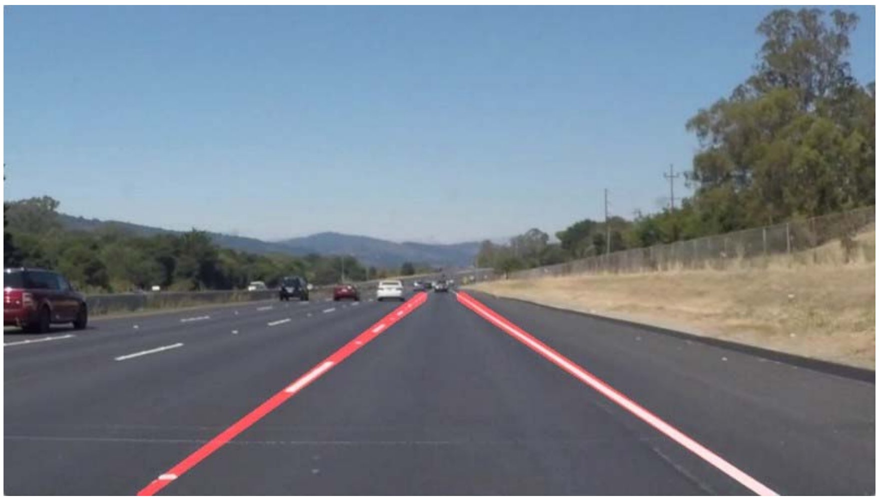
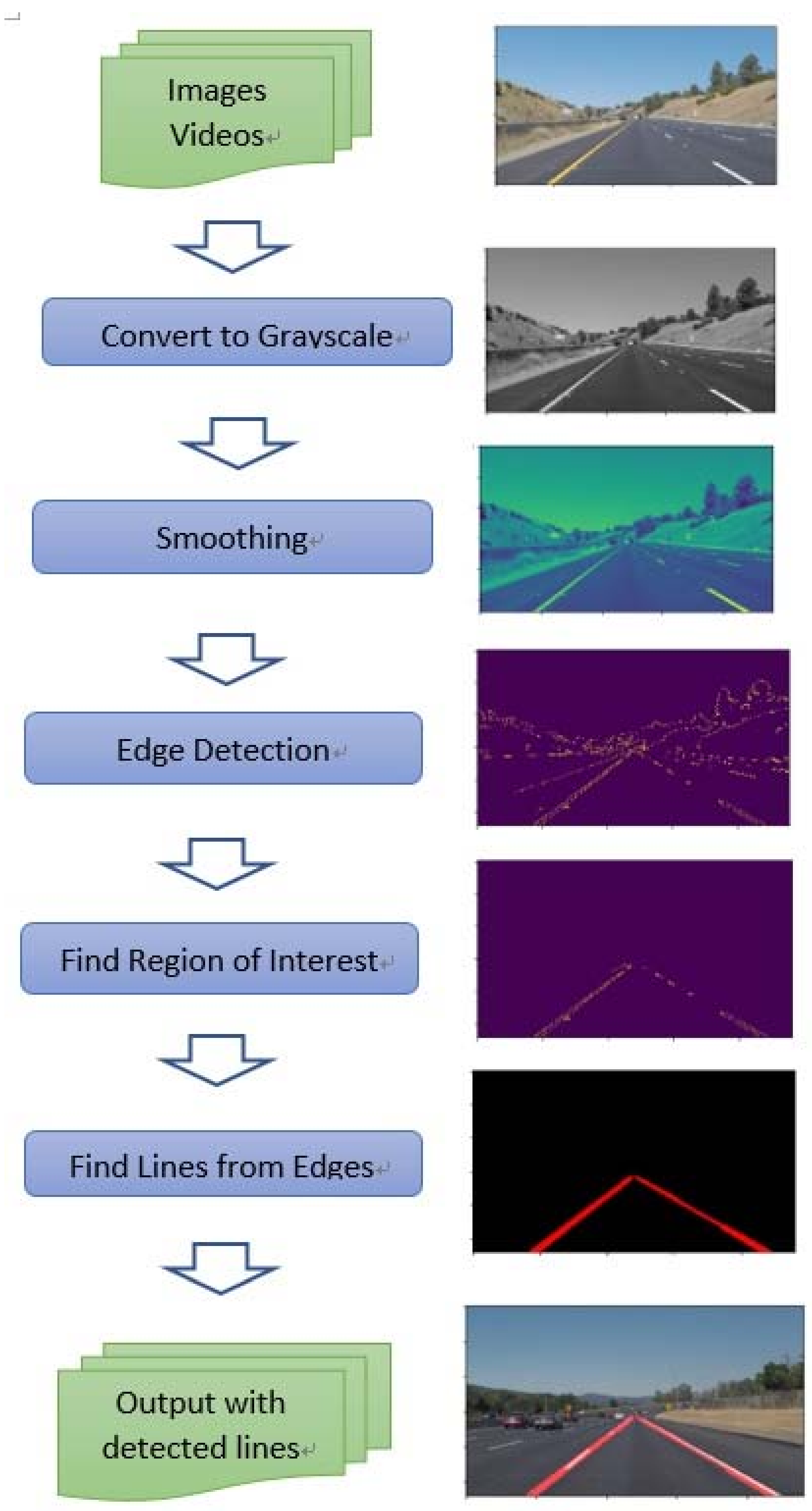
P1: Finding Lane Lines on the Road

This week we need to use Python and openCV to detect the lane line in the images and video, which were recorded with camera mounted on the back-mirror of vehicles. An example of the detection is following:


[illegible] using computer vision [illegible])

1. Overview of Detection Pipeline.

My pipeline consisted of 5 steps as shown in the below:

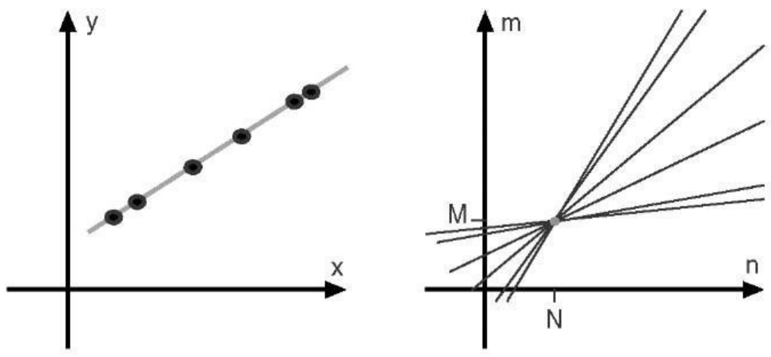


[illegible]

More details on each step:

1. **convert to grayscale**: we process the image or frame within the video by changing colorful image into grayscale. Each pixel can be represented with single 8-bit number (0~255).

2. **Smoothing**: to avoid sharp change in the image, we use Gaussian blurring in this project. We do so to avoid noise because we need to detect edge in next step. Smoothing can significantly improve the accuracy of edge detection later.
3. **Edge Detection**: the lane line has different color from its neighboring region. It is quite natural to detect edges in the image, which are more likely to be the lane lines. Here we use classic *Canny Edge Detection*.
4. **Find Region of Interest**: there are too many edges found from previous step. To narrow down to region that more likely have lane line, we need to restrict our search within more small portion of the image, which is called region of interest. Similar to crop the photos in our iPhone :-)
5. **Find Lines from points**: previous step generates many points consisting of edges. To detect continuous lane line, we need to find line from those points. The most classic approach is to use *Hough Transfrom* and find intersection of different curves in Hough Space.



[illegible]

Extrapolation Issue:

One particular issue I encountered is *how to extrapolate the lane line* to show continuous lines, since the result from Hough Transfrom is a bunch of segments shown in the left image. Our desired result is shown in the right:

[illegible]

Here is my thoughts: a continuous line like Y = a*X + b consists of two things:

1. slope of the line: coefficient "a"
2. intercept of the line: coefficient "b"

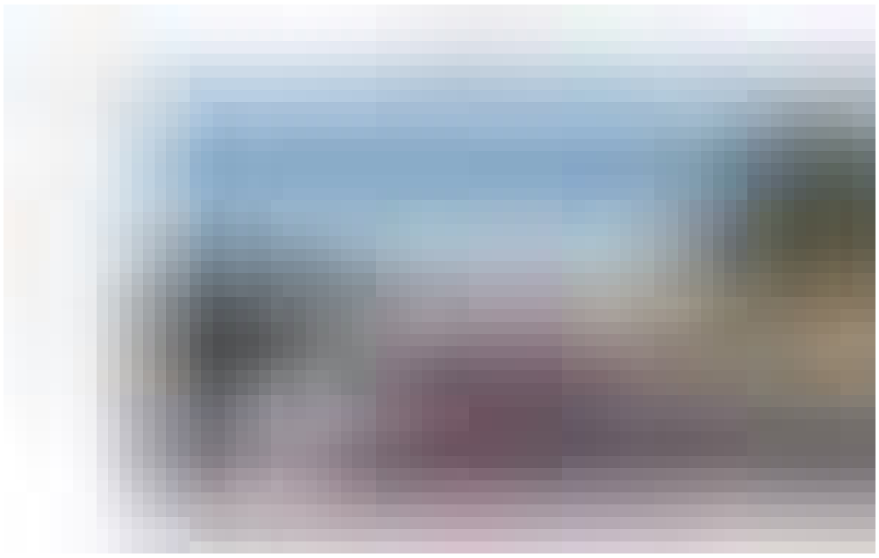
with these two parameters, we can find the top and bottom points on the line, so we can draw them on the image. Here, we use left line as an example, but the same holds true for the right line except its slope has opposite sign.

The illustration is following:

1. **Pre-Processing**: we start with a bunch of segments from Hough Transform, and calculate slope of each segment (shown as directed blue line in the below): *positive slope means the segment belongs to left line, while negative slope means right line*. Meanwhile, we can find the minimum Y-coordinate of all points on both lines ("*minY*" as shown with the dashed red line in the middle of image as below).
2. **Avg. Slope**: starting from a bunch of segments with , we can calculate the slope of each segment. Then average value of them is the average slope of this lane line, that is *the coefficient "a" in Y = a*X + b*.
3. **Avg. Position**: we can compute the average value of X and Y coordinate of all points in the line, which determines the average position of this lane line(as shown with Yellow Dot in the below.) Here, using avg. slope "a", we can easily calculate coefficient "b" using average position (avg_x, avg_y) as: *b = avg_y -a*avg_x*
4. **Determine Top and Bottom Position**: in order to draw lane line on the image, we need to know the start and end points. We call them top and bottom points in the image since the line is vertical. *top_x = (minY -b)/a* ***<note: minY is the minimum Y value of all***

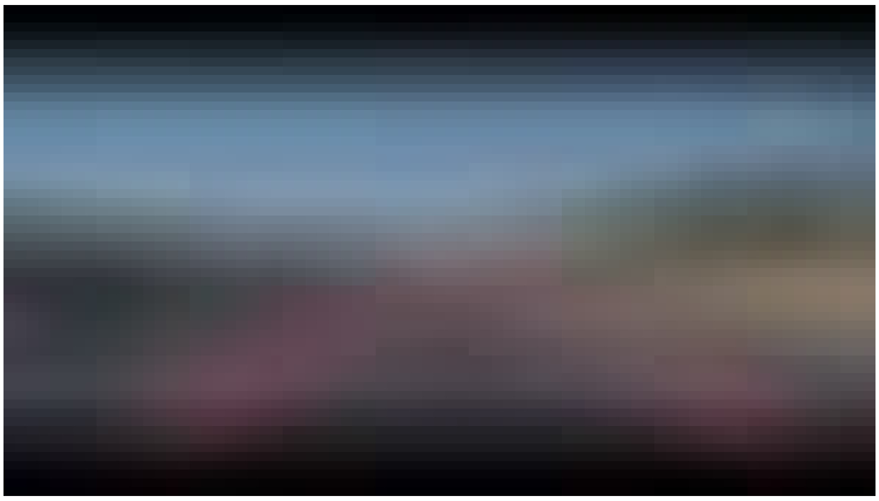
> ***points>***
> *bottom_x=(maxY -b)/a = (image.shape[0] -b)/a*

As such, we can draw left lane line between points:
(top_x, minY) and (bottom_x, maxY)



[illegible]

I modified the draw_lines() function as described in the above.Here is what we have with all these work done:



2. Identify potential shortcomings with your current pipeline

One potential shortcoming would be "***straight line issue***": when the self-driving car moves along the straight line, our detection method works very well. However, *when the car makes turns or move* *uphill/downhill*, the lane line becomes curvy or even disappears, our approach cannot find the straight line in those scenario and may have problem to accurately detect lanes.

Another shortcoming could be "***fixed parameter issue***": all our parameters in aforementioned algorithms are fixed or hard-coded. This is not flexible, or even dangerous, because fixed parameters take the assumption that camera position and view-angle are fixed. However, this is not true if the camera position moves along the bumpy roads, where the region of interest will be completely different from our settings in the code! Moreover, moving up-hill or down-hills can also cause the region of interest changes.

The third shortcoming could be "***view-blocking issue***": our testing images and videos are simplified scenario, because there is nothing in front of our car or block our view. If there is other cars cutting into our lane, edge detection may not work properly because of significant interference. Or, large truck can block most of the lane and our camera cannot capture clear shape of lane marks and may fail.

3. Suggest possible improvements to your pipeline

A possible improvement would be to following:

1. ***straight line issue:*** the simple improvement is to draw non-straight curves that best match the changing lane line. Straight line uses Y=a*X+b to model the lane, while curves can use $Y=a*X^2 + b*X+c$ or even higher order polynomials to build the model. The coefficients "a", "b", and "c" should be computed in real time according to captured images.
   It is noticed that high order polynomial include more terms and needs more computation resource, so the self-driving car system cannot use very high order models. We need to find the balance between the accuracy level and computation requirement.

2. ***fixed parameter issue:*** to improve the flexibility, we can train a neural network model with previous lane detection results (similar to supervised training using marked training set). In this way, the model can extract the lane line information from the marked region inside images or frames of videos.
   Later on, when we test this model in self-driving road test, the model can automatically move the region of interest towards the position that lane line is more likely to be detected. The difficulty

here is how to guarantee the robustness and reliability of our neural network model. I believe it needs huge amount data to train it and test it off line.

3. ***view-blocking issue****:* To resolve the view-blocking issue, we need technique similar to "inference" or "reinforce learning". Because self-driving car has history information along the road, it can use those available information to "*infer*" where is the lane in next few seconds and "*reinforce*" its inference using real situation as feedback.
   For example, when another car merge into our lane and get very close to us, the view of camera is blocked by this car so it cannot detect the lane line. In this situation, our lane detection engine can make use of its history records to "*extend*" the lane line **virtually** for next few seconds. When the car moves forward, our camera can capture the real position of lane line that could match or mismatch with our "inference" or "prediction".
   It can either discard previous inference (for *mismatch case*) and start new one, or reinforce its inference (for *match case*) and continue for next few seconds.