Advanced Lane Finding

The goal of this project 4 of CarND course is to detect the road lane lines on the image and videos. A number of test images and 1 test video (and 1 challenge) are provided to test the final pipeline.

The project is done at 2 main steps with several sub-steps:

I. Detect the lines on the images,

- 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.

II. Extend the pipeline developed for lane detection to videos

Related Files:

- P4_AdvancedLaneFinding.ipynb
- project_video_out.mp4
- This writeup
- output_images

Note: the notebook P4_AdvancedLaneFinding.ipynb includes all the codes and functions for this project; some of them are described in this report.

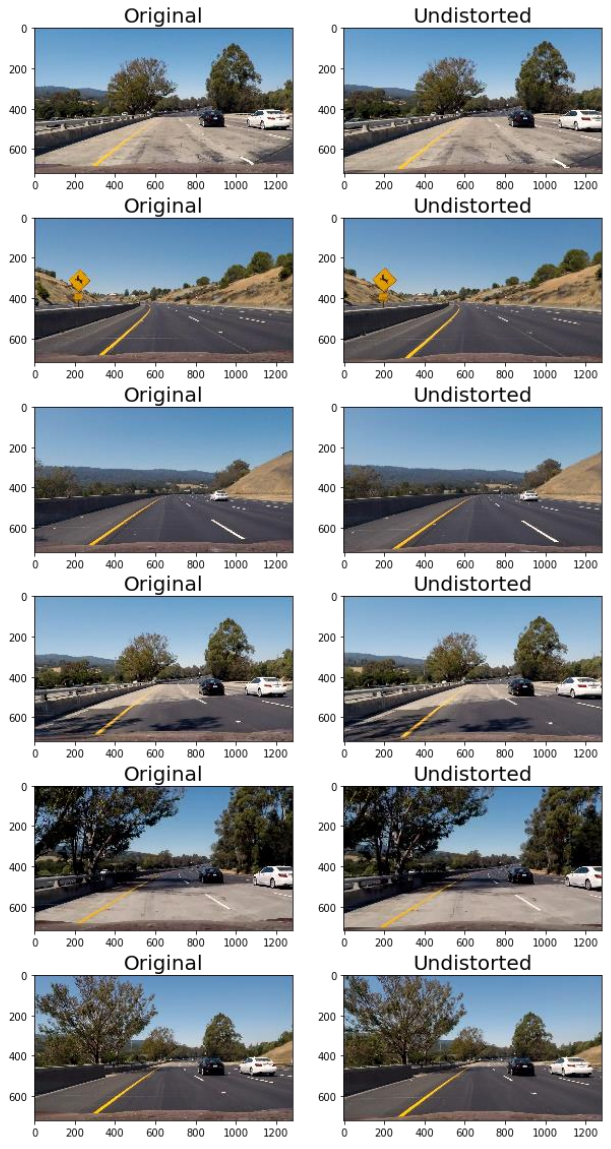
Camera Calibration

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 first code cell of the IPython notebook located in "project_video_out.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.

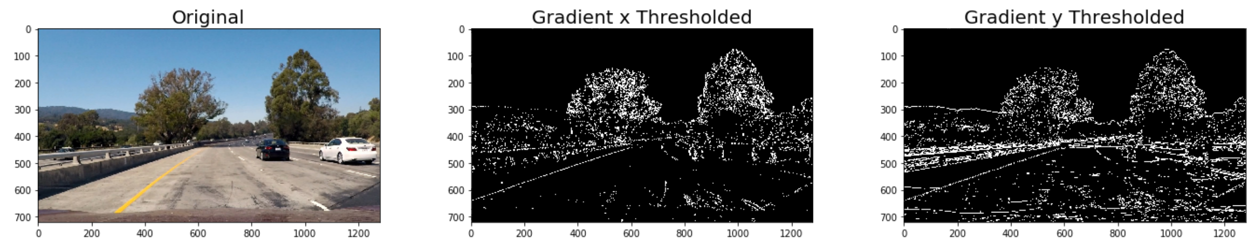
I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

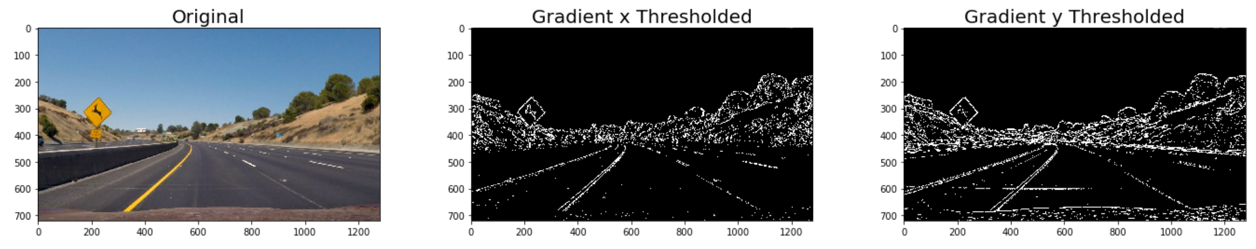


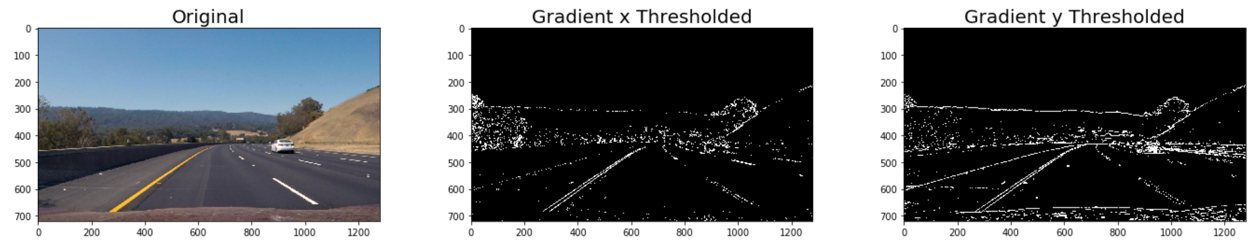
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

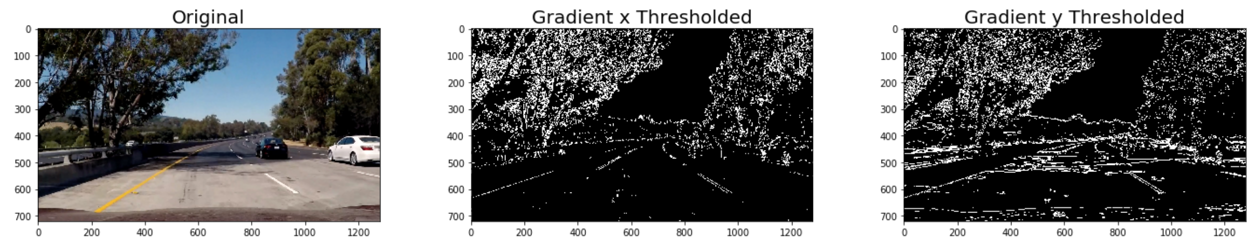
I used a combination of color and gradient thresholds to generate a binary image (thresholding is done at functions abs_sobel_thresh, mag_thresh, dir_threshold and hls_select). Here's an example of my output for each function:

X and y Gradient Thresholding (abs_sobel_thresh):

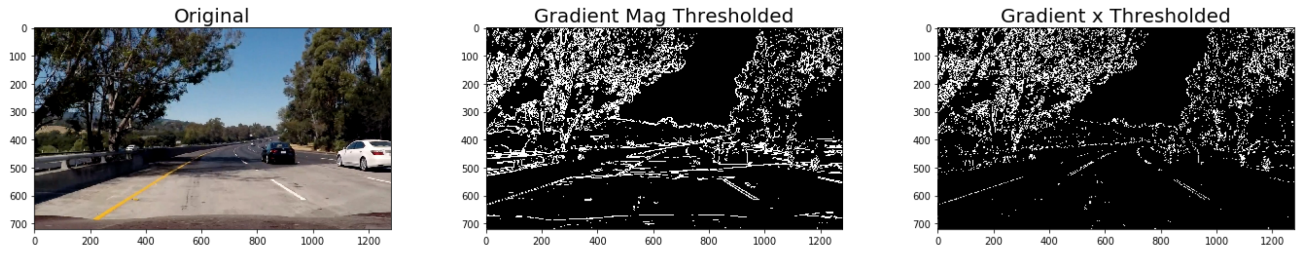


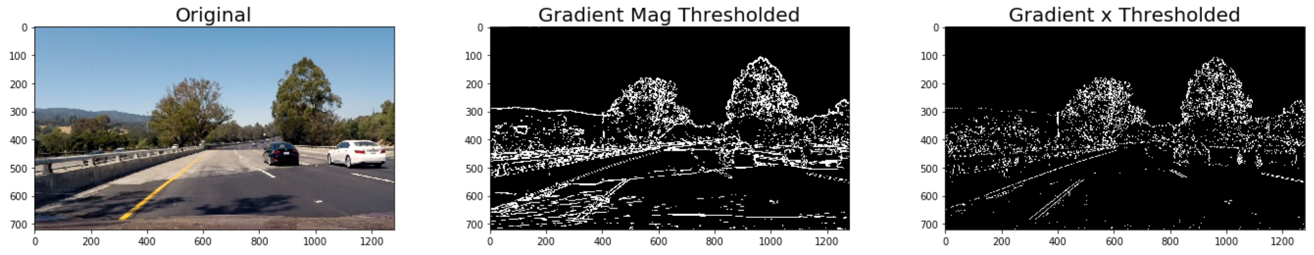


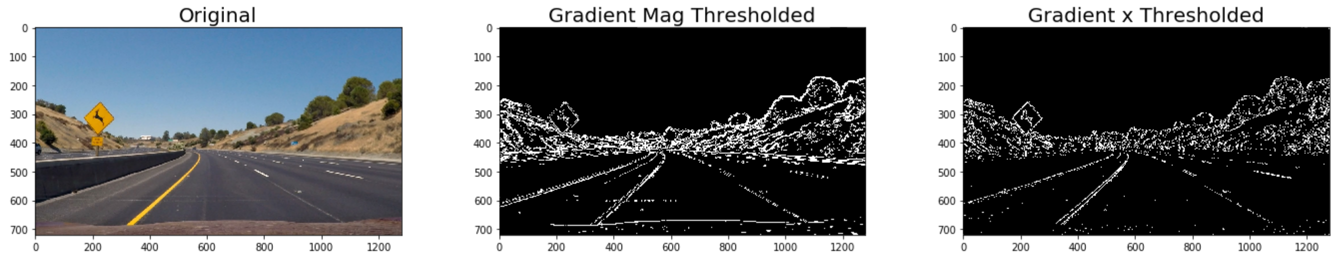


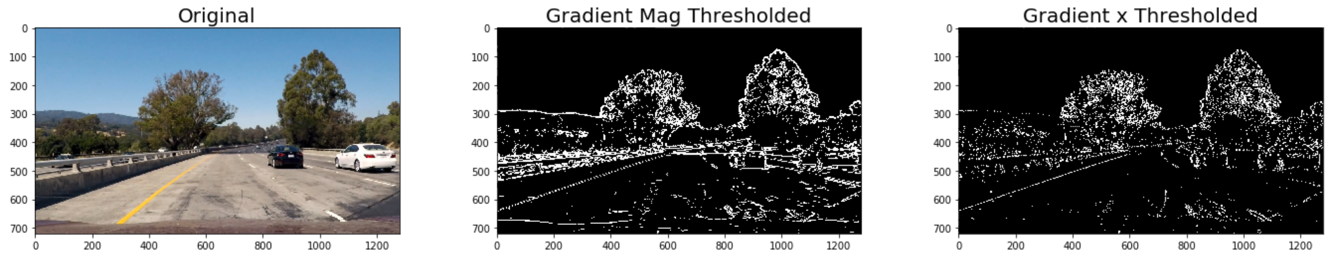


Gradient Magnitude Thresholding (mag_threshold):

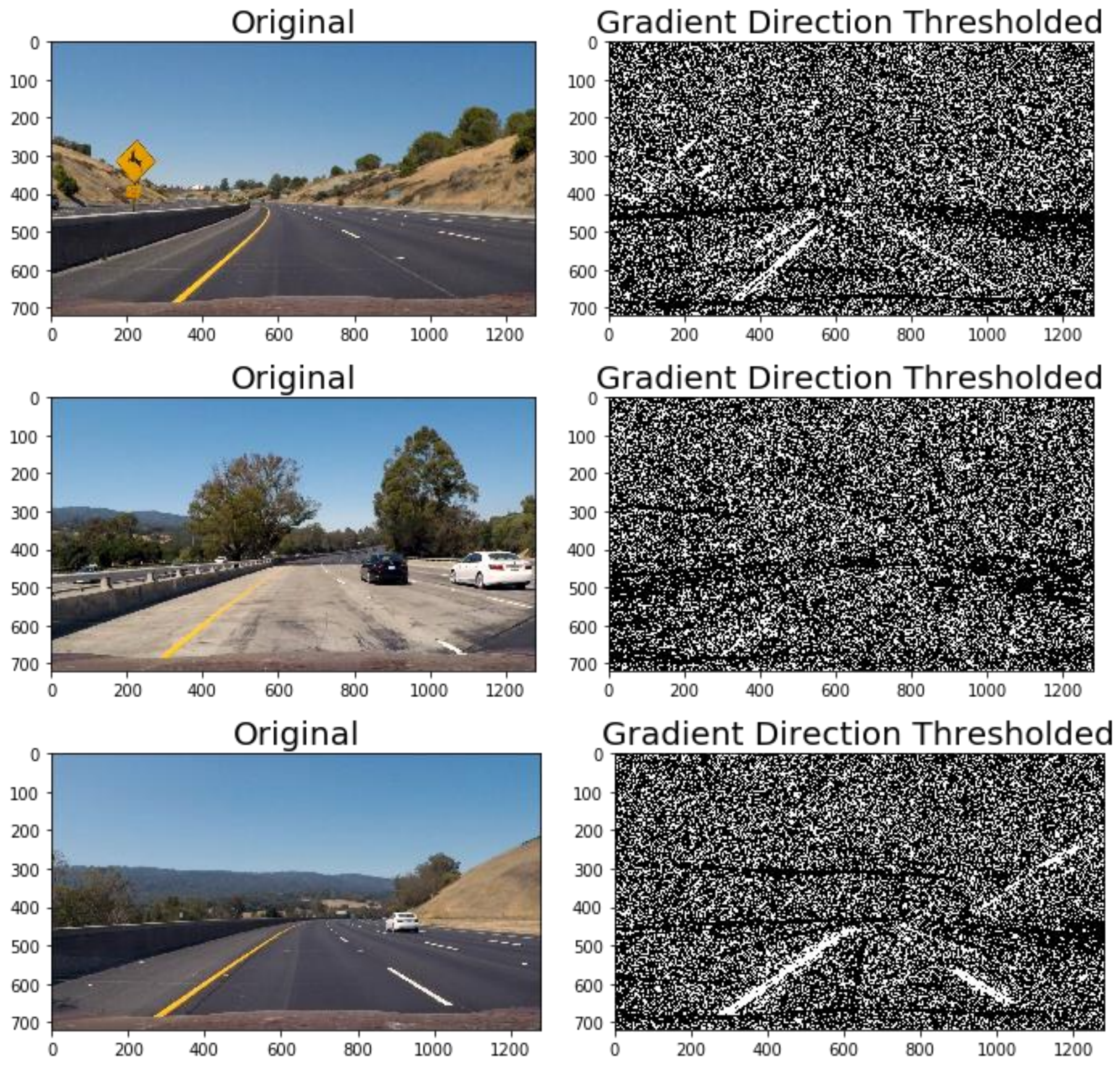




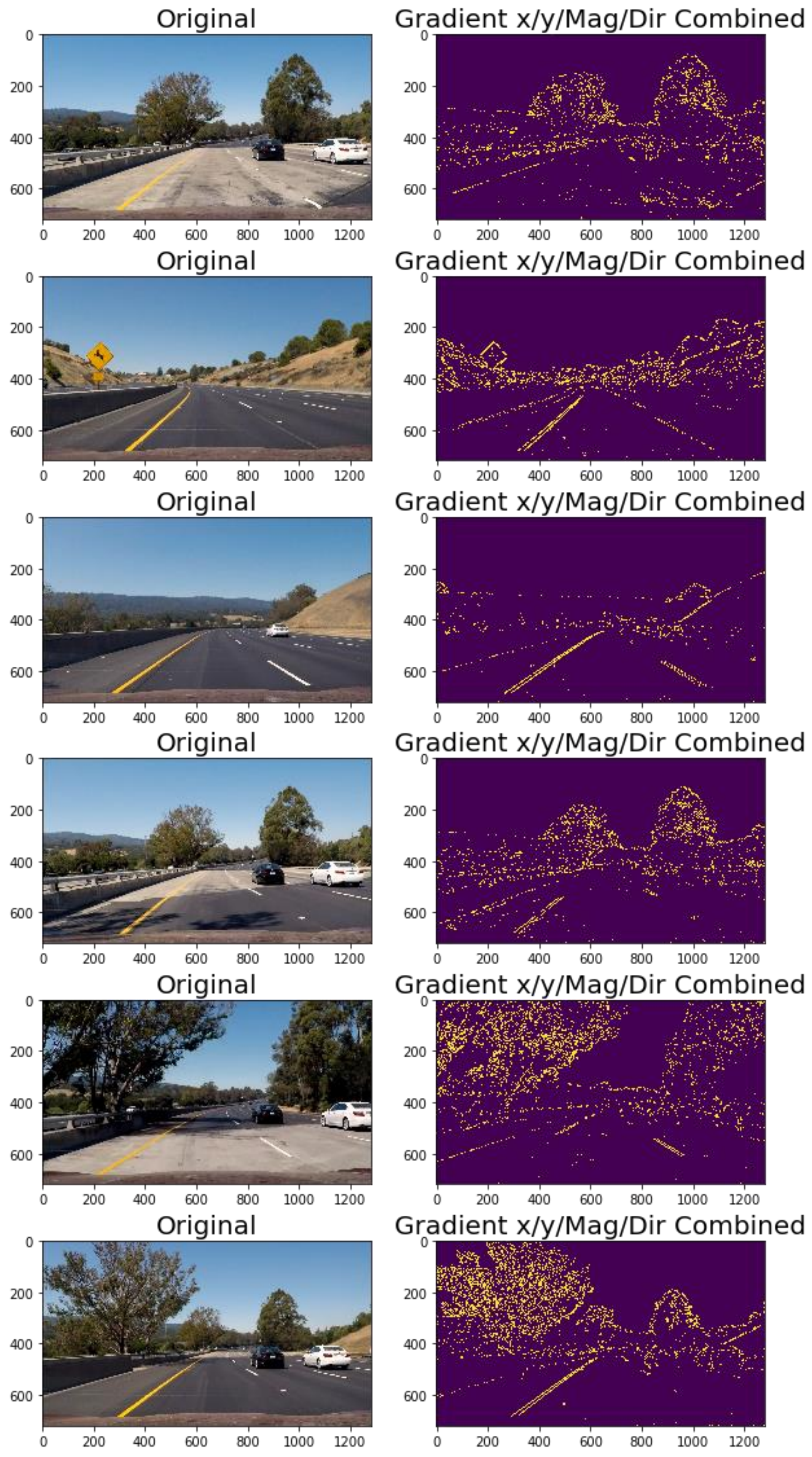




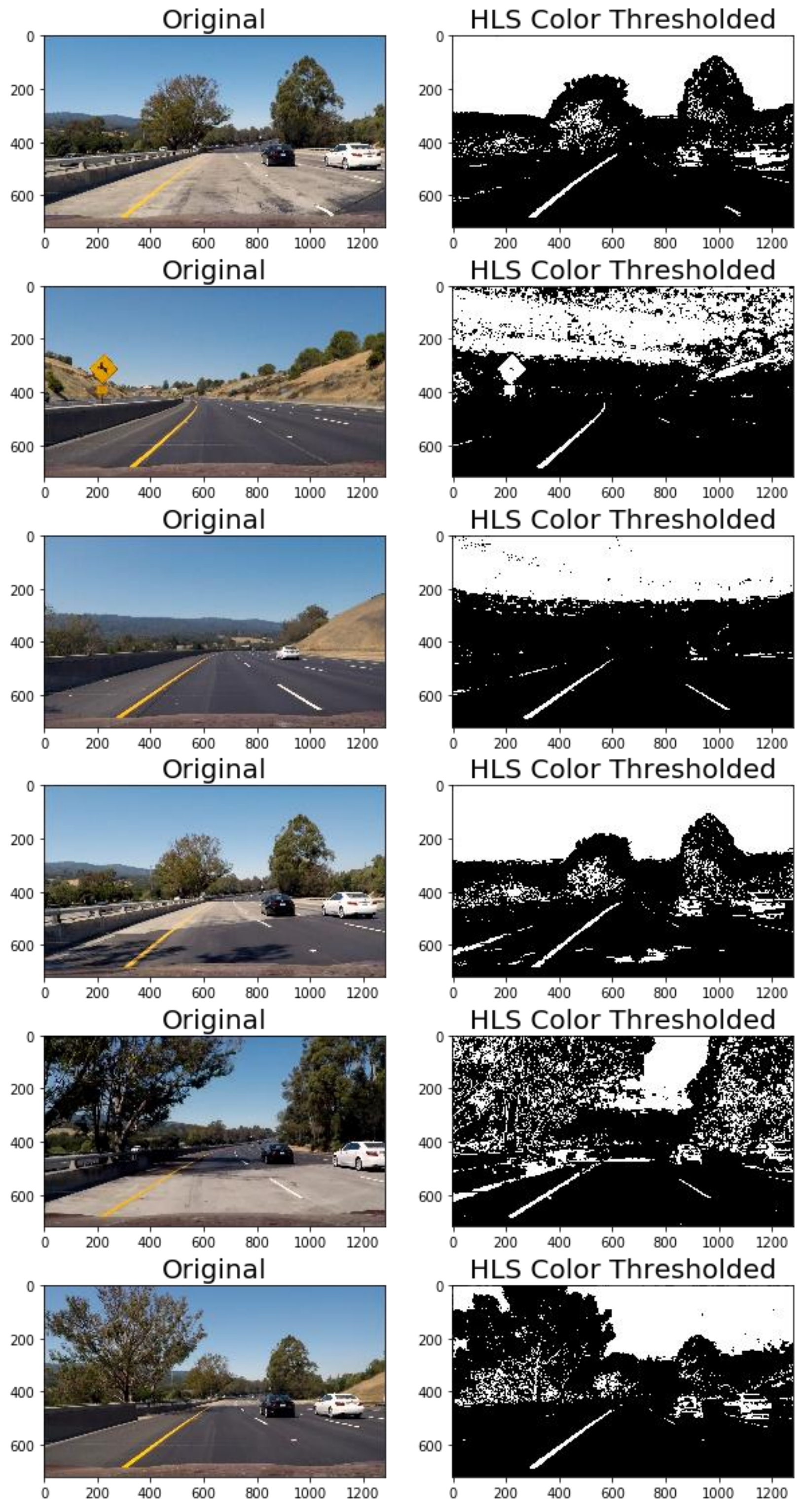
Direction Thresholding (dir_threshold)


Original
Gradient Direction Thresholded
Original
Gradient Direction Thresholded
Original
Gradient Direction Thresholded

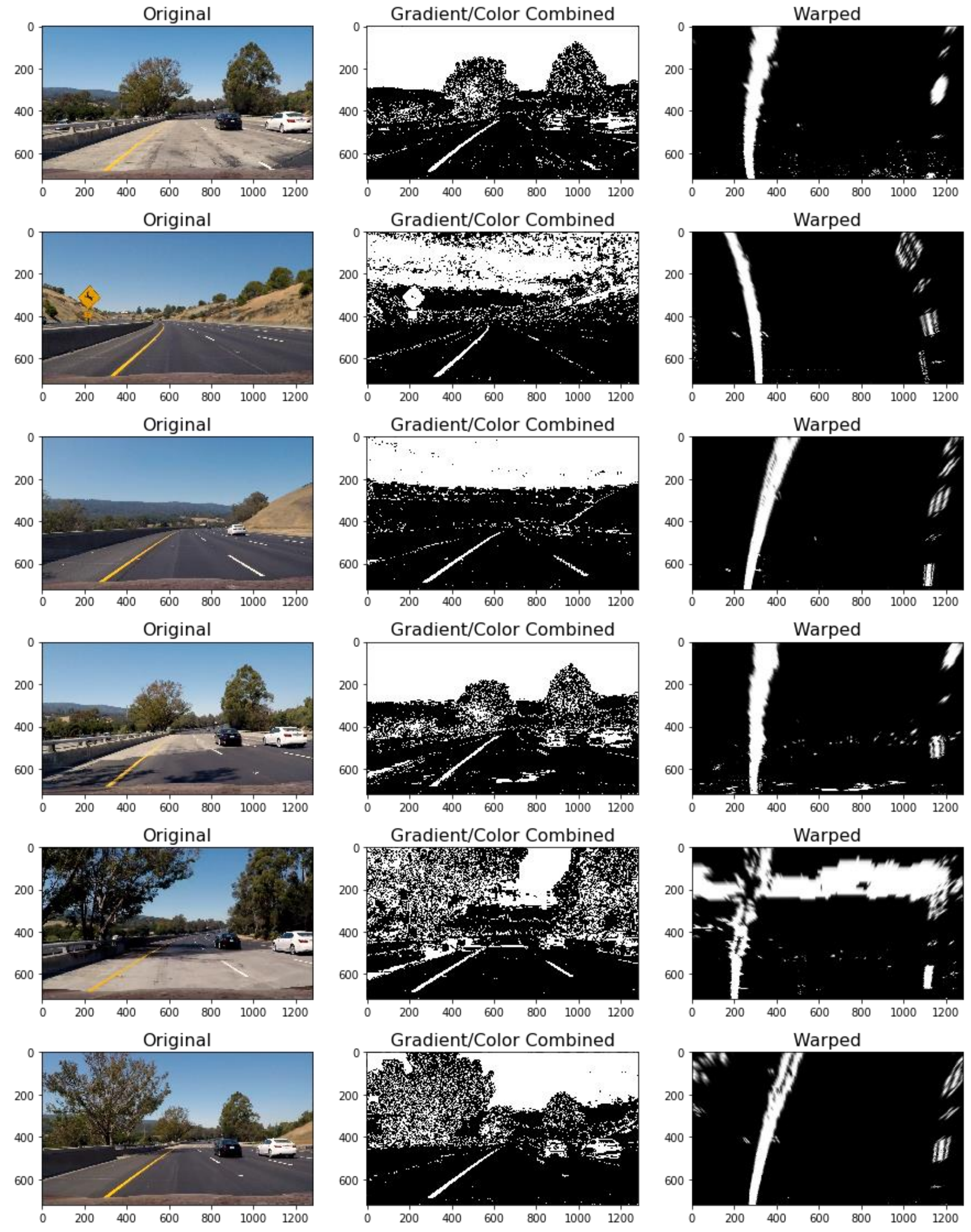
Combining all these gradient thresholdings we get the output below:


Original
Gradient x/y/Mag/Dir Combined
Original
Gradient x/y/Mag/Dir Combined
Original
Gradient x/y/Mag/Dir Combined
Original
Gradient x/y/Mag/Dir Combined
Original
Gradient x/y/Mag/Dir Combined
Original
Gradient x/y/Mag/Dir Combined
0
200
400
600
800
1000
1200

In the next step Color thresholding is performed, the S channel is thresholded in the function hls_select(). The output of S channel thresholding looks like this:

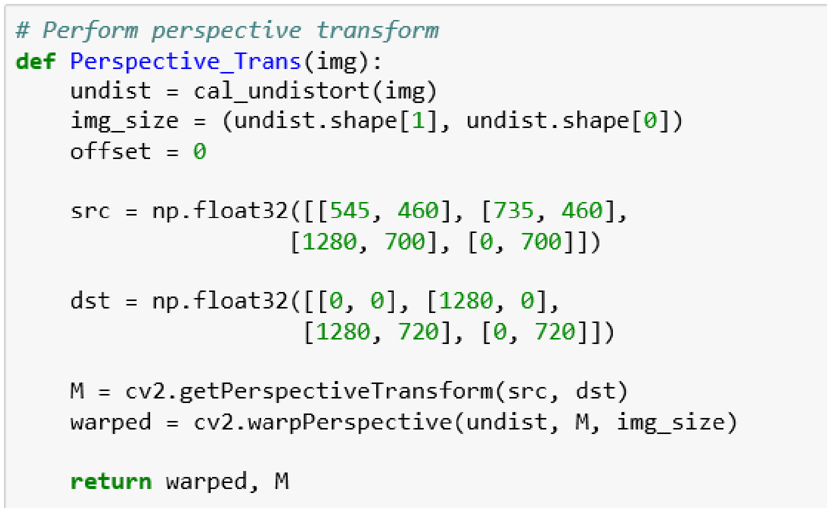


In the next step we combine the results of gradient and color thresholding, the result looks like this (bird's eye view is also shown to evaluate the performance):



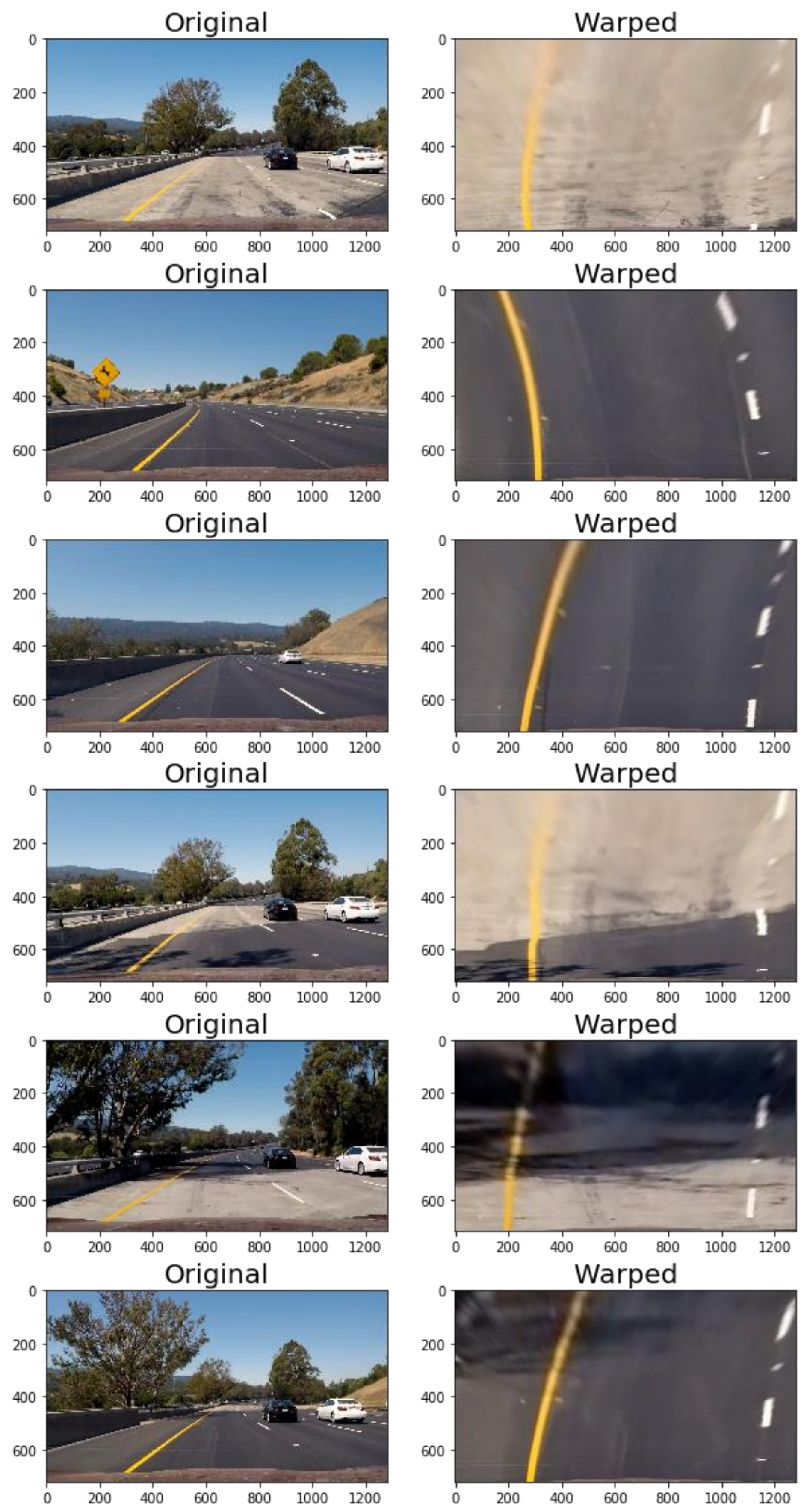
3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

Then the perspective is transformed to get a bird's eye view of the road in a function called `Perspective_Trans()`. To perform this transformation we can imagine a transformation from dash camera view (Trapezoid) to the bird's eye view (rectangle); the camera view is captured in src and the bird's eye view is represented by dst arrays. The `Perspective_Trans()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose the hardcode the source and destination points in the following manner:



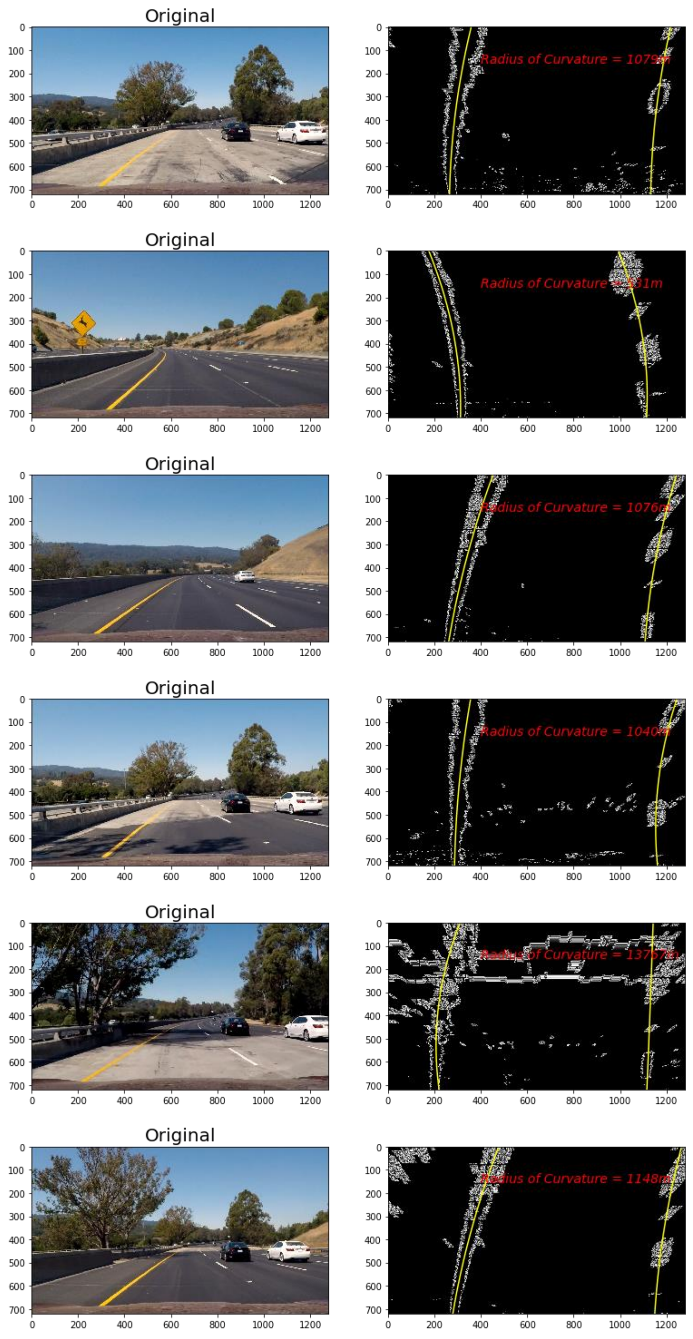
```
# Perform perspective transform
def Perspective_Trans(img):
    undist = cal_undistort(img)
    img_size = (undist.shape[1], undist.shape[0])
    offset = 0

    src = np.float32([[545, 460], [735, 460],
                    [1280, 700], [0, 700]])

    dst = np.float32([[0, 0], [1280, 0],
                     [1280, 720], [0, 720]])

    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(undist, M, img_size)

    return warped, M
```

In the next step I performed perspective transformation in function `Perspective_Trans` to get a bird’s eye view of the lanes, below is the sample results:


Original
Warped
Original
Warped
Original
Warped
Original
Warped
Original
Warped
Original
Warped

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

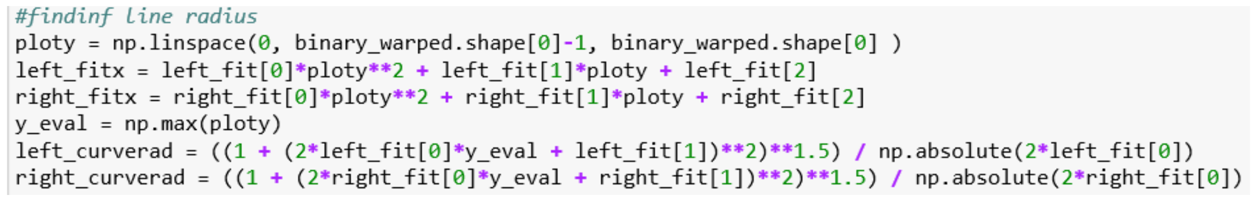
Then in function fit_poly_lane() the pixels of lines are detected and 2nd order polynomial is fit to those pixels. Also, the radius of curvature are calculated from the polynomial coefficient and the translated to real world scale knowing the pixel to actual map scale, the result looks like this:



As we see in the 4^{th} image the lane line is not detected properly.

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.

The line radius is detected in function fit_poly_lane() using the polynomial coefficients in the code below:

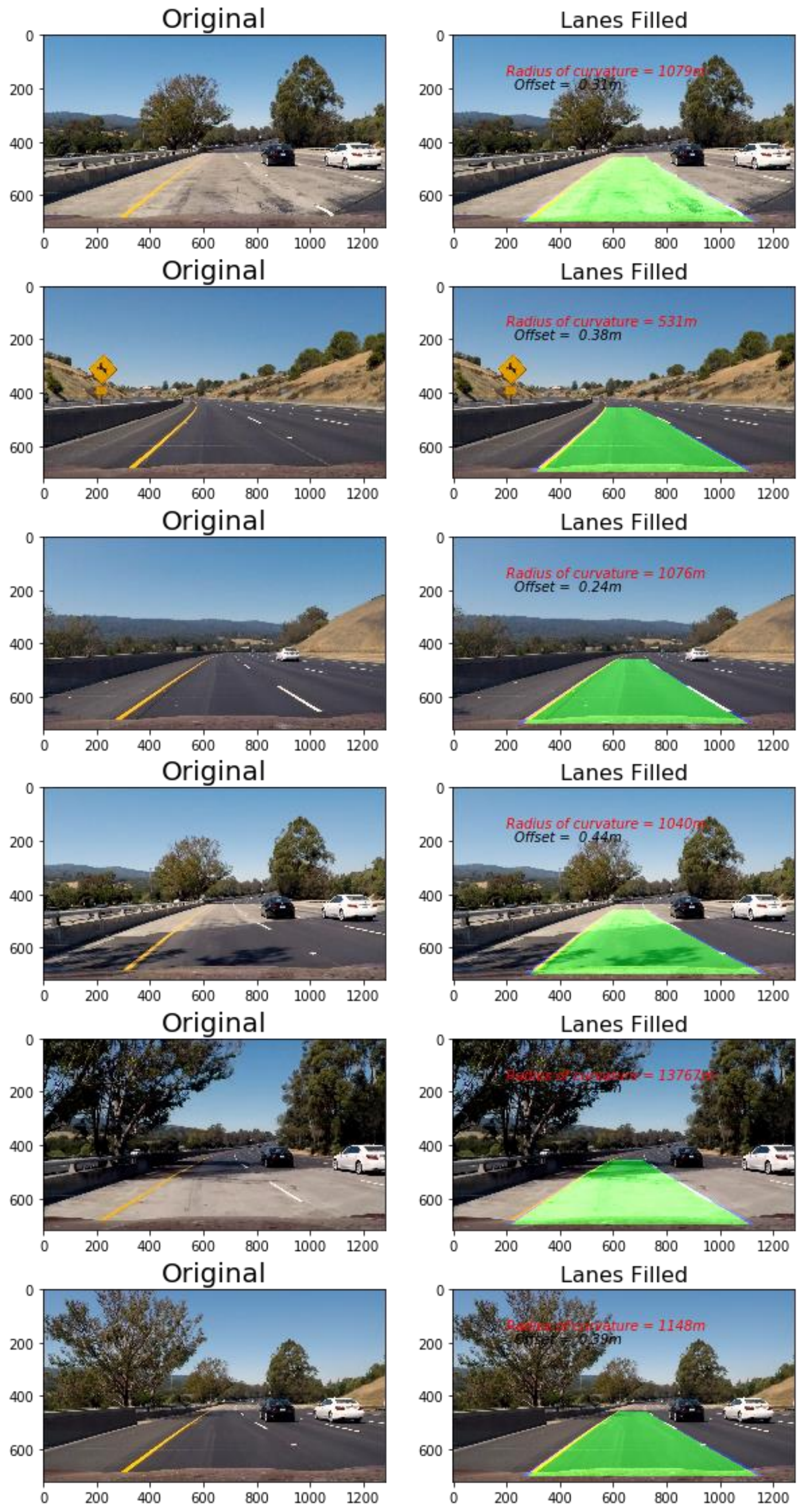


```
#findinf line radius
ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )
left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]
y_eval = np.max(ploty)
left_curverad = ((1 + (2*left_fit[0]*y_eval + left_fit[1])**2)**1.5) / np.absolute(2*left_fit[0])
right_curverad = ((1 + (2*right_fit[0]*y_eval + right_fit[1])**2)**1.5) / np.absolute(2*right_fit[0])
```

Then it's converted to real world scale. This function can be found in the project Jupyter file.

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 lane_filler function. Here is an example of my result on the test images:


Original
Lanes Filled
Radius of curvature = 1079m
Offset = 0.31m
Original
Lanes Filled
Radius of curvature = 531m
Offset = 0.38m
Original
Lanes Filled
Radius of curvature = 1076m
Offset = 0.24m
Original
Lanes Filled
Radius of curvature = 1040m
Offset = 0.44m
Original
Lanes Filled
Radius of curvature = 13767m
Original
Lanes Filled
Radius of curvature = 1148m
Offset = 0.39m

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 function video_processor() is created to process each frame of the video, it's input is the video frame images and it outputs a lane filled image. The pipeline for this function starts with camera calibration and getting the undistorted image, then the image is passed through gradient and color thresholding and then it's warped. The warped image is passed to the fit_poly_lane() function to get the curvature of the lanes and get the filled lane image.

The video is stored in the repository with the name project_video_out.mp4.

Also, the function video_processor2() is created based on the reviewer suggestions to process each frame of the video. The main difference between this function and video_processor() is that it works based on the yellow and white channel thresholding. The result of this function is captured as project_video_out2.mp4.

Also the pipeline is run over the challenge video, the result is stored as harder_challenge_video_out.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?

I used almost all of the provided techniques in the course for lane detection, if I want to improve it more I would work a little on the lane radius detection to make it more stable and less oscillatory, it's better filtered using the fact that at lane lines change with a limited rate.

Also as the challenge video shows the pipeline loses it's accuracy at places where we have very shaded areas, I think it can be improved by looking at other color channels.

Here's also a comment I made in project 1:

To detect curvy lanes such as those presented in the challenge video we can use the polynomial functions rather than the lines...

which is similar to technique I used for this project.