Writeup for advanced lane finding projec

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 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.

(Image References)

[image1]: ./P2/ undistort.png
[image2]: ./ P2/ threshold.png
[image3]: ./ P2/ implement_threshold.png
[image4]: . / P2/ src_points.png
[image5]: ./ P2/perspective_change.png
[image6]: ./ P2/fit_polynomial.png
[video1]: ./project_video_output.mp4 "Video"

[Rubric](https://review.udacity.com/#!/rubrics/571/view) Points

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

1.Camera Calibration:

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 contained in the first code cell of the “P2.ipynb”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:

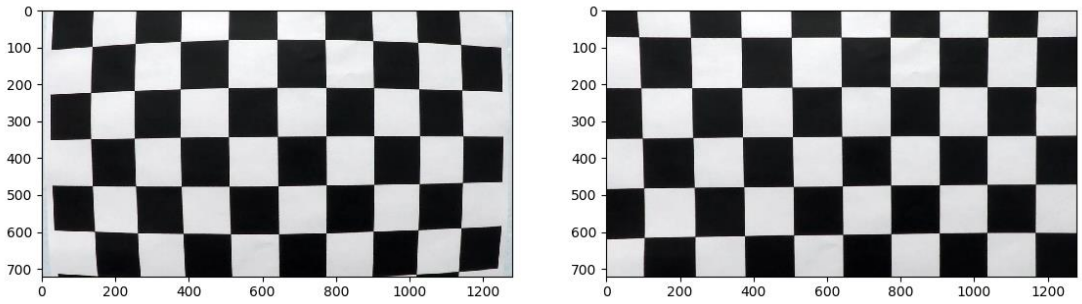

Image1

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.

I used a combination of RGB, HLS, Sobelx thresholds to generate a binary image, these three functions' code are in the 2, 3, 4 code cell of the "P2.ipynb"IPython notebook.
Then in code cell 5, I wrote a ''for" loop to see how six test images are doing at the same time. It seems r channel is great at pick up lane pixels, but it also pick up light-colored pavement. As you can see in image2, third, sixth and seventh image include some unwanted red pixels on pavement. H channel of HLS color space seems to exclude light-colored pavement pixels well. So I use "and" operator to combine them. Sobelx does a good job at finding lane lines at a distance, so I use "or " operator to combine this channel with the other two before. Image3 is what it looks like when I combine these three thresholds. (Actually I use this "for" loop to see how test images are doing though my entire project. It is a pretty good idea to see them all at once)

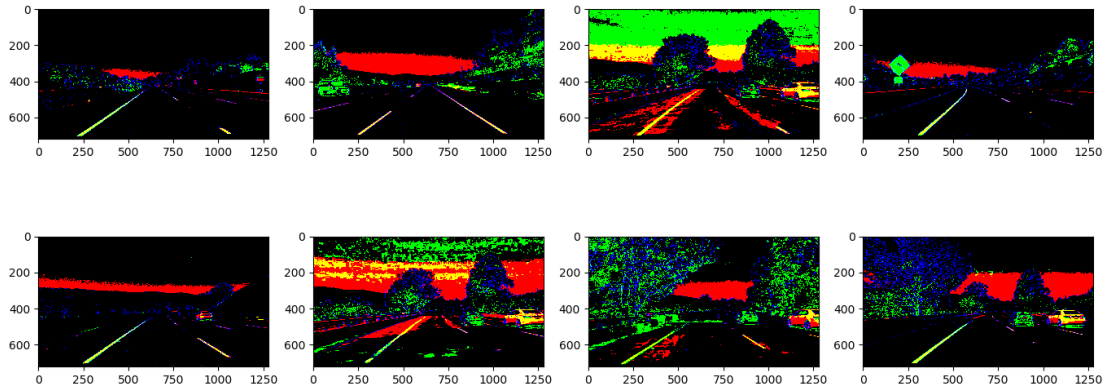

image2

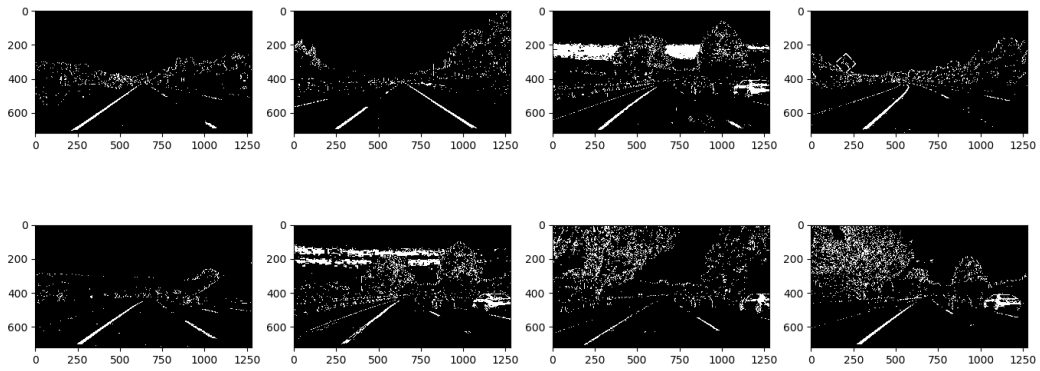

Image3

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 ` persp_trans `, which appears in sixth code cell. I choose a straight lane line picture as shown in image4, and choose four source points along the lines. These four points should appear a rectangle when looking down on the road from above, so my destination is [[300,0],[1000,0],[300,719],[1000,719]].

Image4

I verified that my perspective transform was working as expected by apply perspective transform function on eight test images and their warped images appear to be parallel, as shown in image5 .

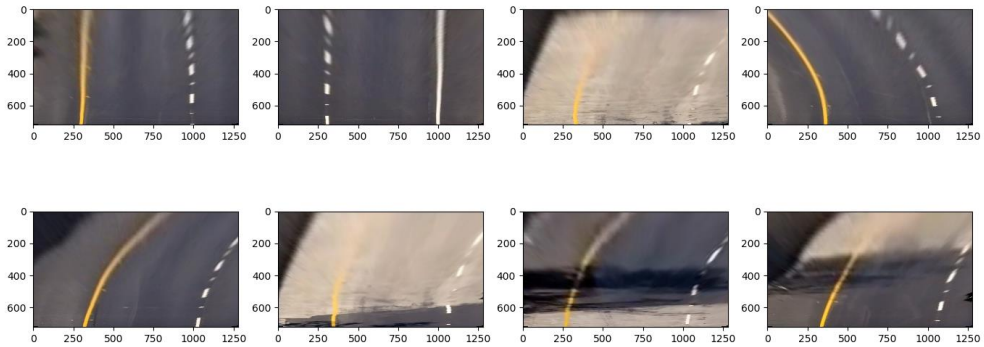

Image5

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

positions with a polynomial?

My function that finds lane pixels using sliding windows is located in code cell 8. First I used a hist function(located in code cell 7) to sum up active pixels vertical, the beginning points of lane lines are likely to be vertical, therefore have larger summation result. Then I set some parameters for sliding windows, and I use two lists “left_index” and “right_index” to store pixels indices that are belonged to lane lines. Them I use a “for” loop to iterate the location of my sliding windows, so it will move from the bottom of the image to the top, and in the meanwhile, the pixels that belonged to lane lines are stored in “left_index” and “right_index” list. If the number of pixels are larger than “minpix”, the x value of the next sliding windows would change based on the mean value of the last set of pixels that are located within the window. At last, my “lane_pixels” function return (leftx, lefty, rightx, righty, image_3d), leftx and lefty represent pixels indices that are belonged to the left lane lines, rightx, righty represent pixels indices that are belonged to the right lane lines. Image_3d is a image with sliding windows drawn on it.

Then I build a “fit_poly_3d” function to fit polynomial and visualize it, which located in code cell 9. It gets lane line pixels from “lane_pixels” function located in code cell 8. And I use polyfit function of numpy to fit a polynomial. Variable “ploty “ is generated as y value, I use “ploty” as y value to caluate x value of the polynomial that I just fit. The result is shown below.

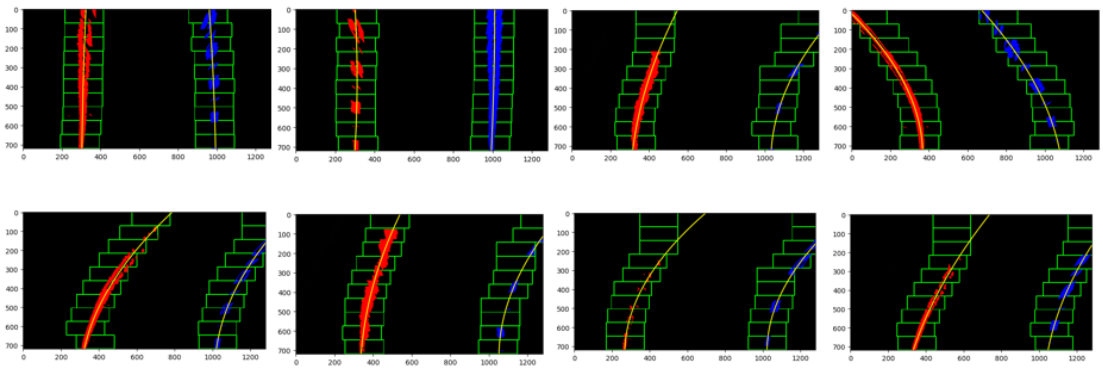

Image6

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.

My function “get_curvature” calculated the radius of curvature of the lane and the position of the vehicle with respect to center, and it is located in code cell 10. First I calculate the meter per pixel ,which is useful to convert number of pixels to meters in real world. Them I use polyfit of numpy to fit a polynomial in real world, which represent real curvature. Then I calculate curvature of the maximum y value because maximum y value is where vehicle located. Then I calculate the midpoint of two lane lines intersection with y maximum value in image, which represent vehicle position. X center point minus vehicle represent how many pixels between car center and the vehicle center. Then multiply it by “xm_per_pix” gives me real distance between vehicle center and road center.

6. Provide an example image of your result plotted back down onto the road such that the

lane area is identified clearly.

I implemented this step in lines # through # in my code in code cell 11 in the function" print_on_original". Here is an example of my result on eight test images:

Image7

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!).

My video named project_video_output.mp4, is provide inside zip file.

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 algorithm has some problem with fitting polynomial (it does well on test images, but in some frame in video it does not fit lane lines well). My guess is there is something wrong my thresholds, it does not recognized enough points to fit a polynomial. Maybe I should extract some more test images from test video to improve my threshold.
I wrote a function called "near_search" to detect lane pixels around the former polynomial, but I did not figure out how to implement it in pipeline, how do I know when it fails to detect and switch to sliding window method? And I did not figure out how to implement in a video. I think search around former frame's polynomial could save some calculation, but does it really improve result?