Queen Mary University of London

Introduction to Computer Vision

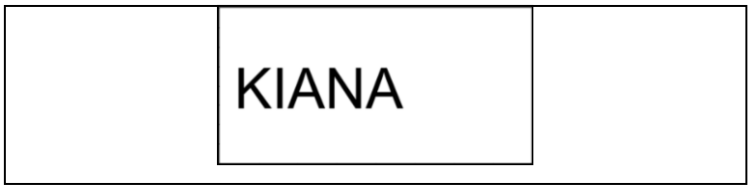
Coursework

Submission

Kiana Hadysadegh

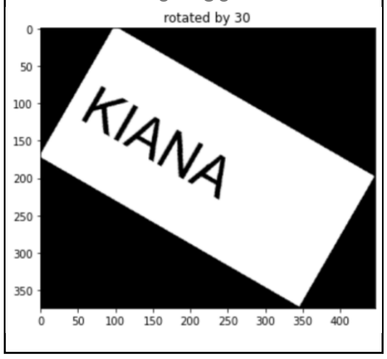
230632224

Transformations
Question 1(b):

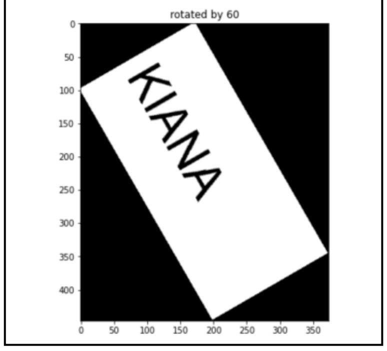


Rotated images:

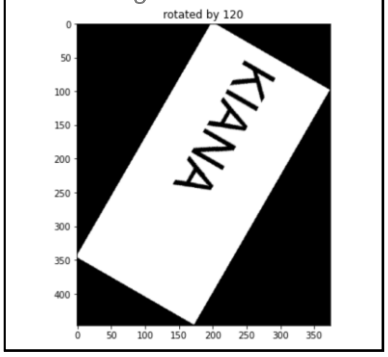
θ = 30



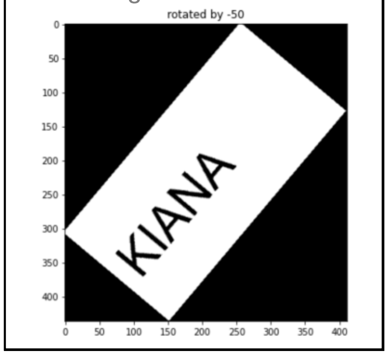
θ = 60 deg



θ = 120 deg

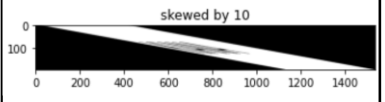


θ = -50 deg

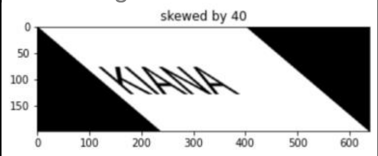


Skewed images:

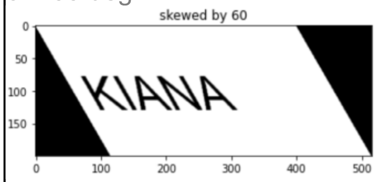
θ = 10 deg



θ = 40 deg



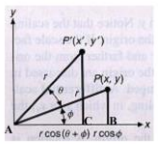
θ = 60 deg



Your comments:

At first we open our image as an numpy array
Rotation Formula:

$\theta > 0$: counterclockwise rotation



from *ABP* triangle: $cos(\phi) = x/r$ or $x = rcos(\phi)$
$sin(\phi) = y/r$ or $y = rsin(\phi)$

from *ACP'* triangle:
$cos(\phi+\theta) = x'/r$ or $x' = rcos(\phi+\theta) = \underbrace{rcos(\phi)}_{x}cos(\theta) - \underbrace{rsin(\phi)}_{y}sin(\theta)$
$sin(\phi+\theta) = y'/r$ or $y' = rsin(\phi+\theta) = \underbrace{rcos(\phi)}_{x}sin(\theta) + \underbrace{rsin(\phi)}_{y}cos(\theta)$

$$x' = xcos(\theta) - ysin(\theta), \quad y' = xsin(\theta) + ycos(\theta)$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \underline{P' = R\ P}$$

Since in the rotation technique the dimensions of the original image changes, we first calculate the dimensions of the new grid (which basically follows the original formula of rotation):
new_height = round(abs(y * cos) + abs(x * sin))+1
new_width = round(abs(x * cos) + abs(y * sin))+1
then we define our output grid: output = np.zeros((new_height, new_width, channels))

Since the only point that its coordinates doesn't change in rotation is the center, we calculate the center of the both grids. Also to use our formulas we should find all points coordinates with respect to the center point.

If we iterate through original grid and copy its pixels values into the corresponding coordinates in the new grid, we face some black dots in the output. This is because we should use round operator in calculating coordinates for the new grid, So some pixel values are copied in the same point in the new grid, and some coordinates in the new grid missed because of the round operation. As a result we can see black dots in our rotated image which means some spots were missed in the new grid. To avoid this problem we use inverse mapping technique, which iterates through new grid points and find their coordinates and pixel values from the original grid. By this method, none of the pixels of the new grid miss:
We inverse our rotation matrix and multiply it by coordinate of the point in the new grid to find the coordinate of the original point.

$$R^{-1} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

original_x = (int(inverse_rotation_matrix[0,0] * x + inverse_rotation_matrix[0,1] * y))
original_y = (int(inverse_rotation_matrix[1,0] * x + inverse_rotation_matrix[1,1] * y))

So, we first change the origin of the new grid to its center and calculate the points coordinates with respect to the center.
Then we find the corresponding points coordinates in the original grid using inverse mapping formula. The points in the original grid also were calculated with respect to its center point, so we convert them with respect to the normal origin(array[0,0]).

In the end we check if our calculated coordinate is in the grid or not since indexes of the points in the borders become out of bound. After that we copy their value into our new grid.

For horiziontal skewness: (shear along x-axis)

$$\begin{bmatrix} 1 & \frac{1}{\tan\theta} \\ 0 & 1 \end{bmatrix} \qquad S_x^{-1} = \begin{bmatrix} 1 & -\frac{1}{\tan(\theta)} \\ 0 & 1 \end{bmatrix}$$

To avoid missing pixels in the new grid, Again we use inverse mapping method and iterate through the new grid and find the value of its points from the original grid:

We use inverse of the above matrix:

```
old_x = round(x - (y/tan(theta))
old_y = y
```

Since it is horizontal skewness only x-values (width) of the points changes. So we create a new grid for output with the same height and new width which is calculated based on the skewness formula:

```
new_width = round(abs(old_width)+abs(old_height/tan))+1
```

We iterate through the new grid and calculate the coordinates in the old grid using the above formula. In the end we copy the pixel value from original image into the output grid.

***Advantages and disadvantages of different approaches:**

Inverse mapping:
In this method since we iterate through new grid and looking for the pixel values in the original grid, we have pixel accuracy (no pixel misses in the new grid, and no pixel filled more than one time) and versatility. Versatility means that we can recover the original input from the output since :

$$f(f^{-1}(x)) = x \text{ and } f^{-1}(f(x)) = x$$

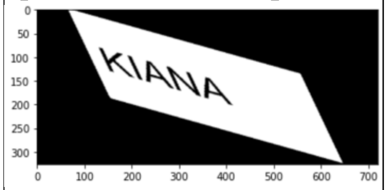
However, implementing this method is complex compared to forward mapping.

Bilinear Interpolation:
It smoothes the image and removes the black dots by putting the average value of the neighbours of one pixel in it. Its result has good quality according to its computation complexity for small rotations. However, it also has computational cost compared to forward mapping and has limited accuracy for handleing large rotations. This is because in this method we assume that the value of the pixels vary smoothly within a small neighbourhood but in large rotations this assumption leads to loss of information. When an image is transformed, sometimes the pixels in the new image have to be obtained from distant or non-adjacent pixels in the original image. This can result in some undesirable side effects such as artifacts and a decrease in overall image quality.

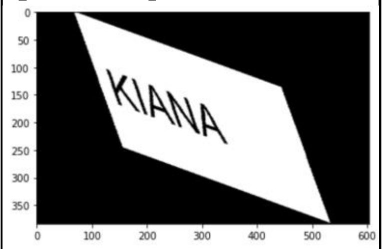
Forward mapping:
It is so simple to implement and it is efficient for real-time applications because it can quickly compute the answer. However we have missing pixels and black dots in our result. It may also have sampling issues when the size of the image after transformation is considerably bigger than the original one. For example since we should convert new coordinates to integers we may find repetitive coordinates in the new grid.

Question 1(c):

θ1=20 clockwise and θ2=50



θ2=50 and θ1=20 clockwise



Your comments:

***Analyse the results when you change the order of the two operators: R(S(I)) and S(R(I)). Are the results of (i) and (ii) the same? Why?**

They are different because the order of these two operations is important. To prove it we can take a look at transformation matrices in these two scenarios:

$$k = \frac{1}{\tan(\theta)}$$

First rotate then shear:

$$\begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix} \times \begin{bmatrix}1 & k\\ 0 & 1\end{bmatrix} \Rightarrow \begin{bmatrix}\cos(\theta) - k\sin(\theta) & \sin(\theta) + k\cos(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}$$

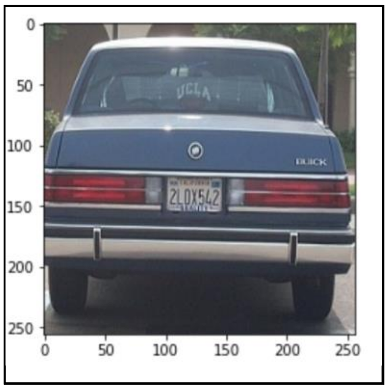
First shear then rotate:

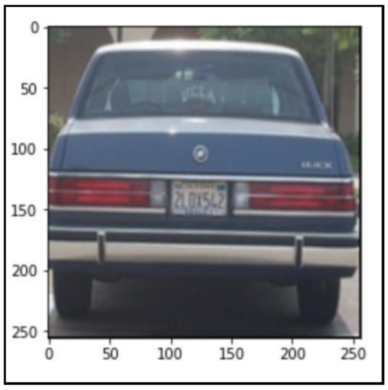
$$\begin{bmatrix}1 & k\\ 0 & 1\end{bmatrix} \times \begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix} \Rightarrow \begin{bmatrix}\cos(\theta) + k\sin(\theta) & -\sin(\theta)\\ \sin(\theta) + k\cos(\theta) & \cos(\theta)\end{bmatrix}$$

It is obvious that the transformation matrix is completeley different in these two scenarios which leads to different new points after applying each matrix to the old points.

Convolution
Question 2(b):

Designed kernel:





Your comments:

Convolution:
To calculate of the convolution of the given image with given kernel, we place the kernel on each pixel of the image so that the center element of the kernel is on that pixel. Then we perform element-wise multiplication between pixels of the kernel and pixels of the image which the kernel is on them. After that we calculate the summation of all multiplied elements and divide it by sum of the values in the kernel. We repeat this process for all pixels of the image and save their convolution value in the same position in another grid (with the same size as image) and return it. If the sum of products value is negative we assume it is zero and if the sum of the elements of the kernel is 0 or negative we do not perform the division to handle negative values and divide by zero problems.

The other problem is that pixels from the border of the image up to [kernel_size/2] from it do not participate in the convolution operation and this called the border problem. To solve this problem I use the zero padding method. I create a new grid which its dimensions is kernel_size bigger than the old grid and fill it with zeros. Then I copy the image in the center of the new grid in a way that the old border is in the [kernel_size/2] distance from the new grid border. By this way all of the pixels in the border to [kernel_size/2] from it also participate in the convolution operation. In the code I iterate through output grid and for each point I calculate the range of the image in the new grid that should participate in the convolution and calculate the convolution of it with the kernel and fill the corresponding point in the output grid with calculated convolution value.

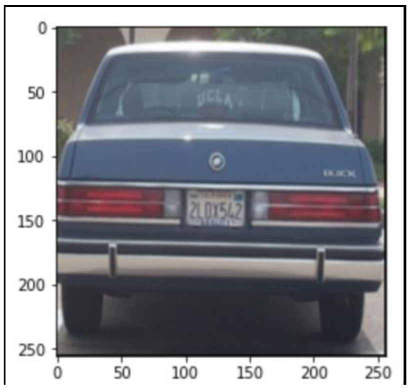
$$x(k,l)**g(k,l)=\sum_{k'=0}^{M_g-1}\sum_{l'=0}^{N_g-1}g(k',l')x(k-k',l-l')$$

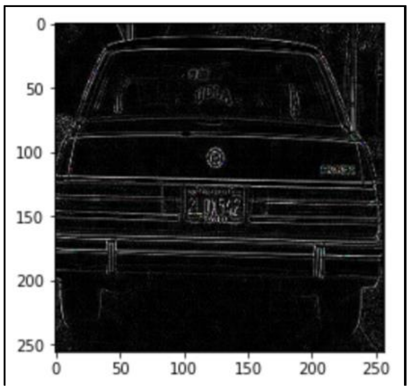
***Convolution kernel that computes, for each pixel, the average intensity value in a 3x3 region.**

$$\begin{matrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{matrix}$$

we can use the above kernel to calculate the average intensity value in 3x3 region for each pixel. This is because each values in the 3x3 region are add together and the result is divided by sum of values in the kernel which is 9. So, by using this average kernel we can calculate average intensity in this region for each pixel.

Question 2(c):





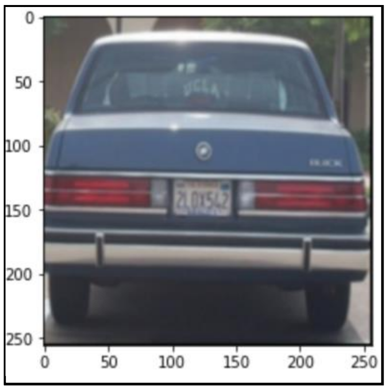
Your comments:

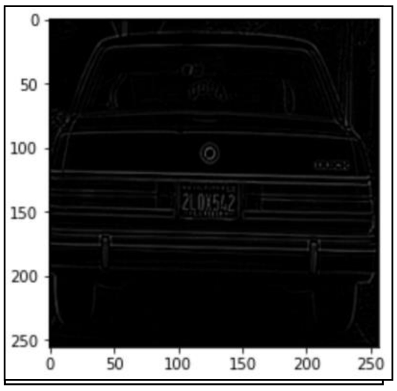
***Comment on the effect of each kernel**

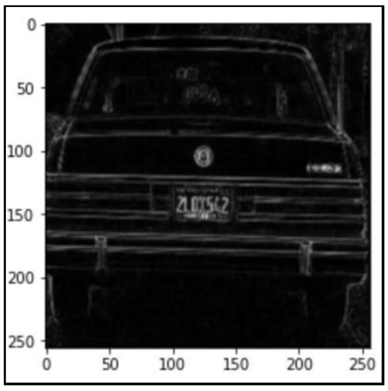
Kernel A is a smoothing kernel (Gaussian blur kernel). We can see the smoothing effect of this kernel on our image (it is obvious that our original image is blurred after applying this kernel). According to the weights of the kernel we can see that center pixel is more emphasized in the convolution operation and as you go far from the center pixel the weights decrease. So this the reason for smoothing effect of this kernel. When edge preservation is important it has better performance than average intensity.

Kernel B is Laplacian edge detection kernel. By using this kernel we can find edges in our picture since it highlights regions with fast intensity change (emphasizes the difference in intensity between a pixel and its neighbours). The edges are amplified by the positive weights, while the center's negative weight contributes to the focus on sudden changes.

Question 2(d):







Your comments:

***Comment the results.**

AA: Our image smoothed more since the first kernel blurres our image and when we apply the kernel A on our blurred image it become more smoothed and blurred. This process could result in a considerable decrease in high-frequency details and edges.

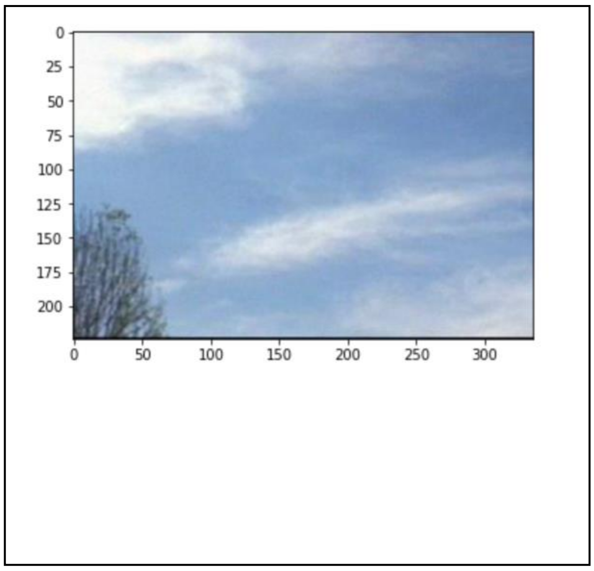
AB: By applying kernel A noises are reduced and image is smoothed. If we apply kernel B on our smoothed image the remaining edges emphasized. This kernel is used in feature enhancement techniques. We can see using this kernel the border is hardly visible.

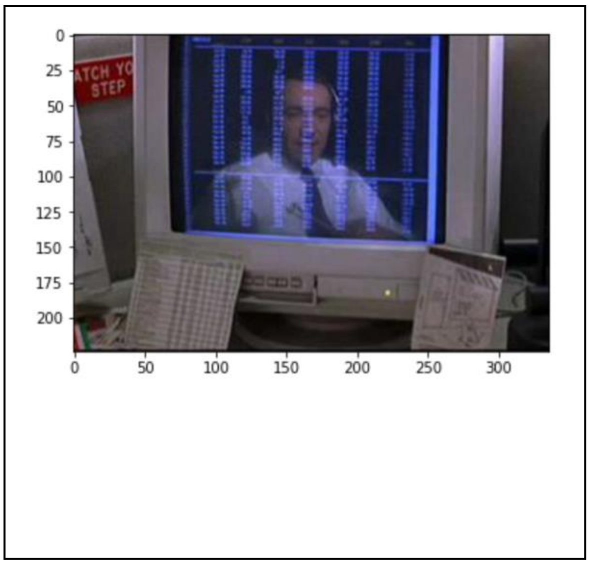
BA: By applying kernel B first we emphasize the edges and then we smooth the result. By using this filter we first detect edges and emphasize them and then by blurring the image we reduce their emphasis. As it is shown above, we can see the border is more visible since we first preserve it by applying kernel B and then smooth the image.

Why the results of AB and BA are not same? This is becasue covolution operation is not commutative. In this operation the order of applying kernel matters, so that is why the result of AB and BA are not same.

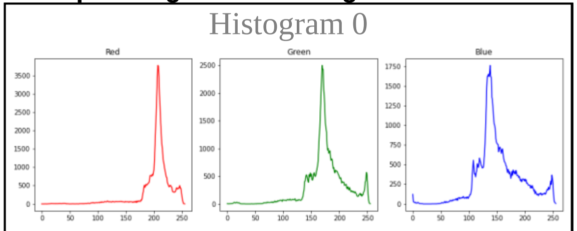
Histograms
Question 3(a):

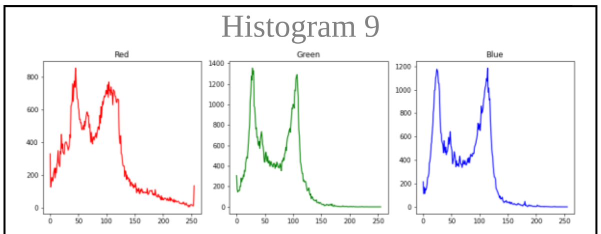
Two non-consecutive frames:





Corresponding colour histograms:





Your comments:

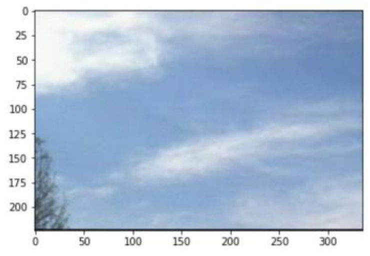
Create histogram for each frame:
1- I defined a 1x256 numpy array (each channel histogram array) for each channel (1x256x3) which represent color values from 0 to 255 in that channel.
2- I iterate through image for each channel and read the value of the pixel (which is a color in that channel) and increase the counter in the histogram array (which its index is the same as the color value).
In practice, we are recording how frequent is a color value in each channel and store it in a array. Then I plot the histogram array for each channel, save and return them.

We can observe that for each channel in frame 0 higher values(brighter colors) are more frequent, means that the image is bright. However, we can see in frame 9 low values (dark colors) are more frequent, since the image is darker.

Question 3(b):

Two consecutive frames:

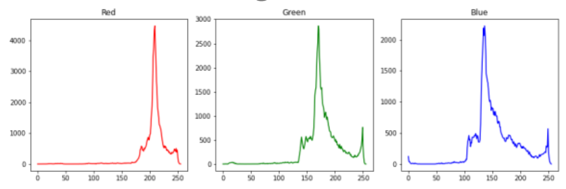
I8



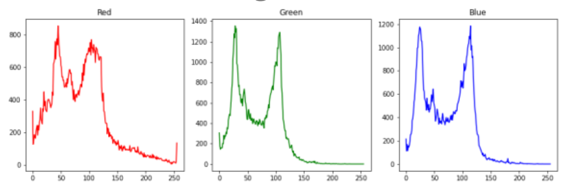
I9

Histograms:

Histogram of I8



Histogram of I9



Intersection result

7562.0

Intersection:

$$HI^{(i)} = \sum_{j=1}^{256} \min(H_1^{(i)}[j], H_2^{(i)}[j])$$

$$HI_{\text{RGB}} = \frac{1}{3} \sum_{i=1}^{3} HI^{(i)}$$

To calculate intersection value for two given histograms, we should
1- compare elements of each channel of each histogram with each other and pick the minimum element.
2- add them together (min_sum).
3-Then we take average of calculated values of each channel(min_sum) .

For normalization, we also calculate sum of max elements of each two elements of every histogram in each channel (max_sum). Then we divide sum of elements in min_sum by sum of elements in max_sum.

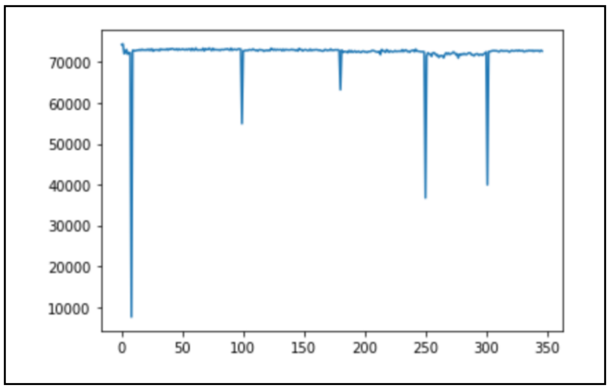
$$NHI^{(i)} = \frac{\sum_{j=1}^{256} \min(H_1^{(i)}[j], H_2^{(i)}[j])}{\sum_{j=1}^{256} \max(H_1^{(i)}[j], H_2^{(i)}[j])}$$

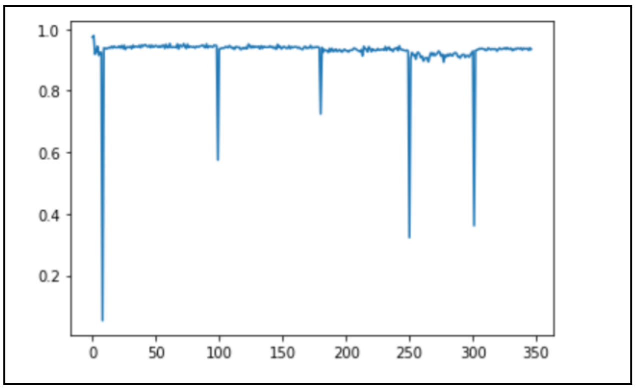
We can observe that for each channel in frame 8 higher values(brighter colors) are more frequent, means that the image is bright. However, we can see in frame 9 low values (dark colors) are more frequent, since the image is darker and the scene has changed.

We calculate the intersection of frame 7 and 8 which are in the same scene: 72263.33333333333
Also, we have a scene change between frame 8 and 9 and their intersection value is : 7562.0
This significant drop in intersection value shows that the scene is changed and the two consecutive frames 8 and 9 do not have same pixel colors and intensities.

Question 3(b):

Intersection result for a video sequence:





Your Comments:

***Does that change the results? Plot the intersection values over time and the normalised intersection values, and save the corresponding figures. Show and comment the figures in the report.**

For each frame we calculate histograms of that frame and store it in a list. Then we calculate intersection and normalized intersection values for each two consecutive frames and store them into array and plot them.

Normalisation does not change the result of the intersection, It just scaled our y-axis values in order to make analysis on the histogram easier. Normalization is helpful when we want see absolute similarity without pay attention the overall intensity into consideration. Since the values are normalised they are between 0 and 1 so if two consectutive frames are so similar their intersection value is 1.

Question 3(c):
Comments:

Discuss in the report the following: What does the intersection value represent for a given input video?

The intersection value shows the amount of similarity between two histograms. So it means how much the color or intenisty in consecutive frames are similar.

Can you use it to make a decision about scene changes?

Intersection value between histograms can show scene changes between frames of the video. For example if intersection value is high, it means that two consecutive frames are so similar. When we see that the intersection value between consecutive frames drops suddenly it may shows that the scene has changed between 2 consecutive frames.

How robust to changes in the video is the histogram intersection?

Histogram intersection can handle slow transitions and gradual changes in lighting in video. However when scene strurctures change without considerable changes in pixel values (color) and intensity, it may not have a good performance. This is because histogram intersection value is sensitive to value of the pixels (color and pixel intensities).

When does it fail?

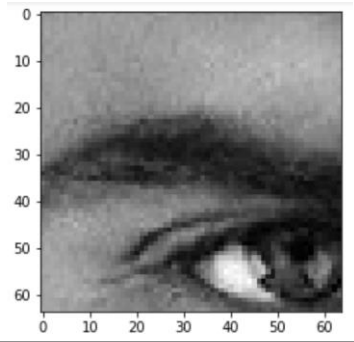
As we said earlier, when we have changes in structures (contents) of scenes without having considerable changes in pixel values (means that colors and intensities are similar), it can fail. For example having 2 frames with same colors and intensities but different scene contents.

When pixel values changes fast, this method may not have a good performance. (example: when the pixel values change fast since we have moving objects)

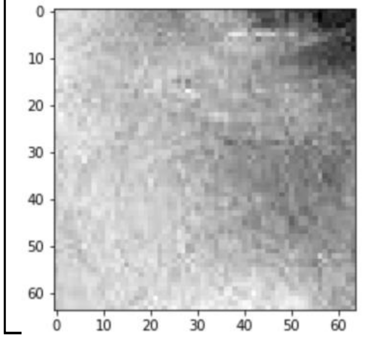
Texture Descriptors and Classification
Question 4(a)

Three non-consecutive windows

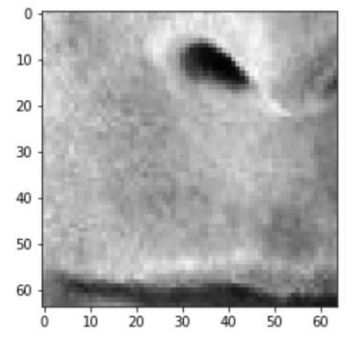
W1



W2

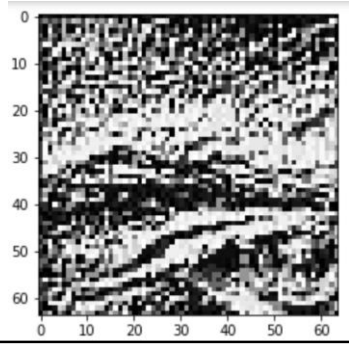


W3

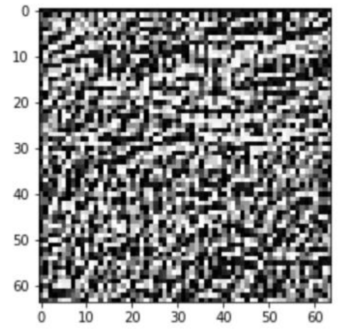


LBP of windows

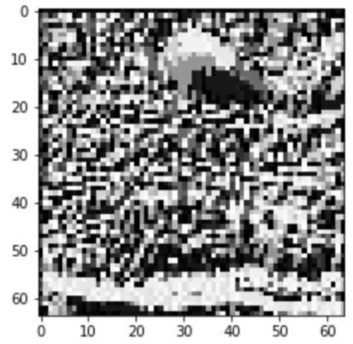
LBP1



LBP2

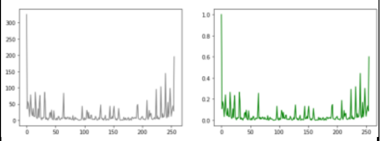


LBP3

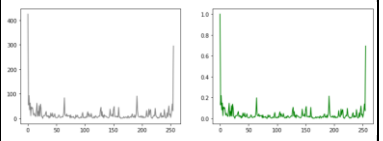


Histograms of LBPs

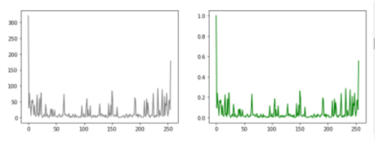
H1



H2



H3



Question 4(b)

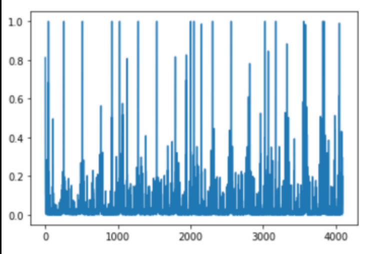
Two example images:

Face image

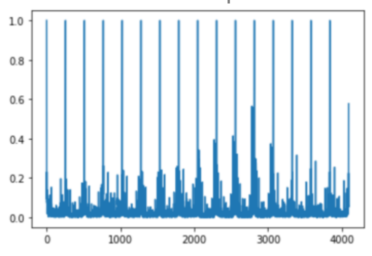
Car image

Descriptors:

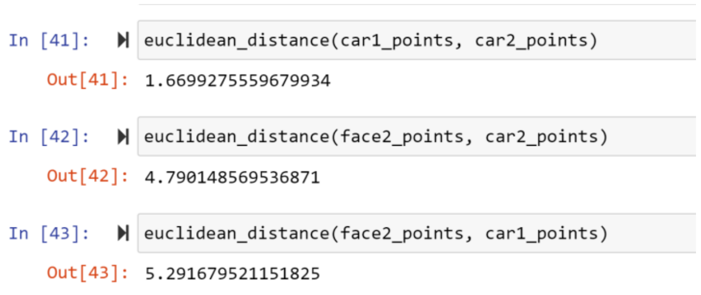
Face descriptor



Car descriptor



Your comments:



```
In [41]:   euclidean_distance(car1_points, car2_points)
   Out[41]: 1.6699275559679934

In [42]:   euclidean_distance(face2_points, car2_points)
   Out[42]: 4.790148569536871

In [43]:   euclidean_distance(face2_points, car1_points)
   Out[43]: 5.291679521151825
```

Window_size = 64
We can observe that face_2 descriptor and car_1 descriptor can do classification using Eulidean distance since their global descriptor values are compeletly different which makes them suitable for classification.
By comparing their Eulidean distance we see that the distances of cars from face are high while the distances of two cars descriptors are low. Since car_2 descriptor is also similar to car_1 so their distance is smaller than face_1 and car_1.

To calculate LBP codes of the image:
1- we first divide the image into non overlapping windows (for example 256x256 image has 64 non overlapping windows with window size 32x32).
2- For each pixel in the window we find its 8 neighbours in the window and check if its value is bigger than its neighbours concat “1” to the binary code otherwise we concat “0” to this pixels’ binary code.
3-So after the iterations finished we have a list of binary codes for each pixel. The challenge here is that if we are in border pixels of the window we cannot find 8 neighbours for it. So to solve this issue we add a zero padding to each window before starting calculations for LBP codes of the window.
4-In the end we change the binary values to decimal values (base 2 to base 10) and return the LBP decimal codes for that window.

To calculate histograms of each window:
1- we pass LBP decimal codes for each window to create_histogram function,
2-then we iterate through LBP codes and record the frequency of occurance of each LBP value in a array.
3-To normalise the array we divide it by maximum number in the array.
4- At the end we plot normalised and raw array as normalised and raw hsitograms.

To plot the texture classified image we copy the decimal LBP codes of each window into the same position in the ouput grid.

To perform classification:

For example we want to classify car images from face images. We should create a global descriptor for face and car and then calculate their differences with a method (like Euclidean distance of histograms).

$$d = \sqrt{\sum_{i=1}^{n} (x_{2i} - x_{1i})^2}$$

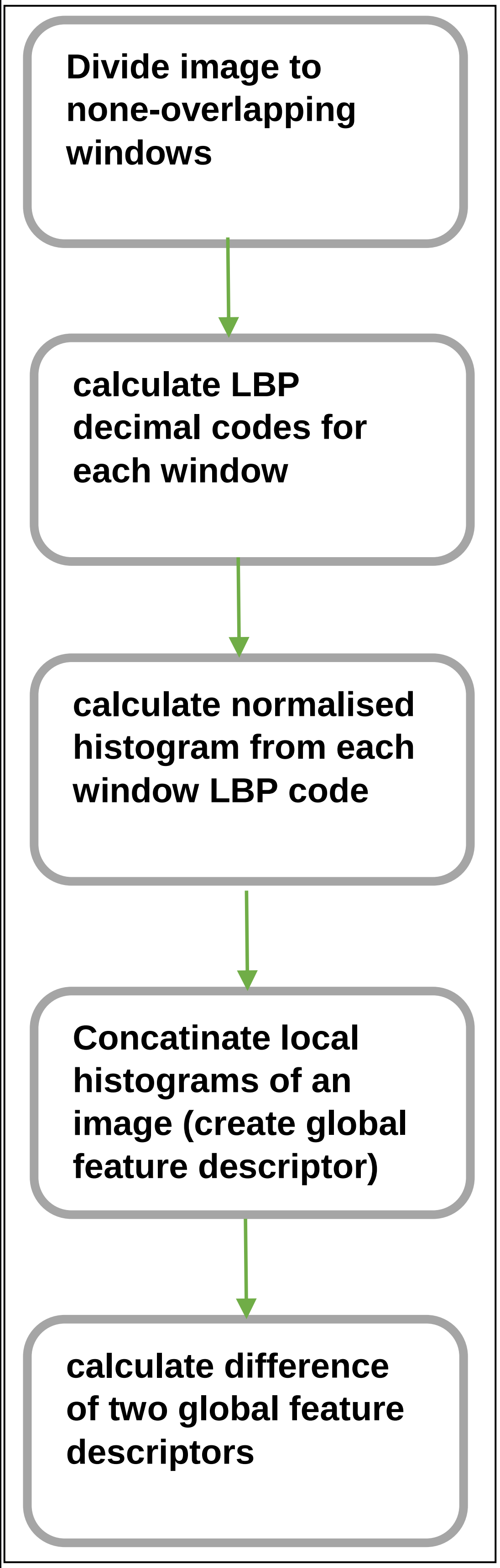
Then based on the value of the difference we can do classification (for example the difference between two car images is significantly less than it between face and car image).

To calculate global feature discriptor:

we iterate through non-overlapping windows of each image and concatenate each windows LBP codes normlaised histogram. Then this concatenated normalised histogram become our global feature descriptor for that kind of image.

Question 4(b)

Block diagram of classification process


Divide image to none-overlapping windows
calculate LBP decimal codes for each window
calculate normalised histogram from each window LBP code
Concatinate local histograms of an image (create global feature descriptor)
calculate difference of two global feature descriptors

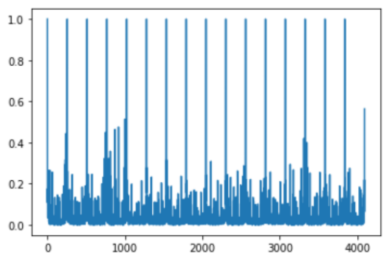
Discuss in the report alternative approaches.
1- Bag-of-Words (BoW) Model:
Using k-means algorithm to group similar local descriptors and assign each local descriptor to its nearest cluster center(called visual vocabulary).To represent an image, we count the occurrences of the cluster centers. we look at each region of the image and compare it to the cluster centers in our vocabulary. In the end, we count how many times each visual word appears in the image, and use this count as a feature vector to represent the image.

2- Spatial Pyramid Matching :
We divide the image into multiple spatial regions at different scales (creating a pyramid structure). Then, for each section, we calculate LBP and use a weighted sum of the local descriptors to create the global descriptor.

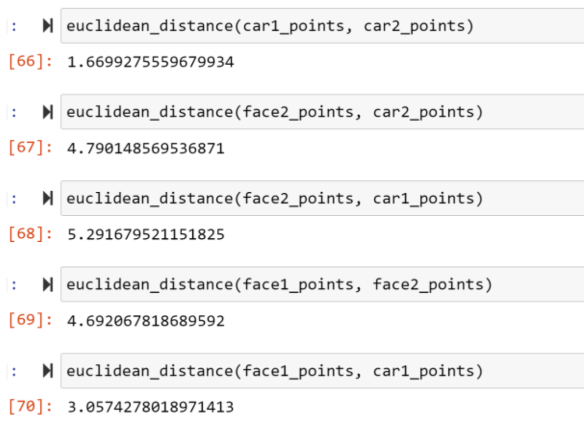
Comment the results in the report.

Is the global descriptor able to represent whole images of different types (e.g. faces vs. cars)? Identify problems (if any), discuss them in the report and suggest possible solutions.

Face_1 global descriptor:



The problem is with face_1 global descriptor because its histogram values is so small and its global descriptor shape is different from other faces so in the classification the euclidean_distance assumes they are in the different classes.
Since cars global descriptor values are also small, using euclidean_distance we cannot classify face_1 and cars classes from eachother. How ever we can see that other faces global descriptors are compeletly different from cars global descriptor and that is why we can classify them with other faces descriptors.
To avoid this issue we can use another approachers to form global descriptor from histograms which are discussed above.



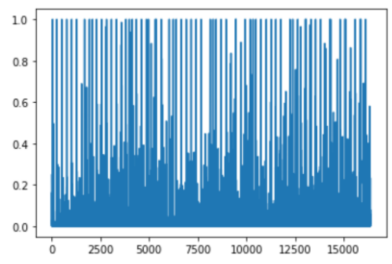
```
euclidean_distance(car1_points, car2_points)
[66]: 1.6699275559679934

euclidean_distance(face2_points, car2_points)
[67]: 4.790148569536871

euclidean_distance(face2_points, car1_points)
[68]: 5.291679521151825

euclidean_distance(face1_points, face2_points)
[69]: 4.692067818689592

euclidean_distance(face1_points, car1_points)
[70]: 3.0574278018971413
```

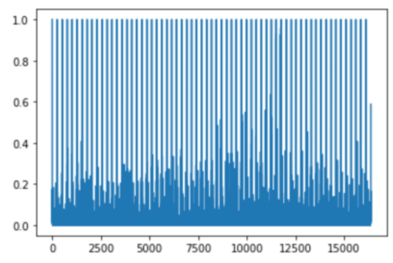
Question 4(c)

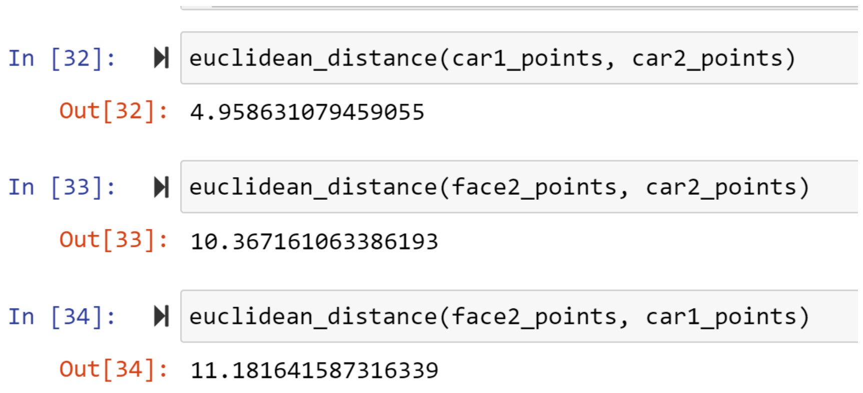
Your comments:

Decrease the window size and perform classification again. Comment the results in the report.

Window size 32
Face 2 , Car 1







```
In [32]:  euclidean_distance(car1_points, car2_points)
Out[32]:  4.958631079459055

In [33]:  euclidean_distance(face2_points, car2_points)
Out[33]:  10.367161063386193

In [34]:  euclidean_distance(face2_points, car1_points)
Out[34]:  11.181641587316339
```

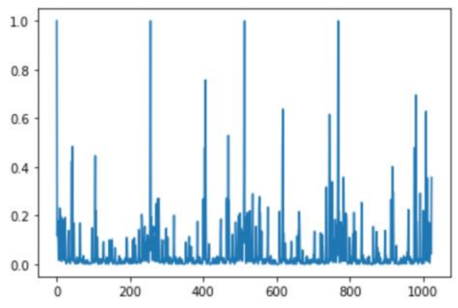
We see that differences values become higher after decreasing the window size. The performance of our classifiers are still as the same. The only change is that by dividing the window size by 2 the differences are multiplied by a number around 2.

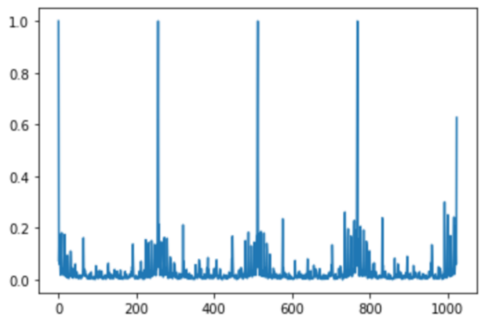
Question 4(d)

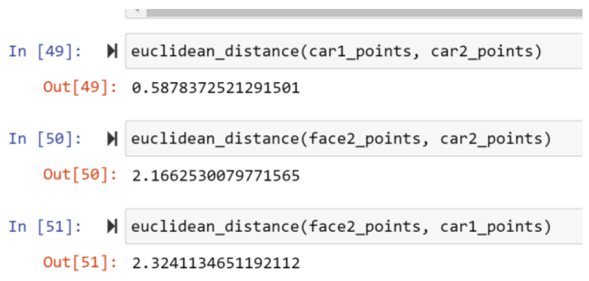
Your comments:

Increase the window size and perform classification again. Comment the results in the report.
Window size 128

Face 2, Car 1







```
In [49]:  euclidean_distance(car1_points, car2_points)
Out[49]:  0.5878372521291501

In [50]:  euclidean_distance(face2_points, car2_points)
Out[50]:  2.1662530079771565

In [51]:  euclidean_distance(face2_points, car1_points)
Out[51]:  2.3241134651192112
```

We see that differences values become lower after increasing the window size. The performance of our classifiers are still as the same. The only change is that by multiplying the window size by 2 the differences are divided by a number around 2. To conclude, it seems that the window_size has inverse relationship with our differences.

Question 4(e)

Your comments:

***Discuss how LBP can be used or modified for the analysis of dynamic textures in a video.**

Spatiotemporal LBP:
In this approach, we not only take into account the pixel's spatial neighborhood in the current frame, but we also incorporate information from the same pixel's neighborhood across consecutive frames.
So 1- for each pixel in the frame we calculate the LBP code for its 8 spatial neighbors.
2- Then we add corresponding pixel's value in the same spatial position in the previous or next frames (or both) in the LBP computation.
3- After that we concatenate LBP codes to form a spatiotemporal LBP code for the pixel.

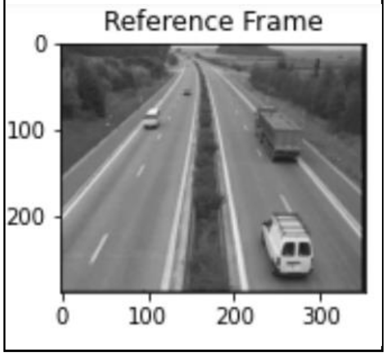
Its advantage is that it can calculate dynamic texture patterns like moving object. However it has more computational complexity and needs tuning hyperparameters like window size.

3D LBP:
like above method we extend LBP computation by including the corresponding pixel's value in the same spatial position across neighboring frames. Then we concatenate or combine the LBP codes from the spatial and temporal dimensions to form a 3D LBP code for the pixel.

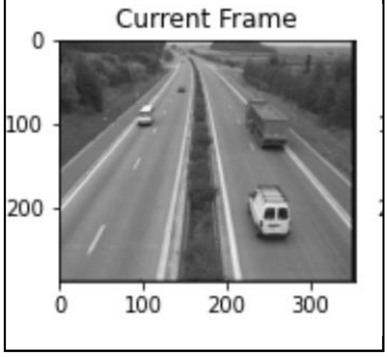
DObject Segmentation and Counting
Question 5(a)

Original frames:

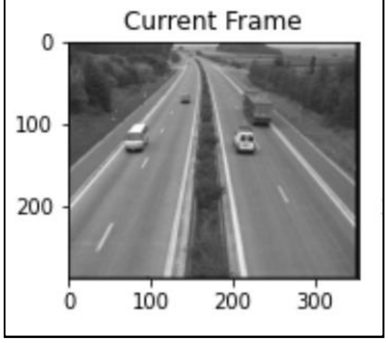
Reference frame



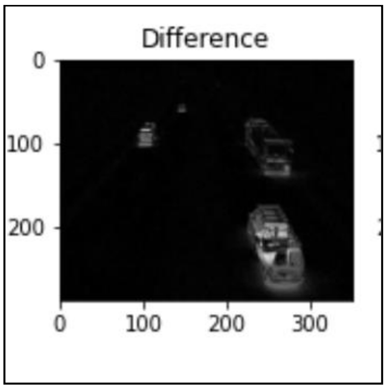
Selected frame 1

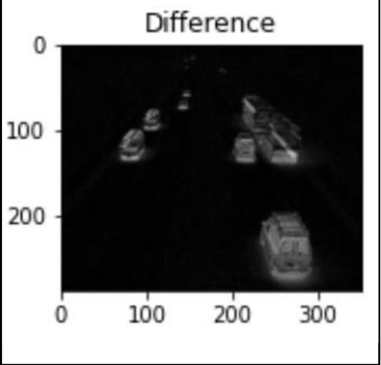


Selected frame 2

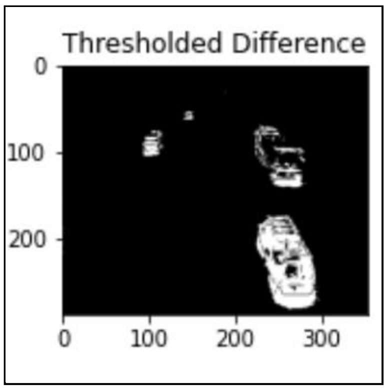


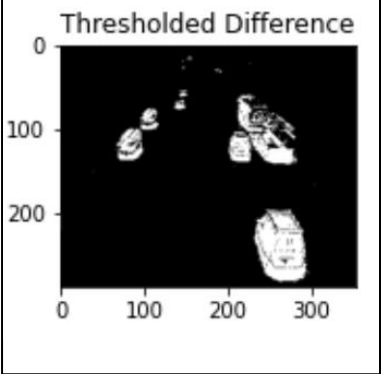
Frame differencing:





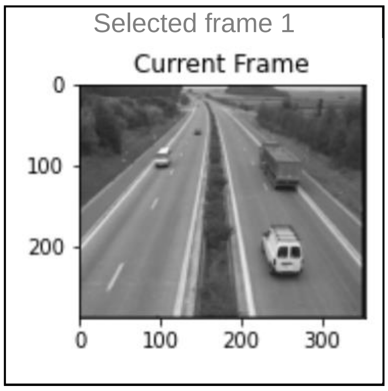
Threshold results:

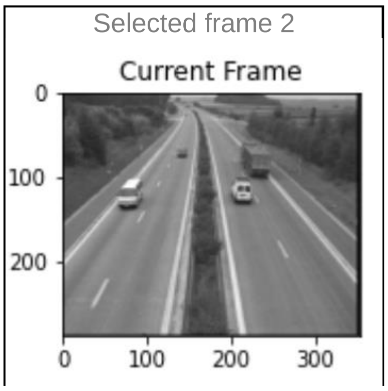




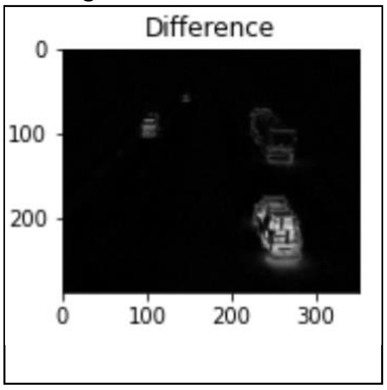
Question 5(b)

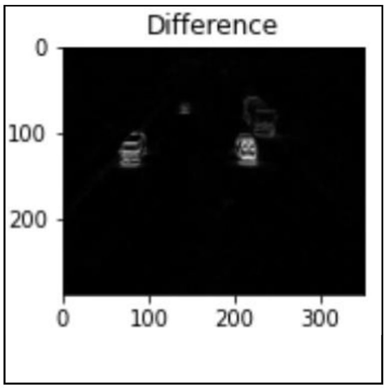
Original frame:



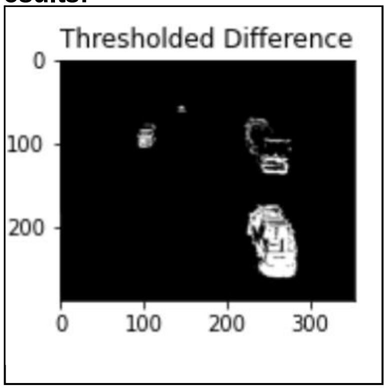


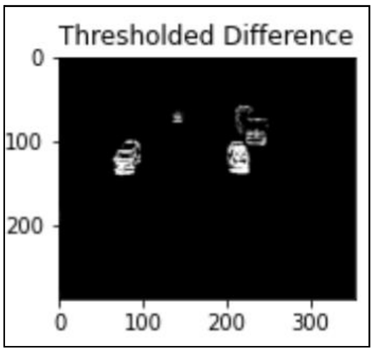
Frame differencing:





Threshold results:



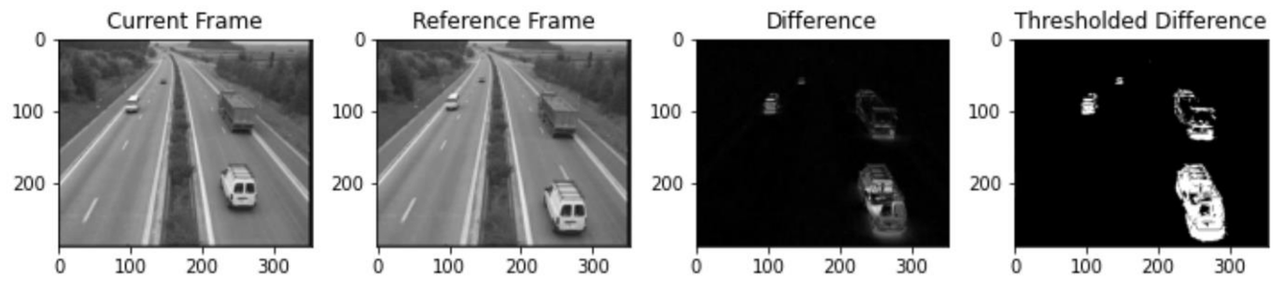


Your comments for 5a,5b:

We first convert each frame of the video to the grayscale image and save it in a list.

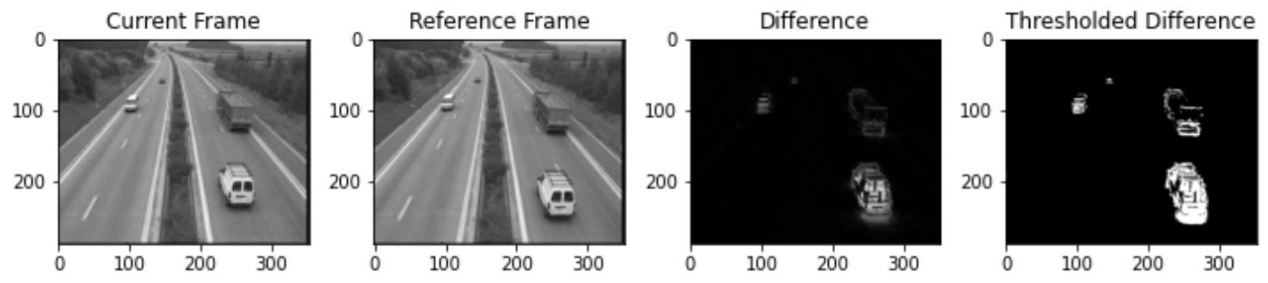
5a:
We pick the first frame of the video as a reference frame and for each frame we calculate the absolute difference between that frame and the reference frame.
One of the challenges here is overflow of the values so we use np.float32 and then cast it to np.unit8 to avoid overflow.
We apply the threshold value on the difference (pixels below threshold are zero (black), Otherwise 255(white)). Using this way we can separate background and foreground.

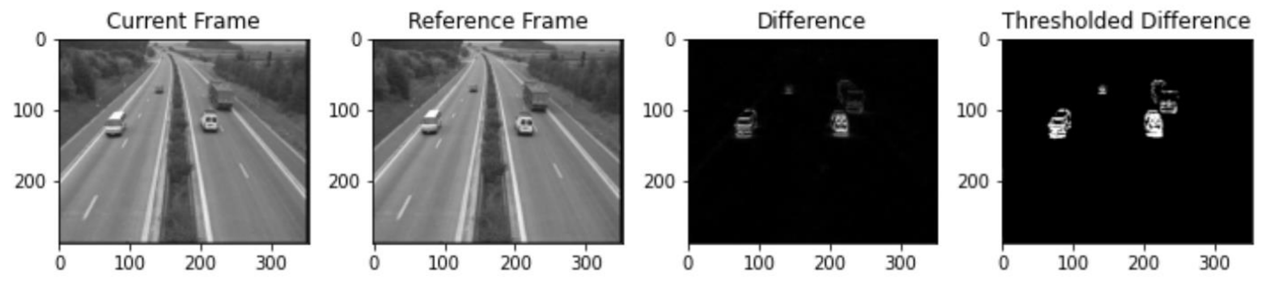
At the end we plot our results and save them.



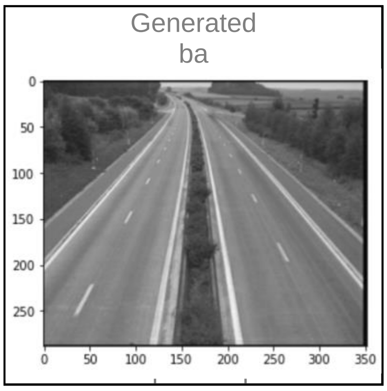
5b:

To calculate the frame differences of two consecutive frames we did the same calculations as part a but we pick previous frame of each frame as a reference frame.





Question 5(c)



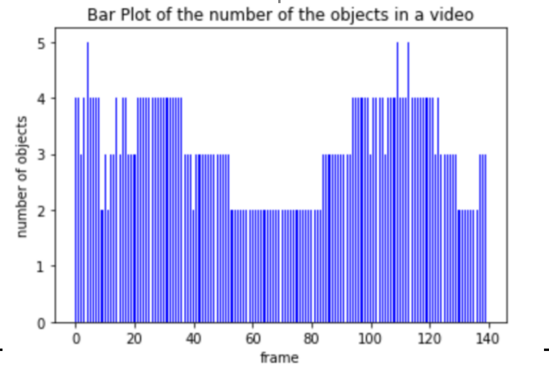
Your comments:

To generate a background frame:

1- we first pick the first frame as a reference frame
2-then we iterate through the frames and update our refrence frame in each iteration using weighted sum of current reference frame and current frame.
reference_frame = (alpha * frame + (1 - alpha) * reference_frame)
3-Then we adjust the alpha value by looking at the result to find the best alpha for the cleanest background picture (alpha = 0.027).

Question 5(d)

Bar plot

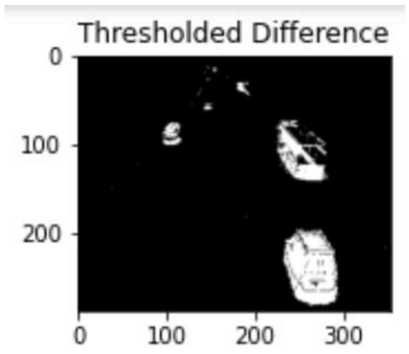


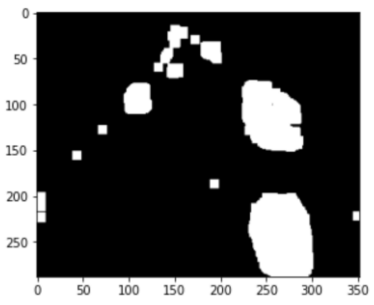
Your comments:

Discuss in the report the implemented solution:

For this part, We first calculate the frame differences from the generated background in the previous part using frame differencing function in part a. We should calculate the number of objects in thresholded frame differences to solve the question. Since in the thresholded results the background is balck, One way is to calculate the white holes in the picture which represent our objects. The problem is that there is some black shadows and streaks in our white holes which makes conuting each white connected component difficult. To solve this we use dilate method to whiten neighbour pixels of each white pixel in the picture. To do this we create an output black grid and iterate through image if we see a white pixel we whiten the specified rectangle indexes below in our black grid:

Rectangle height: [white_pixel_height: whitepixel_height + kernel_size]
Rectangle width: [white_pixel_width: whitepixel_width + kernel_size]





get_contours(image): In the next step we should count and find each connected white component (or contour) in the dilated image. We first iterate through image pixels and if we find a white pixel we use Depth first search algorithm to find the other white neighbours of that pixel in each component.At the end we store each white components objects indexes in an array and append it to our list of white components.

dfs(image, i, j): In the Depth first search algorithm we find the indexes of the obejcts in each white connected component: we first create two lists to store indexes of white objects in a component and to represent frontier (or stack which is a LIFO queue). While our stack is not empty we every time pick the last element from it and check if it is white change its color to black in order not to visit it again(representing visited list in DFS algorithm) and save its indexes into our list. After that, we find its 4 adjacent neighbours and add it to the stcak and repeat this process till the stack become empty.

So for each frame in our video we dilate the frame and create its white component list (which each of its element is an array of the indexes of white objects in every white component (contour)). Then by looking at our original frame we find the threshold for considering how many white objects are representing an independent component. We apply this threshold on the length of the each element of the white component list and count the objects on that frame.
After this operation we have a list that each element of the list represent the number of objects in the corresponding frame. In the end we plot this list.

Include advantages and disadvantages:
Advantages:
By using dilation we reduce the impact of black shadows and streaks in our frame. This leads counting our connected components more robust.
By using connected component concept, we can analyse even close moving object in our frame.
Our solution adapts to variations in scene complexity. since we manually find the threshold value to consider how many objects produce an independent component.

Disadvantages:
This method has computational complexity for large frames and videos since we use dilation and connected components concepts.

The solution performance depends on adjusting the hyperparameter value like thresholds and kernel size and shape. Becasuse of these hyperparameters finding the optimal value for them is hard in some cases.

There are false positive in our method since the algorithm can count some obejcts as countors which are not. Also we may have tiny moving objects in scene which are ignored due to the threshold value that we assigned.

The accuracy of our model is dependent to the generated background and low quality background leads to false detections in counting the moving objects in our method.