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 colour 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 centre.
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.

Note on Output Images and Videos:

- The output image at each stage of pipeline on each test image in the ‘test_images’ folder is in the respective folder of the test image in the ‘output_image’ folder. For e.g. If test image in ‘test_image’ folder is ‘test1.jpg’, then the its output image at each stage of pipeline as specified in examples folder is in the folder named ‘test1’. This ‘test1’ folder is in ‘output_images’ folder.
- The output of test on ‘test_video.mp4’ is ‘test_output.mp4’ and is in the folder ‘output_images’. Before testing the pipeline on ‘project_video.mp4’, I tested the pipeline on ‘P1-challenge.mp4’, the output of this is ‘P1-challenge.mp4’ and is saved in ‘output_images’ folder. Finally, I tested my pipeline on ‘project_video.mp4’, the output of this is ‘project-output.mp4’ and is stored in ‘output_images’ folder.

Note on experiments done:

- To get ‘binary_warped’ image, I tried using ‘cv2.inRange()’ function to detect yellow pixels and white pixels in the area near lane using masking. But, I didn’t get statisfactory result.
- To make the broken the lane line continuous, after getting ‘binary_warped’ image, I tried using hough transform to detect broken lines and join them using the same logic as P1. But, I’m not getting statisfactory result for it too.

All code is in file CarND-Advanced-Lane-Lines.ipynb

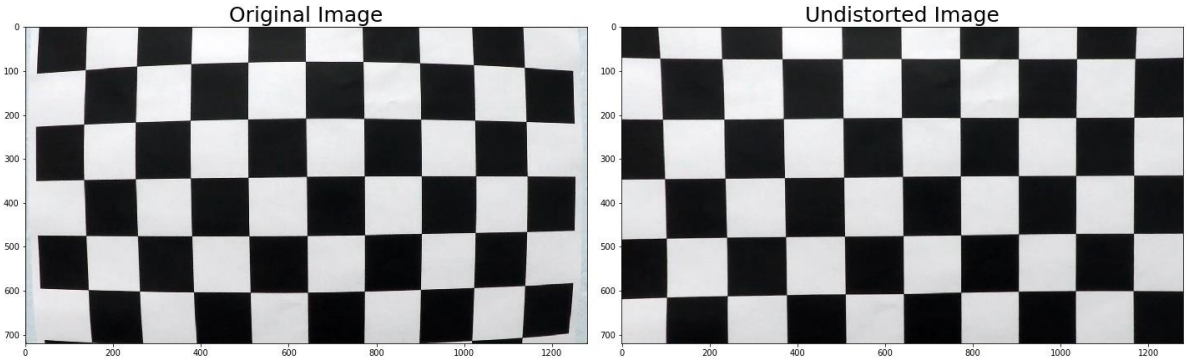
Camera Calibration

1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is in the 2nd code cell of the IPython notebook.

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



Pipeline (single images)

1. Provide an example of a distortion-corrected image.

Using calibration matrix, distortion co-efficients and cv2.undistort() I apply the distortion correction to one of the test images like this one:

2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

My aim at this step was to reduce the computation while maintaining precise detection of lane lines.

The code for my method to create a thresholded binary image includes a function called `get_warped_image()`, which appears in the 5th code cell of the IPython notebook. The `get_warped_image ()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points.

I use a combination of color and gradient thresholds to generate a binary image. Firstly, I convert the image to HLS color space. Then, I extract 'L' channel as 'l_channel' and 'S' channel as 's_channel' from image.

I use 'l_channel' and 'cv2.sobel()' function to get the gradient along X axis. After scaling this image, threshold the image with low threshold = 20 and high threshold = 100 to get a binary thresholded image 'sxbinary'.

I use 's_channel' to color threshold it with low threshold = 150 and high threshold = 255 to get a binary thresholded image 's_binary'.

Then, I combine images 'sxbinary' and 's_binary' using OR operator to get a combined binary image 'combined'.

I wanted to remove the noise and keep only the lane lines for better detection of curve. So, to remove noise, I use masking technique (same as P1). I mask the left and right lane lines by defining vertices (580, 460), (620, 460), (350, 680), (150, 680) and (710,460), (760, 460), (1200, 680), (1000, 680) respectively.

After masking the lane lines, I get a binary image 'combined' which is noise free.

Here's an example of my output for this step.

3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for my perspective transform includes a function called `get_perspective_transform()`, which appears in the 6th code cell of the IPython notebook. The ` get_perspective_transform()` function takes as input an image (`combined`) a binary image to get back output as 'binary_warped', as well as source (`image`) to get a 3-channel 'warped' images.

The code to calculate transform matrix 'M' and inverse transform matrix 'Minv' is in the 4th code cell of the IPython notebook. I chose to hardcode the source and destination points in the following manner:

```
src = np.float32(
   [[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
   [((img_size[0] / 6) - 10), img_size[1]],
   [(img_size[0] * 5 / 6) + 60, img_size[1]],
   [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])
dst = np.float32(
   [[(img_size[0] / 4), 0],
   [(img_size[0] / 4), img_size[1]],
   [(img_size[0] * 3 / 4), img_size[1]],
   [(img_size[0] * 3 / 4), 0]])
```

This resulted in the following source and destination points:

Source	Destination
585, 460	320, 0
203, 720	320, 720
1127, 720	960, 720
695, 460	960, 0

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The code to identify lane lines includes a function called `get_fits()`, which appears in the 8th code cell of the IPython notebook. The `get_fits()` function takes as inputs an image `img`.

After undistorting, thresholding and doing perspective transform, I have a binary image as an input here. I'm using **Sliding Window search** approach to get a 2nd order polynomial fit of left and right lane lines.

I first compute the midpoint, left lane line's base 'leftx_base' and right lane line's base 'rightx_base'. I do this by finding histogram along x axis. Code for this step includes a function get_values() which is in the 7th code cell of the IPython notebook, its takes as input an image 'binary_warped'.

After this, I initialize two empty list for left line and right line 'left_lane_inds' and 'right_lane_inds', which I want to fill and few parameters related to window like No. of windows 'nwindows', window height 'window_height', window width '+/- margin' and threshold to recenter window 'minpix'. Then, I extract nonzero co-ordinates in 'binary_warped' image using 'binary_warped.nonzero()' function. I then filter out x co-ordinates and y co-ordinates.

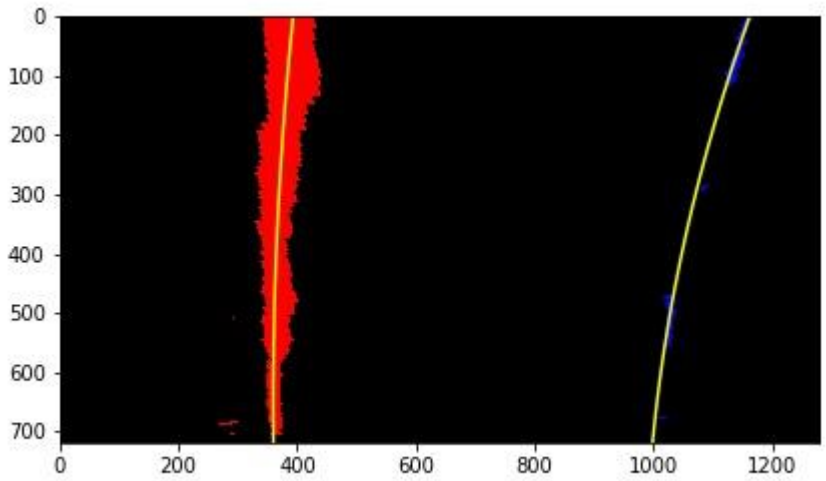
I run a loop over the no. of windows 'nwindows'. I run the loop for window on left line and window on right line simultaneously. In each iteration, I update the four points of both window, then I search for nonzero pixels within the windows. The pixels found are added to their respective list of indices. Lastly, I re-center the window, if the no. of pixels the window are more than the threshold 'minpix'.

After looping, I have left and right lines indices in 'left_lane_inds' and 'right_lane_inds' respectively.

I then extract left and right line pixel positions ‘leftx’, ‘rightx’, ‘lefty’ and ‘righty’.

Using ‘np.polyfit()’ function and these four parameters above, I find a 2nd order polynomial fit for left lane and right lane lines.

Output image for this step looks kinda like this:



5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

Steps to calculate radius of curvature are as follows:-

The code for this steps includes function get_radius_of_curvature () which is in the 9th code cell of the IPython Notebook. The function takes as input list of y co-ordinates ‘ploty’ and co-efficients of both lines left and right as ‘left_fit’ and ‘right_fit’ respectively.

Formula to calculate Radius of Curvature is as follows:

R_curve= (1+ (2*Ay*+*B*)^2) ^3/2 / |2*A*|

I’m fitting for x, so I have my 2nd order line equation in the form of *f(y),*

$x = Ay^2 + By + C.$

I find x co-ordinates of left and right lane as ‘leftx’ and ‘rightx’ corresponding to y co-ordinates using the co-efficients in ‘left_fit’ and ‘right_fit’ respectively.

I want radius of curvature in meters, to get this I define pixel to meters conversion factor for x and y as ‘xm_per_pix’ and ‘ym_per_pix’ respectively.

I convert x co-ordinates in leftx and rightx by multiplying them with ‘xm_per_pix’. Similarly, I do this for y co-ordinates too.

After this, I use cv2.polyfit () function to find new 2nd order polynomial line co-efficients list 'left_fit_cr' and 'right_fit_cr' corresponding to scaled x co-ordinates and y co-ordinates of left lane and right lane.

I want to find radius of curvature at point 'y_eval' (max value of y) i.e. at point near car.

Using y co-ordinate 'y_eval', new co-efficients and formula above, I compute the radius of curvature for left and right lane as 'left_curverad' and 'right_curverad'.

Final radius of curvature is the average of 'left_curverad' and 'right_curverad'.

Steps to find position of vehicle w.r.t to center i.e. calculate offset from center:-

The code for this steps includes function get_offset_from_center() which is in the 10th code cell of the IPython Notebook. The function takes as input a binary warped image 'binary_warped' and co-efficients of both lines left and right as 'left_fit' and 'right_fit' respectively.

Using co-efficients list 'left_fit' and 'right_fit', I find x co-ordinates of left and right line corresponding to 'y' (max value of y) as 'left_point' and 'right_point'.

I take the average of 'right_point' and 'left_point' to get 'car_midpoint'.

Then, I calculate 'pixels_off_center' by taking difference between 'road_midpoint' and 'car_midpoint'. The offset is in pixel here. To convert this to meter(s), I initialize a scaling factor 'xm_per_pix' and multiply it with 'pixels_off_center' to get the position of vehicle with respect to center in meters 'offset_from_center'.

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

The code for this step includes function get_warped_back() which is in the 11th code cell. The function takes as input undistorted image 'dest', x co-ordinates list 'left_fitx' and 'right_fitx' of left and right lane lines and y co-ordinates list 'ploty'.

I create an image 'color_warp' to draw lines on. Then, I recast the x and y points into usable format for cv2.fillPoly(). After this, I use cv2.fillpoly() to color the pixels corresponding to road in 'color_warp' image. Then, using inverse transformation matrix 'Minv' and cv2.warPerspective(), I get image 'newwarp'. Then, using cv2.weighted(), I get final Unwarped image 'warped_back'.

This image 'warped_back' is then passed through function get_result() which is in the 12th code cell of the IPython notebook, to put text on image using cv2.putText().

Here is an example of my result on a test image:

Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here are the link to test-output.mp4, P1-challenge-output.mp4 and project-output.mp4.

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?

My approach was to reduce computation in each iteration as much as possible while maintaining accuracy in detecting left and right lane lines. I decided to use gradient along X axis and color thresholding along with masking technique to accurately detect lane lines. To find line fit for first frame, I used Sliding window approach. Once we have a best fit of previous frames, we then use it to find pixel belonging to lane lines in current frame. The code for this is in the get_new_fits() function which is in the 16th code cell of IPython notebook. I defined a class Line() to keep track of Left and Right lane line with attributes best_fit, detected etc. to better process the frames.

To make the pipeline robust, I will work on better detecting detecting broken lane lines, removing noise.