Advanced Lane Finding

This project is the advanced version of P1 of the Self driving Nanodegree. Here again, we are required to detect the left and right lane markings on the road, and then overlay the identified lanes on the regular camera image. The pipeline of this project is much more involved than the one used for the first project. The pipeline for this project included following steps:

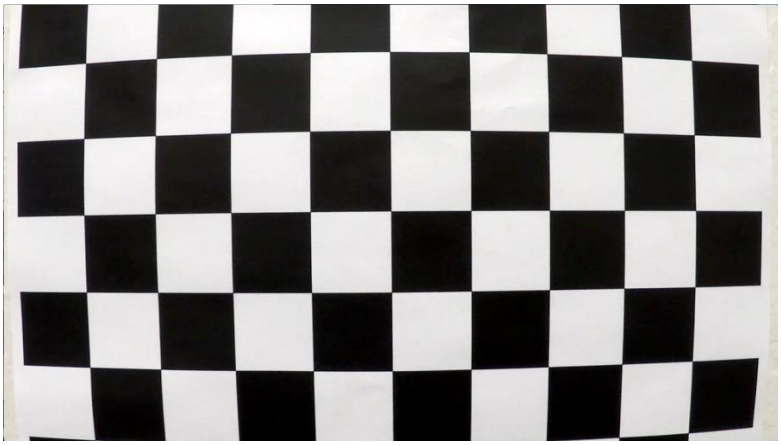
1. Calibrating camera for the possible distortion
2. Correcting distortion in each image
3. Applying filters like Sobel to detect edges (horizontal as well as vertical). Combine filters to convert the image into binary image
4. Perform the perspective transform on the images to get the bird eye's view
5. Identify the both left and right line markers in the transformed image. Fit the polynomial to the identified markers.
6. Measure the radius of curvature of the lines and the deviation of car from center of the lane
7. Overlay the polygon bounded by the left and right lines on the images and the project video.

While working on this project, I tackled all the rubrics points and the explanation of the same can be found below:

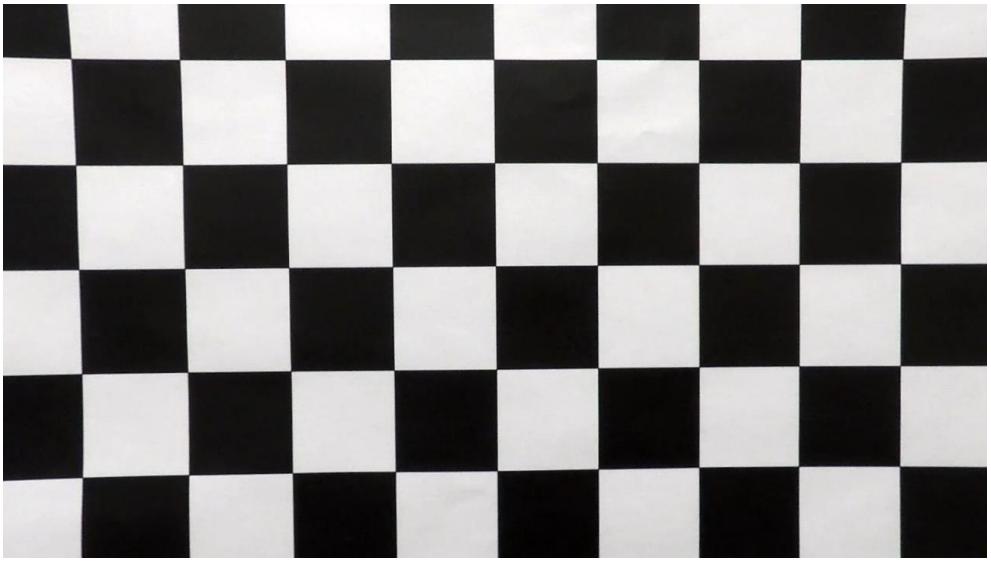
Camera Calibration

1. Have the camera matrix and distortion coefficients been computed correctly and checked on one of the calibration images as test image?
   The code for this step is included in carnd_proj.ipynb jupyter notebook between cells 2 and 8. For calibrating camera, I used the calibration images provided along with the project files. For each image, I tried to the 54(9*6) corners using cv2.findfindChessboardCorners() . Whenever corners were found, I appended them to a list called imgps. In addition, I also created the object points for each image in 3 dimensional (x,y,z) space. I fixed the z=0 for each of the object points. I appended these points into a list called objps, whenever I founded the real corners in the image.
   Post this step, I had to lists—one containing real corners found in the images and the other containing the corners that I specified to be ideal for the image. Using these two lists, I calibrated the camera using cv2. calibrateCamera() and then used the obtained camera matrix and distortion coefficients to undistort each image using cv2.undistort().

The example of a calibrated image can be seen below:



Uncalibrated Image

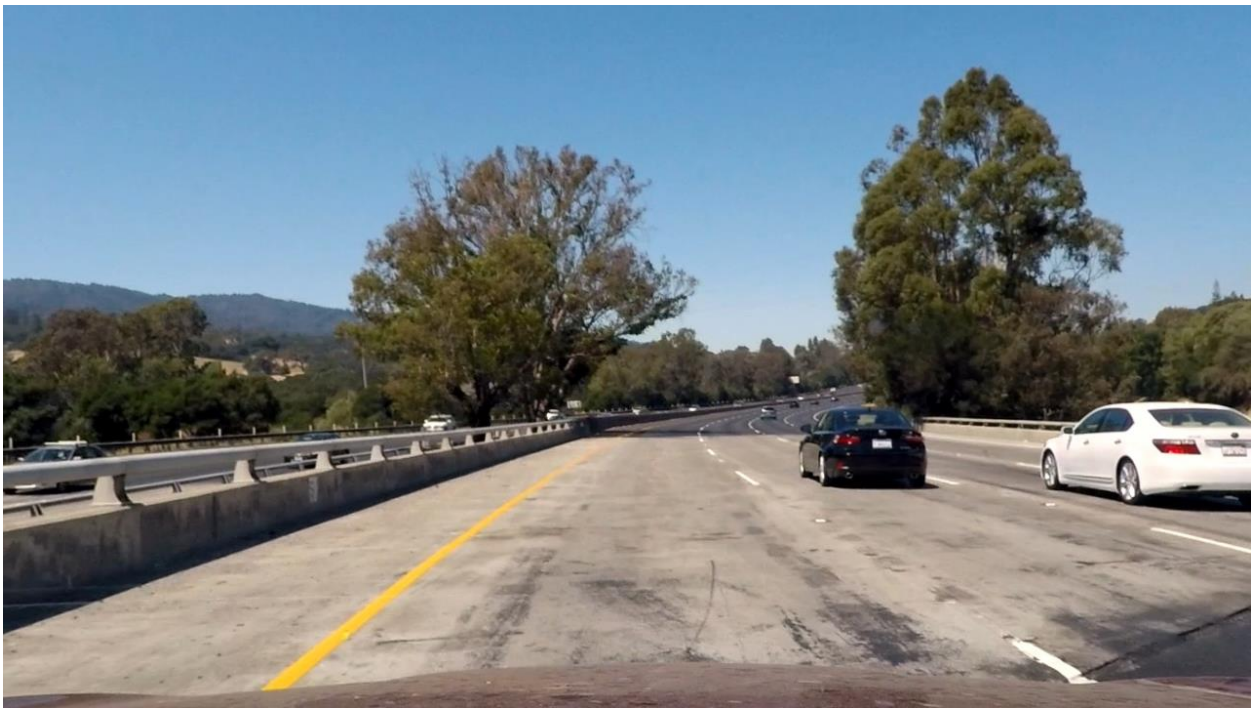


Calibrated Image

Pipeline(Single Images)

1. Has the distortion correction been correctly applied to each image?

I have used the same function cv2.undistort() with same camera matrix and distortion coefficients estimated from the chessboard images. I have then applied that function to image car image as a part of my pipeline. The example output of the undistorted car image is included in the output_images folder by the name of car_undistorted.png and is also shown below:



Undistorted Image(test1 image)

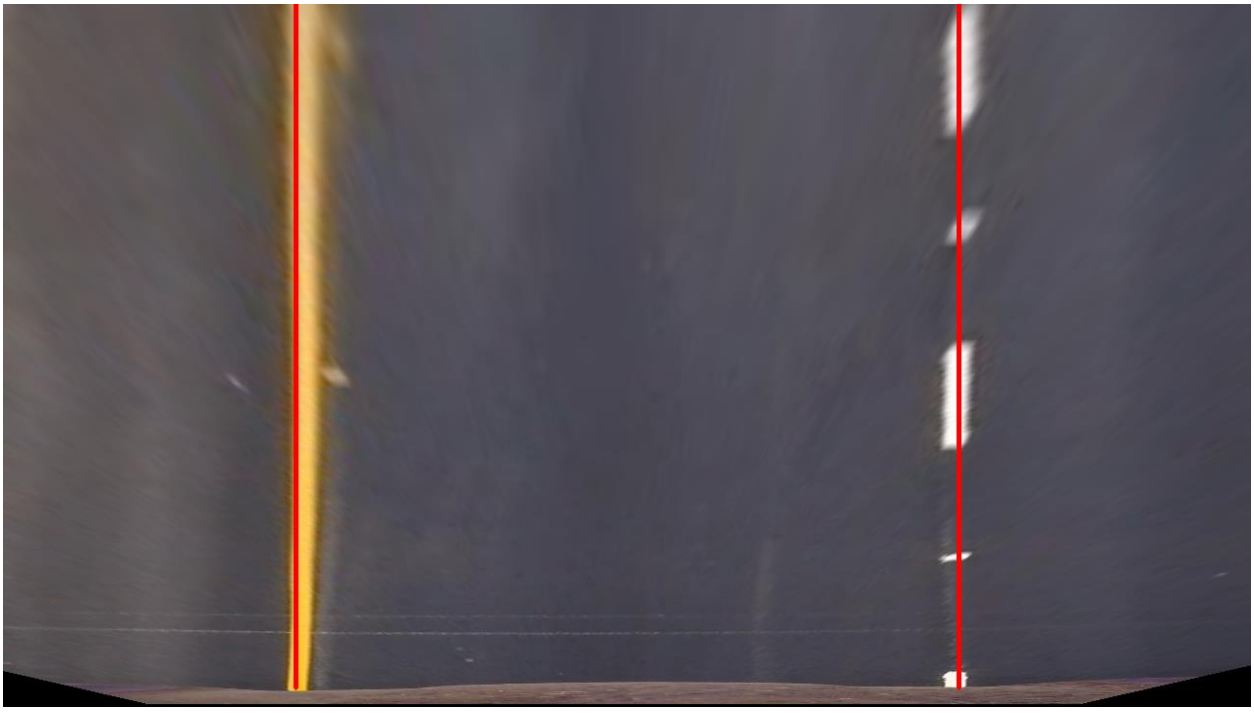
2. Has the binary image been created using color transforms, gradients or other methods?
   For creating binary image, I experimented with color transforms, gradients magnitude, gradient in x and y, and gradient direction. I created separate functions to test each of the possible gradients. The functions can be found in jupyter notebook by the name of thresholdingX, thresholdingY, thresholding_mag, thresholding_dir and thresholding_color. Among many color transforms, the one that worked best was RGB2HLS. In addition, I only used the S channel for my binary image generation.
   I found the best result across image using following filter combination – (gradient X & gradientY) | gradient_color

   For demonstration, I used test3.jpg image provided in test_images folder. The output of the binary image was saved in the output_images folder in the name of binary_image_test3.jpg and is displayed below:

3. Has the perspective transform been applied to rectify the images?
   To perform perspective transform, I manually selected the four source corners by eye balling the images. In addition, I also hard coded the four destination corners. The corners were hard coded and specified in the function perspective_transform() in the code as src = np.float32([[260,680],[580,460],[700,460],[1050,680]]) and dst = np.float32([[offset,img.shape[0]-20],[offset,0],[img.shape[1]-offset,0],[img.shape[1]-offset,img.shape[0]-20]]) with offset=300.

   An example of the image and the transformed image:

Image in regular view with the destination points



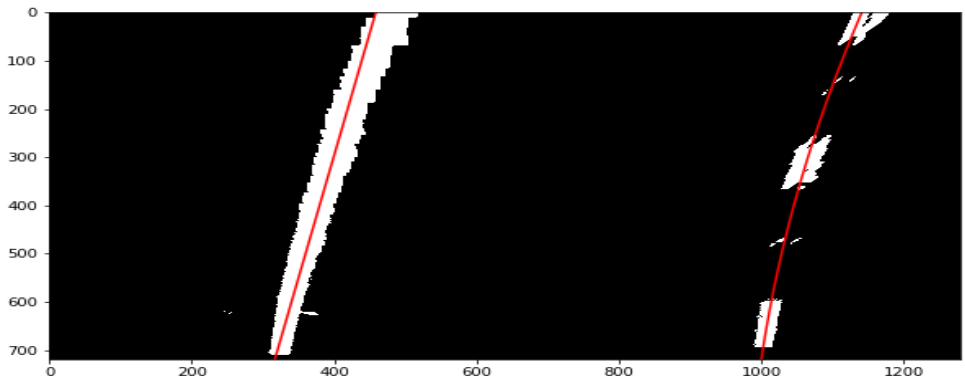
Transformed Image

4. Have the lanes pixels been identified in rectified image and fit with polynomial?

   In order to fit the polynomials on both left and right lines, it was required to identify the pixels associated with each line. For the same, I deployed window search method. I started with identifying base of each line by calculating sum over each column of the image array and then, selecting the x value where the sum was maximum in left half as left base and right half as right base. This technique is similar to the one described in the lecture videos. I started with the lecture code as my starting for finding lane pixels. In addition, I created a line class to keep track of old polynomial fit and new polynomial fit values. The line class also made sure that I used linear interpolation to smooth the changes in fitted line from one frame to another.
   To this end, I created a few helper functions and their definitions are as follows:

   a. ***image2binaryandtransform*** – takes image as an input, converts to binary as described before and then performs perspective transform
   b. ***imageReadAndTransform*** – takes image path, reads image from the path and then call the above function(image2binaryandtransform)
   c. ***fitlineAndproject*** – takes and image as input, calls imageReadAndTransform, searches for left and right line pixels, fits the polynomials and returns a new image containing only the line markings(filled polygon)
   d. ***overlayfittedline*** – Called only when working with single image. Calls fitlineAndproject and then overlays the polygon returned by fitlineAndproject onto the original image
   e. ***process_image*** -- this is the function used to process images for video. It internally calls fitlineAndproject and keeps track of the frame number of the video.

      An example of the fitted polynomial on test3.jpg image:



5. Having identified the lane lines, had the radius of curvature of the road been estimated? And the position of vehicle with respect to center in the lane?

   After identifying the lane lines, the radius of curvature and deviation of car from the center of the lane was also estimated using the same function ***fitlineAndproject***() in the code. For radius of curvature, the line was fitted again after scaling both x and y. The conversion factors used for x and y were 3.7/700 meters/pixels and 3.7/700 meters/pixels respectively.

   For measuring deviation from center of lane, I assumed that the centre of image would be the location of car. Then I calculated the center of left and right fitted line at the max y value of 720, as that would be the nearest to the car. The difference of centre of image and the center of lines was then estimated to be the deviation of vehicle from the center in the lane.

   An example of final image with radius of curvature and center deviation is

Pipeline(video)

1. Does the pipeline established with test images work to process the video?
   Yes, the same pipeline was used with addition of a function described before as well. The function used was ***process_image()*** and can be found near the end of the jupyter notebook. The function internally calls ***fitlineAndproject*** function and also, keep tracks of the frame number so as to enable average of polynomial fit over last few runs.
   The link to the final video:

Summary and Discussion of further improvements

Summary of the steps that I took for developing the pipeline is as follows:

1. Camera calibration using provided calibration images
2. Distortion correction of each image after caibrating camera using matrix and distortion coefficients
3. Applying sobel filter to detect edges in x and y direction. Converting the RGB to HLS channel image. Using only S channel of the converted image. The ideal threshold for each filter was found using trial and error on each test image. The final combination of different filters used to generate binary image was also found using trial and error.
4. Perform perspective transform on each image to get bird eye's view. The source and destination points used for transforming images was found by eye balling provide straight lines images.
5. Then, left and right line pixels were identified by using a technique similar to the one provided in the lecture videos. In addition, a line class was created to make sure that linear interpolation can be applied on each of the polynomial coefficients. Also, after the first frame, the search of the lane line pixels was narrowed down to the area identified before. All this was accomplished in ***fitlineAndproject*** function.

Possible improvements:

The pipeline broke down on the challenge video. I feel if the following improvements are made the pipeline would perform better on challenge video as well as other images.

1. Instead of combining filters and color transform by trial and error on few test images, I feel broader set of images should be used to identify better combination of filters. Infact, this step could utilize unsupervised machine learning techniques to create robust binary images
2. The source and destination points seems to be the weakness in perspective transform step. The src and destination points were hard coded and estimated by eye balling few images. I feel they can only work for certain kind of images. If the lines curve lot more than the test images, this step of the pipeline might break down.
3. Some intelligence can be applied in the ***fitlineAndproject*** function as well. The lane lines marking seem to varying a lot between frame to frame in challenge videos. This shows that the previous steps are not robust enough. In such cases, some sanity checks can be implemented in the function, which identifies if the frame was

difficult or the estimation was really poor or highly different from previous frame. These diagnostics can then be used to ignore the fit obtained in such frames and reuse the ones available from the past frame.