Advanced Lane Finding

Advanced Lane Finding Project (P4)

The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use colour 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 centre.
- 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.

Rubric Points

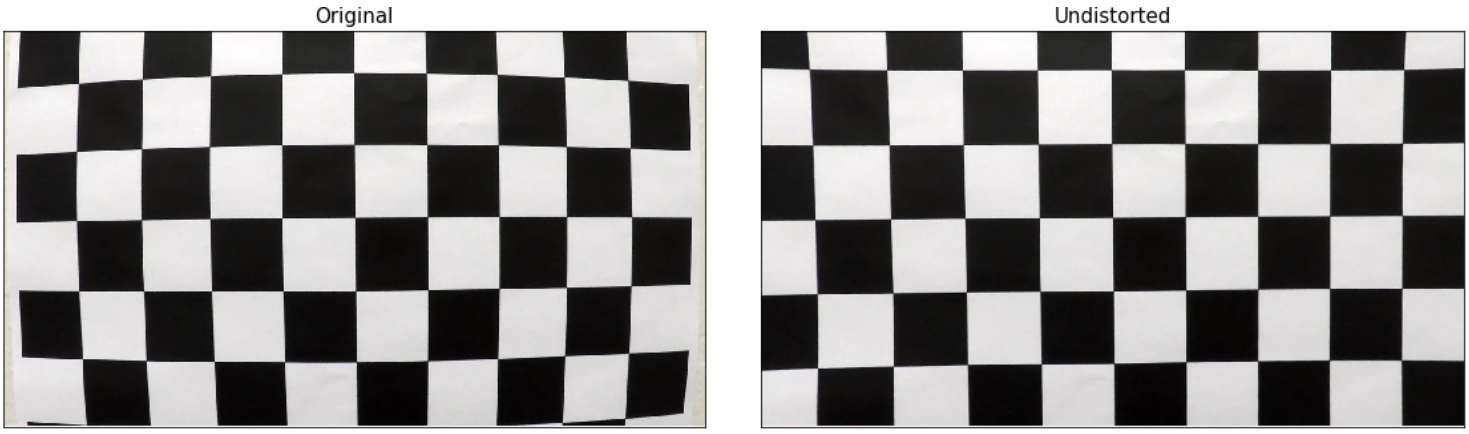
Here I will consider the Rubric points individually and describe how I addressed each point in my implementation.

Camera Calibration

Using glob library I have pulled in the filenames of images captured for camera calibration. Accordingly I have evaluated camera calibration matrix and camera distortion parameters. Here I have used imread method from ‘matplotlib.image’ library to read in the RGB images into workspace.

In order to calibrate camera, I have created an empty object points array in world frame that has 6x9 coordinate points (equal to the number of corners of the given chessboard pattern). Using OpenCV ‘findChessboardcorners()’ and taking grayscale image as input images, I have found the chessboard corners and appended in ‘imgpoints’ array (Box #2, line 19). Using the reference object points and extracted image points I have calculated the camera calibration matrix and distortion parameters. Thanks to OpenCV again for its ‘CameraCalibrate’ function that helped to carry out this task.

The camera parameters are then utilized in ‘cv2.undistort’ function to calculate the undistorted version of the distorted image as seen in (box #3, line 4). Here is the comparison of before and after the undistortion operation as seen using chessboard pattern.



Pipeline (single images)

The pipeline of a single image processing has taken place in the following steps:

- Step 1: Undistort image
- Step 2: Filter image
- Step 3: Take perspective transform
- Step 4: Detect lane lines (both left and right line)
- Step 5: Find curvature of the lines
- Step 5: Overlay information on source image for visualization

Step 1: Undistort Image

In the first step, using the camera calibration parameters I have undistorted the images. Here is an example of undistorted image from given test images.

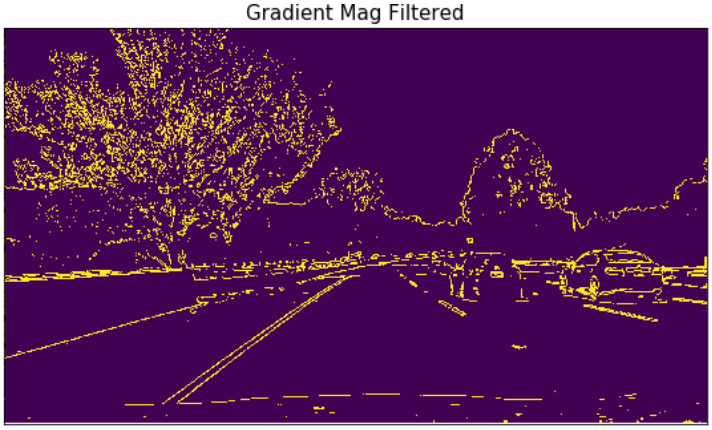
Step 2: Filter Image

In next stage, I have filtered the image for better feature extraction. Here, I have used combination of filters: gradient magnitude filter, colour saturation filter, colour lightness filter, and blue-yellow filter. Gradient filter

helped me find edges, lightness helped me to isolate white lines, blue-yellow filter helped me to isolate yellow lines, and saturation helped to cope with shades. The combination resulted in a robust filter that can handle most of the cases. Here is the results at every stage of filter.

Original

Gradient Mag Filtered

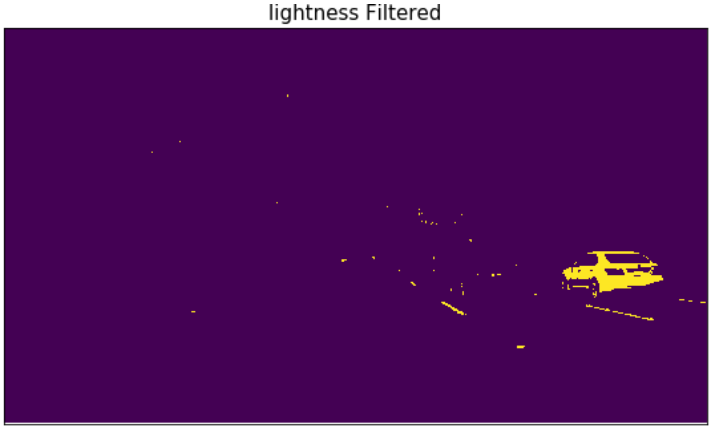


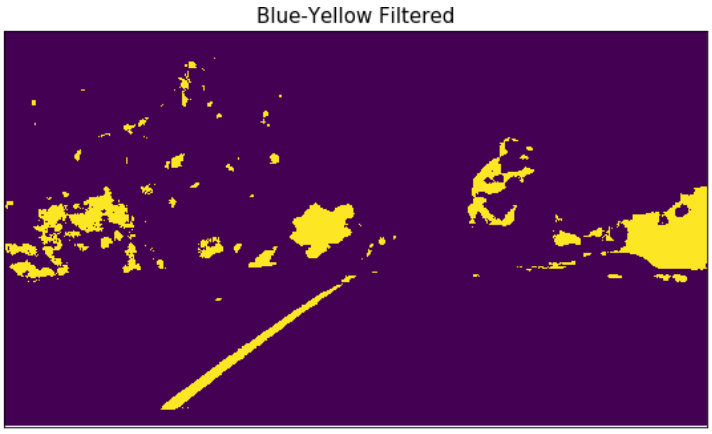
Original

Saturation Filtered

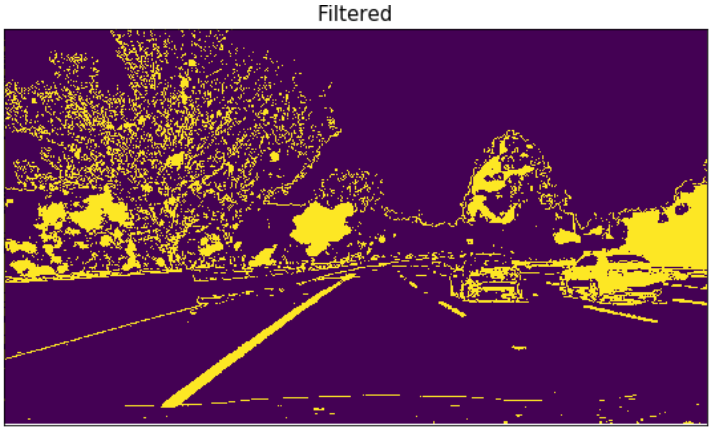
Original

lightness Filtered





Here is the final output of the combination of filters:



The filters are combined with OR operation in line 15 of box #5.

```python
output[ (grad_mag_filtered == 1) | ((s_channel_filtered == 1) | (l_cha
```

Step 3: Perspective Transformation

In order to detect and understand the lane lines better, affine transformation is adopted in this project, which helps to estimate a ‘birds-eye-view’ of the input image. In order to find the transformation matrix, I have taken a ‘straight lane line’ version of input image as benchmark and accordingly I have defined four points (to make two lines) that aligns with the lane lines. Looking at the output image, I have tuned the points, such that, in ouput image lane lines are straight. The tuned version of the four points in source image and destination image are as follows (can be found in Box #7):

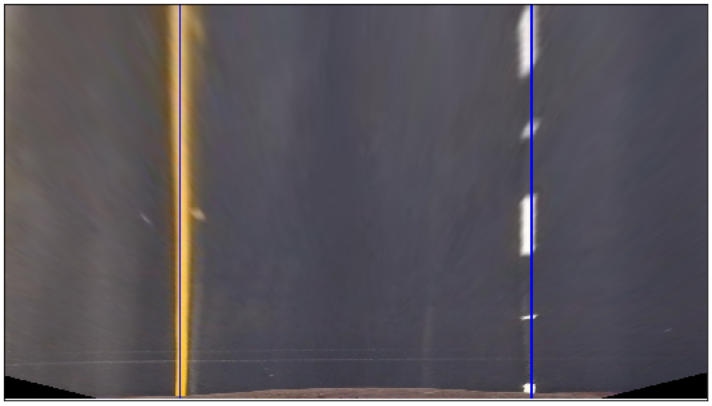
```python
s_left_b = [ (image_shape[0] * (1/6))-10, (image_shape[1] * (1/1))     ]
    s_left_t = [ (image_shape[0] * (1/2))-58, (image_shape[1] * (1/2))+100 ]
    s_right_t= [ (image_shape[0] * (1/2))+62, (image_shape[1] * (1/2))+100 ]
    s_right_b= [ (image_shape[0] * (5/6))+42, (image_shape[1] * (1/1))     ]

    d_left_b = [ (image_shape[0] * (1/4)), (image_shape[1] * (1/1)) ]
    d_left_t = [ (image_shape[0] * (1/4)), (image_shape[1] * (0/1)) ]
    d_right_t= [ (image_shape[0] * (3/4)), (image_shape[1] * (0/1)) ]
    d_right_b= [ (image_shape[0] * (3/4)), (image_shape[1] * (1/1)) ]
```

Here is an example of what the given 'straight_lines1.jpg' looks like after the points are tuned to get a reasonable affine transformation:

Original

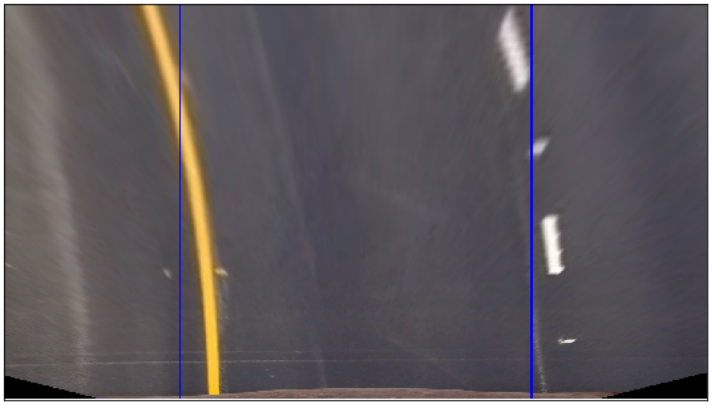
Transformed



After tuning the source and destination points, I have applied the same source and destination points to evaluate a curved line. Here is an example of the curved version of lane lines:

Original

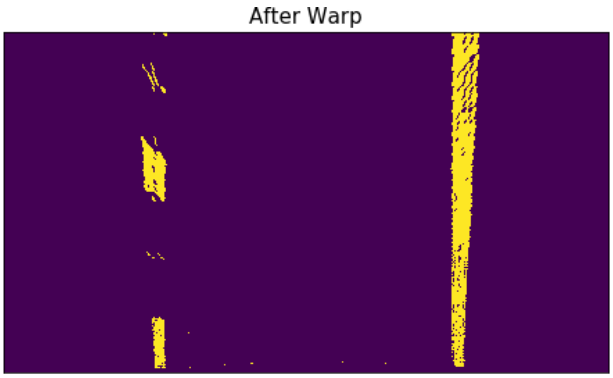
Transformed (Curved)

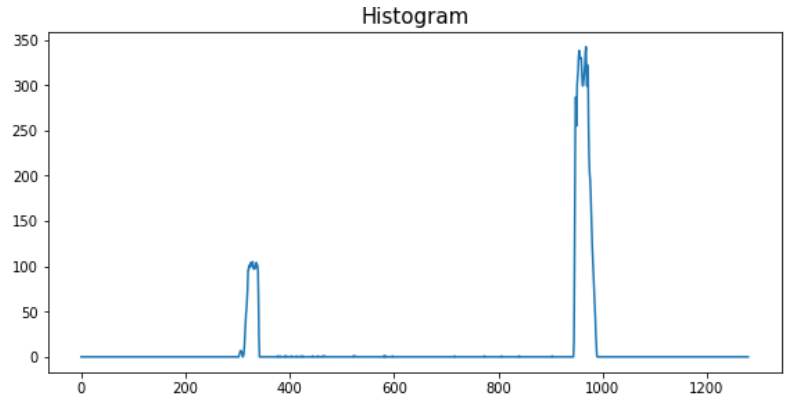


Step 4: Detect Lane Lines

In order to detect the lane lines, I have vertically divided the input image (filtered and warped) into 12 segments. For each segments, using sliding window of histogram method (as presented in classroom materials), I have identified the pixels (represents lane points) on that window. Once all frames are successfully

identified, the points are used to evaluate coefficients of a second order polynomial, which essentially represents the shape/curve of the lane lines (both left and right) as found on the *top-viewed* image. (box # 10 in notebook). Here is what histogram looks like (for this illustration purpose, I am taking a full image as a single sliding window):





At the end of the detection, I do a simple sanity check to make sure I am not reading unreasonable radius of curvature (<200m). In case, I do not get a good data, I skip the working frame and instead move ahead with results from previously known good frame.

Once a lane is successfully detected, the points of the curve are then used to draw/shade the 'lane' using 'cv2.fillpoly' function as seen in Box #11. Since the shade is found in warped/transformed image, it was required to unwarp the image so that, we can see that in a normal view. I have done that using the inverse of transformation matrix found to warp the image in the first place. Here is how it looks like after overlaying the lane between two lines.

In order to calculate radius of curvature, I have used, 'R_curve' equation found in Lecture 35 (Measuring Curvature). The findCurvature method is defined in Box #10, in the notebook. At the end of the pipeline, I overlay the curvature information onto the image as seen in the example below:

This brings the end to the image processing (lane finding) pipeline. The step by step function calls can be found in Box #15 of the notebook.

Pipeline (video)

Here's the link to my video result

Discussion

The pipeline worked well for the given project video. It has managed to work in all lighting and shade conditions found in the video. Here are the some successfully detected lanes line in presence of challenging conditions.


Left line radias: 12377.59
Right line radias: 565.77
Offset from center: -0.14


Left line radias: 1233.46
Right line radias: 301.76
Offset from center: -0.06

Overall, to me the project has been a great experience. As part of future work, I look forward to make tracking more smooth using averaging techniques or Kalman Filters. Also to make the image filtering more robust, I will try averaging (blurring technique). This may take some load off of tuning.