P4 Write-up Report 3rd release

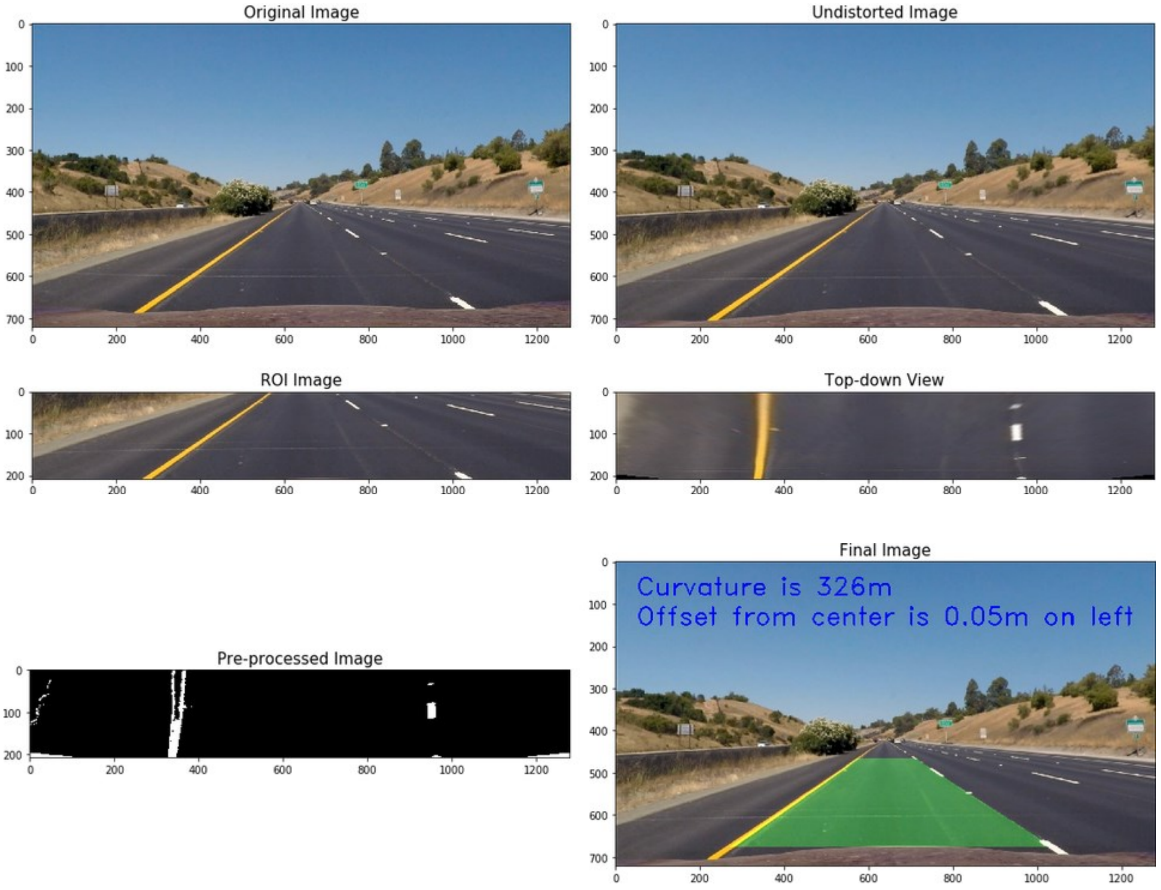
Advanced Lane Finding

jiyang_wang@yahoo.com

1. Release Notes on the 3rd release

The issue in the 2nd release was that lanes are drawn on the distorted frame instead of undistorted one. This issue has been solved by this release. The root cause was that the order of image processing steps was not kept consistent in various parts of the program. Now the processing order below is kept strictly consistent, which results in right image output with lanes drawn on undistorted frames. See below the results of all the processing steps.

Original Image → Distortion Correction → ROI Cropping → Top-down Transform



2. Release Notes on the 2nd release

Two major issues in my first submission are solved in this release:

- Issue with the polynomial fitting in almost every frame of the video where the top corners of the polygon are not lining up with the actual locations of the lane lines. A couple of bugs are fixed.
- Errors in lane detection in the part of the road with brighter colored pavement. A new color thresholding filter is applied that works very well for project_video.mp4 and also works better for challenge_video.mp4 than the filter in the first submission.

3. The Goals and Steps of the Project

- 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.

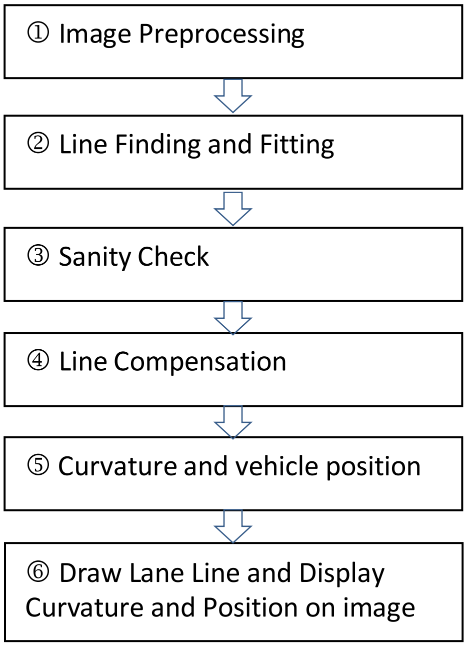
4. Lane Detector Overview

My lane detector consists of the following 6 major steps, which work as a pipeline:

All these steps/functions are called by Lane Detector that is defined as a class:

```
class LaneDetector()
```

Image pre-processing starts in the constructor of this class, which gets camera matrix (`mtx`), distortion coefficients (`dist`), and the tranform matrix (`M`) and its inverse matrix (`Minv`), as part of the initialization of this class.



① Image Preprocessing

② Line Finding and Fitting

③ Sanity Check

④ Line Compensation

⑤ Curvature and vehicle position

⑥ Draw Lane Line and Display Curvature and Position on image

① Image Preprocessing:
- Camera Calibration
- Image Un-distortion and Cropping
- Perspective Transform (top-down view)
- Color Transforming & Thresholding

② Line Finding and Fitting
- Start with sliding-window search
- Next, line fitting with pixels around lane lines found earlier

③ Sanity Check
- Radius of curvature is big enough
- The two lane lines are roughly parallel

④ Line Compensation
- The line that failed sanity check is re-established by another line that succeeded, or both failed lines are dropped.

Every step/function is illustrated in more details below.

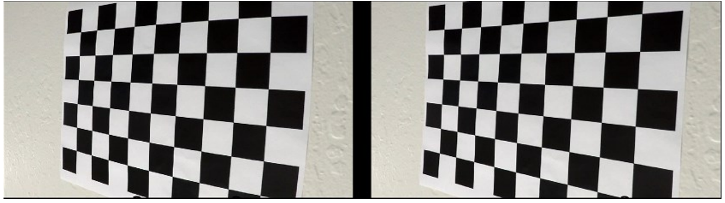
5. Image Preprocessing

This part is in `LaneDetector` class as a method called `img_preprocessing()`, line 44 to 70. The next sections follow the codes of this method.

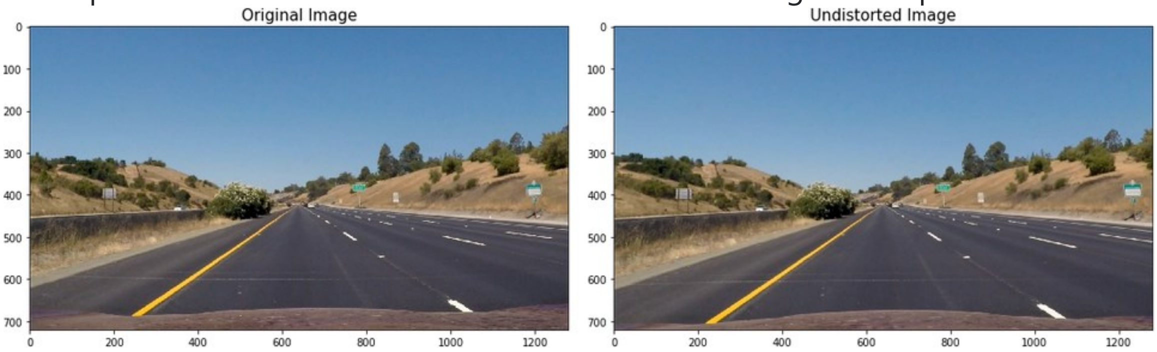
3.1 Camera Calibration and Un-distortion

Function `camera_calibration()` is an external function (line 9 to 54 in the 1st code section) that implements this step. Utilizing the crosscheck board images stored in the folder called `camera_cal`, we can find the distortion coefficients, `dist`, and the camera matrix, `mtx`, that will be used to transform 3D object points to 2D image points.

Original Image Un-distorted Image



The input video frames are undistorted first before being further processed:



3.2 Perspective Transform and Bird's Eye View

The next step is to crop the image for Region of Interest (ROI) simply by sub-slicing the image:

roi = image[467:675, 0:cols]

cols is the number of column of the original image.

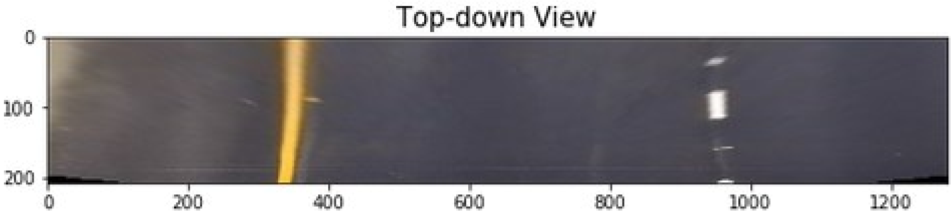
Note that the cropped image has the size of (208, 1280).

Image cropping reduces the image processing time by several folds, but the downside is that the error of line fitting over the cropped, i.e., shrunk, image is also enlarged as compared with line fitting over the image of the original size.

Then we can make a top-down view of the cropped image by calling function `top_down_view()`, in which the `M` matrix and its inverse are obtained by mapping the following four nodes defined as `src` and `dst`:

```
    src  =  np.float32([[570-6,  0],[722-6,  0],[1084-56,  rows-1],
[308-56,rows-1]])
    dst   =   np.float32([[0.25*cols,   0],   [0.75*cols,   0],
[0.75*cols,rows-1],  [0.25*cols, rows-1]])
```

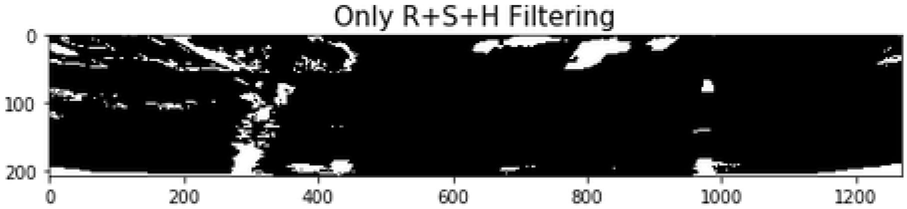
```
    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    top_down_img = cv2.warpPerspective(img, M, img_size,
flags=cv2.INTER_LINEAR)
```



3.3 Color Transforming and Thresholding

At this step, we aim at highlighting the lane lines while suppressing other parts on the road as much as possible. This is not an easy task because the brightness, contrast, sunlight, even the reflection of windshield to video camera change in very large and varied ranges from image to image, and one set of parameters of image processing method are tune to work well in one scenario but they may not work at all in another scenario.

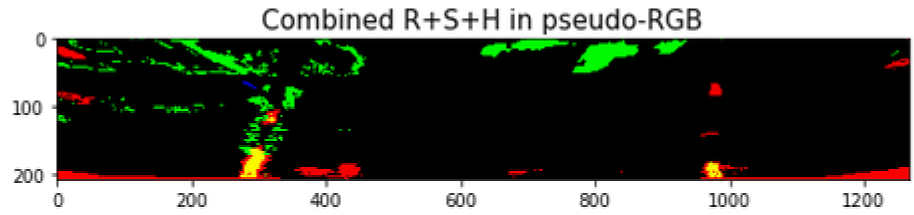
After many tests, such as using gradient thresholding by Sobel kernel, color transforming from RGB to HSV, and color thresholding, I found that a combination of Red filtering with Hue and Saturation filtering (image is converted from RGB to HSV before being split into H, S, V) can reliably single out yellow and white lane lines. This method is implanted by `combine_color()` function. This is understandable because yellow and white lane lines have more “red” ingredients than other colors. However, the parts of road that have strong sunlight on them can also be selected by this compound filter, an example as shown below.



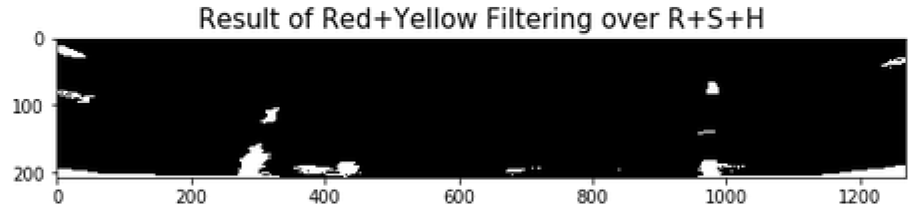
To address this issue, pseudo R, G, B color are added onto each contributor of the compound filters in `combine_color()`, i.e.,

```
color_binary = 250 * np.dstack((binary_r, binary_s, binary_h))
```

in which `binary_r` is the result of “red” filtering, `binary_s` is the result of Saturation filtering, and `binary_h` is the result of Hue filtering. Each takes one of the RGB channels, forming a single 3-channel image. The outcome looks like below:



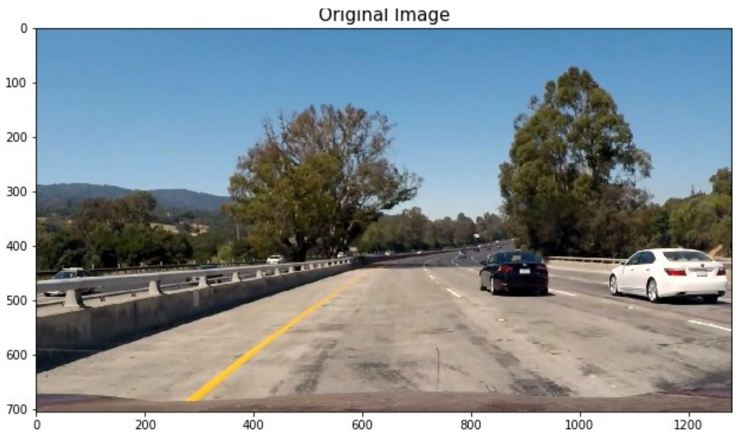
I then apply yellow+red filter, `red_yellow_filter()` , to this image and single out red and yellow while dropping other colors. The result is as below:

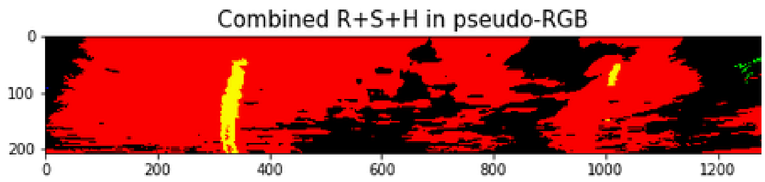


Thus we have the lane lines identified. This filter is called Red+White over R+S+H filter and was implemented in my 1st submission.

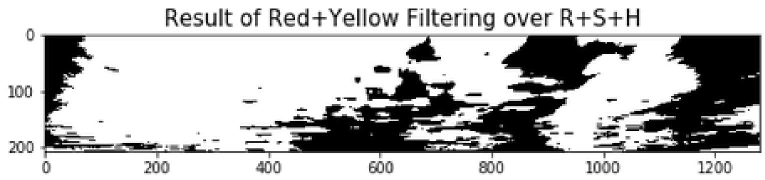
3.4 Issues and Improvements

The above color filter is not immune to the noises in the case where sunlight over the road also has strong response to filters in Red channel. See below an example.





Apparently, the current color filter cannot filter out the parts of the road in red color in the image. Below is the filtering result:



No lines can be correctly found on both sides of the road in this case.

Two approaches, combined yellow + white thresholding with brightness equalization, and combined black + white thresholding, can be exploited to address this issue and a comparison between them is made in next sections.

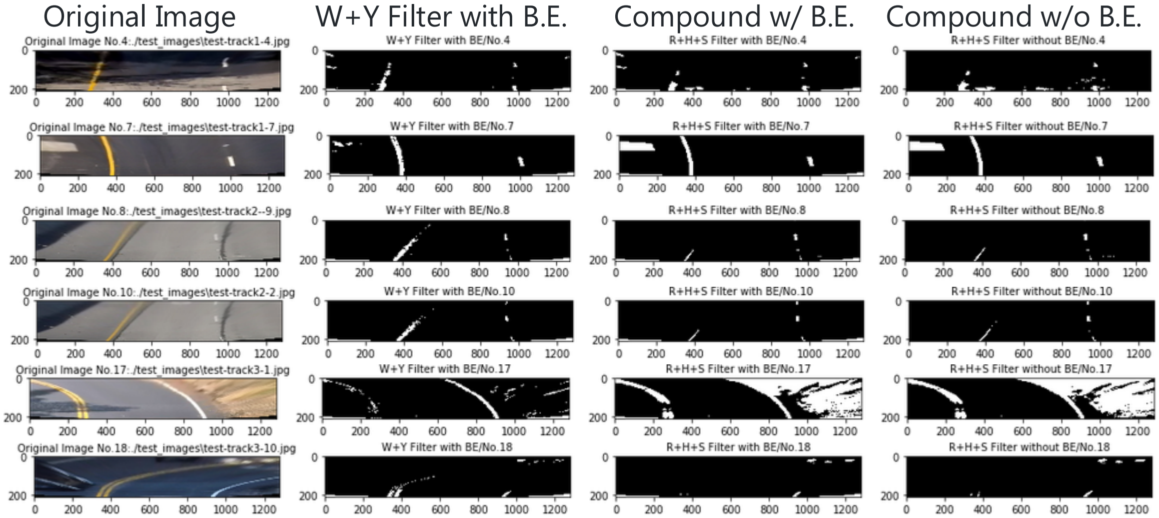
3.4 Yellow+White Thresholding with Brightness Equalization

The Red+White over R+S+H filter works fine for project_video.mp4 and challenge_video.mp4, but the threshold for red channel must be manually changed, for example, (210, 255) for project_video.mp4 and (175, 255) for challenge_video.mp4, while all the other parameters are the same for both videos. Why do they require different red-channel thresholds? Perhaps it is because the ambient lights of the roads have different strength.

So it may help if we can equalize the brightness of the images before using the above technique. This can be done by processing the V channel as following:

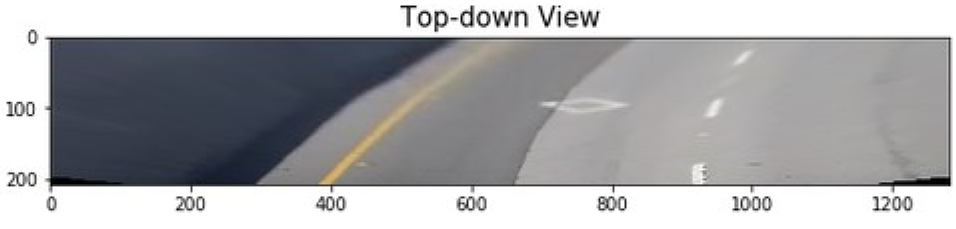
```
# Equalize brightness
    hsv = cv2.cvtColor(top_down_img, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)
    v += 255
    final_hsv = cv2.merge((h, s, v))
    img = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)
```

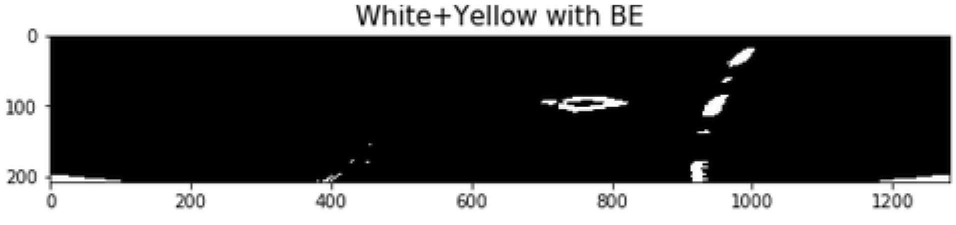
I then apply white+yellow filter to the brightness-equalized images and build a new filter `yellow_white_be(image)`. Some of the results and their comparisons with the filters introduced earlier are shown here (without changing any other parameters):


Original Image W+Y Filter with B.E. Compound w/ B.E. Compound w/o B.E.

From the pictures above we can see clear improvement of brightness equalization (B.E., the 2nd column above) on noise suppression.

I applied this filter to project_video.mp4 and it worked very well. However, it does not work well on challenge_video.mp4. Let’s look into the details on one part of the road going after the bridge. Note that challenge_video.mp4 does not seem to be taken by the same video camera as the one for project_video.mp4. The image below was transformed by the M matrix that was generated by the video frame of project_video.mp4 so the two lane lines in the top-down view do not look parallel to each other. But this does not affect lane detection.


Top-down View


White+Yellow with BE

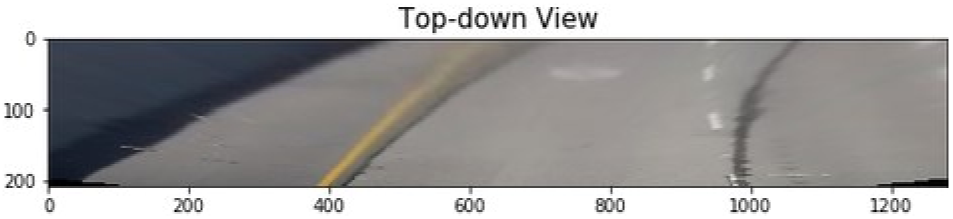
The filter fails to detect the left lane line. This is why it doesn’t work well.

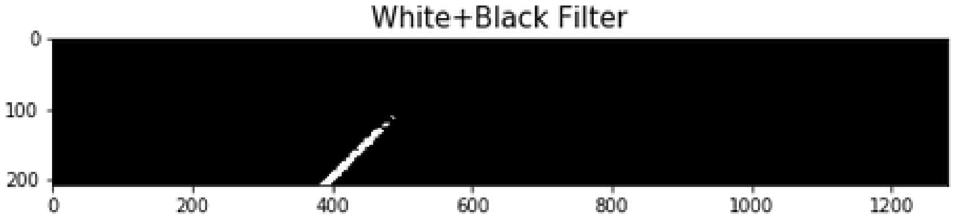
3.5 Black+White Thresholding

In the review of my first submission, reviewer provides me with a better filter with thresholds based on the B channel of LAB (for yellow lines) and the L channel of LUV (for white lines), which do a nice job of finding the lane lines without getting distracted by shadows and color changes on the road.

This filter is implemented by function `white_black_filter(image)`. It is applied to the input video frames as the sole pre-processing filter and it does work very well for project_video .mp4.

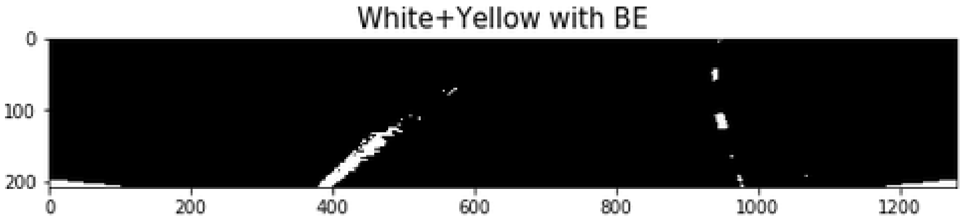
When I applied Black+White Thresholding to challenge_video.mp4, it didn’t work well. In fact, it cannot detect any lane lines at all in the 1st half of the video. Scrutinizing these image frames, I found that the filter cannot detect the right lane line. See below a part of road before bridge:





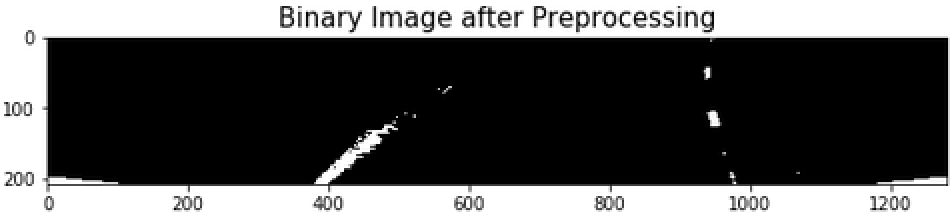
3.6 Further Enhancement to Binary Color Thresholding

If we apply filter `yellow_white_be` to the case above we can see that it works fine:



The studies above suggest that the two filters, `white_black_filter` and `yellow_white_be` , are complimentary. Merging them together to form a new filter may address the issues in detection of lane lines in challenge_video.mp4.

The final submission exploits this new "merged" filter for image preprocessing. It works well for project_video.mp4 and gets promising, even though not perfect, results on challenge_video.mp4, which is a big improvement as compared to the outcome of using the above two filters individually.



6. Line Finding and Fitting

Line finding starts with sliding window search using histogram for deciding the starting base position of each lane line. Once a line is found, it is fitted by a quadratic line using `cv2.polyfit()`, which returns the coefficients of the quadratic function. In the subsequent frames of videos, the search for the position of the lines takes place only around the quadratic lines found over previous frame, instead of throughout the whole frame.

To increase the credibility of the lines found, I first check a couple of cases.

Case 1: too many or too few non-zero pixels in the frame.

Too many such pixels suggest that there are a lot of noises surrounding the lane lines and line fitting may not be over the lanes. Too few, on the other hand, lead to line fitting not very precise.

Case 2: the base position of the two lane lines are aligned so that distance between them is around 700 pixels.

7. Sanity Check

Two things are checked. One is the radius of the curvature of each line found. The radius should be larger than 50m.

Radius of curvature is calculated by the coefficients of quadratic function as following:

```
curverad = ((1 + (2*a*y + b)**2)**1.5) / abs(2*a)
```

Another check is whether the two lines are roughly parallel. As their curvature has already been checked, we can assume that these curvatures are similar. So to check whether the two lines are parallel, we simply get their x values on top and at bottom of the cropped image and whether the distances of two lines on top and at bottom are roughly the same (i.e., their difference is in a small range).

Then we need to update the information of each line, which is achieved by updating the variables in `class Line()`.

8. Line Compensation

When one of the two lines is either not detected successfully or is abnormal (i.e., failed to pass sanity check), it continues to use the fitted line from the last video frame , which is saved in `Line.allx` (if not None, the most recent fitted x values) or `Line.bestx` (average x values).

9. Curvature and Vehicle Position

Radius of curvature is calculated as introduced above in sanity check, which is also introduced in the lecture. Vehicle position is from the base position of each lane line at the bottom of the image, which is computed by function `base_pos()`. Then the vehicle position is calculated by:

```
offset = (a*y*y + b*y + c)
# car offset from lane center
car_offset = center_position - (offset_l + offset_r) * 0.5
```

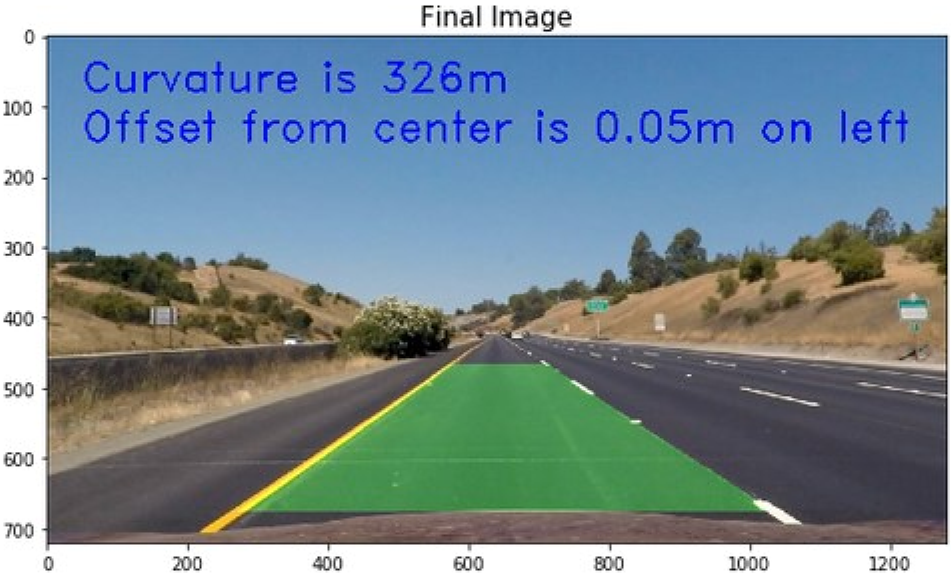
Of course, y is the value at the bottom of the cropped image) and is in meters.

10. Display Results

Finally, the lane lines are marked in the original image by transparent green polygon, with curvature and vehicle position showing up in the image as well. This is done by `self.draw_lines()` that has the following steps:

Cropped warped_binary image → plot lines →inverse top-down → paste to an blank image of the size of original image → merge with the original image already undistorted → result

The vertices of the polygon are generated by the mean of x values over each line corresponding to the evenly sliced y values along the vertical axis. Then the image with the results (which has the size of the cropped image) is perspective-transformed using `Minv` (inverse of M matrix) and pasted to the cropping positon of a blank image of the same size as the original video frame. Lastly, it is merged with the original image that has been undistorted.



11. Issues and Discussions

Issue 1 – Parameters not feasible for all environments and must be tuned manually

There are many parameters involved in image pre-processing, line detection, sanity check, etc., of lane detector and their values are assigned manually (after lots of trials). This means that there are always cases where the lane detector does not work well.

There may be two approaches to this issue. One is adaptive parameter tuning. The other is to find features that are invariant to ambient environments of the roads.

For example, I tried **Adaptive Local Threshold** introduced in paper “A Layered Approach To Robust Lane Detection At Night”, by Amol Borkar et, al, unfortunately it doesn’t work well in my project. Plus, it computes the threshold for each pixel one by one, making it a very slow process. OpenCV2 also provides a couple of adaptive thresholds but they are not useful to this project.

While it seems very difficult to come up with adaptive thresholding, we may find some invariant features in the video frames and utilize these features to detect lane lines no matter how the environment changes. For example, paper “Robust Lane Detection in Shadows and Low Illumination Conditions using Local Gradient Features” by Avishek Parajuli, et. al., presents a method for lane boundaries detection which is not affected by the shadows, illumination and uneven road conditions. This method is based upon processing grayscale images using local gradient features, characteristic spectrum of lanes, and linear prediction.

Issue 2 – prediction of lane line movements in video camera

Another issue is the prediction of the lane line in case lane detector fails. Currently I’m using the fitted lines from the previous frames but they may have large offset from the lane lines as the vehicle is moving very fast on a curve. Prediction can provide more precise “next position” of the lane line. This issue may be addressed by using Kalman Filter and Particle Filter.

Take Kalman Filter as an example. Kalman Filter requires states and measurements in order to make prediction. States are the fitted lines over the last video frames. Measurements may be from ICU of the vehicle or by computing 3D velocities from Optical Flow captured by the video camera.

Issue 3 – Missing and broken lane lines

If video camera sees just a small fraction of a lane line or even nothing as all, it is difficult or impossible for it to fit a quadratic line correctly to the lane. This is the major blocking point of my current lane detector.

However, the broken lane lines, either dashed or erased, or covered by dirt, are not really isolated. They have neighbors, i.e., the lane lines before and after them. Instead of processing video frames individually, it’s better to process them collectively so that more “global” information can be utilized to help re-construct the lane lines.

One good idea from the article on adaptive local threshold, which is the so-called **Temporal Blurring** that can extend the dashed or even dotted traffic lanes and give the appearance of a long and continuous line. More details can be found in that paper.