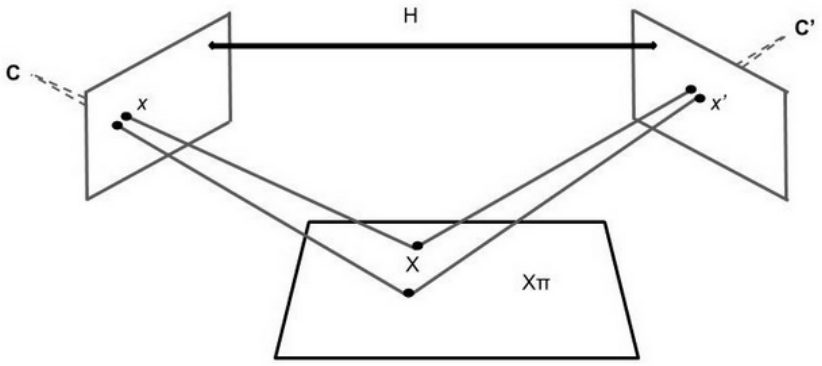
ENPM673 - PERCEPTION FOR AUTONOMOUS ROBOTS

PROJECT-2

NAME: DIVYANSH AGRAWAL
UID: 117730692

HOMOGRAPHY

Homography describes the projective geometry of two cameras and a world plane. In simple terms, homography maps images of points which lie on a world plane from one camera view to another. It is a projective relationship since it depends only on the intersection of planes with lines. As shown in the figure below:



Planar Homography

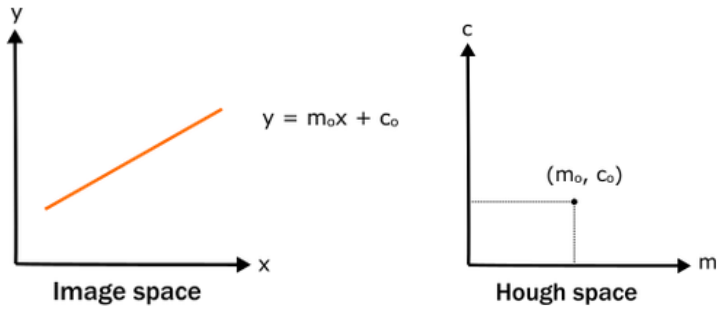
In OpenCV, the homography matrix can be calculated using cv2.findHomography. This function takes two attributes, source points and destination points and returns the homography matrix. The image below shows the matrix required to calculate the homography. The 'H' in the image below is the final homography matrix. [x1, y1] ….. [xn, yn] are the points in the first image and [x1', y1'] …. [xn', yn'] are the corresponding points in the second image.

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1' \cdot x_1 & -x_1' \cdot y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_1' \cdot x_1 & -y_1' \cdot y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2' \cdot x_2 & -x_2' \cdot y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y_2' \cdot x_2 & -y_2' \cdot y_2 \\ & & & & \vdots & & & \\ x_n & y_n & 1 & 0 & 0 & 0 & -x_n' \cdot x_n & -x_n' \cdot y_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y_n' \cdot x_n & -y_n' \cdot y_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ \vdots \\ x_n' \\ y_n' \end{bmatrix}$$

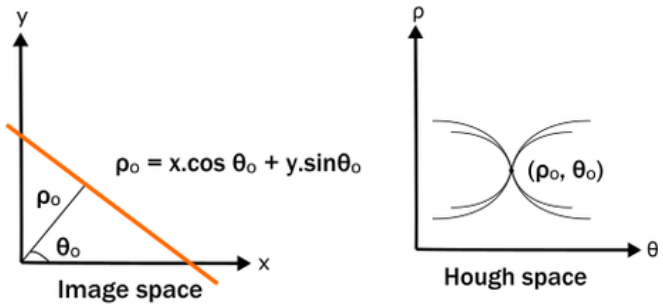
$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}$$

HOUGH TRANSFORM

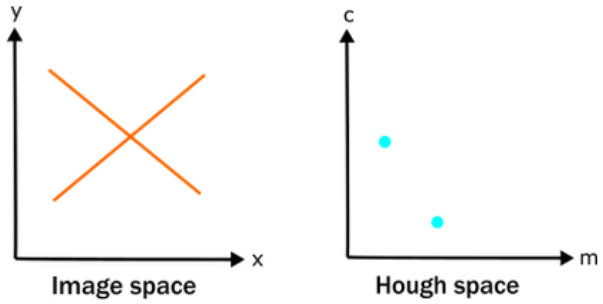
We transform the image space into hough space. By doing this we convert a line in image space to a point on hough space.



The equation of the line in the image space is of the form y = mx + c where m is the slope and c is the y-intercept of the line. This line will be transformed to a point of the form (m, c) in the hough space. But in this representation m goes to infinity for vertical lines. So let us use the polar coordinates instead.



The line is represented by the length of that segment ρ , and the angle θ it makes with the x-axis. This line will be transformed to a point of the form (ρ,θ) in the hough space. The intersection of multiple lines in image space represent corresponding multiple points in hough space.



Similarly the reverse i.e lines intersecting at a point (m, c) in hough space can be transformed to a line y = mx + c in image space.

Usage of Hough Transform:

In OpenCV, line detection using Hough Transform is implemented in the function **HoughLines** and **HoughLinesP** [Probabilistic Hough Transform]. This function takes the following arguments:

- *edges*: Output of the edge detector.
- *lines*: A vector to store the coordinates of the start and end of the line.
- *rho:* The resolution parameter ρ in pixels.
- *theta*: The resolution of the parameter θ in radians.
- *threshold*: The minimum number of intersecting points to detect a line.

The only difference between the two functions is that one uses the standard Hough transform, and the second uses the probabilistic Hough transform (hence P in the name). The probabilistic version is so-called because it only analyzes a subset of points and estimates the probability of these points all belonging to the same line. This implementation is an optimized version of the standard Hough transform, and in this case, it's less computationally intensive and executes faster.

QUESTION 1 - HISTOGRAM EQUALIZATION

Pipeline:

1. Reading the image as an array.
2. Flattening the array since it is easy to perform calculations on a 1D array.
3. Creating an empty array [variable in code: histogram] of 256 bins.
4. Iterating through each pixel of the flattened array and checking all intensities and increasing that intensity counter by 1 in the empty array [histogram]. This way I did not have to break the image into multiple channels.
5. Finding the cumulative intensity array (the sum of all values in the histogram up to that point, taking into account all previous values).
6. The values in the cumulative array are very large. So I matched these values to the original image (0-255) using normalization.
7. Now, I used the normalized cumulative sum to modify the intensity values of the original image.
8. Finally, I reshaped the array to match the original image.
9. For adaptive histogram equalization, I created a window of a specific size, the output submitted has a window of 100*100 pixels.
10. Run simple histogram equalization on each window and then update the original image window by window (histogram equalized window).

The outputs below show adaptive histogram equalization with different window sizes.
Window size: 200*200 pixel

Window size: 50*50 pixel

Window size: 1*1 pixel

Comparison: On comparing the outputs of Simple Histogram Equalization and Adaptive Histogram Equalization, it can be seen that Simple Histogram Eq. is better than the Adaptive one because the windows are visible in the adaptive one. In order to improve this, either CLAHE (Contrast Limited Adaptive Histogram Equalization) should be used or interpolation should be done on the output of adaptive hist. eq. to make the edges smooth.

Problems faced: In adaptive, I broke the image into multiple segments and then processed each segment and planned to join them after processing, but when I used join from PIL, I got an error "numpy.ndarray has no attribute image". So I asked a TA and he suggested that I should go for the sliding window method. I made a window of a specific size and then performed histogram equalization on that window and then updated the window in the original image. This way I didn't have to join the different image segments.

Video Link for Simple Histogram Equalization:
https://drive.google.com/file/d/16SbnIzWQWoy3k9FUT3uMOt86GFO_9OC-/view?usp=sharing

Video Link for Adaptive Histogram Equalization:
https://drive.google.com/file/d/1mypukgET67y0JElFewxyxz_JA5D71CpR/view?usp=sharing

QUESTION 2 - LANE DETECTION

Pipeline:

1. Reading the video frame by frame, converting each frame into gray image (single channel), and then applying gaussian blur to reduce noise.
2. Next, I used Canny edge detection to detect all the edges.
3. Next, cropping a Triangular region (region of interest) from the frame, this region contains only the lane lines, no other feature.
4. For cropping, I created a mask of black color (using fillPoly), whose shape is the same as the region of interest. I took bitwise_and of this mask and my frame, so I got an image (cropped image) which only has the lane lines.
5. Using houghLinesP on the cropped image to detect the lines. Hough lines P was used instead of hough Lines because I wanted the lines in coordinate form and not the polar form. This gives both solid and dashed lanes in the cropped image.
6. For separating the solid and dashed lines
    - Find the line which has maximum length, this will always be a solid lane, no matter which frame is being processed.
    - Calculate the slope of this line, and find its sign using np.sign.
    - Calculate the slope of all lines and check if its slope matches that of the longest line.
    - If slope of any line [i] is the same as that of the longest line then that line [i] is a solid line, else it is a dashed line.
    - Segregate them and color them according to solid or dashed using cv2.lines.

This pipeline works for the flipped video (this video after flipping) too as shown in the images below, though it might throw an error if in any other video, the triangular region (region of interest) does on have lane lines. This is because the current video has lane lines in the center of the frame, but if the lane lines are on either side of the frame, then the vertices of the region of interest would have to be changed and the pipeline might give an error. The image below shows that the pipeline works for the flipped video too.

Problems Faced:

Finding the region of interest took a lot of time, since if the region was inaccurate then houghLinesP detected lines other than lane lines.

Video Link:

https://drive.google.com/file/d/1v7uP9ZiCeBu_krxbaD4BGQxUzYTIIixd/view?usp=sharing

QUESTION 3 - TURN PREDICTION AND CURVATURE CALCULATION

Pipeline:

1. Read the video frame by frame, converting the color to HSL - hsl_image (hue, saturation and lightness). The HSL color space abstracts color (hue) by separating it from saturation and pseudo-illumination and thus is preferred for practical applications like lane detection and turn prediction here.
2. Creating a mask for yellow and white (this can be called thresholding too as I am giving a range of colors for yellow and white)
3. This yellow mask is 'bitwise_and'ed with initial hsl_image to get only the yellow lines. The white mask is 'bitwise_and'ed with initial hsl_image to get only the white lines. These two images are 'bitwise_or'ed to get both the lines in a single image.
4. This hsl_image is converted into grayscale for further processing.
5. Now using canny edge detection to detect the edges (lanes) in the grayscale image.
6. Next, crop the image using the region of interest method (here, the region of interest is a trapezium or a 4 sided convex polygon since warping will be used in the near future).
7. Next, warping the cropped image (grayscale) [variable name: warped_img] to a specific window size (width = 600 and height = 300).
8. I also warped the original image (frame from the video) to the window size [variable name: warped_img_color]. This was done because hough lines will be detected on the grayscale image but will be plotted on the coloured (BGR image) so they need to be warped in the same way.
9. Next, using houghLinesP to detect the lines. Hough lines P was used instead of hough Lines because I wanted the lines in coordinate form and not the polar form.
10. Next, I separated the coordinates from the lines returned by houghLinesP.
11. I separated these coordinates into right_lane_coordinates and left_lane_coordinates using an 'if' condition based on their location in the window.
12. Now I have coordinates of the left lane and the right lane.
13. So, I used 'np.polyfit' to fit a second degree curve on these coordinates. I got the coefficients on the curve. Curve fitting was done separately for both the left and right lanes.
14. Using this curve, the radius of curvature was calculated and displayed on the video. While calculating this, a lane length of 10ft was assumed (standard data for US road lanes) to map pixels to meters.
15. Now as I have the curve, I plotted the left lane using cv2.line. For the left lane, the x values range from 0 to 120 (0,120) and for the right lane, the x values range from 480 to 580 (480, 580).

16. For turn prediction
    - When I separated the coordinates for left and right lanes, I sorted them into an order of descending Y coordinates. That is, points with higher Y value will be present first in the list.
    - Next, I calculated the slope of the line present between two consecutive points.
    - If the value of the slope of this line decreases as we pass through the coordinates list, this means that the lane is moving right, so the vehicle should move right too.
    - If the value of the slope of this line increases as we pass through the coordinates list, this means that the lane is moving left, so the vehicle should move left too.
17. After all these steps, next I stacked different frames according to the output video required. This involved resizing the frames to the same size and converting single channel frames to multi channel by giving the same single channel as input for all the channels of the new image.

This pipeline might throw errors if run on other videos, because at many places in the pipeline, things have been specified according to this video. Like in the region of interest, the coordinates present here might not give an accurate region of interest in a different video. When I am segregating the right and left lanes, the points have been given according to this video. This pipeline might not be the best when it comes to turn detection but I tried my best to implement my own pipeline.

Problems faced:

Getting an accurate region of interest (ROI) was the biggest issue. I had to tune x and y values of all the 4 coordinates of the ROI multiple times to make sure that no points other than the lanes are present.

In the middle of the video, there is some shadow of a tree and the type of material used for making lane changes, there I had to figure out a way so that these things don't interfere with my pipeline. Using HSL also helped me in tackling this issue.

The curve for the right lane was not fitting properly. There were frames where the curve went out of the lanes (shown below), so I had to tune the X values for the right lane multiple times. I came out with (480,580) after testing about 50 combinations.

Video Link:
https://drive.google.com/file/d/1Midxwu7vqy0VJqUgjg4qB3Z398YpocuN/view?usp=sharing

References:

1. https://medium.com/hackernoon/histogram-equalization-in-python-from-scratch-ebb9c8aa3f23
2. https://medium.com/@kyawsawhtoon/a-tutorial-to-histogram-equalization-497600f270e2
3. https://stackoverflow.com/questions/22685274/divide-an-image-into-5x5-blocks-in-python-and-compute-histogram-for-each-block
4. https://stackoverflow.com/questions/44650888/resize-an-image-without-distortion-opencv
5. https://medium.com/@st1739/hough-transform-287b2dac0c70
6. https://learnopencv.com/hough-transform-with-opencv-c-python/
7. https://medium.com/@mrhwick/simple-lane-detection-with-opencv-bfeb6ae54ec0
8. https://www.intmath.com/applications-differentiation/8-radius-curvature.php