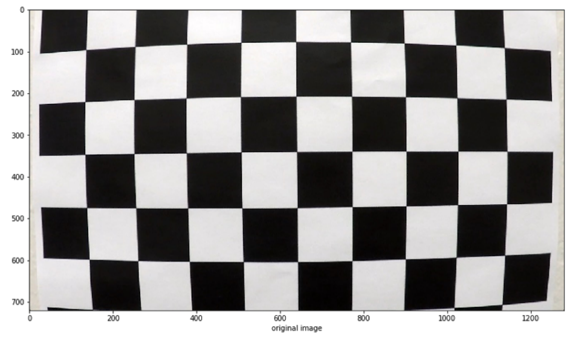
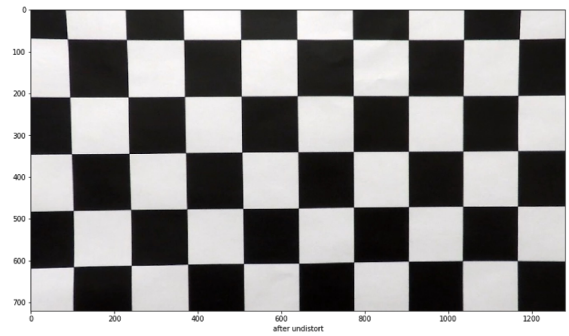
The project includes:
project.ipynb
c_params.p.npz
project_video_output.mp4
challenge_video_output.mp4
harder_challenge_video_output.mp4

Camera Calibration

All code is in project.ipynb.
calibrate_from_chessboard_images () is to calibrate camera by using pictures under camera_cal/*.jpg.
Using the course's example code, cv2.findChessboardCorners from every picture and cv2.calibrateCamera to get mtx, dist.
I saved the params as file c_params.p.npz by using np.savez_compressed
So I don't need to calculate them every time just load them by calling my function load_calibrate_params()

Here is a image after calibration:





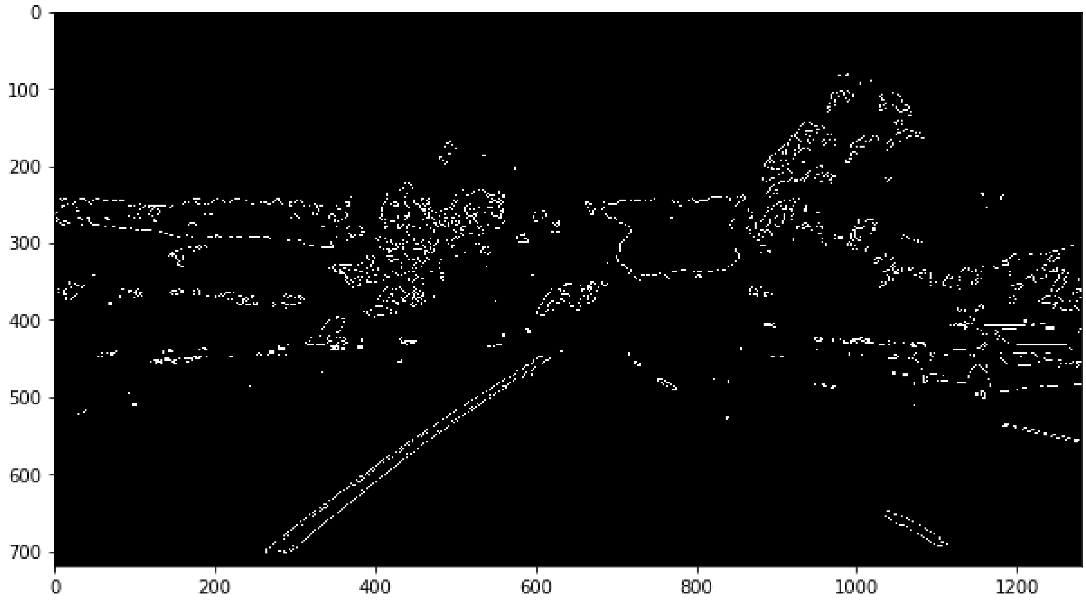
Pipeline (single images)

By undistorting an image, I first filter out all white and yellow colors, by using HSV color range:

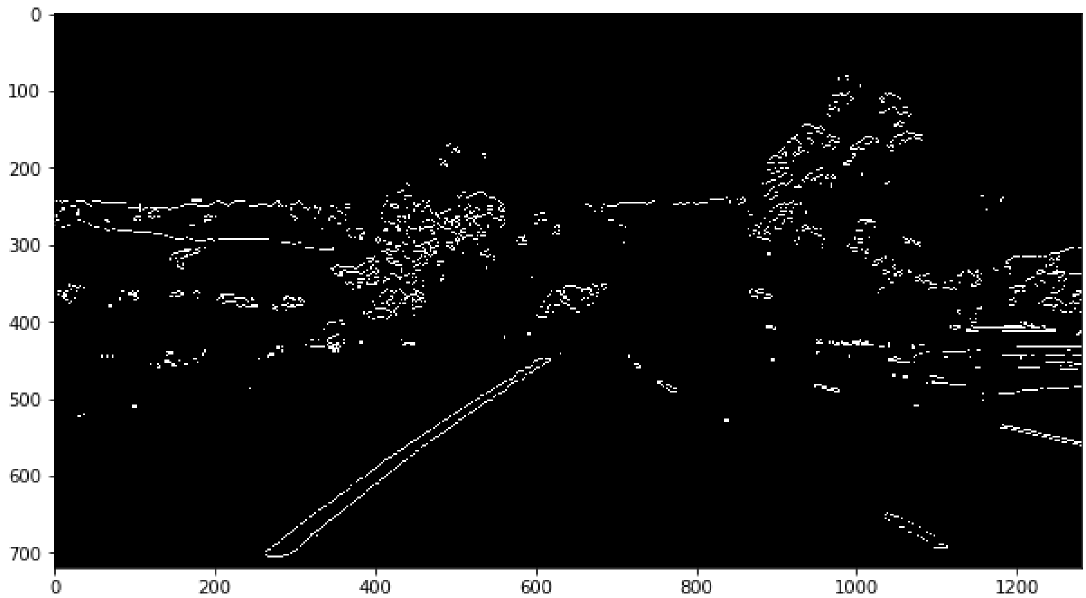
```
yellow_min = np.array([65, 80, 80], np.uint8)
yellow_max = np.array([105, 255, 255], np.uint8)
yellow_mask = cv2.inRange(blur_img, yellow_min, yellow_max);
white_min = np.array([0, 0, 220], np.uint8)
white_max = np.array([255, 80, 255], np.uint8)
white_mask = cv2.inRange(blur_img, white_min, white_max)
```

In the function filter_by_color I also use gaussian blur to smooth the picture better.

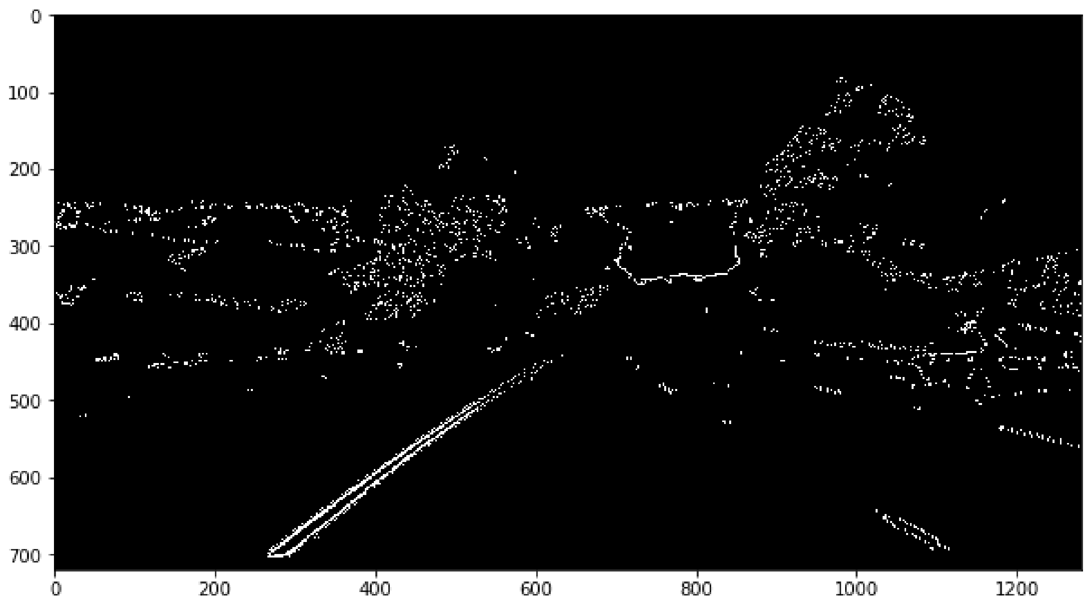
Then I use the first class's cv2.Canny function to generate edges, and also used sobel operator and magnitude gradient to detect edges. Here is the result of three detected results. We can see three different detecting ways have good edges detection performance, and the sobel operator looks like having more smooth edges than the other two ways.



canny_binary



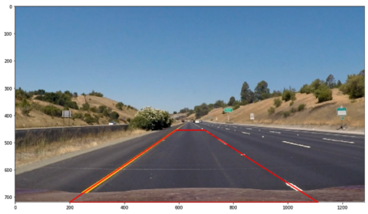
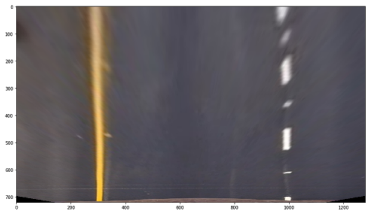
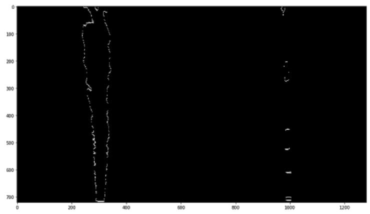
sobel image



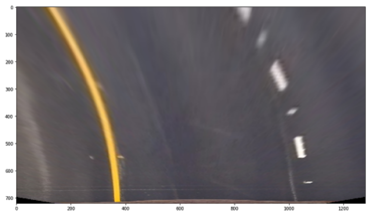
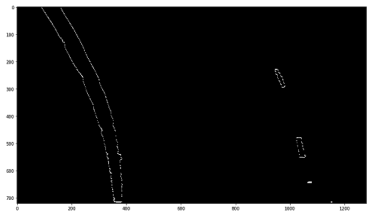
magnitude image

Then I use the straight_lines1.jpg to transfer picture to bird eyes view.
Sine the straight line picture is easy to manual choose 4 point, there the src and des I choose:

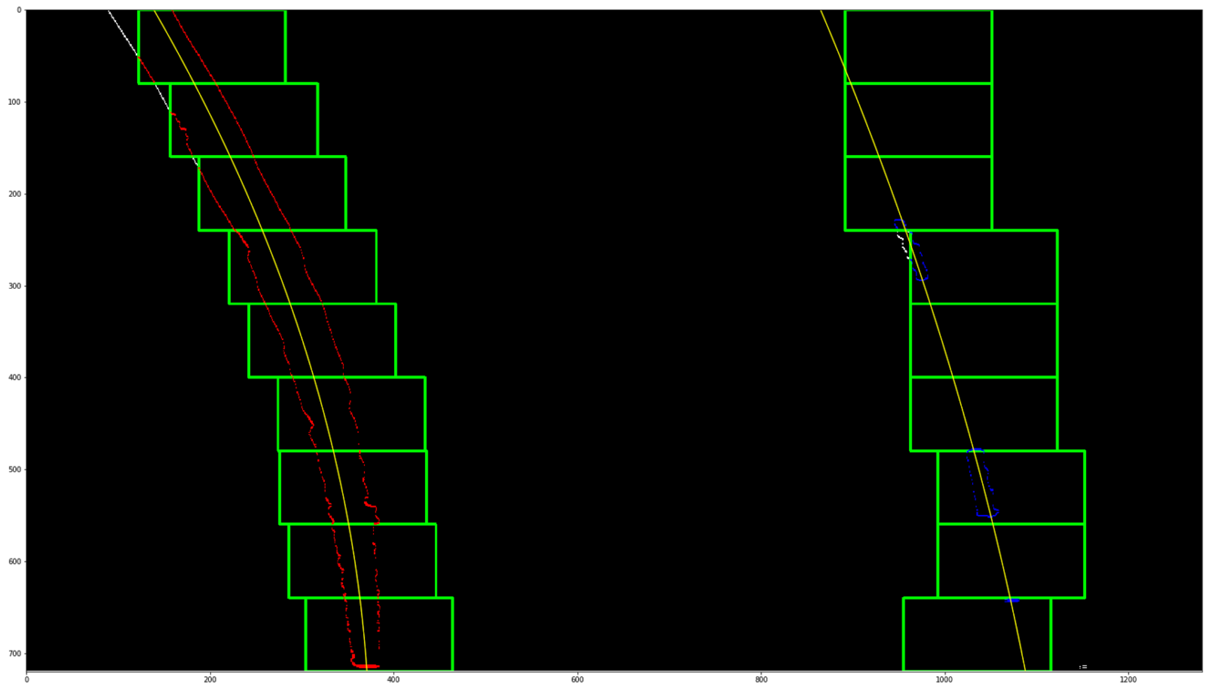
```
src = np.float32([(590.0 /1280. * w,  455. / 720. * h),
                  (695.0 /1280. * w,  455. / 720. * h),
                  (1110.0 / 1280. * w, 1. * h),
                  (200.0 / 1280. * w, 1. * h)])

dst = np.float32([(300.0 /1280. * w,  100. / 720. * h),
                  (1000.0 /1280. * w, 100. / 720. * h),
                  (1000.0 / 1280. * w, 1. * h),
                  (300.0 / 1280. * w, 1. * h)])
```

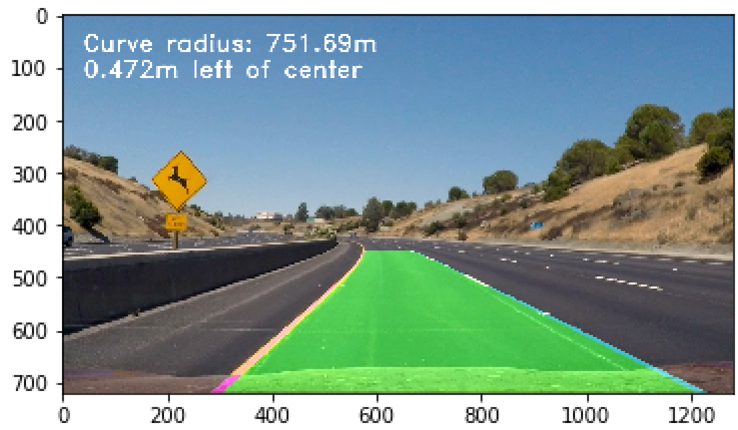
It works well, by testing other images:

Then I use the sliding window detection method to generate two lanes



The I draw the two lane back to the original image, I got this:



I used find_lanes() function to generate left_fit,right_fit,left_curverad,
right_curverad,center_dist
I used:
ym_per_pix = 30/720
xm_per_pix = 3.7/700
to convert pixel to meters

Then I used cv2.putText to print the curvature and center distant in the image.
With function draw_lane()

Finally I write a function pipeline() to integrate all this together, and I also write a function to smooth two continues continuous images' lanes.
Fine_lane_by_prev_fit()

Which use the previous the left and right fit to map y coordinate to x coordinate and use the new points to update next round of left, right fit.

I have a function test_one() to generate a image lanes like this:

Using test_video(input_video, output_video)
I generated
project_video_output.mp4
challenge_video_output.mp4
harder_challenge_video_output.mp4

The project_video_output.mp4 looks having a good results.
But challenge_video_output and harder challenge_video_output is not good, only detected partial of the driving mile's lanes.

So I think a key process of this pipeline is how get a well edge detected image. Either having a better color filtering method and tuning the weight of sobel or magnitude gradient methods to get better results. Maybe potentially we can use deep learning method to predict the lane's line function parameters like y=Ay^2 + By + C, to predict A,B,C by using picture as input, generated A,B,C to train the data.