# Advanced Lane Finding Project

I only completed the first project(project_video.mp4), and to be honest, I am not enough satisfied with the detection algorithm, but it is the deadline.
All code is in the Advanced_Lane_Finding.ipynb file, it is in CarND-Advanced-Lane-Lines/examples/Advanced_Lane_Finding.ipynb, you can find it in my workspace.

The framework of code is from the Udacity lesson.

## Step 1: Compute the camera calibration matrix

For adjusting the distortion images, the first thing is to compute the camera calibration matrix, and why are the images need to undo distorted? Because there are images distortion in the camera.

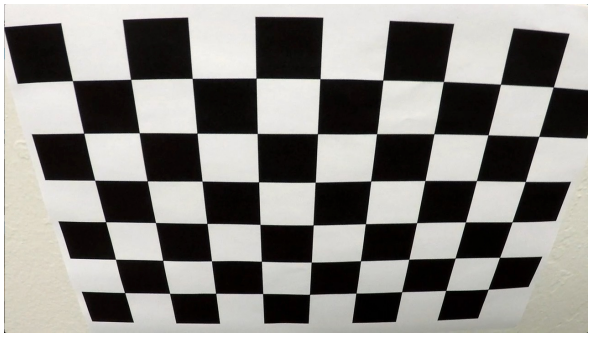
Image distortion occurs when a camera looks at 3D objects in the real world and transforms them into a 2D image; This transformation isn't perfect. Distortion actually changes what the shape and size of these 3D objects appear to be. So, the first step in analyzing camera images, is to undo this distortion so that we can get correct and useful information out of them.

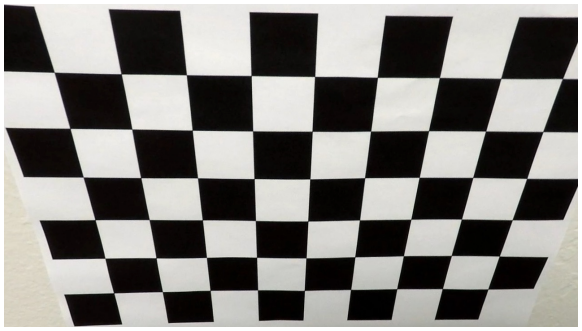
The main types of distortion:
1) Radial distortion, Real cameras use curved lenses to form an image, and light rays often bend a little too much or too little at the edges of these lenses. This creates an effect that distorts the edges of images, so these lines or objects appear more or less curved than they actually are. This is called radial distortion, and it's the most common type of distortion.
2) Tangential distortion, this occurs when a camera's lens is not aligned perfectly parallel to the imaging plane, where the camera film or sensor is. This makes an image look tilted so that some objects appear farther away or closer than they actually are.

I defined the **get_camera_parameters**() function to get the camera intrinsic and extrinsic parameters from several views of a calibration pattern, these parameters are [retval, cameraMatrix, distCoeffs, rvecs, tvecs] with the **cv2.calibrateCamera**()

## Step 2: Get undistortion image

I defined the **get_undistort**() function to get the undistortion images with [retval, cameraMatrix, distCoeffs, rvecs, tvecs] parameters and **cv2.undistort**() function.





## Step 3: Get binary image

For identify pixels that were likely to be part of a line in an image, I used the combination of sobel operation(gradient orientation) and HSL color space to do it. (If you have more time, you should try out some more colorspaces to experiment with like channels from LAB or LUV. You might be able to get some really good results without utilizing sobel at all, in fact)

It took much time to tweak parameter and threshold for the quality, which I tried the below:
sobel x(x-direction),
sobel y(y-direction),
magnitude of the gradient(the gradient is just the square root of the squares of the individual x and y gradients),
direction of the gradient,
HLS color space.

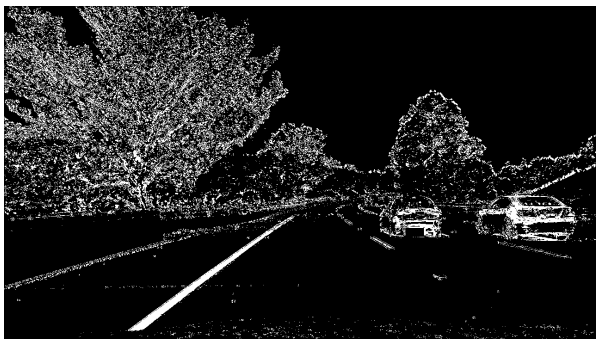
Finally, I select the combination sobel x, direction of gradient and HLS, the reason:
1) sobel x, I find the x-gradient does a cleaner job of picking up the lane lines and it is suitable for the vertical line, gradx = abs_sobel_thresh(image, orient='x', sobel_kernel=ksize, thresh=(10, 100));
2) direction of the gradient, it is much noisier than the others, but it can pick out particular features by orientation, dir_binary = dir_threshold(image, sobel_kernel=ksize, thresh=(0.2, 1.2));
3) HLS color space, choose the  S channel, it is doing a fairly robust job of picking up the lines under very different color and contrast conditions, while the other selections look messy, s_binary[(s_channel >= 80) & (s_channel <= 170)] = 1.

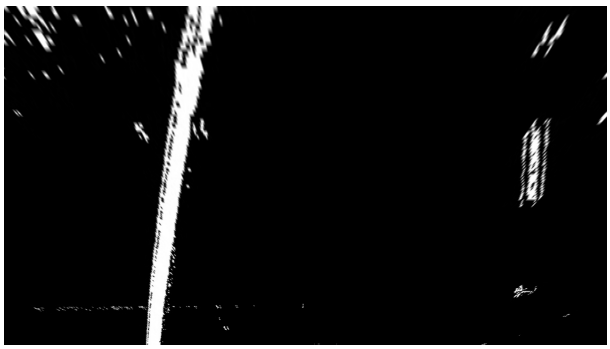
The detail is in the **combined_binary**() function.

## Step 4: Get perspective transform image

For measure the curvature of the lane lines, need to transform the image to a top-down view.

The key point is perspective Matrix in the step:
Firstly, assumed the road is a flat plane, it isn't strictly true, but it can serve as an approximation for the step;
Secondly, pick up four points in a trapezoidal shape that would represent a rectangle when looking down on the road from above,
Thirdly,  selected a trapezoid by using image dimensions ratios as an input, refer to https://cmlpr.github.io/blog/2017/03/14/advanced-lane-lines.
fourth, used cv2.getPerspectiveTransform() function to get perspective Matrix, then used cv2.warpPerspective() function to transform the image to a birds eye view image.





After perspective transform, make the lines look straight and vertical from a bird's-eye view perspective.The detail is in the get_src_des_coordinate(), perspective_M() and get_birds_eye_view() function.

# Step 5: Find the Lane Lines.

Based on the step 4, detect lane pixels and find the lane lines.

1) Line Finding Method: peaks in a histogram, it can find a starting point of left and right of lane line;
2) Line Finding Method: Sliding Window method, it can tell us where the lane lines go;
3) Line Finding Method: fit a polynomial to all the relevant pixels found in the sliding windows with polyfit.

### Peaks in a histogram

After applying calibration, thresholding, and a perspective transform to a road image, I have the binary images where the lane lines stand out clearly. However, I still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

In the peaks histogram algorithm, I improved the one points:

- Take a histogram along all the columns in the of the image, it is different from udacity lesson that it take all the columns in the **lower half** of the image.

### Sliding Window method

As shown in the previous, we can use the two highest peaks from our histogram as a starting point for determining where the lane lines are, and then use **sliding windows** moving upward in the image (further along the road) to determine where the lane lines go.

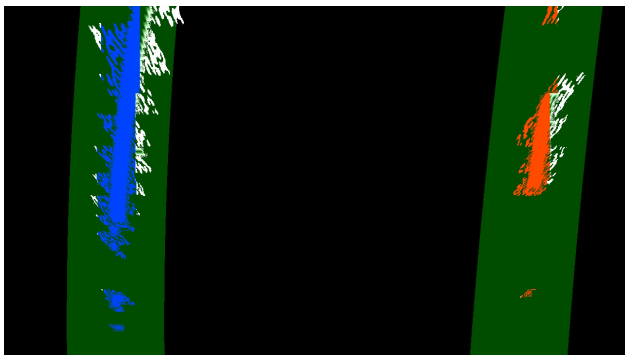
Set the hyperparameters related to the sliding windows, and set them up to iterate across the binary activations in the image.

**nwindows = 9,** is the number of sliding windows
**margin = 30**, is the width of the windows +/- margin
**minpix = 500**, is minimum number of pixels found to recenter window
Iterate through nwindows to track curvature, with the given window sliding left or right if it finds the mean position of activated pixels within the window to have shifted.

### Fit a polynomial to all the relevant pixels

Fit a polynomial to all the relevant pixels with np.polyfit.

Detail is in the **find_lines**() function.



### Step 6: Measure curvature and vehicle position

### Measure curvature

$$f(y) = Ay^2 + By + C$$

The radius of curvature is defined as follow it By M. Bourne.

I have located the lane line pixels, used their x and y pixel positions to fit a second order polynomial curve.

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

fitting for f(y), rather than f(x), because the lane lines in the warped image are near vertical and

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

may have the same x value for more than one y value.

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

The radius of curvature at any point x of the function x = f(y) is given as follows:
In the case of the second order polynomial above, the first and second derivatives are:
So, the equation for radius of curvature becomes:

From Pixels to Real-World, the radius reporting is in pixel space, which is not the same as real world space, so need to repeat this calculation after converting x and y values to real world space.For this project, we can assume that if we are projecting a section of lane similar to the images above, the lane is about **30 meters long and 3.7 meters wide**, which followed the Udacity lesson.

### Vehicle position

For this project, I assume the camera is mounted at the center of the car, such that the lane center is the midpoint at the bottom of the image between the two lines detected. The offset of the lane center from the center of the image (converted from pixels to meters) is the distance from the center of the lane, so we can get the vehicle position by “lane_center - center_image”

Details are the **find_lines**() function and **radius_curvature**() function.

### Step 7: Pipeline

combination the above all steps to the pipeline() function.

The project_video.mp4 of pipeline handle is in here.
link: https://www.youtube.com/watch?v=QG3dy_bCC2U&feature=youtu.be

# 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?

Very much!

Firstly, for get binary image, it takes a lot time to tweak the parameters, and the result is not good to me, because while I got the properly parameters, I unexpectedly forget to save it, my god, only save the video, so I only can give you best result video( but it's gray color), the link: https://www.youtube.com/watch?v=aWn-gJPW8gk, which tell myself that I am still not familiar with the principle how to tweak it.

Secondly, not familiar with the OpenCV and compute vision, about how to properly save the image with gray color and RGB color, and how to convert them, which I learned a lot.

Thirdly, I got to know it is a trade off to tweak parameter and tweak thresh and combination binary image.

I have no time to handle challenge_video.mp4 and harder_challenge_video.mp4, but when I coded for the project, I find a paper that IEEE IV 2018 excellent <<Towards End-to-End Lane Detection an Instance Segmentation Approach>>, it can handle the harder_challenge_video.mp4.

**Some questions from the project reviews, I should to handle these:**

1. A question for you to think about - Do you think applying the transform before thresholding would be more or less helpful than applying the transform after thresholding? Which one might be a better approach if you were to implement this on actual hardware, assuming language of choice?

2. For calculating the radius of curvature and vehicle position, since the results update with every frame, do you think you could apply a low-pass filter (weighted average) here to smooth out these results as well? Do you think that's of any value to an actual SDC, if you smooth out these values instead of the absolute/exact value at each timestep?

3. Do you think dynamic thresholding would help with the challenge videos?
If you look at your results video, you notice how the detected lane seems to spread out as the car movies over a bumpy surface? If you notice that as well, why do you think that happens or what part of your code contributes that particular behavior? The reason I am asking this is so that you try to think about how exactly the camera movements because of the car can (or can not) affect the detection. Do you think that could be improved somehow? Think about it :)

4. Do you think having 3D road data/information could help produce a more robust pipeline for lane detection in actual autonomous vehicles? Or are 2D images, like for this project, reasonable for such a task in almost all cases/scenarios?

5. How could or would you go about implementing the pipeline using a Deep Neural Network/ Model approach instead of using classical Computer Vision? See, if you could try this out as an additional project to up your skills whenever you feel you have sufficient knowledge with ML/DL, if you'd like to!(The suggestion is very important, Lane Detection with Deep Learning and Towards End-to-End Lane Detection: an Instance SegmentationApproach)