PRACTICAL 3

VIDEO PROCESSING AND MOTION DETECTION

Video Definition

A video signal is a multi-dimensional analog or digital signal varying over time, whose spatiotemporal content represent a sequence of images (or frames) according to a predefined scanning convention. Mathematically, a continuous video signal will be denoted by $V(x, y, t)$, where t is the temporal variable.

Video Processing in Matlab

MATLAB provides the necessary functionality for basic video processing using short video clips and a limited number of video formats. The main video container supported by built-in MATLAB functions was the AVI container, through functions such as `Videoreader`, `movie2avi`, and `aviinfo`. This support is operating system dependent and limited only to a few codecs (you may need to re-compress the video before using it in Matlab), although this changes from one version to another.

Operative System	Windows	Mac	Unix/Linux
Supported formats	AVI (Compress/uncompress), MPEG, WMV (.wmv, .asf, .asx), other Microsoft DirectShow formats	AVI (Compress/uncompress), MPEG-1, MPEG-4 (.mp4, .m4v), MOV, other QuickTime formats	AVI (uncompressed)

MATLAB's ability to handle matrices makes it easy to create and manipulate 3D or 4D data structures to represent monochrome and colour video, respectively, provided that the video sequences are short (no more than a few minutes of video, depending on our RAM memory). Moreover, once a frame needs to be processed individually, it can be converted to an image using the `frame2im` function, which can then be processed using any of the functions available in the Image Processing Toolbox (IPT).

TASK 1: READING VIDEO FILES

The MATLAB functions associated with reading video files are as follows:

• `Videoreader`: constructs a multimedia reader object that can read video data from a variety of multimedia file formats.
• `aviinfo`: returns a structure whose fields contain information (e.g., frame width and height, total number of frames, frame rate, file size, etc.) about the AVI file passed as a parameter. (This function will become **obsolete** in future versions).

Please create a new script for this task when you will be adding all the commands from the different steps. This will make easy to repeat and debug your code.

Please remember that the two hour duration of a practical class will probably not be enough time to complete all sections. There is an expectation that you will work outside of class as an independent learner.

STEP 1: Download the video viptraffic.avi to your current folder in Matlab. Create a string variable containing the name of the video and read information about this file

```
file_name = 'viptraffic.avi';
file_info = aviinfo(file_name)
```

or

```
videoObj=VideoReader(file_name)
```

Please note that, while aviinfo returns and Structure with different variables containing only the information, VideoReader creates and object containing the full video.

STEP 2: Download now the video shopping_center.mpg. Try viewing information for this video file. Is it working? What parameters are different for the new file?

The function `VideoReader` also allows us to load a file into the MATLAB workspace. The data are stored as an object, which holds information about the video plus the video itself. You can have access to some of this info using command `get(videoObj)`. To retrieve the video itself (frames and pixels), we will need to run an extra command:

STEP 3: Read all the frames from viptraffic.avi using:

```
videoObj =VideoReader(file_name)

vidFrames = read(videoObj);
```

This step should generate a 4D array containing all the frames.

STEP 4: Run the command

```
size(vidFrames)
```

and try to understand what is each of the dimensions of the array. You can do that by comparing its dimensionality with the info from Step 1. Explain how each frame is stored in `vidFrames`

In order to visualise the video, we can call Matlab internal player.

STEP 5: Open the video player, explore the different options and run the video

```
implay(vidFrames)
```

STEP 6: Run the command `help implay` and also explorethe user interface of the movie player. How can we specify the frame rate of playback? Run the movie at half of its frame rate.

We can also view all the frames simultaneously using the montage function. This function will display images in an array all at once in a grid-like fashion

STEP 7: Use the montage function to display all images in a grid.

```
montage(vidFrames)
```

Viewing Individual Frames. To extract one of the frames from the full video stream, we can isolate it in a variable as an image

STEP 8: Extract the first frame of the video:

```
frame = vidFrames(:,:,:,1);
```

Please note that this new variable `frame` is indeed a colour image.

STEP 9: Visualise the image of the first frame in a new figure window using `imshow`

TASK 2: MOTION DETECTION USING BACKGROUND SUBTRACTION

As discussed in the lectures, in order to calculate background subtraction we need first to obtain a background image. As a first approach we could use an empty frame of the video. If you use the `montage` function, you will notice that for the viptraffic video, the last frame does not have any moving object on it. Therefore we can select this last frame as background image

STEP 1: Create a new script for this task. Load the viptraffic video and extract all the frames. Now create a variable `Bkg` and assign the last frame image into it. Finally, display it in a new figure.

[HINT: If you are having troubles to load the video file, please follow the instructions given in the previous task]

To perform background subtraction we do not need colour images. Operating with grayscale images in normally more efficient and comfortable

STEP 2: Convert the `Bkg` image into gray scale. [HINT: Matlab provides a function to do this: `rgb2gray]`. Store the result in a new image `BkgGray` and display it in a new figure. Check the size of the new image and compare it with the previous background size. Can you explain the difference?

Another way to convert into gray scale consists on playing directly with the raw pixel values in each of the channels. The average of the pixel value over the colour channels will give you a good approximation to the gray scale image.

$$Gray = \frac{R + G + B}{3}$$

STEP 3: Calculate a second background image `BkgGray2`, using the previous equation. To do that you can use the following piece of code:

```
BkgGray2 = Bkg(:,:,1)/3 + Bkg(:,:,2)/3 + Bkg(:,:,3)/3;
```

STEP 4: Compare the 3 background images visually and extract your own conclusions. Is it a good approximation? You can also measure their efficiency using `tic/toc`

```
figure
subplot(1,3,1), imshow(Bkg), title('Colour Bkg')
subplot(1,3,2), imshow(BkgGray), title('Gray Bkg'), colormap(gray)
subplot(1,3,3), imshow(BkgGray2), title('Gray Bkg Approx'), colormap(gray)
```

Once we have a background, we can perform the background subtraction at every frame. A straightforward way to perform the processing on all frames is to use a for loop.

STEP 5: Create a loop that extract a single frame at each iteration into a variable called `currentFrame`. Fill the gaps in the following code

```
figure
for t = 1:____________
      currentFrame= ____________;

end
```

STEP 6: Since the background is in gray scale, we will need to convert the current frame too. Convert it into a variable inside the loop using the same procedure that you decided to the background in Step 4.

```
currentFrameGray= ____________;
subplot(2,3,1), imshow(currentFrameGray), title(['Frame: ',num2str(t)])
subplot(2,3,2), imshow(BkgGray), title('Background')
pause(0.2)
```

STEP 7: Now we are ready to implement the background subtraction itself. The whole operation can be written in a single line thanks to Matlab Matrix operations:

```
Blobs=abs(currentFrameGray - BkgGray) > Th;

subplot(2,3,3), imshow(Blobs), title('Blobs´), colormap(gray)
```

Test different values of `Th` until you are happy the car is well detected and the amount of noise is not unbearable. Please, note that, because of the comparison, the resulting image is binary (boolean).

STEP 8: Pay attention to the black car towards the end of the video. Is it well detected? Why do you think this is?

[HINT: Think of the datatype of the variables you are dealing with and the value of the pixels for the road, the white car and the black car.]

STEP 9: We can solve the previous issue by converting the variables to doubles. Add the previous lines before you perform the background subtraction:

```
BkgGray = double(BkgGray);
currentFrameGray = double(currentFrameGray);
```

Observe now if the car is detected.

[HINT: Remember than the plotting function imshow() only works for uint8. So if we are working with doubles, you can either use imagesc() for plotting, or, alternatively, just cast to uint8 before plotting: imshow(uint8())]

STEP 10: As a final step, we can save the result in a video. To do that, we will use the function `VideoWriter`. To make it work, you will need to create and empty video object before the loop:

```
vidObj2 = VideoWriter('resultTraffic.avi');
open(vidObj2);
MAP=colormap(gray(256));
```

add each individual image to the video within the loop:

```
frame = im2frame(uint8(Blobs)*255, MAP);
writeVideo(vidObj2,frame);
```

and finally, after the loop has finished, to indicate the object is ready:

```
close(vidObj2);
```

TASK 3: BACKGROUND UPDATE

Although the previous background made the job, it is not always possible to find an empty frame. Moreover, the background may need to be updated to deal with different light conditions over time (sunny/cloudy/night). A better and more common approach is to calculate the background using a median filter.

STEP 1: Create a new function call bckGenerator. The input parameters will be the 4D image array and a sampling rate integer.

```
function [Bkg] = bckGenerator(videoStream, sampling)
```

STEP 2: Within the function, we will need a loop to extract the individual frames at the input sampling rate, which will be accumulated into a buffer. Remember that the extracted frames need to be converted into **grayscale** and to **double** to ensure calculations are correct:

```
buffer=[];
counter=0;
for t = 1:sampling:size(videoStream,4)
      counter=counter+1;
      buffer(:,:,counter)=_________________;

end
```

Please, be sure you understand all the elements and their reasoning in the previous code.

STEP 3: To finish our function, we just need to apply median filter on the buffer images, after the loop has finished. Matlab provides a very helpful function to do this, called `median`.

```
Bkg = median(buffer,3);
```

The parameter with value 3 indicates the dimension over which we want to apply the median filter, since we do not want to do the median over the rows (dimension 1) or the columns (dimension 2), but over the time (dimension 3), we need to specify a 3 in there.

STEP 4: Replace the background image in the Task 2 script by the one generated by this function. When calling the function, try different sampling values between 1 and 20. Observe the quality of the generated background and its computation time (tic/toc). Which one do you think is the optimal value?

TASK 4: POSTPROCESSING

In the previous tasks, we have implemented a full motion detection algorithm that we can use for more high level understanding of the scene, such as counting the number of vehicles or detecting their speed.

However, as you can see in your results, some of the detected motion blobs are a bit noisy, with both holes and noise isolated pixels around the image. The severity of each of these problems will depend on the threshold you chose in Task 2 step 7.

Do not worry! We can solve these issues, at least partially, by applying morphology. Matlab provides implementations of these functions:

```
imdilate()
imerode()
imclose()
imopen()
```

We will use the last 2 functions, since the work similarly to the first ones but without changing the size of the blob. All the functions require 2 parameters: a **binary** image where the morphological operator will be applied, and a mask defining the size and connectivity and extension of the operator when applying it. Usually, the mask is defined as:

```
Mask= ones(rows,columns);
```

where the most usual values for rows and columns are 3x3, 5x5, 7x7 and so on.

STEP 1: Apply the open and close morphological operators until you are happy with the resulting blobs and plot the result. There are 3 elements that you can vary:

- How many times you apply each operator (calling the function several times), if any
- The order in which you apply the operators. This will depend on what is your predominant problem (see theory)
- The size of the mask of each operator.

```
subplot(2,3,4), imshow(BlobsCorrect), title('Post-processed Blobs´), colormap(gray)
```

Now that we have better and compact blobs, we can proceed to extract each individual vehicle. To do that, we need to apply the connected component labelling algorithm. Luckily, Matlab also provides an implementation of this algorithm in the function `bwlabel()`.

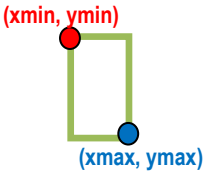
STEP 2: Read the documentation of the `bwlabel` function and applied to your post-processed blobs. Display the result:

```
subplot(2,3,5), imshow(BlobsLabel), title('Labelling´)
```

STEP 3: To count the number of vehicles at each frame, we just need to detect the maximum label returned by the connected component algorithm:

```
NumVehicles = max(max(BlobsLabel));
```

We are also going to extract and draw a bounding box around each vehicle in order to show we have detected it. For each blob labelled, we will create a bounding box in the form:




```
BB = [xmin ymin xmax ymax];
```

which stores the upper left corner and lower right corner, which defines the bounding box.

STEP 4: For each blob labelled, we will create a bounding box and accumulate them into a matrix. All the pixels belonging to a blob can be easily extracting making use of the fact that they are labelled and using the function `find()`.

```
BBs = [];
for b = 1: NumVehicles

    [ys xs]=find(BlobsLabel == b);

    xmax=_______;
    ymax=_______;
    xmin=_______;
    ymin=_______;

   BB = [xmin ymin xmax ymax];

    BBs = [BBs; BB];

end
```

STEP 5: Finally plot the BBs on the top of the original image using the function `rectangle()`.

```
subplot(2,3,6), imshow(currentFrame), title('Detections´), hold on
for b = 1: NumVehicles

  rectangle('Position', [BBs(b,1) BBs(b,2) BBs(b,3)-BBs(b,1)+1 BBs(b,4)-BBs(b,2)+1])

end
hold off
```

TASK 5: NEW VIDEOS

STEP 1: Now run all the previous scripts on the video shopping_center.mpg. Does it work? How many parameters did you have to change/ tune?