Writeup report for Advanced Lane Finding project

Camera calibration

The pipeline for camera calibration is as follows:

1) Given a set of distorted images with chessboard views, find chessboard corners with cv2. findChessboardCorners() function. Note: for the most of given images (20 images total) cv2. findChessboardCorners() succeeded in finding 9x6 chessboard corners, but for 3 images it was unsuccessful, so these images were just excluded from further calibration coefficients calculation.
   Fig.1 and 2 show initial chessboard images and images with corners drawn where successfully found.

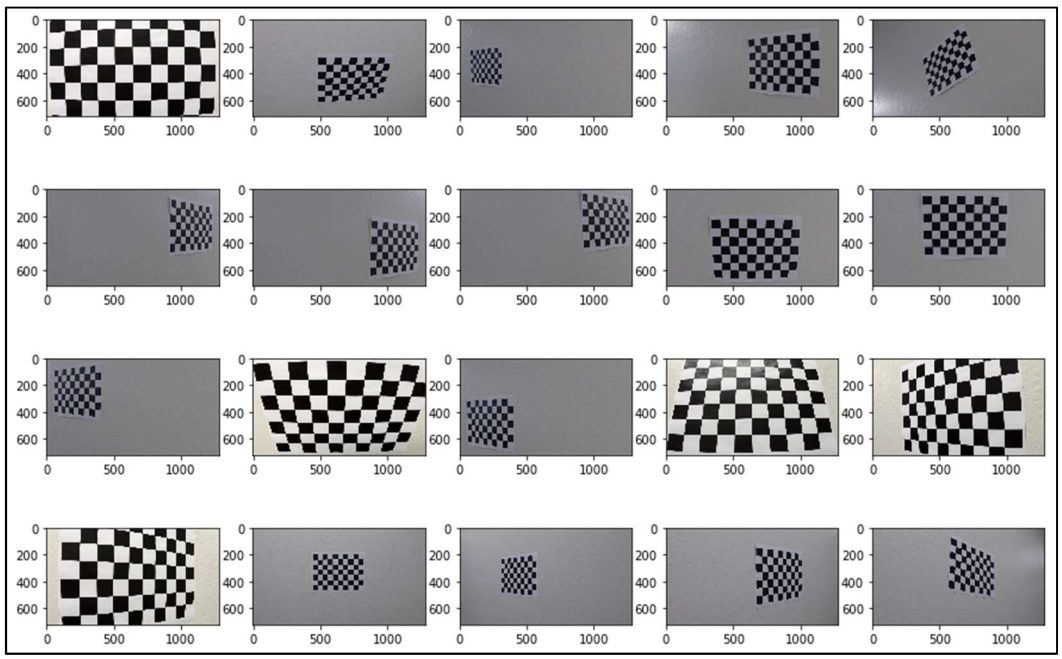


Fig.1. Chessboard images

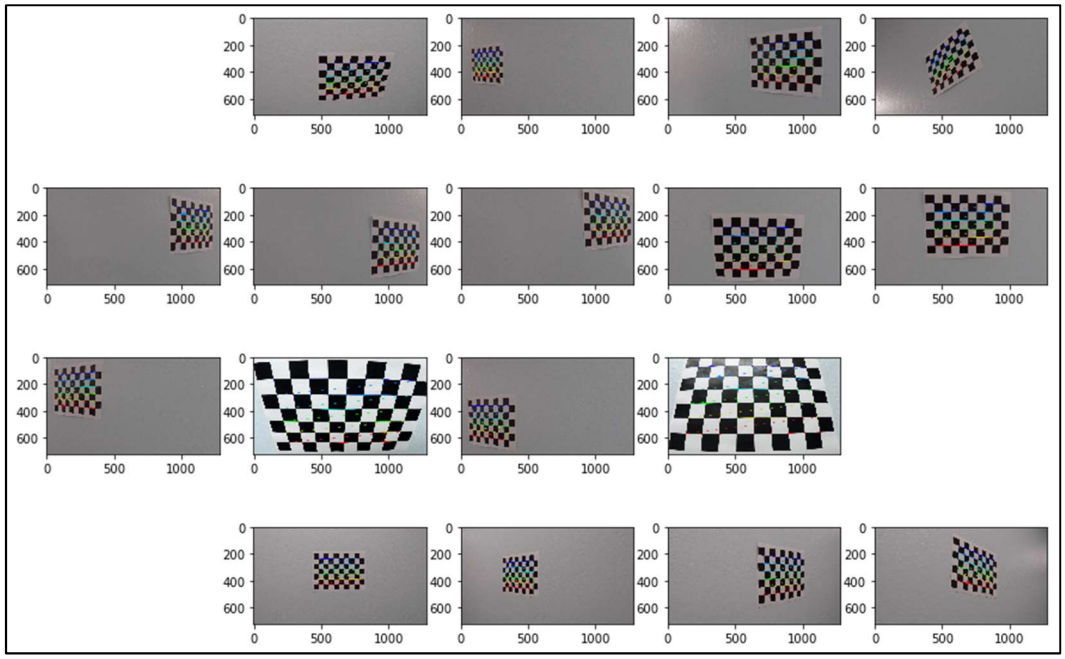


Fig.2. Chessboard images with corners drawn

It can be seen from Fig.2 that three images are missing.

2) Once corner coordinates have been found, use cv2. calibrateCamera() function to get calibration matrix and distortion coefficients:
ret, **mtx**, **dist**, **rvecs**, **tvecs** = **cv2.calibrateCamera(object_points**, **image_points**, (**img.shape**[0], **img.shape**[1]),**None**,**None**)
where **object_points** – regular 9x6 grid coordinates and **image_points** – stacked coordinates of corners found, **mtx**, **dst** – matrix camera and distortion coefficients, respectively.
3) With **mtx**, **dst** found, apply **cv2.undistort()** function to initial images to get undistorted images. Sample undistorted images shown on Fig.3

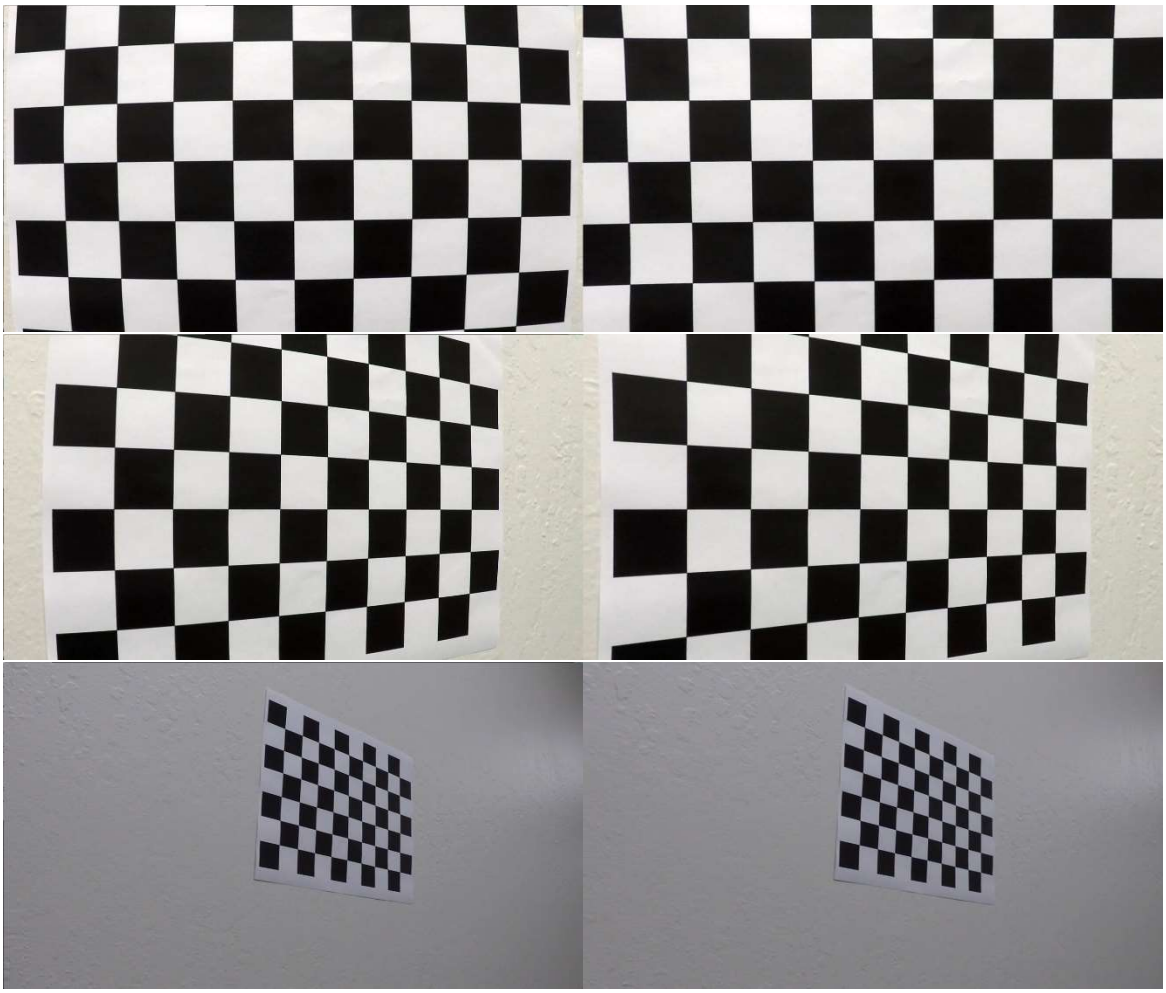


Fig 3. Initial (left) and undistorted (right) images of chessboard.

Image processing and filtering

For the image processing the following steps were done:

1) Convert to HLS colormap and select one more channels for further processing. During experiments it was found that the most robust result can be obtained from combined L and S channels, though L channel can add more noise to filtered image. This was implemented in function

to_hls_scaled(img, **init_cmap**='**BGR**', **out_channels**=['**S**'], **weights**=[1]),

where **img** – input 3-channel image, **init_cmap** – colormap of input image, **out_channels** – output channels in HLS space, **weights** – respective weights of **out_channels**
Output of this function is scaled to [0,1] 1-channel image, where each value is weighted sum of output channels.

2) After converting to 'HLS-weighted' image, sobel filter is applied to get gradients. This was implemented in function

combined_sobel(img, mag_min, mag_max, dir_min, dir_max, init_cmap='BGR'),

where **img** – input image, **mag_min** – minimum value of gradient to retain, **mag_max** – maximum value of gradient (both values are in [0,1] interval), **dir_min**, **dir_max** – minimum and maximum angles of gradient direction in [0, pi/2] interval, init_cmap – colormap of input image (for 'HLS-weighted' image from the previous step **init_cmap=='GRAY'**, as it has only 1 channel). Further experiments were about selecting best weights for to_hls_scaled function and **mag_min**, **mag_max**, **dir_min**, **dir_max** values for combined_sobel. During searching is was a tradeoff between robustness and noise on filtered images. A compromise was found with the following values:

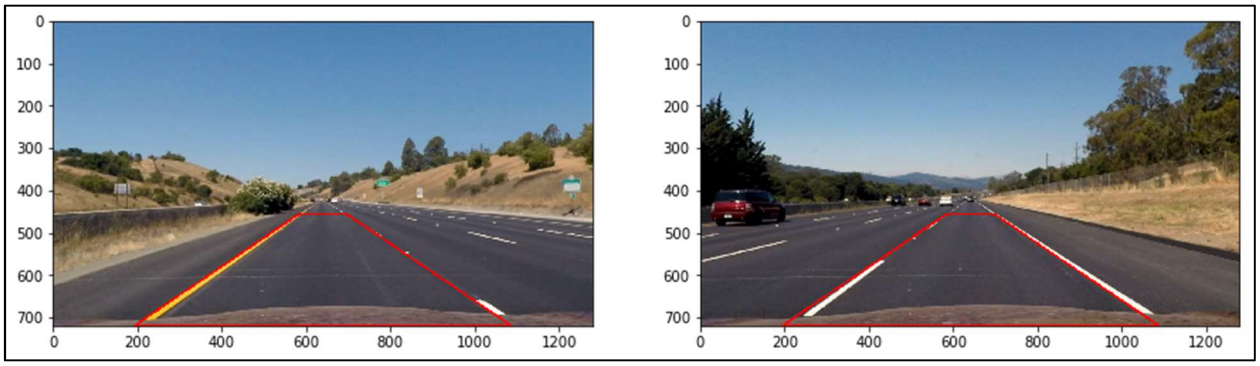
weights = [1,1]
mag_min = 0.095
mag_max = 1
dir_min = 0.1*np.pi/2.
dir_max = 0.7*np.pi/2.

The results of image processing and output test images are shown if Fig.4 and Fig.5, respectively.

Perspective transform

Perspective transformation was performed with cv2.warpPerspective() function. For defining source and transformed points coordinates a simple helper function get_polygon_vertices() was implemented wich returns four polygon vertices for desired values of parts of the image to be warped.

Below are the sample images with polygon drawn.



The result of applying perspective transform to filtered images is shown on Fig.6.

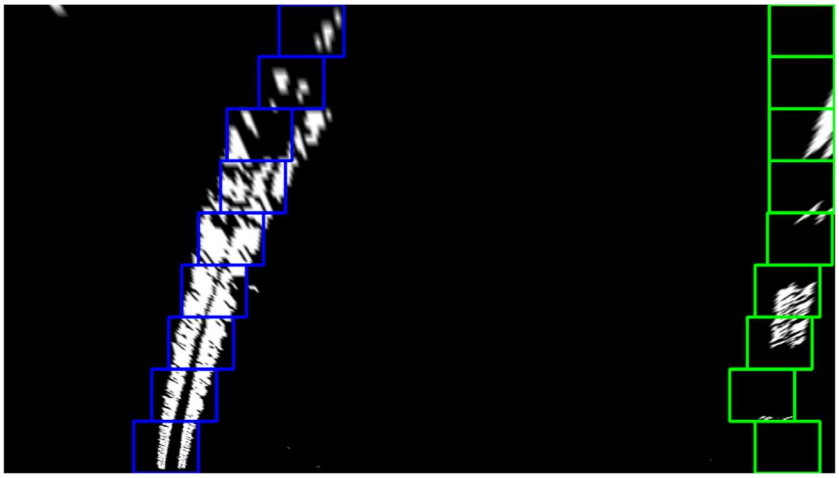
The quality of images is quite satisfactory for further processing and lane finding.

Lane finding

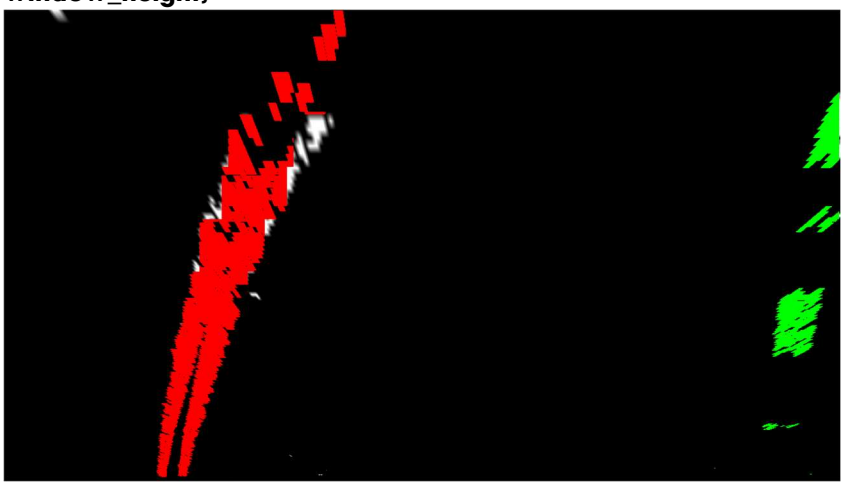
My implementation of lane finding algorithm is based on convolution of grayscale image with sliding window and finding peaks among convolution values. These peaks are considered to be the best centers of sliding window to select points that are inside this window. Then the selected points are used to get a second order fit which represents the best fit for road line. The pipeline of this algorithm consists of the following steps:

1) Convert initial camera-view image to bird-view filtered image, as described in the previous sections
2) Divide the image to horizontal stripes in accordance with the sliding window height.

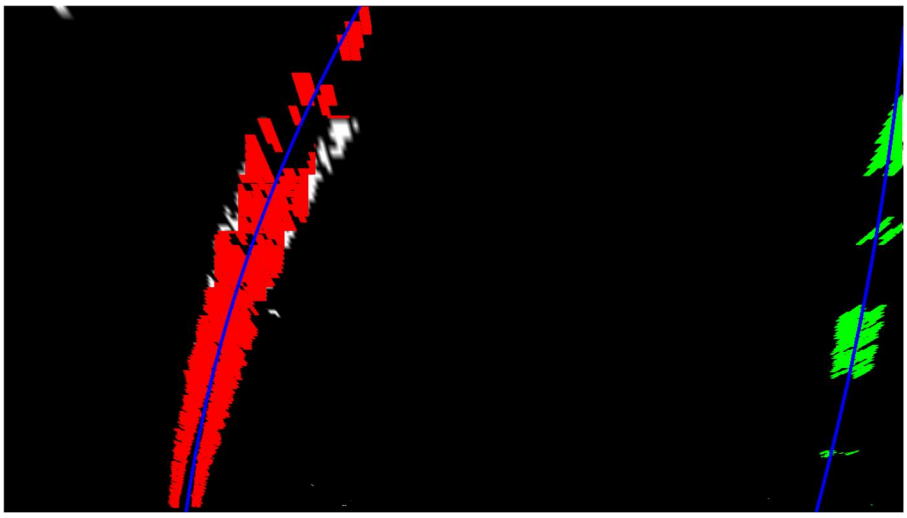
3) Find initial position of sliding windows (centroids) which represent the positions of road lines at the bottom of the picture. This procedure is implemented in **find_init_centroids(img, window_width)** function, where **img** – grayscaled filtered birdeye-view image, **window_width** – width of the sliding window in pixels.
4) For the bottom stripe set the initial guess of centroid coordinates from the previous step. Make convolution of the stripe with the sliding window in a small region around the initial guess. Set the coordinates of bottom stripe centroids to the maximum values of convolutions.
5) Move to the next stripe and set the initial guess for centroid coordinates equal to the coordinates from the previous stripe. Make convolution in a certain region around the initial guess and find the best position of centroids. Make this operation for all the stripes. This procedure is implemented in function **find_centroids(img, window_width, window_height, margin, init_left_centroid, init_right_centroid, min_points=50)**. The meaning of the parameters is quite obvious except '**margin**' that represents an area around the initial guess of centroid position to make convolution in (measured in pixels) and '**min_points**' that is the minimum number of points in a sliding window to 'believe' that it might represent a part of road line. Sample result is presented below:



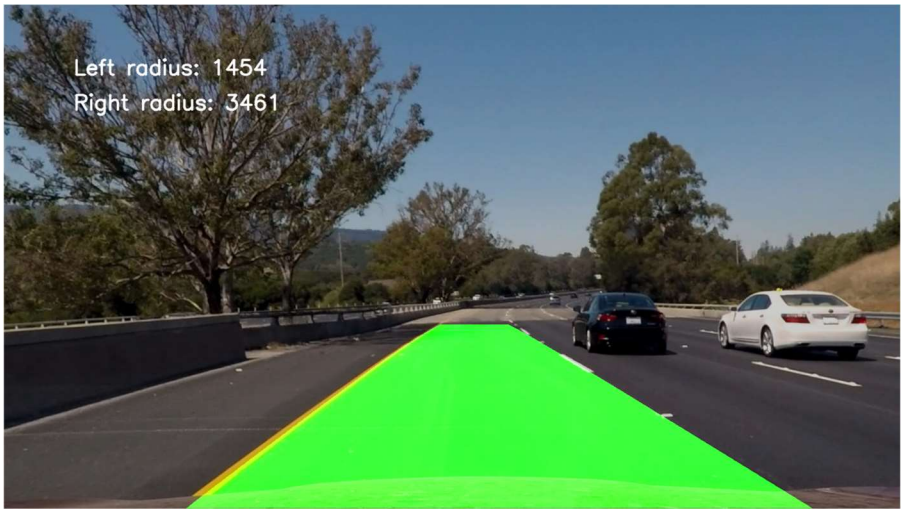
6) Having the coordinates of centroids, select the points from the image to make fit on. This is implemented in function **select_points_by_centroids(img, centroids, window_width, window_height)**



7) Fit lines on selected points. This is implemented with function **get_polylines(img, l_points, r_points)**



8) Draw a polygon on fitted lines which would represent the lane area.
9) Unwarp the polygon with inverse perspective transform and combine the result with the initial camera image.



All the functions listed above are implemented inside **lines** class.

There are three basic methods:

1) **__init__(self, dist, mtx, M, invM)** – initialize an instance, dist, mtx – distortion coefficients and camera matrix, respectively, M, invM – matrices of prespective and inverse perspective transforms, respectively
2) **set_params(self, draw='region', single_mode=False, window_width=50, window_height=80, margin=50, curve_line_margin=50, mag_min=0.05, mag_max=1, dir_min=0., dir_max=0.9*np.pi, init_cmap='BGR', hls_channels=['S', 'L'], max_to_keep=5, min_points=50)** – set parameters for video processing

parameters:

draw ['region', 'lines'] – draw a filled polygon or just lines

single_mode – if set to true, searches lines from scratch for every frame, otherwise uses lines from the previous frame and makes search only in curve_line_margin (in pixels) region around them

window_width – width of search window

window_height – height of search window

margin – width of search zone for **single_mode == True**

curve_line_margin - margin for **single_mode == False**

mag_min, **mag_max**, **dir_min**, **dir_max** – parameters for sobel filtering

hls_channels – which of HLS channels to use for processing

max_to_keep – number of frames used to average line drawing

min_points – threshold number of points for line searching

3) **get_lines(self, img)** – finds lines and draws then on the image

Example usage:

```
line_gen = lines(dist, mtx, M, invM)
line_gen.set_params()
result = line_gen.get_lines(img)
```

Discussion

There are few issues with the approach listed above:

1) Need for accurate search of parameters for filtering with no guarantee of their robustness. Tradeoff between stability and noise effects.
2) Perspective transform makes farther part of the lines blur, rather accurate perspective calibration required to eliminate bias effects (i.e., if far points are chosen inaccurately, parallel lines become curved). For very broken ground additional bias emerge from the fact that in that case the road isn't a plane, maybe more sophisticated transform needed.
3) If there is a long object parallel to the lane, i.e. road safety barrier, there might be wrong detection of the lane border especially when the barrier is placed near the lane

Ways to improve:

1) Use "weighted" convolution during line detection that would use the fact that car tends to be close to the center of the lane, so the weights decrease towards the edges of the picture, making the algorithm choose points closer to the center of image
2) Use lines predictions obtained from different filer schemes (i.e. one from HLS and another one from RGB colormaps) then estimate confidence of these predictions and combine best ones

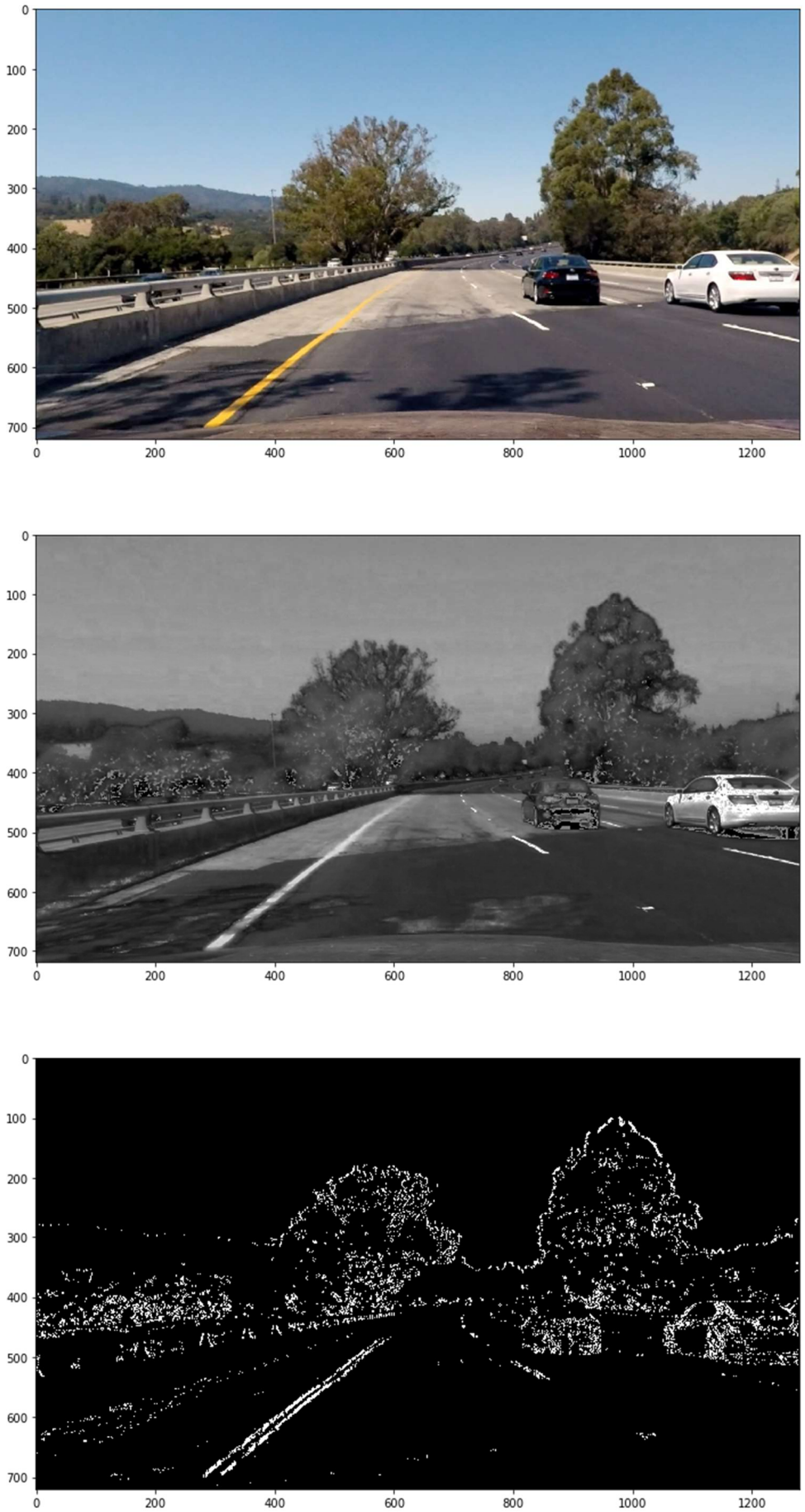


Fig.4. Initial (top), HLS-weighted (middle) and sobel gradient (bottom) images.

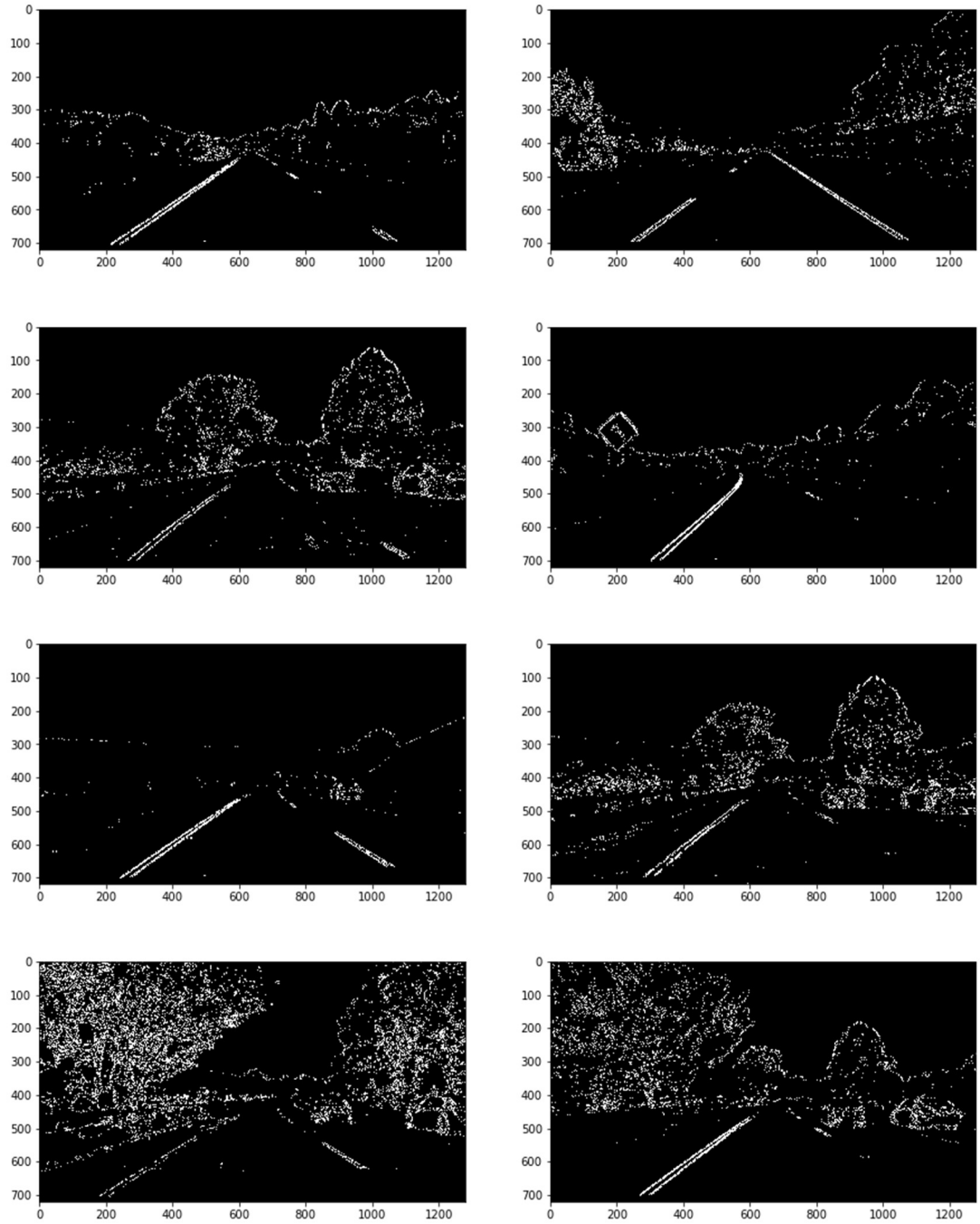


Fig.5. Filtered test images.

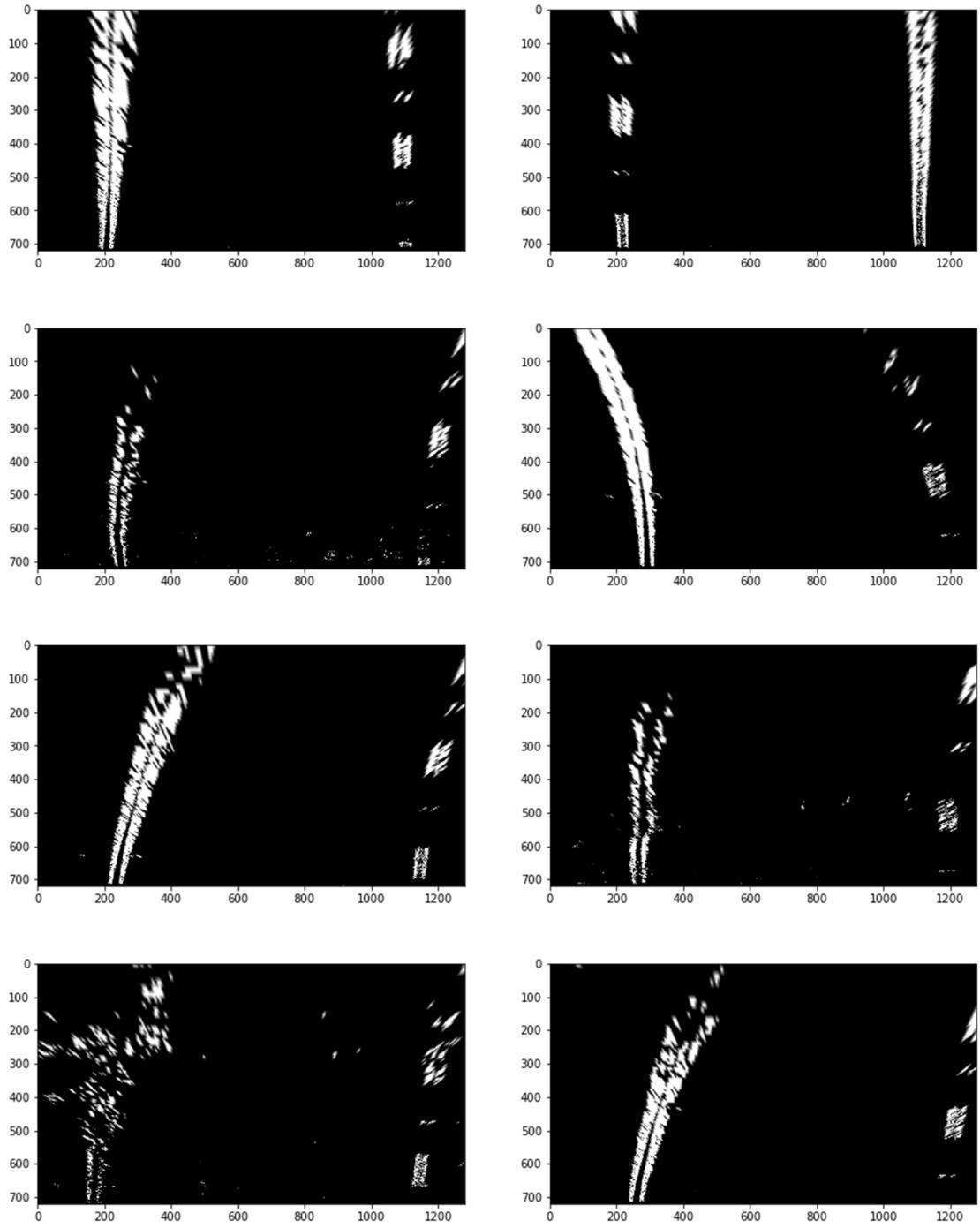


Fig.6. Filtered images transformed to bird-eye view.