TOSHKENT SHAHRIDAGI INHA UNIVERSITETI
INHA UNIVERSITY IN TASHKENT

FINAL REPORT ON PROJECT

Capstone Design (SOC4150)

Comprehensive Assignment
(Line Detection)

Team Members:

1. Anvarjon Yusupov U1610026
2. Boburjon Iskandarov U1610054
3. Oybek Amonov* U1610176
4. Kamronbek Jurayev U1610261

TOSHKENT SHAHRIDAGI INHA UNIVERSITETI
INHA UNIVERSITY IN TASHKENT

Content:

1. Code Structure

There is main loop and five function:

Pixels – retrieve pixel coordinates from average slope and intercept
regression_on_lines - right and left line separation
filters - color correction, image denoising and edge detection is applied
ROI - masking of the frame
display_lines – lines are added on the original image
prediction_turn – predicts potential turn

2. Algorithm

Project is fully written using Python language. It requires image, matrix, and trigonometrical manipulations, that's why we used three libraries: 1) cv2 – open source computer vision library; 2) NumPy – python library for numerical manipulations 3) math – for mathematical manipulations. The whole process is concentrated in one loop. And each iteration is sequence of analyzes and manipulations over each frame. Loop is stopped either if frames would end (video ends) or if user manually interrupts process by entering key 'q'.

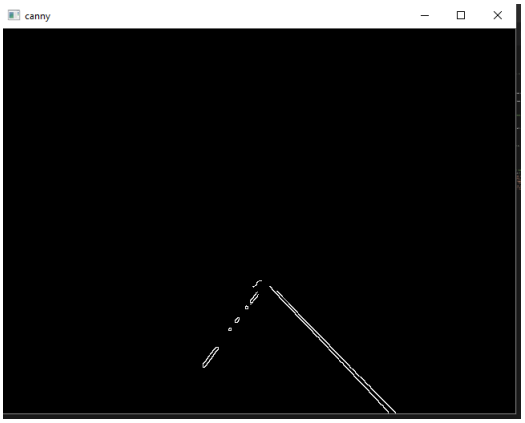
First of all, we create variable **cap** that captures the video from source. Afterwards we use this variable to read frames, we achieve that using command *cap.read(),*it gives parameters of the frame and return value. We break the loop if return value is False, because it means problems occurred while reading the frame. The first manipulation applied to the frame is resizing. We manually resize the frame into resolutions that are given in the task (1024x768, 800x600, 640x480, 400x300). Before starting further manipulations, by using *cv.getTickCount()* we keep the number of clock-cycles at that exact moment(in variable - **timer**), it will be required afterwards to find the frames per second (fps).

Next step is calling the function *filters(frame).* In that function, we first convert BGR image into grayscale one, it is required because edge detecting function (cv2.Canny*(image, threshold1, threshold2)*) takes as input only 8-bit imaginaries. After color correction, we remove noise from our grayscale image, for that we apply Gaussian blurring function (temp = *cv2.GaussianBlur(image, matrix size, value of matrix)*). After removing high-frequency components, we apply to our image edge detecting technique mentioned above. *Cv2.Canny()* – function used for edge detection. In the function we specify the parameters of unit pixel range for edge linking. As a result, we get the image which contains only edges, or in other words those juxtaposed pixels that has significant difference in their value.

1) Edge detection applied image

The next phase is masking process. For that we call the function *ROI() – region of interest.* First we need to distinguish coordinates of our ROI. We manually found out the coordinates of it and use them in code. We created two arrays. Array number 1, keeps the vertices of ROI (which is area in

shape of triangle) as our coordinates. Second array has the same dimension as our frame, but the values are 0. We achieved that with numpy function *np.zeros_like(variable dimension of which we copy).* And finally in order to get the mask, we call function *cv2.fillPoly(array#2 resulting mask, array#1 that has only specific area of interest, value that will be assigned to specific region within the array1)*, which creates a matrix that has the dimension of the original frame, but the values of are all zeros, besides the ROI that we assigned all as 255 (white color). By the end of the function, we return our mask to main loop.



After acquiring mask, we need to apply it to our **canny_Image**, to have edge detected image only within the ROI. It very easy to do, *bitwise_and* operation must be applied between two matrices:

canny_image = cv2.bitwise_and(canny_image, roi_mask) # ROI-masked-canny image

2) Mask applied on edges detected image

Next goal to achieve is detecting the Hough lines. *cv2.HoughLinesP()* – is the function that we would use, but we need to specify the distance precision (rho - given in pixels) and angular precision (theta - given in radians). These data are essential because Hough Line detection encapsulates polar system where line and angle are only parameters. Besides rho and theta, we need to provide the largest allowed gaps between segments and minimum length of the line.

lines = cv2.HoughLinesP(canny_image, 2, np.pi/1000, 108, np.array([]), minLineLength=40, maxLineGap=100)

As a result, we get all the lines that fit to given criteria. Afterwards, we call the function *regression_on_lines(),* where we apply principles of mathematics, specifically properties of linear functions. We need to pass as argument variables **lines -** contains coordinates of Hough lines and **frame** in numpy matrix form.

Within the called function, first we are checking whether lines were detected or not. In case of failure in detection, we return empty array. Otherwise, we create three arrays: **left_line**, **right_line**, and **road_lines** (last one stores return value). There are two possible lines, in order to identify whether it is right line or left we use principles of linear functions. Here are the conditions:

x1, x2 – first and second coordinates of detected line on x-axis

y1, y2 – first and second coordinates of detected line on y-axis

Left Region	Right Region
$x_1 < x_2$ $y_2 < y_1$ $m= (y_2 — y_1) / (x_2 — x_1) < 0$ (Slope)	$x_2 > x_1$ $y_2 > y_1$ $m= (y_2 — y_1) / (x_2 — x_1) > 0$ (Slope)

According to mathematical properties of linear functions these conditions must be followed for lines in left region.	Right line must follow these conditions, according to properties of linear functions.
If x1 and x2 are the same (means vertical line) then we skip further analyzes of that line, cause scope will be equal to infinity. It will cause logical error.	

The exact same conditions described above are checked in the code. Through several test, we came to conclusion that these conditions were not enough. For increasing accuracy manually calculated approximate boundaries for left and right line region (variables are **right_reg** and **left_reg**), so by comparing x-coordinate of line with those boundaries, we identify in which region that specific line lies.

Nested *for* loop is used for lines analysis. We inspect coordinates that are stored in **lines**. First conditional operation is to skip vertical lines, that we mentioned above (x1=x2). After that we calculate slope and interception for coordinates taken at that iteration. Slope is calculated as given above; interception is equal to y1 - (slope * x1). Conditions for region identification that were mentioned before:

After checking for vertical lines, we test the value of slope. Then we check the location of coordinates relatively to our calculated boundaries. After identifying the side where line lies, we add the values of a slope and an intercept into respectively named array that we initially defined.

```
if x1 == x2:
    continue

slope = (y2 - y1) / (x2 - x1)
intercept = y1 - (slope * x1)

if slope > 0:
    if x1 > rignt_reg and x2 > rignt_reg:
        right_line.append((slope, intercept))
else:
    if x1 < left_reg and x2 < left_reg:
        left_line.append((slope, intercept))
```

It is not the end of the function yet. We should find the average value of right and left intercepts and slope. We use numpy function:

left_avg = np.average(left_line, axis=0)

right_avg = np.average(right_line, axis=0)

Axis = 0, means that we are computing vertical average or in other words average value of columns in each array. Goal is to get one overall line for each side. Although we found slope, intercept and coordinates on each side, it is not enough yet to draw line on image. That's why before storing final result to our return variable (**road_lines** defined at the beginning), we call another function *pixels(image, left_avg* or *right_avg)*. We call this function twice, once for right side and once for left side.

In function pixels, we find x1, x2, y1 and y2 coordinates for lines in both sides. Finding vertical coordinates are not hard, for y1 we assigned the height of the image, or in other words y-axis coordinate of the last row. Finding y2 is not hard as well. We manually assigned height*3/4 as y2 (means minimum length of line would be 25 percent of the whole image height). Finding x1 and x2 requires calculation:

$$x_n = \frac{y_n - \text{intercept}}{\text{slope}}$$

n – either 1 or 2 for both right and left lines

It is so because intercept is yN – (slope*xN), so by subtracting intercept from yN we get slope*xN. Now we just divide result by slope and get xN. We return coordinates back to *regression_on_lines()* where *pixels()* was called.

In *regression_on_lines()* after receiving coordinates for left line and right line, we store them in **road_lines** and send them back to main loop.

In main loop, after acquiring coordinates of lines (store them in **regression_lines**), we call *display_lines(original frame, regression_lines).* In that function, we create a matrix (**line_image**) with the sizes as original frame, but all values are zero - *np.zeros_like(image)* , first we will draw line on this matrix. Drawing line is done with function:

cv2.line(line_image (explained above), start_point, end_point, color, thickness, line_Type)

We know starting and ending points, for color and thickness we gave green and 3 respectively. For line type we assigned cv2.LINE_AA - anti-aliased line which is used for curves. This particular function is repeated twice for both lines in right and left. After that we blend **line_image -** matrix that has all pixels black, besides drawn lines (green colored) and **frame –** original image that hasn't been modified in any way besides resizing. They both have the same size. In order to blend we use

line_image = cv2.addWeighted(frame, weight of the frame, line_image, weight of the line_image, scalar added to each sum)

The reason we use matrix with values zero is that values of respective pixels will be added, so some value N of original frame added to zero value of matrix, the result will be the pixel value of original frame (that is not the case when we are adding pixels of line and original frame). Weight for both images we gave one, so the lines would be visible but will not stand out from rest of the image. We return result of the blending back to the main loop.

Our final image is stored in **line_image.** However, prediction part yet to be done. We call function *prediction_turn(frame – original image, regression_lines – where we store our coordinates of lines)*. Regression lines keeps coordinates of detected lines.

In order to predict the turn, we need to find incline/decline angle of the lines (radian angle is arctangent of division between x-offset and y-offset). First we acquire height and width of original image. Height is needed to find y-offset, which would always be the half of the height. Width would be needed to find horizontal middle boundary, which we took as the half of width (**middle = width/2**). X-offset is a bit complicated because it depends on the number of detected lines:

Case 1 - 2 lines are detected:

Then x-offset is **left_x2** + **right_x2** (horizontal end point of left and right lines) divided by 2 - **middle**

$$\frac{left\_x2+right\_x2}{2-middle}$$

Case 2 – One line is detected:

X offset is difference between values of x2 and x1.

Case 3 - No lines are detected:

X offset is zero

There are only these three cases. Now after getting x and y offset, we may find incline/decline angle. Which is simply arctangent of x-offset/y-offset. There is code:

math.atan(x_move_axis/y_move_axis)

Math.atan() is function for arctangent, **x_move_axis** and **y_move_axis** our variables in code that keeps the values of the x and y offsets. The result that we get is in radians, but for convenience we convert it to degrees. It is done by this code: *int ((radian_angle * 180)/3.1416).* Now after acquiring incline/decline angle in degrees we send them back to the main loop.

Before displaying everything on the screen, we calculate frame per second (fps). Which is easy to do:

fps = cv2.getTickFrequency() / (cv2.getTickCount() - timer)

We are diving frequency of clock-cycles by the difference of number of clock-cycles (we defined **timer** at the beginning of the code). So by this simple calculation we get fps. In order to display fps value on the image, we use *cv2.putText()* that puts required text on the frame. The same function will be used for adding to the screen information about prediction, which depends on the angle that we calculated.

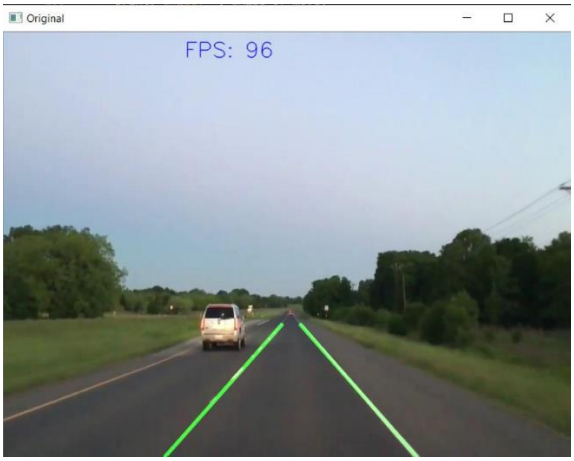
Case 1 – angle is less than -2:

Turning to left

Case 2 – angle is greater than 2:

Turning to right

At the end we are using *cv2.imshow()* to show the final result, that is stored in **line_image**. Additionally, we display fps and prediction. The loop would repeat the same actions with the other frames. Final result is below:



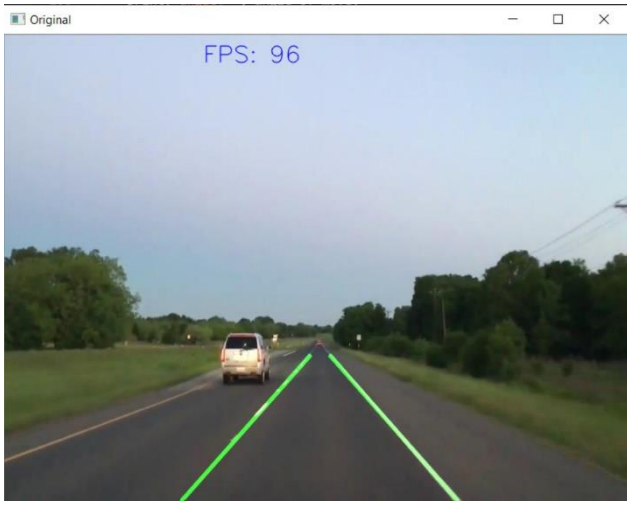
3. Resolution and FPS

Resolution	FPS range
1024x768	60 - 70
800x600	90 - 100
640x480	160 - 170
400x300	400 - 410

CPU characteristics: Intel(R) Core (TM) i7-6500U CPU @ 2.60GHz

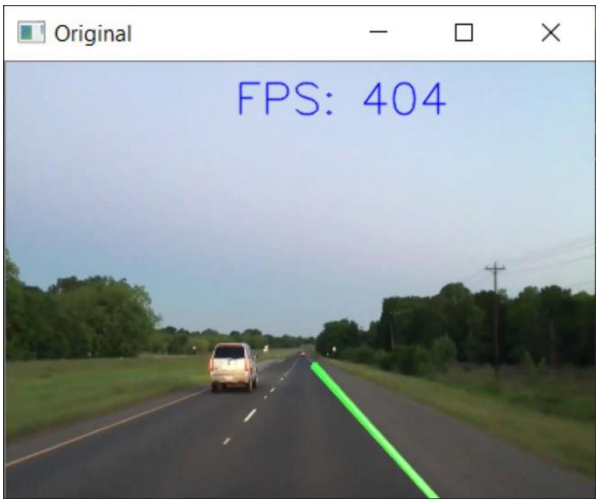
Below you find screenshots of frames with different resolutions and their frame per seconds:

1) Resolution: 1024x768 and FPS: 70



2) Resolution: 800x600 and FPS: 96

3) Resolution: 640x480 and FPS: 161



4) Resolution: 400x300 and FPS: 404

4. Video Link

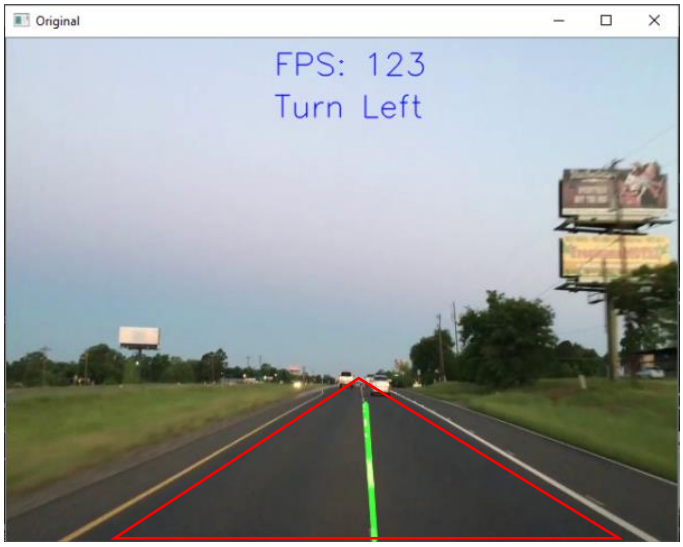
You Tube video Link - https://youtu.be/1y5YdolvYGA

5. Team Contribution

Name	Roles	Contribution
Oybek Amonov* U1610176	**Team Leader** Software Developer	Conducted online meetings, motivated and guided the team towards success. Developed denoising, blurring, binarization, masking, line detection and separation parts
Anvarjon Yusupov U1610026	Report Writer Software Developer	Worked on report, collected data from all team members. Developed denoising, blurring, binarization, masking, line detection and separation parts
Boburjon Iskandarov U1610054	Optimizer Software Developer	Concentrated on optimization and testing of the final code. Developed line drawing and prediction parts
Kamronbek Juraev U1610261	Software Designer Software Developer	Provided software design of the code and documented requirement analysis. Developed line drawing and prediction parts

6. Future Work based on Errors

During the project development, we tried to get the correct result for real-time lane detection; however, we had some inexplicable errors due to some factors. As you have read in previous sections, we chose our ROI to be a triangle so there will be less noise from outside the ROI. But, this triangle has it own disadvantages too, one of which is when a car is changing its moving track (right lane or left lane), the ROI loses one of the side lines it was detecting. So, there might be a wrong assumption on the prediction of turning. We have explained the way we are detecting the turn; it is based on the angle move of the final Hough lines we draw on the original frame. The problem can be seen in the figure. You see that ROI can not cover the whole road width, because it a triangular ROI. If we used a rectangular ROI instead, our working ROI would have noise from road edges with the ground. We tried to work with the road only, so we used triangular ROI. Anyway, the ROI came with it own drawbacks. Another important issue related to the correctness and robustness of our project is the road lines variation. They are not always smooth, and straight. Sometimes edge road lines form a curved line approaching the road edge with the ground. It has also influenced the correct detection of Hough Lines, and gave a false impression that car is changing its track. Since the solution to this problem needs specific case correction, we did not intended to get the project too complex. Otherwise, the whole project might have been complicated to developed and debug. Another minor error occurs when the road track width is not standardized. You can see the problem in the left figure below, right road line is outside the ROI. However, we will keep on improving by adding procedures to handle the case discussed.



Another inconsistency in line detection worth mentioning is the left track and its yellow line. When a car is moving on the left track, the yellow line and the road edge is included as ROI because car sometimes gets to close to the left road edge. This ensures that Canny edge detection finds two possible edges one is the road edge, and the next is yellow or white line. Hough lines merge these two edges into one, so the left line is not consistent. Its pixel coordinate fluctuates continuously until either the car changes its track to right track or road edge goes outside the ROI. As you can see from the figure, car tyre is almost going on the yellow line which means there is some space in ROI to detect the left road edge. When car goes near the centre dashed lines, road edge with the ground will have no effect on the system. Considering the quality factors, the bridges and the road nearby them are the main issue, since that area is difficult to analyze due to little intensity between the road and while lines. We applied blurring, canny edge detection, and closing operations on the grayscale image. The applied techniques are highly dependant on the intensity level difference of the frame. Thus, those regions are almost to analyze using the applied techniques. The problem can be seen in the next figures. Every autonomous system is dependant on the perception of outside world, thus, our system is also dependant on road quality factor a lot.