UDACITY

PROJECT

Advanced Lane Finding

A part of the Self Driving Car Engineer Nanodegree Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Meets Specifications

Overall great job! I don't know why I should be lenient to you as it is one of the best submission I saw Congratulations and good luck on your next project!

If you are interested more in this approach here is an excellent video from Udacity that describes almost all process:
https://www.youtube.com/watch?v=vWY8YUayf9Q

Other approaches:
http://www.vision.caltech.edu/malaa/publications/aly08realtime.pdf
http://campar.in.tum.de/pub/gackstatter2010amaa/gackstatter2010amaa.pdf

And another approach - here are several papers about lane detection with deep learning:
http://www.cv-foundation.org/openaccess/content_cvpr_2016_workshops/w3/papers/Gurghian_DeepLanes_End-To-End_Lane_CVPR_2016_paper.pdf
http://lmb.informatik.uni-freiburg.de/Publications/2016/OB16b/oliveira16iros.pdf
http://link.springer.com/chapter/10.1007/978-3-319-12637-1_57 (chapter in the book `Neural Information Processing`)
http://ocean.kisti.re.kr/downfile/volume/ieek1/OBDDBE/2016/v11n3/OBDDBE_2016_v11n3_163.pdf (in Korean, but some interesting insights can be found from illustrations)

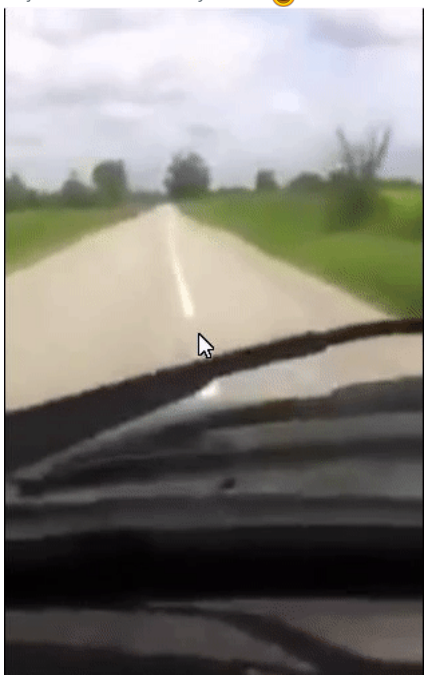
If you are interested in other approaches in computer/deep vision you can follow this repo:
https://github.com/kjw0612/awesome-deep-vision (can be useful in project 5 - vehicle detection)

If you want more info about Computer Vision - here is an excellent free book:
http://programmingcomputervision.com/downloads/ProgrammingComputerVision_CCdraft.pdf

And some other implementations of this project:
https://medium.com/towards-data-science/robust-lane-finding-using-advanced-computer-vision-techniques-mid-project-update-540387e95ed3#.kvvpejkzq
https://medium.com/@heratypaul/udacity-sdcnd-advanced-lane-finding-45012da5ca7d#.bwyelnc4h

And here is another video if you need mega challenge:

Or just for fun - Russian style roads 😄



Writeup / README

The writeup / README should include a statement and supporting figures / images that explain how each rubric item was addressed, and specifically where in the code each step was handled.

Well done with writeup! 👍
It is always important to understand advantages and limitations of your algorithm and know ways to improve it.

Camera Calibration

OpenCV functions or other methods were used to calculate the correct camera matrix and distortion coefficients using the calibration chessboard images provided in the repository (note these are 9x6 chessboard images, unlike the 8x6 images used in the lesson). The distortion matrix should be used to un-distort one of the calibration images provided as a demonstration that the calibration is correct. Example of undistorted calibration image is Included in the writeup (or saved to a folder).

Well done with camera calibration process! Here is more info about this process in documentation if you are interested:
http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html?
http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
http://docs.opencv.org/3.1.0/dc/dbb/tutorial_py_calibration.html

And very detailed Microsoft research:
http://research.microsoft.com/en-us/um/people/zhang/calib/

Pipeline (test images)

Distortion correction that was calculated via camera calibration has been correctly applied to each image. An example of a distortion corrected image should be included in the writeup (or saved to a folder) and submitted with the project.

`undistort` function is successfully used for images.

Remember that usually Python is used by self-driving car engineers for prototyping and final algorithms are usually implemented in C/C++ as it is faster than Python and almost all data are handled in self-driving car in real time. Here you can find article about distortion correction in C++:
https://medium.com/@mimoralea/but-self-driving-car-engineers-dont-need-to-know-c-c-right-3230725a7542#.ge488ni33

Note please that you will apply C++ in term 2 (though in other context).

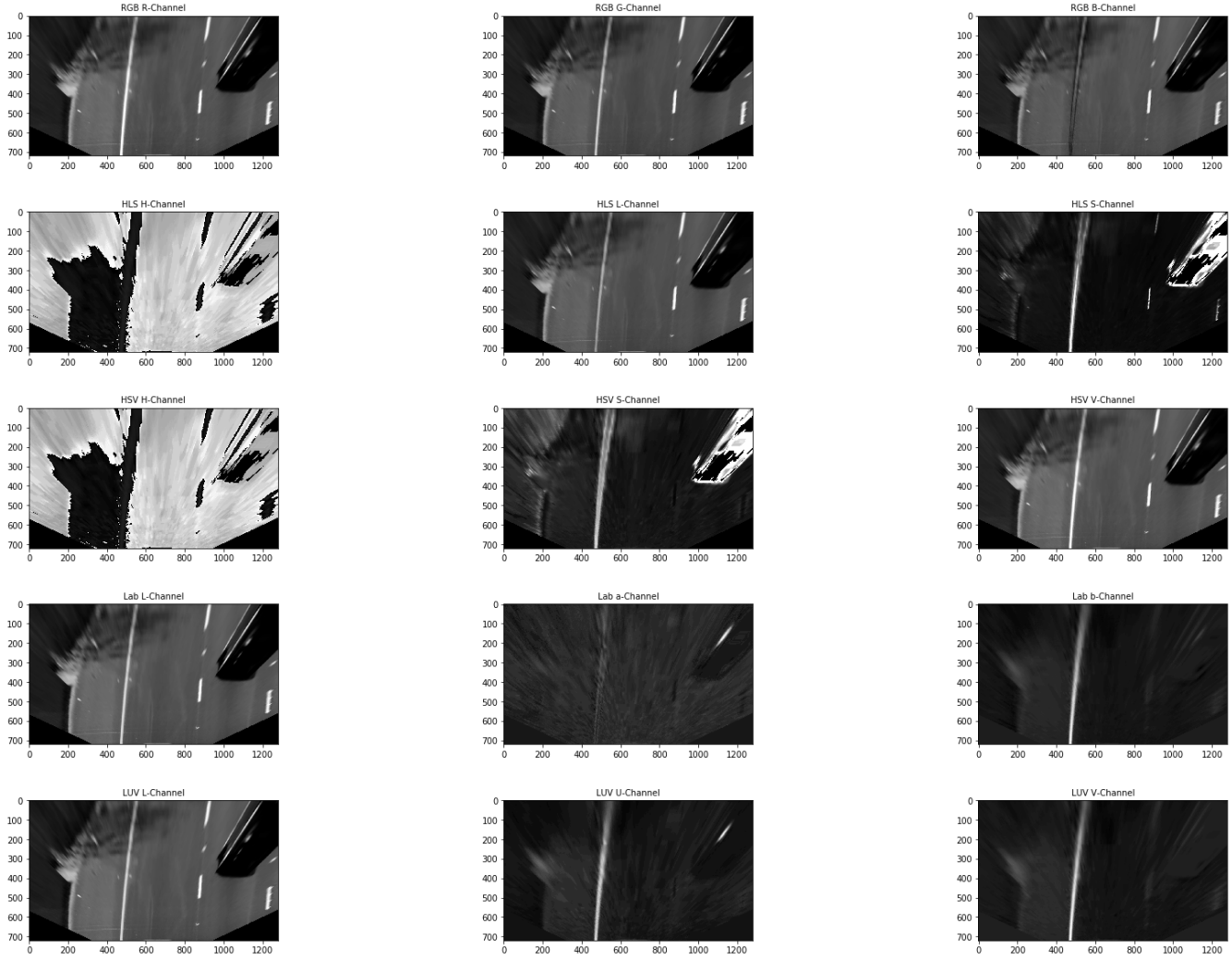
A method or combination of methods (i.e., color transforms, gradients) has been used to create a binary image containing likely lane pixels. There is no "ground truth" here, just visual verification that the pixels identified as part of the lane lines are, in fact, part of the lines. Example binary images should be included in the writeup (or saved to a folder) and submitted with the project.

Well done with color transforms and gradient methods to create binary images for further lane detection!

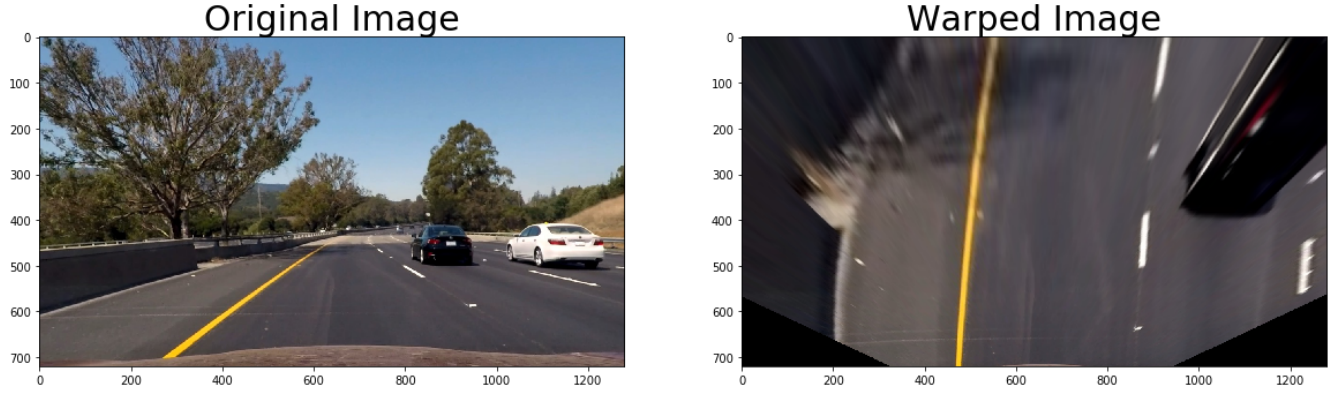
Briefly about lines, colorspaces, and channels in different colorspaces:

- S-channel of HLS colorspace is good to find the yellow line and in combination with gradients, it can give you a very good result.
- R-channel of RGB colorspace is pretty good to find required lines in some conditions.
- L-channel of LUV colorspace for the white line.
- B-channel of LAB colorspace may be good for the yellow line.
  You just need to choose good thresholds for them.

Here are some examples of mentioned channels (in birds-eye view for the original image provided below):



Origianl image for channels shown above:



Note please also that gradients can give big noise in some conditions (for example with shadowed road or when lightning conditions changes drastically).
If you use Sobel remember that kernel size is limited to 1, 3, 5, or 7 values only:
http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html#cv.Sobel

Here is a code to build a widget for parameters fine tuning (for jupyter notebook only):

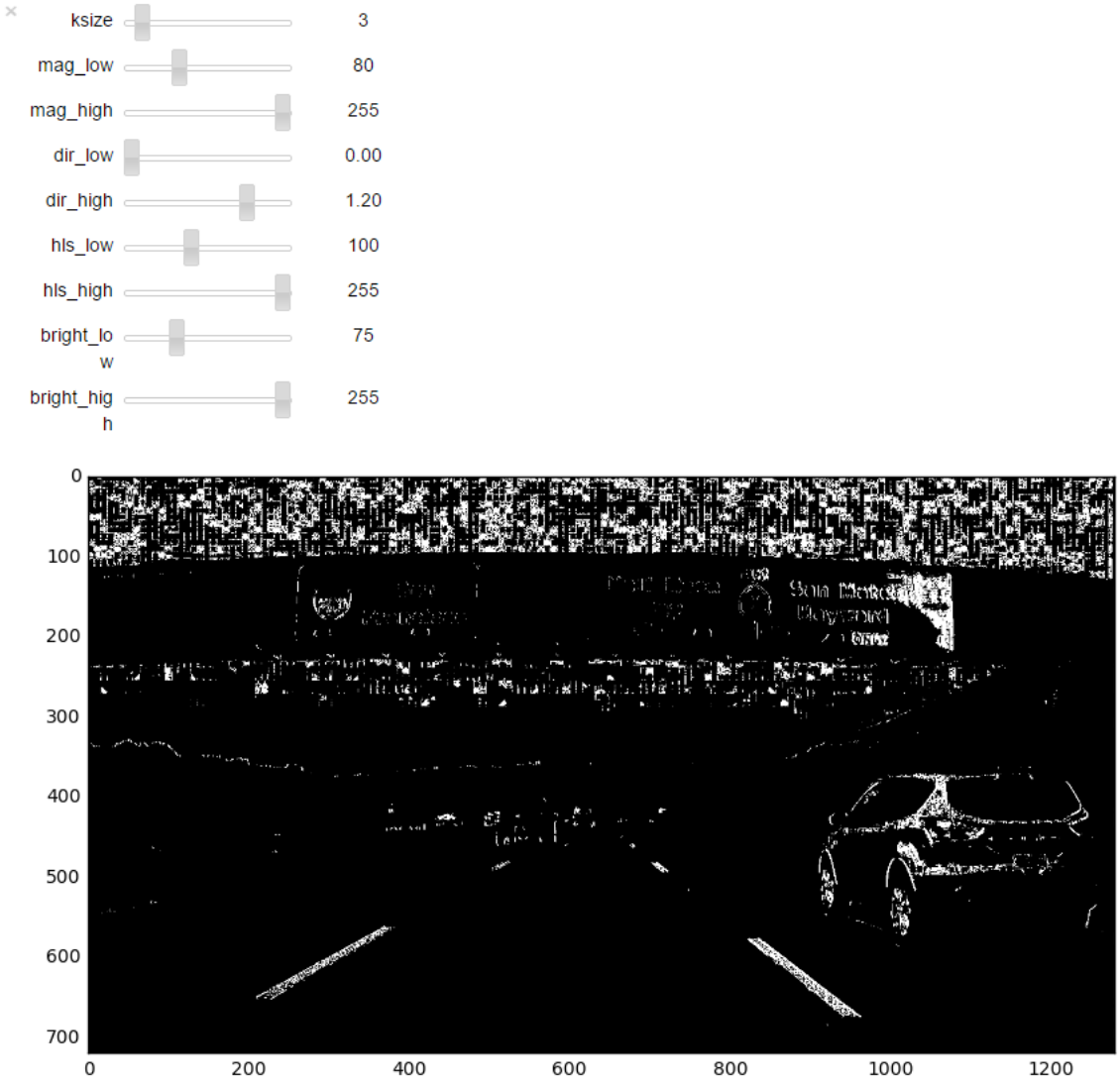
```
from IPython.html import widgets
from IPython.html.widgets import interact
from IPython.display import display

image = mpimg.imread('path_to_your_image')
image = cv2.undistort(image, mtx, dist, None, mtx)

def interactive_mask(ksize, mag_low, mag_high, dir_low, dir_high, hls_low, hls_high, bright_low, bright_high):
    combined = combined_binary_mask(image, ksize, mag_low, mag_high, dir_low, dir_high,\
                                    hls_low, hls_high, bright_low, bright_high)
    plt.figure(figsize=(10,10))
    plt.imshow(combined,cmap='gray')

interact(interactive_mask, ksize=(1,31,2), mag_low=(0,255), mag_high=(0,255),\
         dir_low=(0, np.pi/2), dir_high=(0, np.pi/2), hls_low=(0,255),\
         hls_high=(0,255), bright_low=(0,255), bright_high=(0,255))
```

Where `combined_binary_mask` function returns binary combination of different color and/or gradient thresholds and

```
ksize,
mag_low,
mag_high,
dir_low,
dir_high,
hls_low,
hls_high,
bright_low,
bright_high
```

are parameters for thresholding (if you use other, you can change this part of code).

As a result, you will receive something like that and will be able to fine-tune parameters on the fly:



and will be able to see an impact of each parameter on the resulting binary image with defined combination of thresholds.

Here is similar decision for plain python (without notebook):

```
def adjust_threshold():
    image1 = cv2.imread("./test_images/test1.jpg")
    image2 = cv2.imread("./test_images/test2.jpg")
    image3 = cv2.imread("./test_images/test3.jpg")
    image4 = cv2.imread("./test_images/test4.jpg")
    image5 = cv2.imread("./test_images/test5.jpg")
    image6 = cv2.imread("./test_images/test6.jpg")

    image = np.concatenate((np.concatenate((image1, image2), axis=0),np.concatenate((image3, image4), axis=0)), axis=1)
    image = np.concatenate((np.concatenate((image5, image6), axis=0), image), axis=1)

    threshold = [0 , 68]

    kernel = 9

    direction = [0.7, 1.3]
    direction_delta = 0.01

    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)


    cv2.namedWindow('test', cv2.WINDOW_NORMAL)

    cv2.waitKey(0)
```

```
    while True:
        key = cv2.waitKey(1000)
        print("key = ", key)
        if key == 55: # key "Home"
            if threshold[0] > 0:
                threshold[0] = threshold[0] - 1
            if direction[0] > 0:
                direction[0] = direction[0] - direction_delta

        if key == 57: # key "PgUp"
            if threshold[0] < threshold[1]:
                threshold[0] = threshold[0] + 1

            if direction[0] < direction[1] - direction_delta:
                direction[0] = direction[0] + direction_delta

        if key == 2424832: # left arrow
            if threshold[1] > threshold[0]:
                threshold[1] = threshold[1] - 1

            if direction[1] > direction[0] + direction_delta:
                direction[1] = direction[1] - direction_delta

        if key == 2555904: # right arrow
            if threshold[1] < 255:
                threshold[1] = threshold[1] + 1

            if direction[1] < np.pi/2:
                direction[1] = direction[1] + direction_delta

        if key == 49: # key "End"
            if(kernel > 2):
                kernel = kernel - 2
        if key == 51: # key "PgDn"
            if(kernel < 31):
                kernel = kernel + 2

        if key == 27: # ESC
            break

        binary = np.zeros_like(image)
        binary[(gray >= threshold[0]) & (gray <= threshold[1])] = 1

        cv2.imshow("test", 255*binary)
        print(threshold)
        print(direction)
        print(kernel)
```

You will be able to tune different parameters using keypress. The script works in an infinite loop. Instead of this plain python, you can also use cv2 user interface API for this purpose:
http://docs.opencv.org/2.4.11/modules/highgui/doc/user_interface.html#

One more approach is to use a voting system for the final thresholding instead of the logical combination of different binary images. When the threshold is very clean like B channel from LAB space, it will have a relatively high confidence number. At the end, all the confidence number are summed up and compared with a total threshold. However, this method takes more run time than simply taking some combinations:

```
    conf_1 = 1
    conf_2 = 2
    conf_3 = 3
    threshold_vote = 3
    addup = img_sobelAbs*conf_1 +img_sobelMag*conf_1 +img_SThresh*conf_1 +img_LThresh*conf_2 +img_sobelDir*conf_1 + img_BThr
esh*conf_3

    combined[(addup >= threshold_vote)]=1
```

Where `img_sobelAbs`, `img_sobelMag`, `img_SThresh`, `img_LThresh`, `img_sobelDir`, `img_BThresh` are different color or sobel thresholds.

Here you can find an interesting article about other methods (LDA and LLC) to detect lines in different lighting conditions:
https://otik.uk.zcu.cz/bitstream/handle/11025/11945/Jang.pdf?sequence=1

OpenCV function or other method has been used to correctly rectify each image to a "birds-eye view". Transformed images should be included in the writeup (or saved to a folder) and submitted with the project.

Good job with "birds-eye view"!

Remember that the next step to improve this part of algorithm is to take into account road slope and use dynamic points to transform image to "birds-eye view".

Here are several interesting papers about obtaining and usage of birds-eye view:
http://www.ijser.org/researchpaper%5CA-Simple-Birds-Eye-View-Transformation-Technique.pdf
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3355419/
https://pdfs.semanticscholar.org/4964/9006f2d643c0fb613db4167f9e49462546dc.pdf
https://pdfs.semanticscholar.org/4074/183ce3b303ac4bb879af8d400a71e27e4f0b.pdf

Methods have been used to identify lane line pixels in the rectified binary image. The left and right line have been identified and fit with a curved functional form (e.g., spine or polynomial). Example images with line pixels identified and a fit overplotted should be included in the writeup (or saved to a folder) and submitted with the project.

Well done using histogram method to find right and left lane lines!

Here are some good links for other methods described:
https://www.researchgate.net/publication/257291768_A_Much_Advanced_and_Efficient_Lane_Detection_Algorithm_for_Intelligent_Highway_Safety
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5017478/

Algorithm based on current project😄:
https://chatbotslife.com/robust-lane-finding-using-advanced-computer-vision-techniques-46875bb3c8aa#.l2uxq26sn

Well done with sanity check implemented! The following validation criteria also can be used to remove incorrect lines:

- lane lines have the same concavity
- ratio of lines curvature is not too big (be careful here with straight lane)
- distance between left and right lines at the base of the image is roughly the same as at the top of the image (in birds-eye view)
- lane curvature, distance from the center, polynomial coefficients and so on.. don't differ a lot from the same values from the previous frame

Here the idea is to take the measurements of where the lane lines are and estimate how much the road is curving and where the vehicle is located with respect to the center of the lane. The radius of curvature may be given in meters assuming the curve of the road follows a circle. For the position of the vehicle, you may assume the camera is mounted at the center of the car and the deviation of the midpoint of the lane from the center of the image is the offset you're looking for. As with the polynomial fitting, convert from pixels to meters.

Well done with radius of curvature!

Here you can find more theory behind method to find radius of curvature for polynomial if you missed this link in lessons:
http://www.intmath.com/applications-differentiation/8-radius-curvature.php

Also I have noticed that you use the following code snippet that was provided by Udacity to convert pixels in meters on x-axis:

```
xm_per_pix = 3.7/700
```

`700` here is a width of lane in pixels.

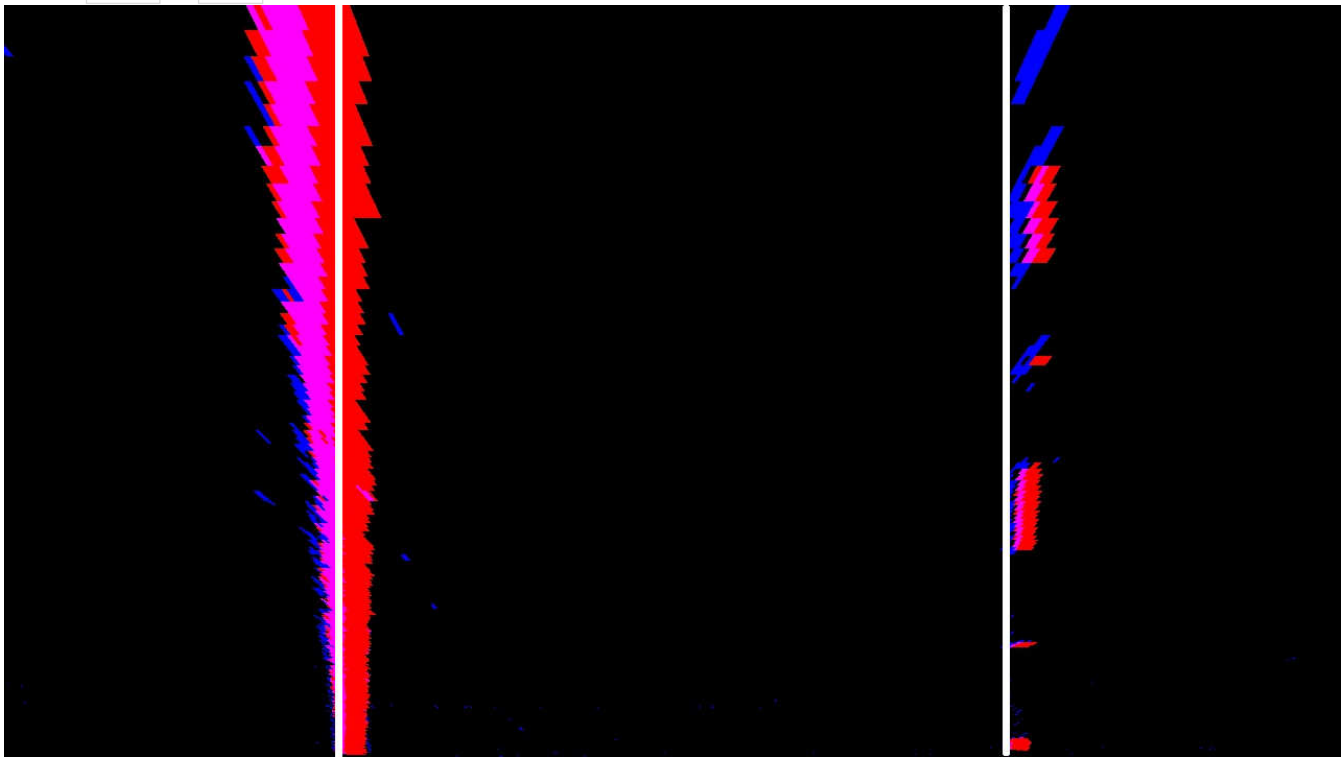
But lane width in pixels on your birds-eye view image can differ from 700px. I would recommend you to determine lane width in pixels using previously determined polynomial coefficients and use this width to translate it to meters.
For example:

```
left_c = left_fit[0] * image_size[0] ** 2 + left_fit[1] * image_size[0] + left_fit[2]
right_c = right_fit[0] * image_size[0] ** 2 + right_fit[1] * image_size[0] + right_fit[2]
width = right_c - left_c
xm_per_pix = 3.7 / width
```

where:

- `left_fit` - coefficients for left line
- `right_fit` - coefficients for right line
- `image_size` - size of image

Note please that the same you can make for y-coordinate. You know that dash line is about 3 meter. You can calculate number of dashed lines in wrapped image and then convert them to meters. It seems you have almost `7` dashed lines (2 colored in white and 5 spaces - see image below) - they are placed in `720` pixels, so you can use the following coefficient `3*7/720`, not `30/720`.



This will make your calculations more accurate.

The fit from the rectified image has been warped back onto the original image and plotted to identify the lane boundaries. This should demonstrate that the lane boundaries were correctly identified. An example image with lanes, curvature, and position from center should be included in the writeup (or saved to a folder) and submitted with the project.

Pipeline (video)

The image processing pipeline that was established to find the lane lines in images successfully processes the video. The output here should be a new video where the lanes are identified in every frame, and outputs are generated regarding the radius of curvature of the lane and vehicle position within the lane. The pipeline should correctly map out curved lines and not fail when shadows or pavement color changes are present. The output video should be linked to in the writeup and/or saved and submitted with the project.

Excellent pipeline implementation!

Note please that besides (or inside) `Line` class mentioned in lecture you can try to use exponential smoothing between current and previous values:

```
def smooth(self, prev, curr, coeficient = 0.4):
        '''
         exponential smoothing
        :param prev: old value
        :param curr: new value
        :param coeficient: smoothing coef.
        :return:
        '''
        return curr*coeficient + prev*(1-coeficient)
```

https://en.wikipedia.org/wiki/Exponential_smoothing

This can be applied to radius of curvature and/or to polynomial coefficients. Just pass to `smooth` method previous (`prev`) and current (`curr`) values you want to smooth.

Also for debugging purposes if you add some sanity check/validation criteria mentioned above you can save images where lines are not detected for further analysis and fine tune your algorithm particular for these images. You can use for it the following code for example:

```
if here_is_some_condition_that_indicates_that_sanity_check_fails:
    # found problematic image - cannot find the line at all, save it for later analysis
    fname = str(random.randint(0, 1000000)) + ".jpg"
    outImg = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
    cv2.imwrite(fname, outImg)
    return img
```

If this method is perfect to tune parameters for current video, it is not good (if not say bad at all) to generalize it for all possible situation on the road. Deep Learning techniques is much better for generalization!

Remember that in real world you should detect lane in real time. So it can be useful to measure time that take your algorithm (for example how many seconds/milliseconds you spend on one frame) and adjust it to improve these values. But also note please that Python is usually used for prototyping in such class of tasks but in real car where you should process data in real time some precompiled language (C/C++ for example) are used.

Here is a very detailed paper about it:
http://cdn.intechopen.com/pdfs/46518.pdf

Discussion

Discussion includes some consideration of problems/issues faced, what could be improved about their algorithm/pipeline, and what hypothetical cases would cause their pipeline to fail.

Well done with discussion!

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

Student FAQ