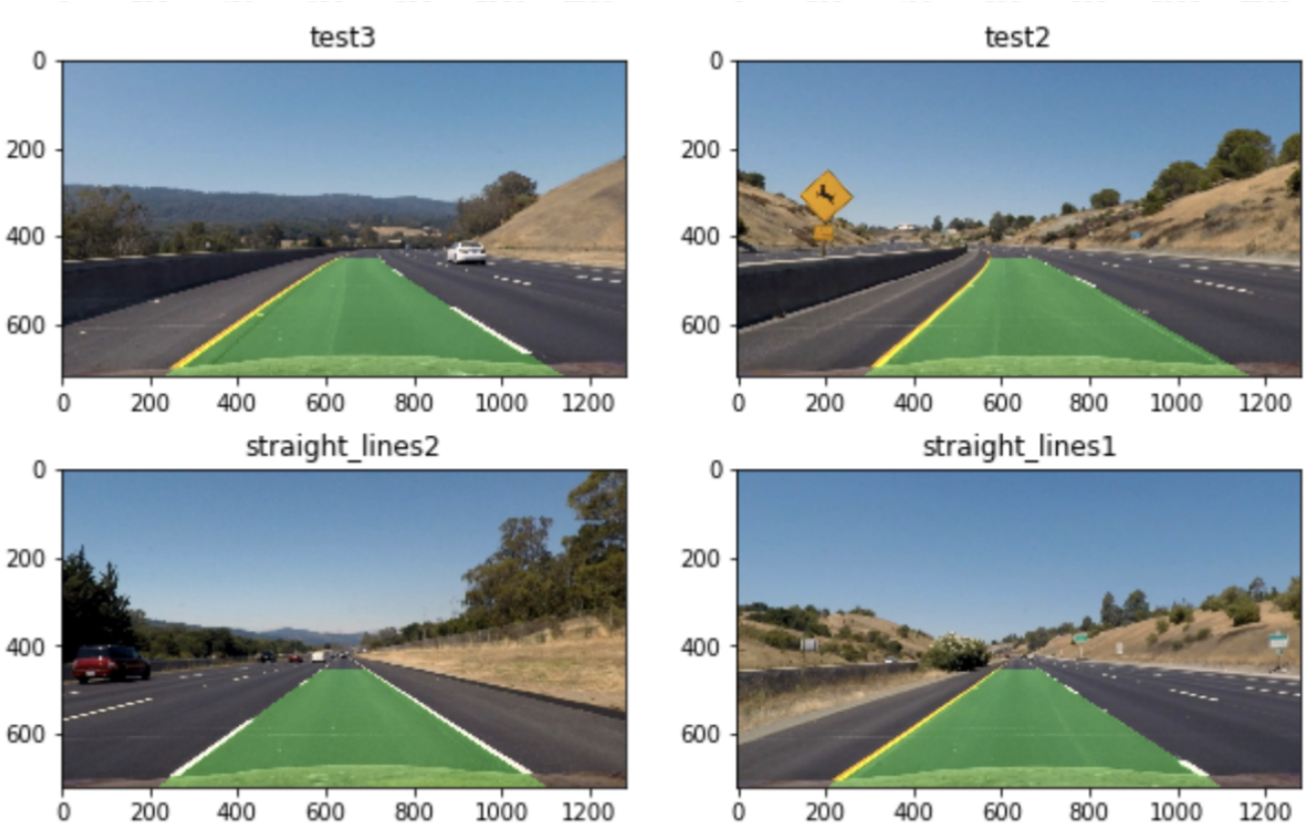

Left Curvature 4867.42 m
Right Curvature 5799.17 m
0
200
400
600
0
200
400
600
800
1000
1200


Left Curvature 5466.23 m
Right Curvature 4835.28 m
0
200
400
600
0
200
400
600
800
1000
1200

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 35th cell in `p4.ipnb`. I basically used the code in the lecture, mapped the warped lines back to the original image using `cv2.warpPerspective` and the inverse martix `Minv`. Here is the result on test images:



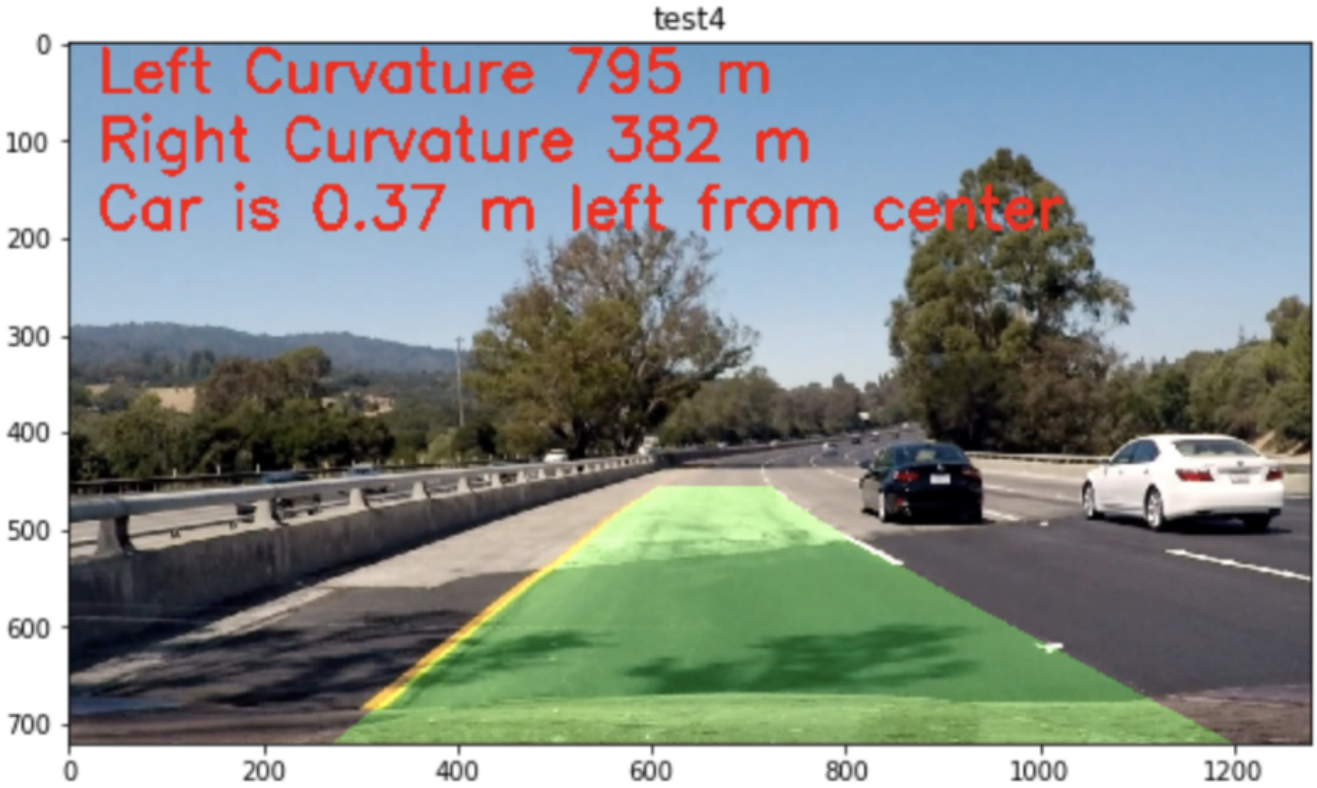


7. Computed distance of car (center of image) to center of lane lines

I assume that the camera is mounted at the center of the car so that `carCenter = xlen / 2`. Then I computed laneCenter this way:

```
leftLane = left_fit_m[0]*ylen**2 + left_fit_m[1]*ylen + left_fit_m[2]
rightLane = right_fit_m[0]*ylen**2 + right_fit_m[1]*ylen + right_fit_m[2]
laneCenter = (rightLane - leftLane)/2 + leftLane
```

Finally, the distance is just `diff = laneCenter - carCenter`. Then I displayed the message on images, indicating if the car is on the left or right and how far it is from the center of the lane.



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!).

My final video output is called `video_output.mp4` in the zip file.

In the video pipeline, the lane lines always looked very nice. But I found that the curvature sometimes got very huge, with is not very resonable according the U.S. government specifications for highway curvature. (http://onlinemanuals.txdot.gov/txdotmanuals/rdw/horizontal_alignment.htm#BGBHGEGC) So I decided to use a `Lane` object to keep track of the current curvature and fit parameters. When the curvature computed in the current image is too large (larger then 5500 meters), I would use the previous curvature. This made the result looked more stable.