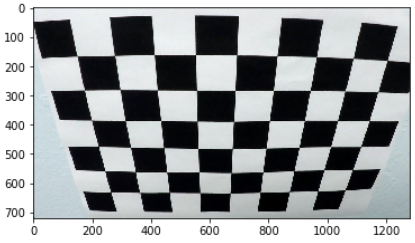
Advanced Lane Finding Project

The goals / steps of this project are the following:

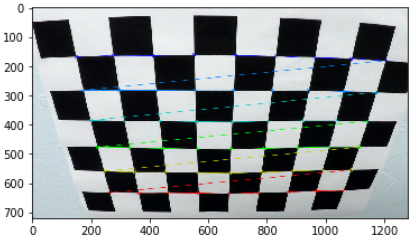
1. Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
2. Apply a distortion correction to raw images.
3. Use color transforms, gradients, etc., to create a thresholded binary image.
4. Apply a perspective transform to rectify binary image ("birds-eye view").
5. Detect lane pixels and fit to find the lane boundary.
6. Determine the curvature of the lane and vehicle position with respect to center.
7. Warp the detected lane boundaries back onto the original image.
8. Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

1. Calibrate Camera

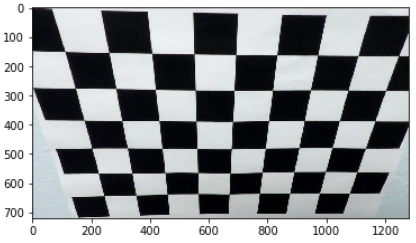
I use a 9x6 chessboard to calibrate the camera. Firstly, I prepare “object points” by generate a grid matrix of 9x6 dimension assuming that z = 0. Then I `cv2.findChessboardCorners()` to detect all corner points in a chessboard image. Finally, I compute the camera matrix and distortion coefficients by using the `cv2.calibrateCamera()` function. I applied the distortion correction to a test image using the `cv2.undistort()` function and obtained this result:



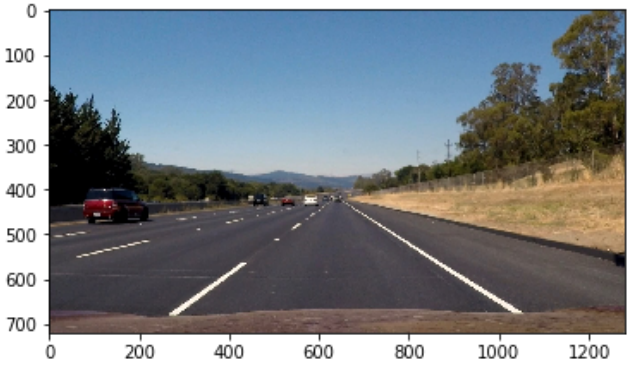
9x6 chessboard



Detect corners



Undistorted chessboard



Original image

Undistort image

2. Create a binary image

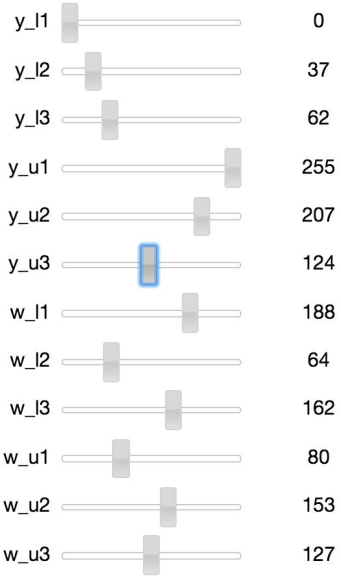
I use color and gradient selection to generate a binary image that contains lane pixels.

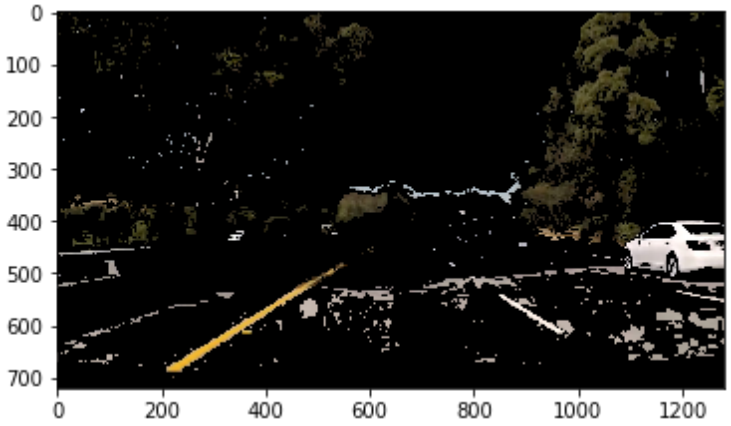
2.1 Color selection

I tried many color spaces other RGB. Here are some summary:

a. **HLS**: Hue, Lightness, Saturation
b. **HSV**: Hue, Saturation, Value (similar to lightness)
c. **CIE LUV**: L - luminance, U - blue luminance, V - red luminance
d. **CIE LAB**: L - luminance, a: green–red, b: blue–yellow

In order to pick the right lower bound and upper bound value for each color space, I create a GUI that allows me to play with the value and see how it affects final result:





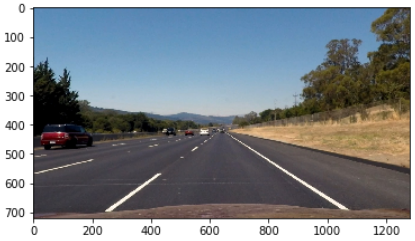
I pick the best value in each of the following color space: HSV, LUV and LAB:

Color space	Lower yellow	Upper yellow	Lower white	Upper white
HSV	[10, 100 ,90]	[22, 220, 255]	[0, 0, 180]	[180, 25, 255]
LUV	[0, 0 , 25]	[255, 95, 110]	[200, 0, 0]	[255, 255, 255]
LAB	[0, 0 , 80]	[255, 255, 110]	[196, 0, 0]	[255, 255, 255]

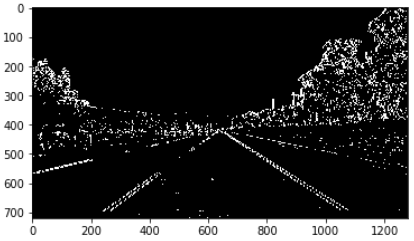
Then I applied the filter to both the project video and the challenge video and I upload the threshold output for the challenge video: challenge_thresh.mp4. I found LAB color space works stable and better in both videos.

2.2 Gradient selection

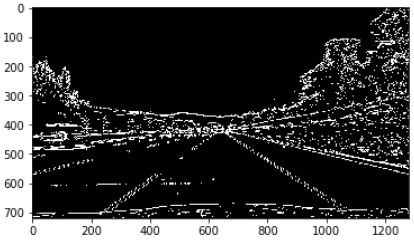
Gradient selection is more robust than color selection since lane color may be more than yellow and white. Here we use Sobel operator the calculate the gradient.



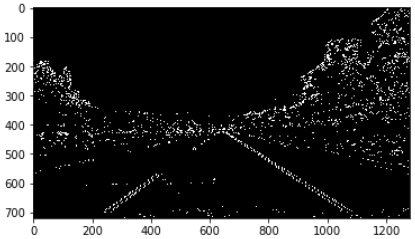
Original image



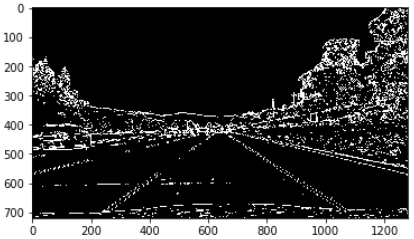
Sobel x



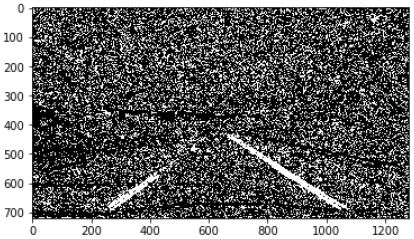
Sobel y



Gradient selection



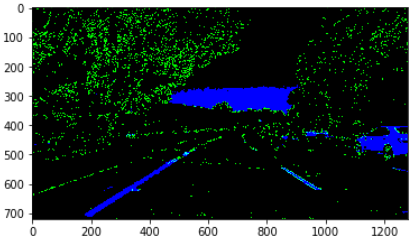
Gradient magnitude

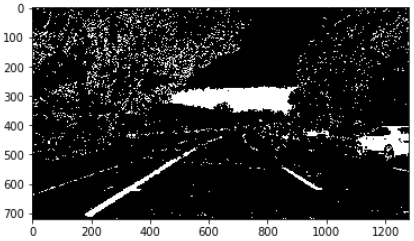


Gradient direction

2.3 Combine color and gradient selection methods

We can combine both the color and gradient selection methods to create a better binary image. The following example illustrates how combination can improve the final result (The green layer is the pixels selected by gradient method while the blue layer is the pixels selected by color method):



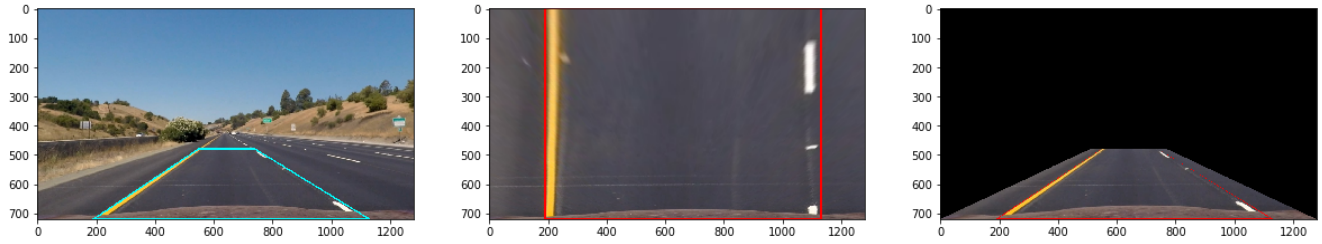


3. Perspective Transform

In this step, I apply a perspective transform to an image so that it shows a birds' eye view.
Here are the source and destination points I pick:

src: [(190, 720), (548, 480), (740, 480), (1130, 720)]
dst: [(190, 720), (190, 0), (1130, 0), (1130, 720)]

I use `cv2.getPerspectiveTransform()` to get the transform matrix and `cv2.warpPerspective()` to warp the image.



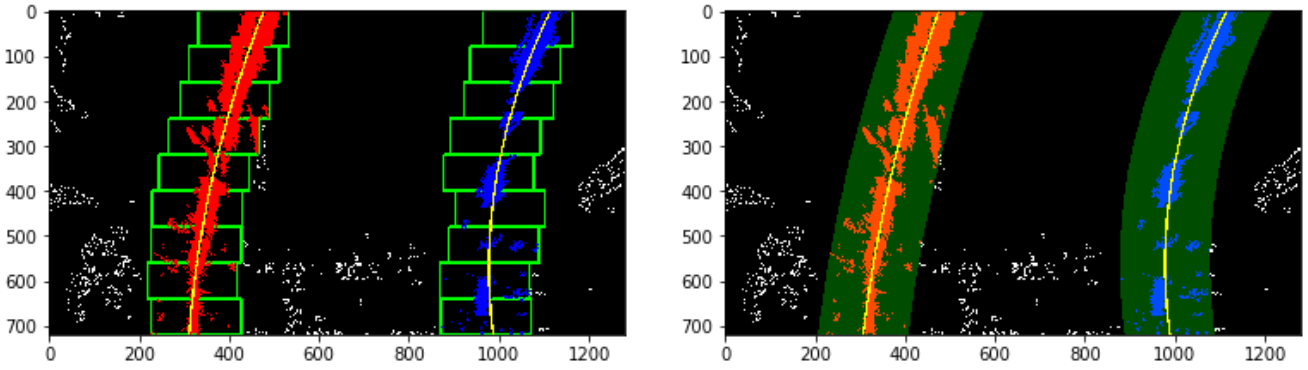
4. Find Lane Pixels

There are two searching algorithms:

1. Sliding window search
2. Polynomial ROI search

To search lane pixels from scratch, we use a sliding window method. We basically divide an image into N windows and for each window, we calculate the histogram and use two prominent peaks as the left and right lane.

If a polynomial ROI is given, we could search within the region.



Sliding window search

Polynomial ROI search

5. Curvature and Offset

Now, let's get more information from the warped image. From it, we can calculate the curvature of the road and also the distance of vehicle center from the road center (the offset).

5.1 Curvature

I use the following formula to calculate the curvature:

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

In case of a second order polynomial curve, the derivatives are:

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

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

Thus, the equation of curvature is:

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

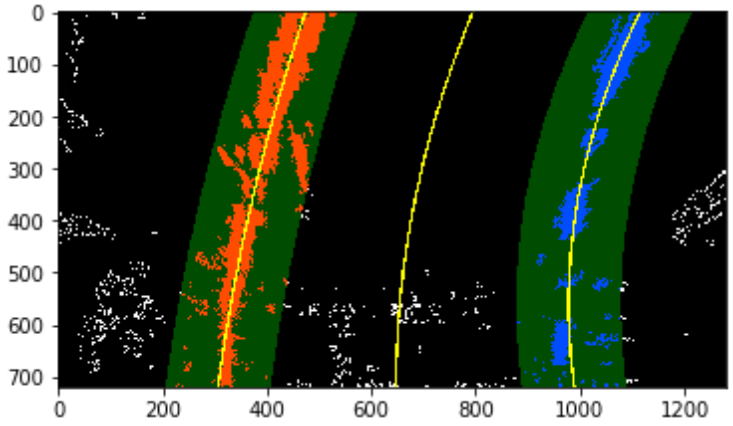
I take the average of left and right lane curvature to calculate the curvature for the lane.

5.2 Offset

Since we assume that the camera is mounted at the center of the car, the offset can be calculated as follows:

```
lane_center = (left_fitx[-1] + right_fitx[-1])/2
offset = np.abs(lane_center - image.shape[1]/2)
```

Finally, we need to convert the curvature and offset into the unit of meter. According to Udacity, we can assume that meter per pixel in y direction is 30/720 and in x direction is 3.7/700.



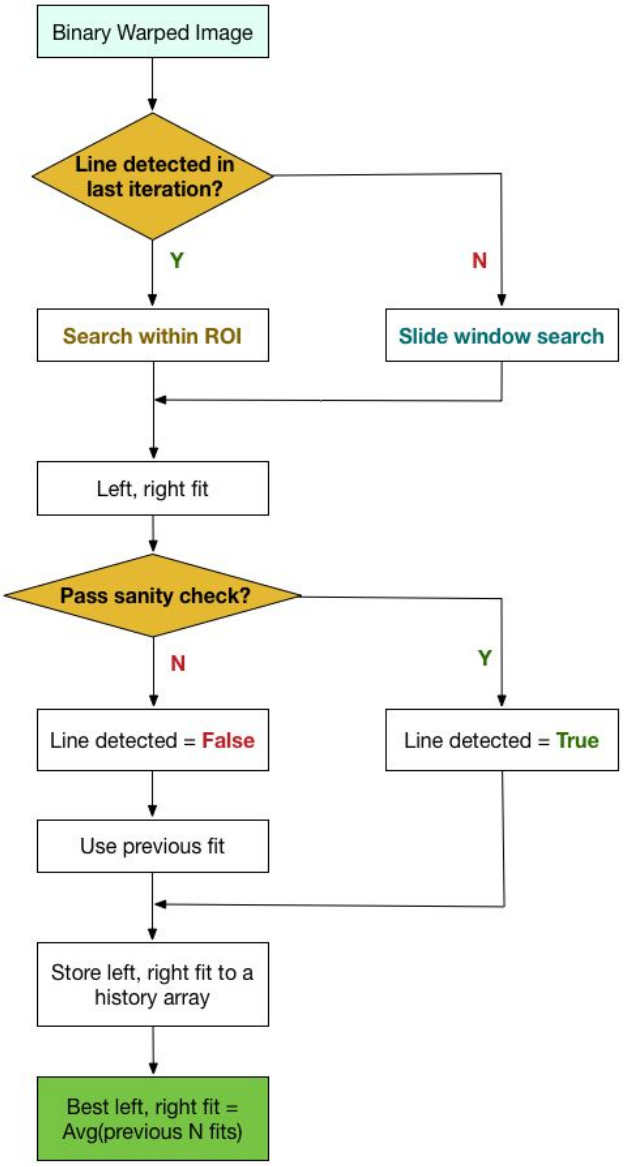
Left curvature: 1827.36463344 m
Right curvature: 592.68868776 m
Curvature: 894.908390349 m
Offset: 0.0276901176104 m

6. Line object

In order to produce a more stable output, I create an object called Line. Its responsibilities include:

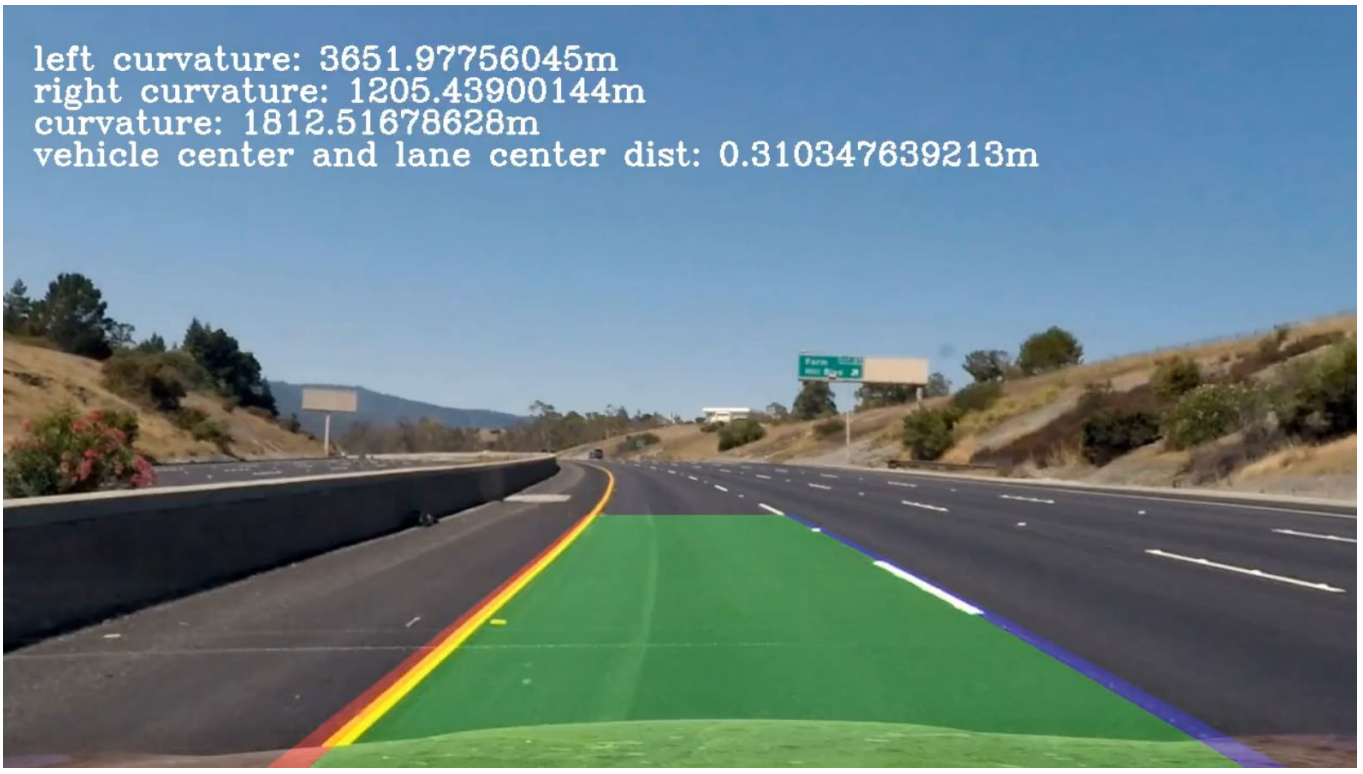
1. Keep track of lane polynomial functions for previous N frame
2. Intellectually choose sliding window algorithm or ROI search algorithm to locate lane lines
3. Perform sanity check on lane lines
4. Take average on current N frame to produce a more stable output
5. Return curvature and offset of the lane

The core algorithm of how a line object detect the polynomial functions for a line is as follows:



7. Annotate Image

To annotate on an undistort image, we need to create a warped layer that contains all the information (drivable region, left lane, right lane, left curvature, right curvature, curvature and offset) and then use the inverse matrix to unwarp the layer. Then add the layer back to the undistort image. Here's an example:



I also create a diagnosis image when tuning the parameter:

8. Discussion

After I introduce the Line object, I can get a perfect result for video 1. What's more, it can reuse previous result in case that lane pixels can't be found! If you check "challenge video screen-shot 1", you can see that the sliding window method returns zero pixels and the algorithm can still be able to generate output.

Challenge video screen-shot 1

However, I am still not able to generate satisfying output for the challenge video and the harder-challenge video. One of the reasons is that the shadow sometimes aligns with the lane very well and produce even stronger binary pixels than the lane pixels. This can trick the self-driving car.

I am still experimenting with some improvements:

1. Restrict the search margin to lower value
2. Adjust color range so that more yellow pixels can be included
3. Modify sanity check condition

Anyways, this is a really awesome project and I do learn a lot from it!