Introduction to Computer Vision

Coursework

1

Manuela Valencia Toro

240429850

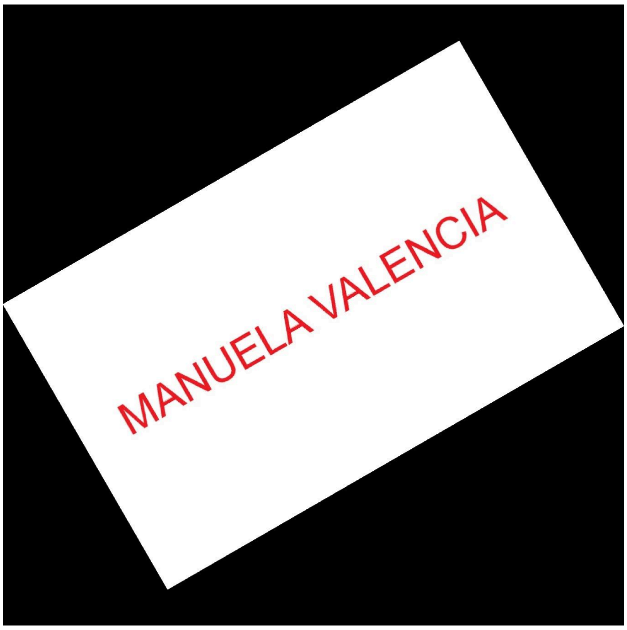
Transformations

Question 1(b):

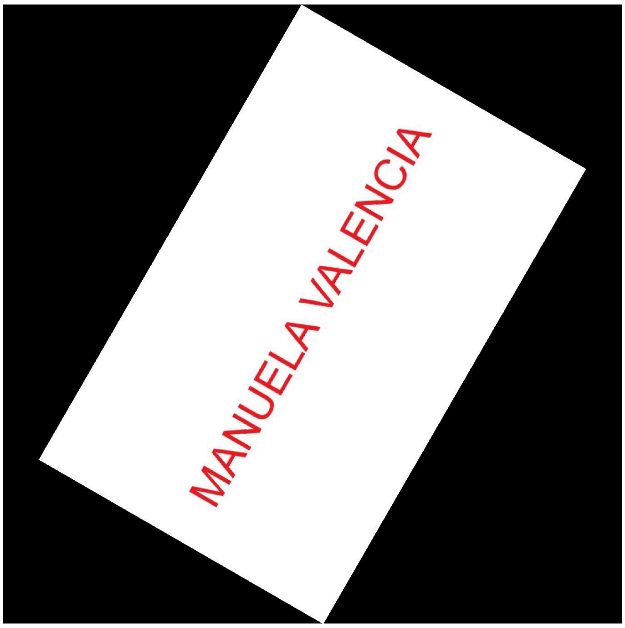
MANUELA VALENCIA

Rotated images:

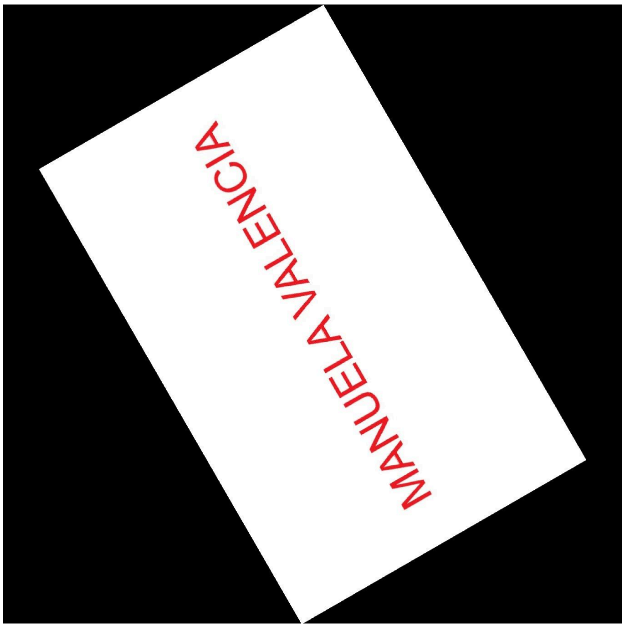
θ = 30 deg



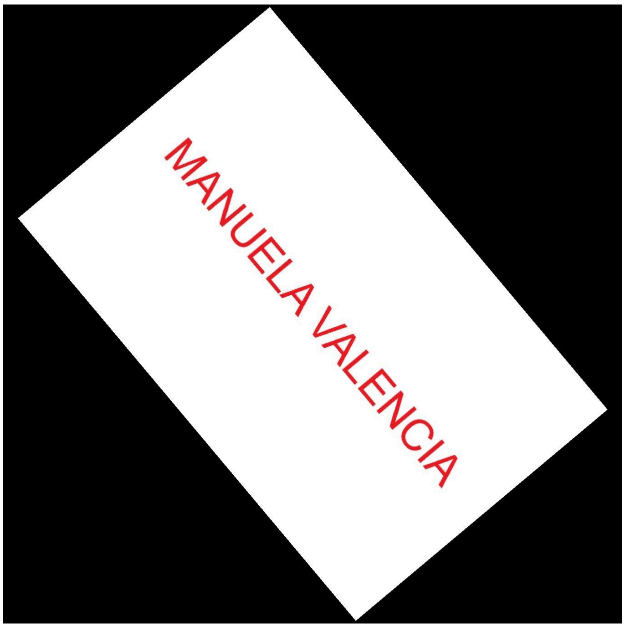
θ = 60 deg



θ = 120 deg



θ = -50 deg



This first part was about the process of image rotation, where using an image as a reference and a certain angle in degrees, each position is multiplied by the rotation matrix defined in the class to get the position that pixel will take in the rotated image. My first implementation took the position 0,0 of the image as the origin, but the returned image was not what I expected. After thinking about it for a while, I figured out that the origin should be exactly in the centre of the image. Firstly, to make it symmetrical and avoid translations, and secondly, to ensure that the rotation is performed around the central axis, because the matrix multiplication will not return a new position for 0,0. So finally it made a lot of sense.

After applying this logic to my code, it was possible to start iterating around each pixel and calculating the new x and y. Next step was to assign the value of this new position to the output image.

There are two ways to assign it, the forward and the inverse mapping. Forward mapping consists of applying the transformation to the input image and mapping each new value in the output. This is a fairly simple implementation, but since the calculated x and y should approximate an integer, there are some pixels that will not be assigned a value after the transformation and this means that a lot of data would be lost.

To improve it I applied the median kernel to finally get an image with non-empty pixels. This solved my problem, but required a lot of additional calculations. Therefore, I finally decided to implement inverse mapping to avoid extra calculations and extra execution time, because this way I can make sure that every position in the output image has a value.

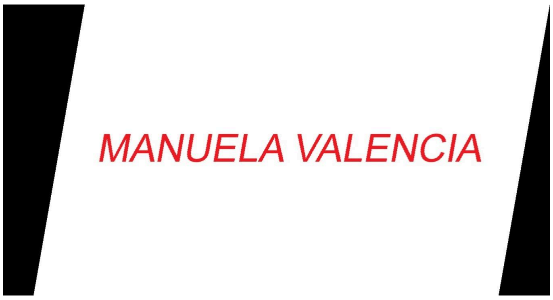
Inverse mapping iterates around each position of the output image, makes the computations to get new x and new y, and finally assigns the value from the input image in that calculated position. This method was used by all next image transformation methods to finally assign the corresponding values to the output, mainly because of its execution time.

After the transformation, I found that even if the image was rotated correctly and each pixel was assigned correctly, depending on the angle, some edges were lost because the new rotated image did not have the same dimensions as the input. To solve this problem, I developed a method that compares the diagonal of the image with the length and width and adds the required columns and rows. After this conversion, my output image looks as desired.

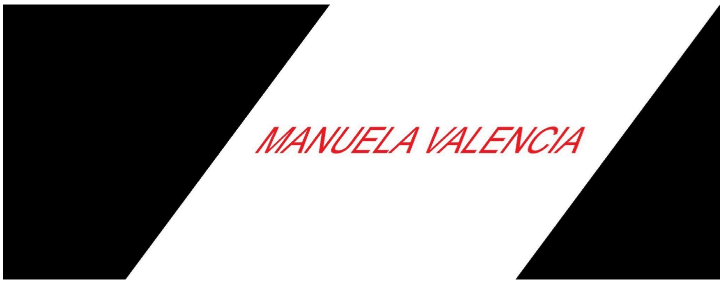
Regarding the output images, they are rotated clockwise Theta degrees. It was pretty interesting to implement this method because it involved a pretty simple process and returns a correctly rotated image.

Skewed images:

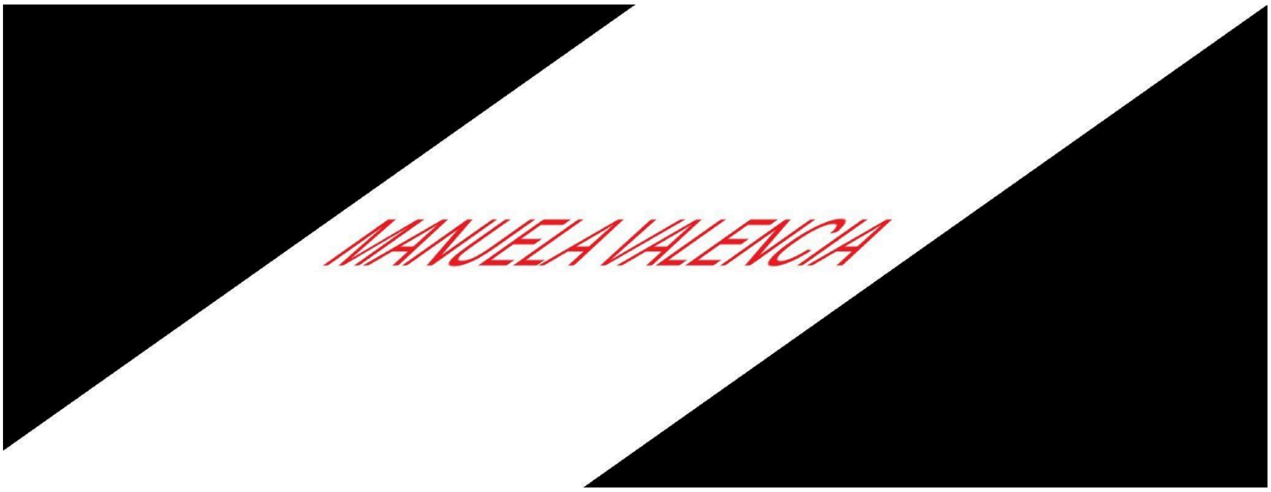
θ = 10 deg



θ = 40 deg



θ = 60 deg



This is the second transformation, Skew. In this exercise, our image was transformed with respect to the x-axis, which basically means y-coordinates were shifted to the right while the x-coordinates remained the same. During implementation, I set the base angle to 90. The transformation will start from this angle, as this is the angle that our image has at each edge before the transformation. So the desired angle transformation is added to our base angle before the iteration around each point of our image begins.

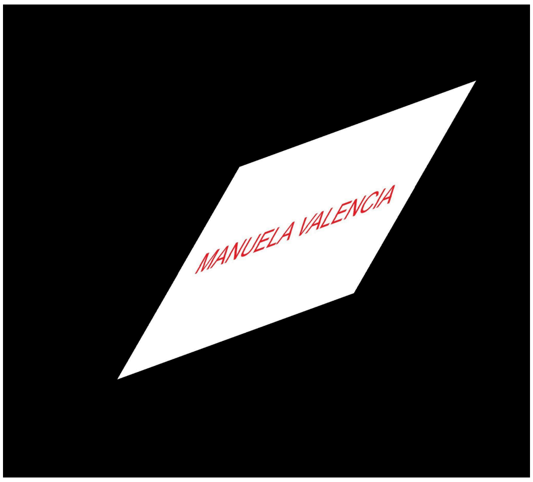
Just like rotation, an image can be sheared by calculating each new position through a defined matrix, the shear matrix, which basically shifts the pixels to the right based on the input theta. After the transformation, as it was mentioned in the first question, inverse mapping was used to map the data to the output. Since the shape of the new image is different from that of the input, some columns were added, in this case by using the triangle at the bottom right of the output.

After these changes the algorithm takes more time to execute the transformation because now it has to visit more pixels, but it is the only way I found to avoid losing data.

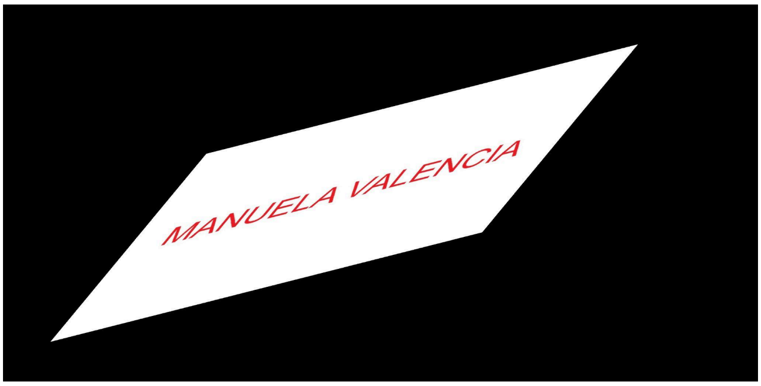
Regarding the output image, the top left and bottom right edges take the desired angle, if the angle is positive they are expanded, while the top right and bottom left corners are reduced to finally obtain an image with the top shifted to the right, as can be seen in the images above.

Question 1(c):

θ_1=20 clockwise and θ_2=50



θ_2=50 and θ_1=20 clockwise



To achieve this, I created 2 methods that call the rotation and skew methods in different order to transform the image.

The first method calculates the rotated matrix and assigns the resulting matrix to a variable to finally call the skew method with the rotated matrix as parameter. The second method calculates the skew matrix and passes it as a parameter to the rotation method. Both images are saved.

One important thing here to note: although the angles and processes are the same, the images are different. This can be explained by one of the properties of matrix multiplication.

Let's say rotation transformation is A, and shear transformation is B.

According to the properties, the matrix multiplication is noncommutative.

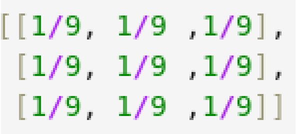
$$A*B \mathrel{!}= B*A$$

This means the order of the matrices while they are multiplied affects the result. Our transformations are basically multiplications by specific matrices, the only thing that changes in the methods is the order and this is why they are different.

Convolution

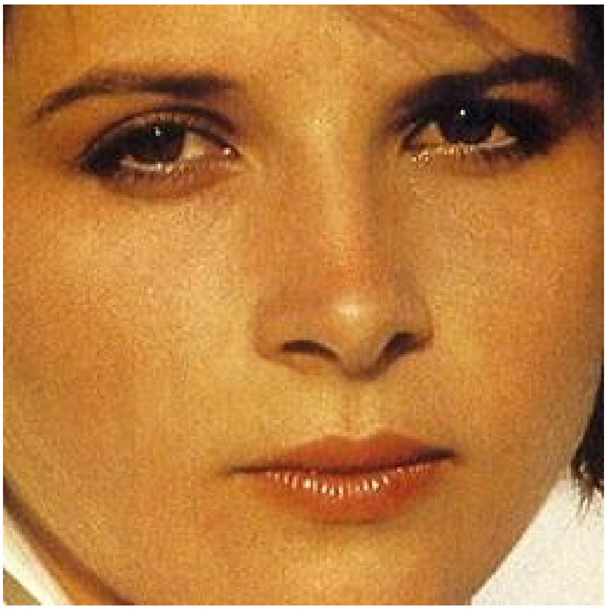
Question 2(b):

Designed kernel:

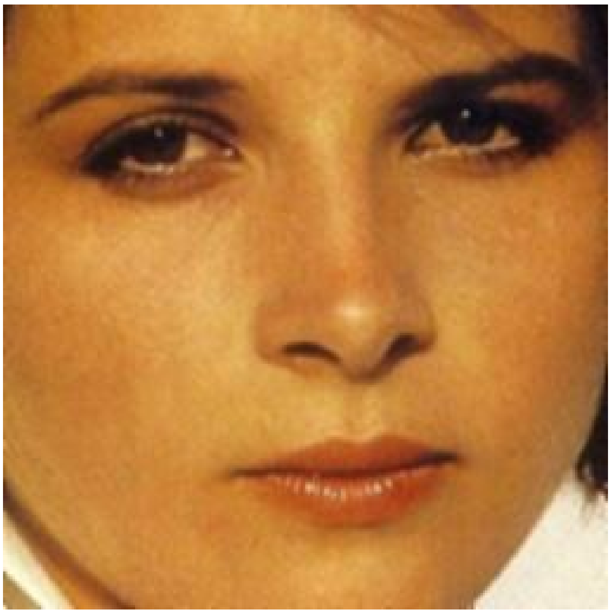


```
[[1/9, 1/9 ,1/9],
 [1/9, 1/9 ,1/9],
 [1/9, 1/9 ,1/9]]
```

Original



Average



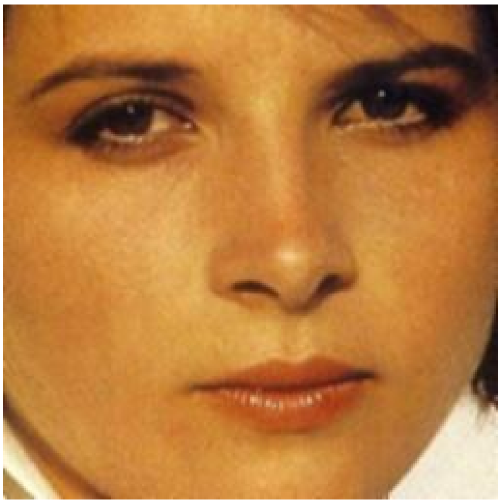
In this task I have implemented 2 different methods.

The first method uses the mean kernel defined above, that can be interpreted as giving equal importance to each pixel in the window and is used to calculate the mean value in a 3x3 window over all the image pixels and to update them with that calculated mean. As each pixel to update must be in the centre of the 3x3 window, the pixels at the border are lost since the iterations will start in column and row 1 and will end in column and row length-1.

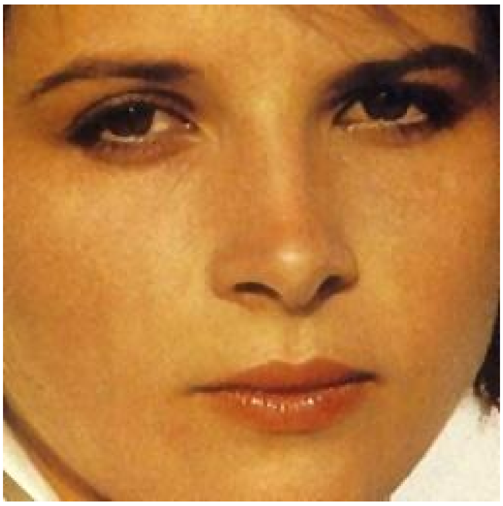
To avoid losing this data, I developed a method that basically assigns this margin to the output image and is able to work with kernels of any size by simply calculating how many columns and rows are lost. In this particular exercise, the lost data may not be a problem, but if you apply a large kernel or a small kernel multiple times, the amount of lost data will increase significantly. This is why I preferred to use this method. From this point on, this method will be used for every convolution. After applying the mean kernel to the image, I obtained a smoothed image. This kernel can be used to reduce noise and blur any image.

To compare the resulting images, I applied the median kernel too. For this, my algorithm iterates in every pixel by creating a nxn window around it, taking the median rather than the mean and assigning it to the centre. The process is basically the same to get the output image. It was a little bit hard to implement because unlike the mean, it was not possible to define a matrix and make simple computations to obtain the value.

Mean



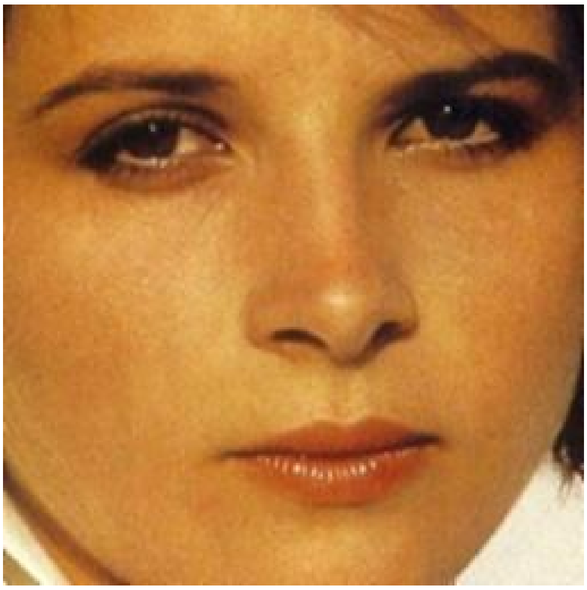
Median

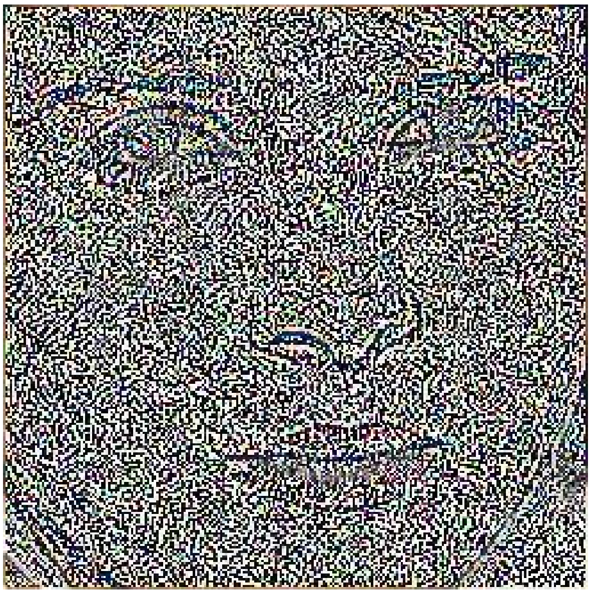


In the case of the median kernel, it is easy to note that it is more effective by reducing the noise in the image, but, at the same time it preserves better all the edges. From this I can say even if it is harder to implement, it could show a better performance most of the time, of course, depending what the objective is. For example, if you want a smooth image it is better to use a mean filter, but if you only want to reduce noise, you should use the median kernel.

Question 2(c):

Kernel A **Kernel B**





Here I first created both kernels as matrices and assigned them to variables, because this is necessary in order to use the method I defined before.

Once I had both variables, I could apply each kernel to the same image I used in the last exercise. When I ran my method for the first time, the resulting image of the A kernel was quite strange. With lots of colourful pixels around the image and no particular pattern that would describe in which cases it could be used.

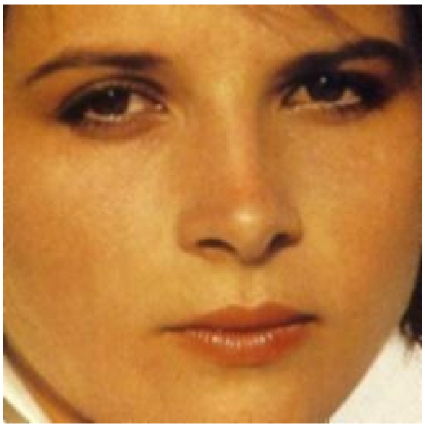
I started trying different things and checking my code, but I remembered how important it is to normalise any kernel before applying it to an image, especially to avoid noise.
After normalising only the first kernel, because the second one can not be normalised since the sum is 0, I got the above images.
From the first image, I can conclude that kernel A can be interpreted as a mean filter that gives more importance to the centre value and less importance to the diagonals.

As for kernel B, the second image basically shows all the horizontal and vertical contours of the input image. Kernel B is a Laplacian kernel, which can basically be defined as the two-dimensional equivalent of the second derivative, which, as we saw in class, is very useful for finding edges. It calculates the difference between the horizontal and vertical values and normalises them (the 4 in the middle is used to normalise the kernel).When a change is detected, this is, when the value of the difference is high, the pixel is going to take a high value, but when the difference is low, meaning an homogeneous region, the pixel is going to take a value near to 0, given an image with all the vertical and horizontal contours highlighted.

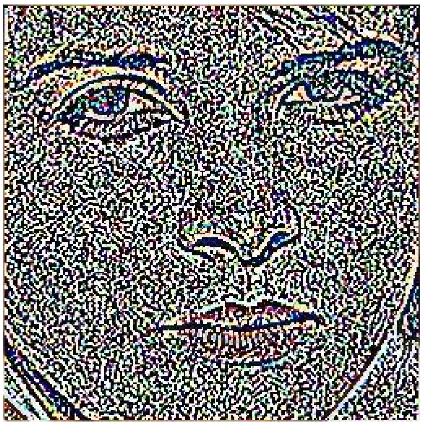
Fill the available spaces for your answers.

Question 2(d):

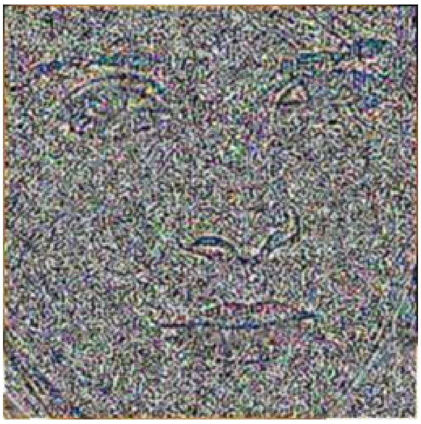
A followed by A



A followed by B



B followed by A



A followed by A: results in an even smoother and blur image that looks quite similar to the image obtained by applying the mean filter. If I apply the A kernel 2 times, I can remove additional noise that was not removed during the first application. However, at the same time, it is possible to lose some details from the input image

A followed by B: Since the Laplacian kernel is very sensitive to noise, applying it to a denoised image, A, ensures better performance to finally obtain an image where the edges are even more emphasised.

B followed by A: Since the order of the operators matters, this image is different from the last one, even if the kernels are the same. Here I have obtained an image with less noise and blurred edges, so it is more difficult to recognise them visually. I looked for an application for this kernel combination, but in the end it did not make sense to me. If I want to see the edges, I want to highlight them as much as possible.

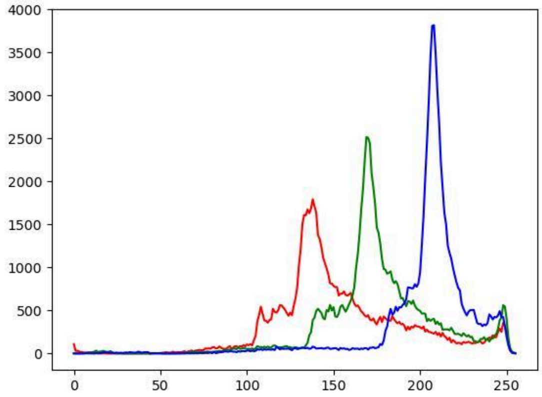
Histograms

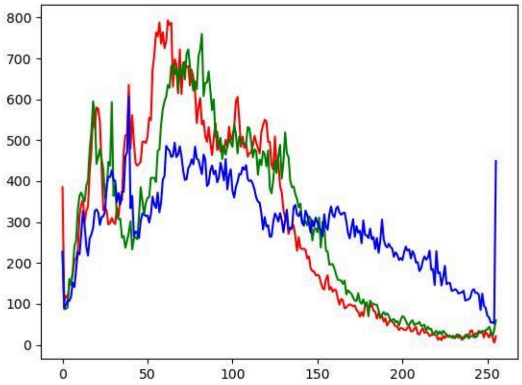
Question 3(a):

Two non-consecutive frames:

frame 1 **Frame 100**

Corresponding colour histograms:





To handle this part of the coursework, I have implemented some helper methods:

Get_frames: returns a list of all frames of a given video.

Histogram: given a frame and bins, returns the colour histogram of the image.

Normalise: returns each histogram (one for each colour), dividing each bin by the total sum.

Using these helper methods I have created 2 main methods. One to create the histogram of each frame, and the other to calculate the intersection between 2 given histograms. This intersection is given by the minimum value between the two histograms for each bin. Even though these implementations were quite simple, I sometimes had to use some libraries such as np.histogram to compare my results, mainly because when working with many images it is a bit difficult to detect errors in the code.

Now as for the histograms of the images shown above, it is easy to see that they are quite different and there are a few aspects worth mentioning here.

One important difference between the images is the colour.

The first image has mainly white and light blue pixels, while the second image is a bit more complex and has a combination of light pixels, such as the desk, and dark objects, such as the people's clothes.

As the histogram counts the number of pixels with each specific value from 0 to 255, if an image has a lot of light, as the first image, its corresponding histogram will have a concentration in the beans near to 255, and an image with more darkness and black objects will have the highest values at the beginning of the histogram.

Now, by analysing the histograms, it is easy to conclude the first one belongs to an image with a lot of white and blue pixels, while the second, shows more colour complexity and a combination of dark pixels and lightedpixels. Even if you can define to which image belongs to a specific histogram, you can't recreate an image from it because a lot of information is lost.

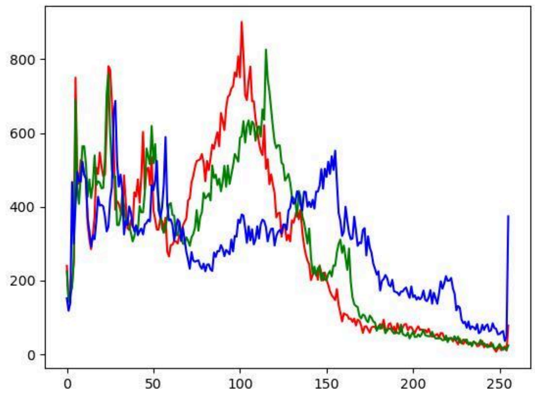
Question 3(b):

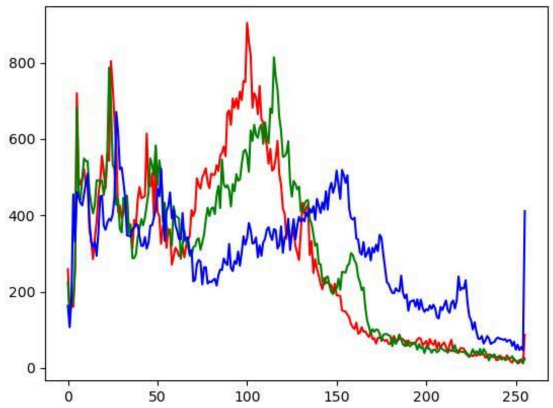
Two consecutive frames:

Frame 187

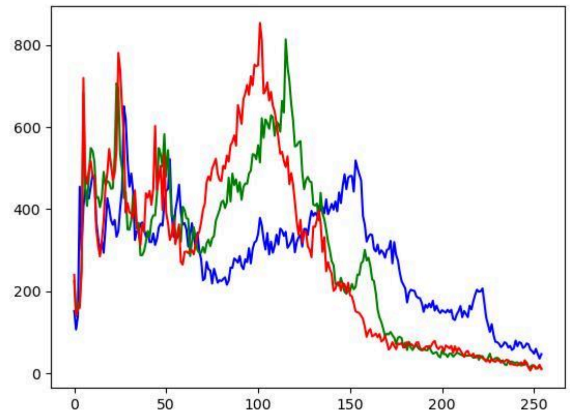
Frame 188

Histograms:





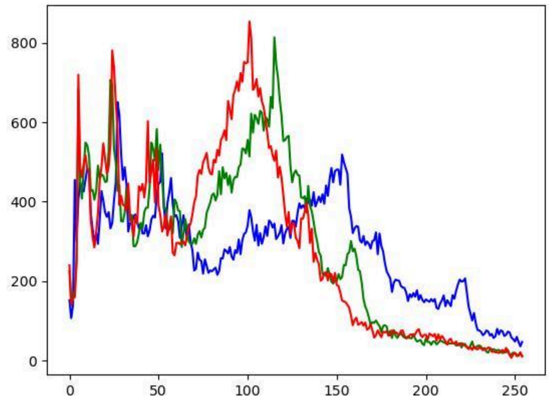
intersection



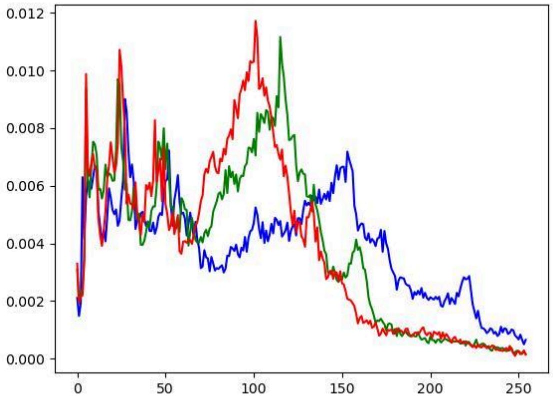
3(b):

Intersection result for a video sequence:

Intersection result



Normalised intersection result



In this section, the histograms belong to two consecutive images that are quite similar. They have the same objects and scenarios, but some areas change a little, such as the woman on the left, as she walks in the video, in the second image she is a few pixels to the right.

If you look at the histograms, they look pretty similar as you would expect, only some bins show a small difference, but in general the distribution is the same, with a high number of dark tones for each of the colours and more light tones of blue than red and green.

Now as for the intersection of these images, it looks pretty similar to the individual histograms, but only has the peaks that are included in both.

The shape is still the same, with the same peaks and a concentration between 0 and 150 and a downward trend from about 170 until 255, showing that there are not many white or colourful pixels.

Finally, the normalised histogram shows the same trends, peaks and shape as the intersection but the ranges are changed. For this case the bins are still the same, between 0 and 255, however, the Y axis goes from around 0.002 and 0.012. This is basically the only thing that changes and it is done to facilitate data manipulation and interpretation.

Question 3(c):

What does the intersection value represent for a given input video?

The intersection represents the common values between frames, this is, it will represent the similarity or overlap of the colour distribution between them. A value of 0 will mean the frames don't overlap, while, if the value of the intersection is the same as the total count for that bin, it will mean the frames overlap completely in that specific bin.

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

By analysing the intersection between the frames I can conclude how much the scene has changed between the frames with respect to the colour, if the change is big, that is to say, the intersection value is near to 0 for most of the bins, I can conclude the changes are abrupt between frames and make decisions about that.

How robust to changes in the video is the histogram intersection? When does it fail?

The intersection is robust to changes of colour in the video, if any section of the frame changes its colour suddenly it will be represented in the histogram, but if for example, the frame content has an abrupt change with respect to the scenario or characters but the relationship between the colours is the same the intersection will not show it.

In these scenarios it will be needed extra techniques to analyse changes in the video, like for example edges and object detection.

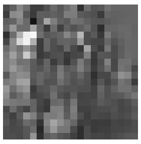
From this section of the coursework I can conclude how useful are the colour histograms when working with images and videos. It was fascinating to work with a video and finally get a graph that represents the colour change between frames and this all by my own hands, without any library. At the beginning one of the hardest things was to debug my code, because the process behind this is a little bit abstract and working with a lot of images makes it difficult to find bugs in the code. I had to analyse most of the histograms before concluding my method was working as expected, but finally, I was able to generate each single histogram for each frame and each pair of frames without any problem.

Texture Descriptors and Classification

Question 4(a)

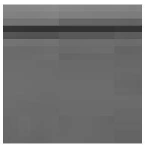
Three non-consecutive windows (window size = 100)

W1-0

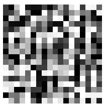


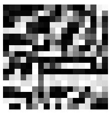
W2-100

W3-140

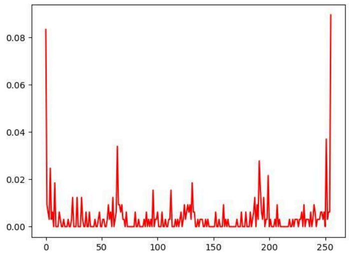


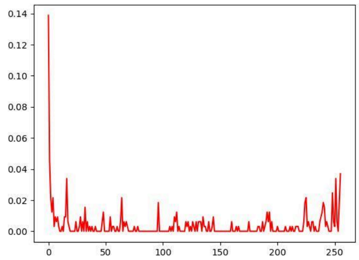
LBP of windows

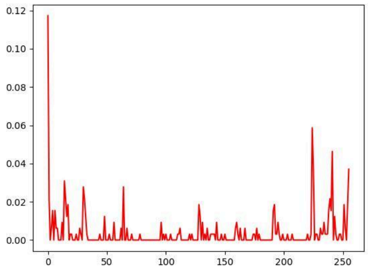




Histograms of LBPs







These are the corresponding windows for the image face-1 in DatasetA, with a window size of 100 pixels.

To get these windows, I first had to convert the image to grayscale to work with each pixel as a single value.

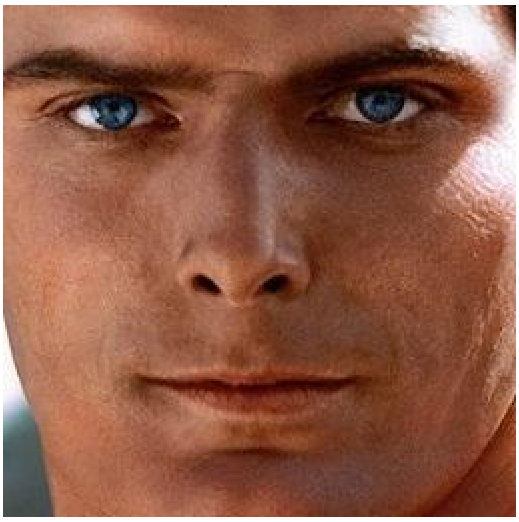
The first images correspond to the windows in the original image, the second part, lbp windows, are the matrices created after applying the lbp process, which basically updates the value of each pixel with the binary relationship to its neighbours converted to decimals.

As you can see, although the input windows do not have much information about the image, the corresponding lbp matrices show certain patterns for each window, such as the line at the top of the second window. This type of pattern determines the similarity between 2 images.

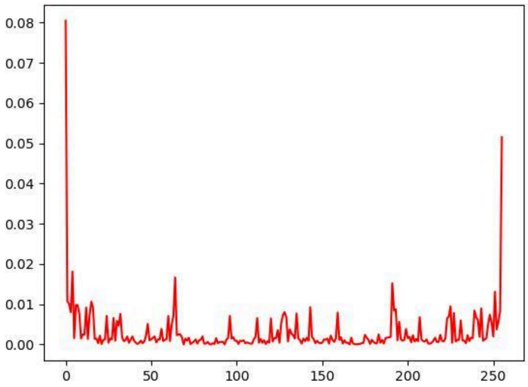
Since these lbp windows are basically matrices, it was possible to represent them as a histogram showing the number of pixels with each colour intensity, which can be seen as a map of the window.

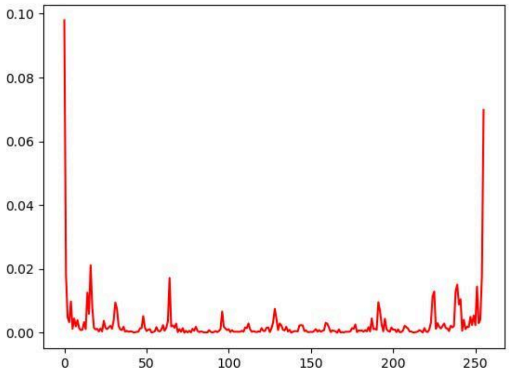
Question 4(b)

Two example images:



Descriptors (window size = 100)





These are the face-1 and car-1 images from the dataset, as the above description, it was necessary to first convert them into grayscale to start iterating over each window of a defined size. In this particular case windows of 100 pixels were used to create each single LBP and finally get a global descriptor that represents the entire image.

Even though the histograms look similar, there are some remarkable differences. For example, the first graph shows symmetry around the bins. If I draw a vertical line in 125, I could see that the right side and the left side tend to be symmetric, however, if I do the same with the second plot, I could find there is no symmetry. Regarding the second plot, I can notice there are more 0 pixels in the image and at the same time, there are more bins around the graph with low values.

Visually, by comparing the peaks, the symmetry, the distribution and the ranges of two global histograms it would be possible to define if they belong to the same category, here, cars or faces. In code, this could be done by computing the distance between them. A small distance will mean the histograms could belong to the same class.

There are multiple factors that will affect the classification based on the distance.

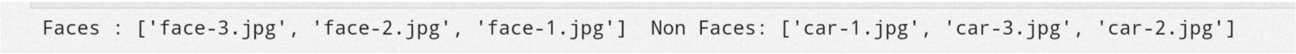
The first factor is the way of unifying the local descriptors into a global. For this, I tested the classification with two different methods. The first method used concatenation to unify each local histogram. It basically stacks every local descriptor. A good point here is the global descriptor will have every single detail of each window and the classification could be more accurate, however, after testing it, it was difficult to visually analyse patterns and data in general.

Second implemented method used intersection to finally get the global descriptor. Even if by using this technique there is some descriptive data that could be lost, by using an appropriate window size it had an accurate performance while classifying all the images, and visually it was more intuitive and easy to analyse. Finally I decided to use this method.

The second factor, the way of comparing the histograms to defying if they belong to the same class. If we ensure our histograms are descriptive enough, the next thing to worry about is the differentiation process. Not using an appropriate method could cause bad classification.

To avoid this, I calculated the difference with euclidean distance, which shows a good performance by giving 0.03 as the minimum difference accepted, this is, when the euclidean distance returns less than 0.03, they are going to be considered the same class.

By using these methods, intersection and Euclidean distance, I found a good performance by iterating over windows of 30 pixels. Every classification of faces and non faces images were performed accurately.



```
Faces : ['face-3.jpg', 'face-2.jpg', 'face-1.jpg']  Non Faces: ['car-1.jpg', 'car-3.jpg', 'car-2.jpg']
```

Question 4(b)

Block diagram of classification process

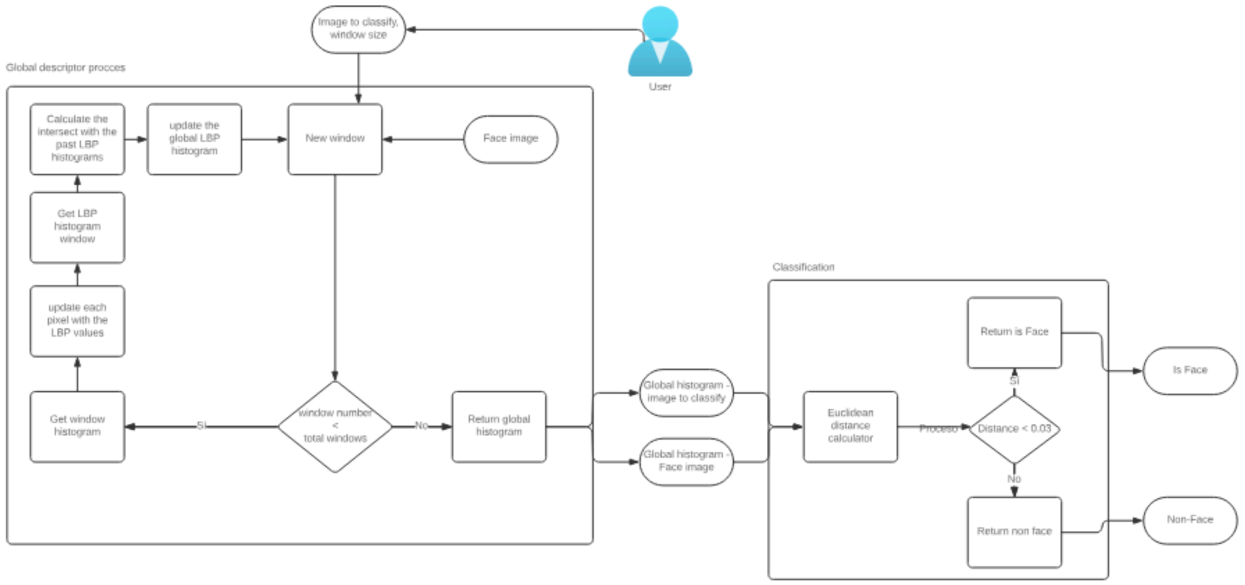


Image to classify, window size
User
Global descriptor procces
Calculate the intersect with the past LBP histograms
update the global LBP histogram
New window
Face image
Get LBP histogram window
update each pixel with the LBP values
Get window histogram
Si
window number < total windows
No
Return global histogram
Global histogram - image to classify
Global histogram - Face image
Classification
Euclidean distance calculator
Proceso
Distance < 0.03
Si
Return is Face
No
Return non face
Is Face
Non-Face

This is the structure of the classification process implemented to classify images as faces and non faces.

First the user starts the process with an image or a list of images, once the process is initialised, the system takes an image of a face to take as base to classify, and computes its lbp histogram.

After this, the system starts iterating over the input image, by dividing it into windows of a given size, creating its local histograms and finally calculating the intersection between them, after finishing the iterations over the windows, and having the global lbp of the input image, the system calculates the euclidean distance between the input image and the face image and returns “face” if the distance is less than 0.03, and non-face otherwise.

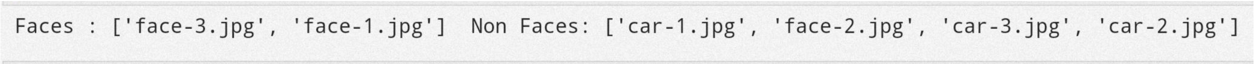
And this process will be repeated until all the images, in case the input is a list, are classified. Finally it returns two lists, one with the images classified as faces and the other one with the non faces images.

Question 4(c)

Here the window size was changed to 100. Even though some of the face images were classified correctly and no car image was classified as a face, the performance was lower for this window size because one of the face images was returned in the list of non-faces.

This could be because there is more data to compare at this window size so the comparison is stronger and focuses on some image and face specific details or expressions rather than general details of a face. For this reason, some face images could not be classified correctly, as it was noticed after executing it.

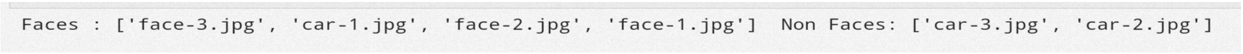
In some cases this window size is necessary, for example when we want to compare expressions or even a specific face. But in this case, where we are comparing something general, this approach was not useful.



```
Faces : ['face-3.jpg', 'face-1.jpg']  Non Faces: ['car-1.jpg', 'face-2.jpg', 'car-3.jpg', 'car-2.jpg']
```

Question 4(d)

For this test I used windows of 20 pixels. Here some of the car images were classified as faces, and this is because small windows cannot capture enough information to compare large objects, such as big faces or cars, and in this case, the global descriptor does not have enough data to accurately describe a face. This window size could be helpful when comparing small scale objects or textures as it is less sensitive to noise.



```
Faces : ['face-3.jpg', 'car-1.jpg', 'face-2.jpg', 'face-1.jpg']  Non Faces: ['car-3.jpg', 'car-2.jpg']
```

Question 4(e)

As has been shown, LBP is a very useful tool for finding patterns, extracting data and classifying images based on their descriptor. All of the examples shown here have used single images, but could we extend this process to work with dynamic textures in videos?

The answer is yes, maybe it is a bit more complex to implement, but considering that a video is a sequence of images, by using LBP to compare objects and implementing a method to analyse the motion in a video it could be done

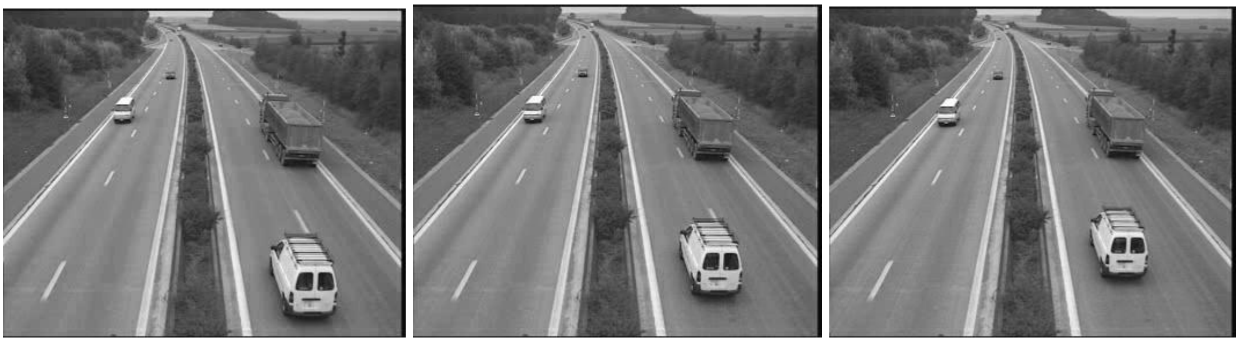
The motion information would give us information about the location of the single object in the video, and by using the lbp global descriptor, this moving object could be extracted and compared to something specific, like in the previous examples, with a face or a car.

This approach could be used to classify videos depending on its content, or to analyse the change between frames by complementing it with the colour histogram defined before. By doing this, the limitations of the colour histogram to define changes and enable a scene understanding would be solved.

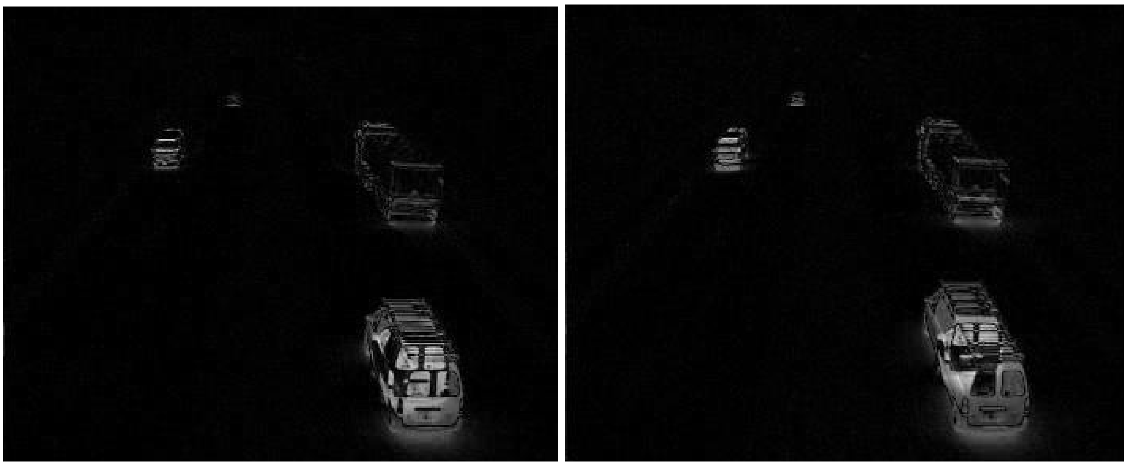
Object Segmentation and Counting

Question 5(a)

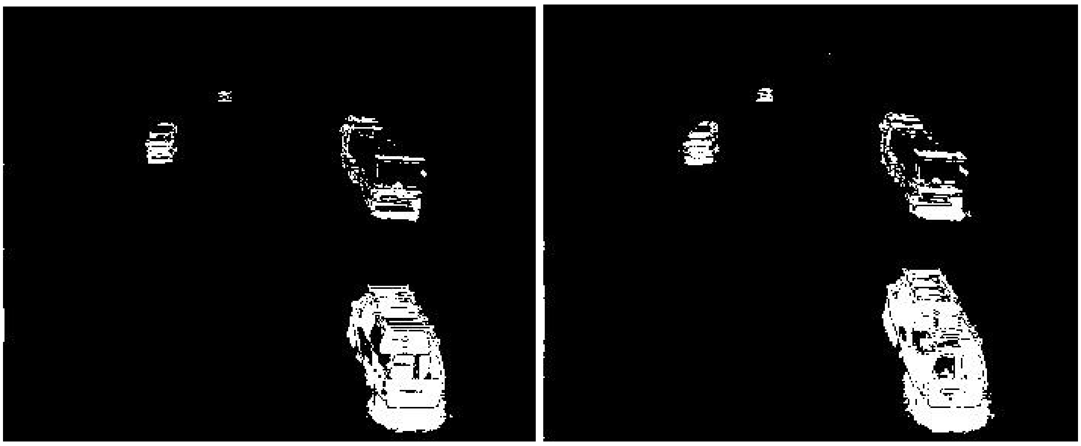
Original frames: 0, 1 and 2



Frame differencing: 0 and 1, 0 and 2

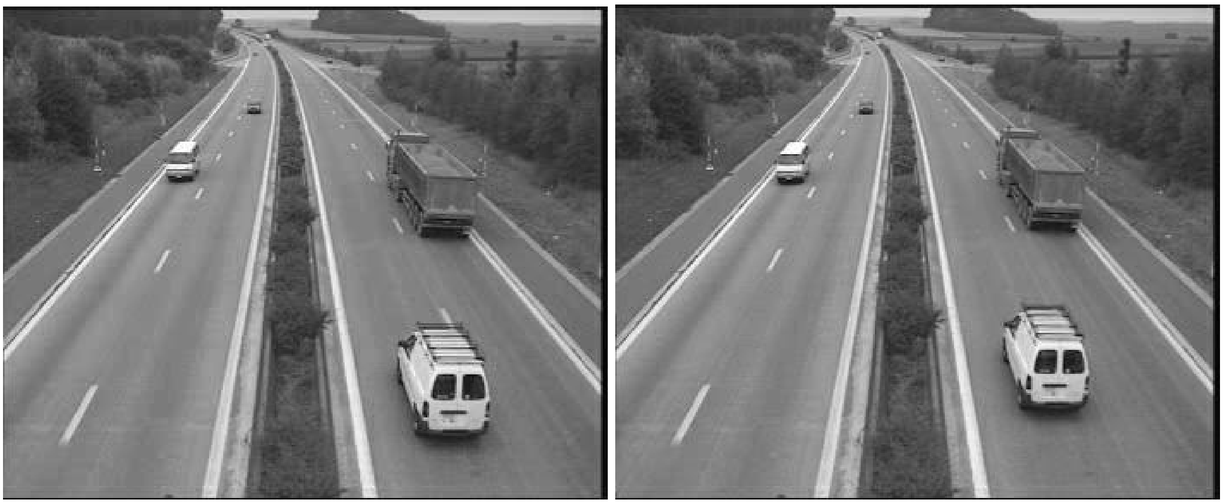


Threshold results:0 and 1, 0 and 2

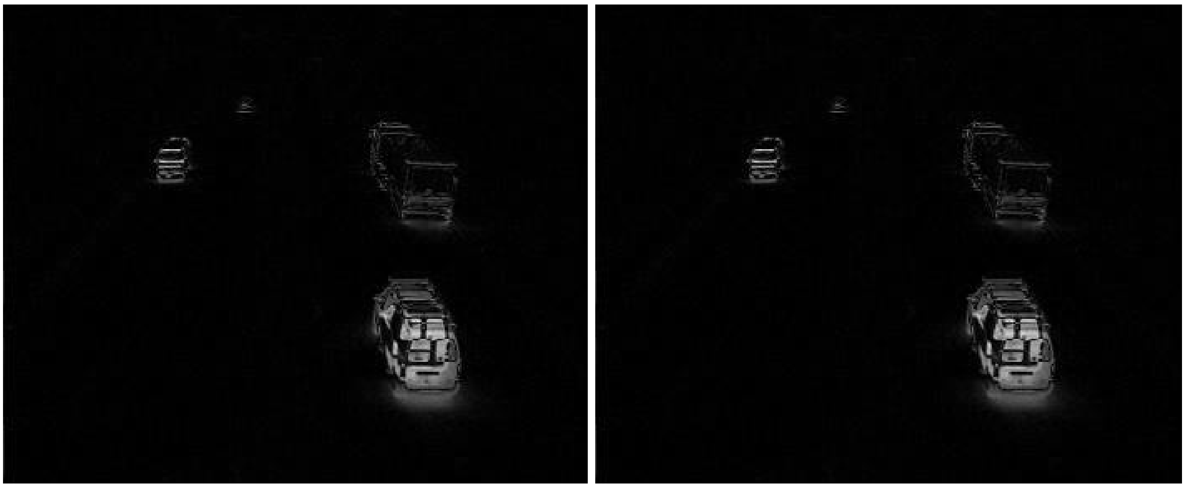


Question 5(b)

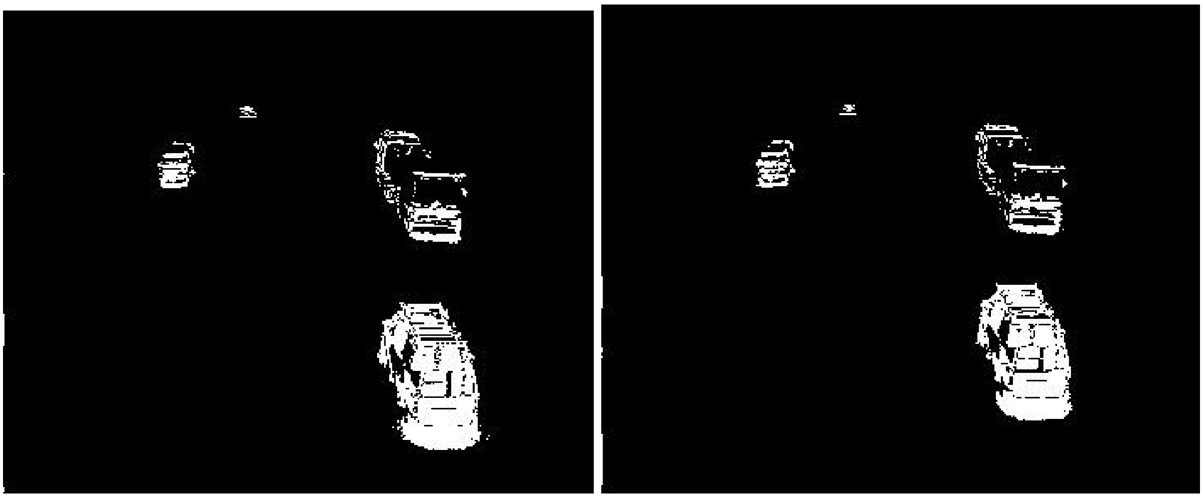
Original frame: frames 1 and 2



Frame differencing: 1 and 2, 2 and 3



Threshold results:



In my opinion, this is the most interesting and impressive part of this coursework, as it involves simple calculations between two images and provides a lot of visual and descriptive information about them. In this particular case, we have used a video of a road with cars passing by, and the idea is to detect every single moving object to eventually count them.

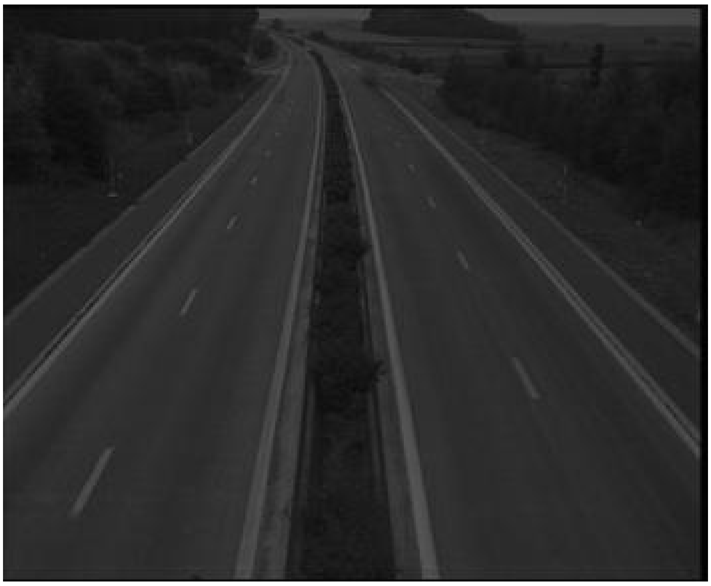
The method implemented is essentially to first calculate the difference between two given frames; the pixels with a high difference mean a change between the frames. Next, the difference matrix is transformed by comparing each value with a defined threshold. All numbers above the threshold are converted to 255, while the pixels with numbers below the threshold are given a value of 0. The resulting image has a black background and all moving objects are white, as shown above.

During implementation, I encountered some problems with the resulting images. The first time, images were returned with a lot of noise, but after changing some parts of my code, I realised that I had to convert each value to an integer before I could differentiate the images.

Another challenge I faced in the implementation was the specific number I should define as the threshold. I had to run the code several times with different thresholds until I finally found the optimal one.

Now as for the difference between exercises A and B attached above, it is easy to notice that in B, where two consecutive frames were used, the threshold matrix only shows the moving car in white, while in A, where I compare each frame to the first frame, there is a white zone in addition to the car, showing the trajectory from that first point in the video to the final position in the second frame. This shows that you have to use consecutive frames if you want to get an accurate picture of the moving object.

Question 5(c)

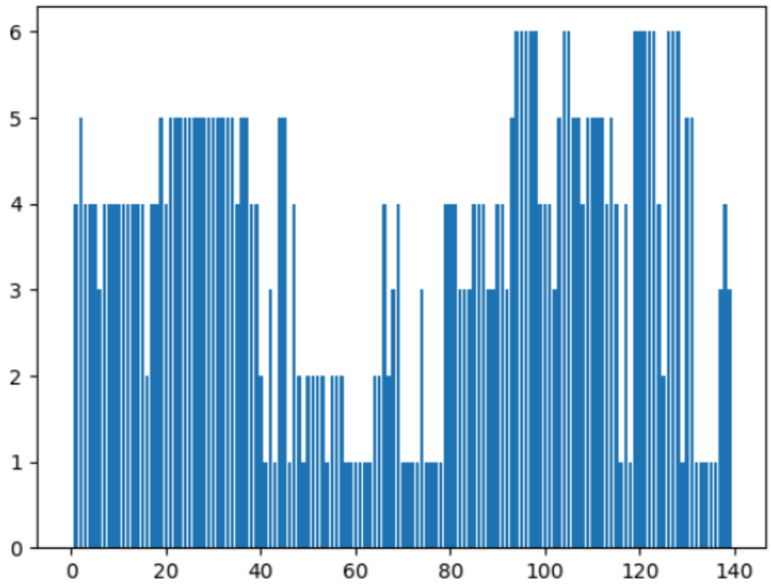


To achieve this, I have developed a method that receives a list of frames and calculates the background matrix by averaging all differences between consecutive frames.

This is done by first assigning image 0 to the background variable, iterating between each image, calculating the difference from the previous image, and averaging, assigning more weight (0.08) to the background image and less weight (0.02) to each new differentiated image. This ensures that the background, which is basically averaged difference matrices, is given more weight than the new image meaning that the pixels that are already filled with the background do not lose their value.

This logic will work only if the camera remains fixed throughout the video. Otherwise the final image would not make sense.

Question 5(d)



This plot shows the number of moving objects in each frame. Since we know that the video is of a road with cars passing by, we can notice some interesting aspects and conclude that the maximum number of cars detected by the camera at the same time is 6, while the average is around 3. From this I can conclude that although it is a busy road, it flows well as there are no more than 6 cars on the road at any one time.

The process behind this plot involves a series of iterations and comparisons to determine whether a particular pixel belongs to an already found object or whether it should be considered a new object.

As for the implemented code, it iterates over each pixel of a given threshold matrix, taking the pixels with the value 255 and grouping them according to the distance between them. If the distance between two given white pixels is less than a defined threshold, they are considered the same object, otherwise a new object is created. This process is repeated until all pixels have been visited. After iteration, all stored objects with less than 50 pixels are discarded and the sum of objects with more than 50 pixels is returned.

By applying this logic, the code would ensure that a single white pixel or a small group of white pixels is not defined as an object to avoid possible noise, while ensuring that all objects are visited and accounted for.

A disadvantage of this is that the returned count is not gonna be accurate with small moving objects as it only returns the count of big objects, and for big images it would take a lot of time to return the count because of the complexity of the implemented code.