

**CSC2014 DIGITAL IMAGE PROCESSING**

**Group Assignment**

|  |  |
| --- | --- |
| Name | Student ID |
| Chan Kah Gin | 23038367 |
| Ho Zi Shan | 22116347 |
| Lee Wen Xuan | 22115737 |
| Thit Sar Zin | 21082755 |

# Introduction

This project consists of two tasks, namely task A (YouTube video processing) and task B (paragraph extraction). Task A involves developing a program that processes four videos to enhance and modify their contents by:

1. Increasing the brightness of the video if it was shot at night.
2. Blurring every face in the video.
3. Resizing and overlaying a talking video (talking.mp4) on the top left of each video.
4. Adding watermarks to the videos to protect video ownership.
5. Adding an end screen video (endscreen.mp4) at the end of each video.

On the other hand, task B involves developing a program that processes a set of scientific papers in image format, extracts all paragraphs, and stores them in the correct order. The detailed process includes:

1. Extracting columns from the images.
2. Extracting paragraphs from the columns.
3. Storing paragraphs in the correct order.
4. Identifying and removing invalid content from the paragraphs.

# Task A

## Proposed Approach

**Increase brightness for nighttime videos:**

To calculate the brightness of each video, every frame in the video was first converted into grayscale using **cv2.cvtColor()**. Then, their brightness values were computed by calculating the mean pixel intensity of the grayscale frames using **np.mean()**. These values were stored in a list and were computed later to determine the brightness values of the entire video by using **np.mean()** again.

def calculate\_brightness(frame):

    # Convert current frame to grayscale (colour information is not necessary to calculate brightness)

    gray\_frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

    # Calculate average brightness of the frame by calculating the mean of all pixel values

    brightness = np.mean(gray\_frame)

    return brightness

Since nighttime videos have lower average brightness values, a threshold of 100 is used to differentiate daytime and nighttime videos. Any videos with average brightness values lower than the threshold are defined as videos shot during nighttime.

# If the average brightness is less than 100, then it is considered as nighttime

    is\_nighttime = avg\_brightness < 100

    day\_night\_status = "NIGHT" if is\_nighttime else "DAY"

    print(f"Detected as: {day\_night\_status}")

If nighttime video were detected, its brightness was increased using the **cv2.convertScaleAbs()** function, where the pixel intensities of each frame were scaled by a factor of two to double its brightness.

def increase\_brightness(frame, factor=2.0):

    # Increase the brightness of the frame by multiplying all pixel values by the factor value 2.0

    bright\_frame = cv2.convertScaleAbs(frame, alpha=factor, beta=0)

    return bright\_frame

After all the steps mentioned above, every frame of the videos was appended to a list for storage, and the processed video was then released to free up memory.

# Loop through the frames again and adjust brightness if it's nighttime

vid.set(cv2.CAP\_PROP\_POS\_FRAMES, 0) # Reset to the first frame

frames = [] # List to store processed frames

for frame\_count in range(total\_no\_frames):

    success, frame = vid.read()

    if not success:

        break

    if is\_nighttime:

        frame = increase\_brightness(frame, 2.0) # Increase the brightness by a factor of 2

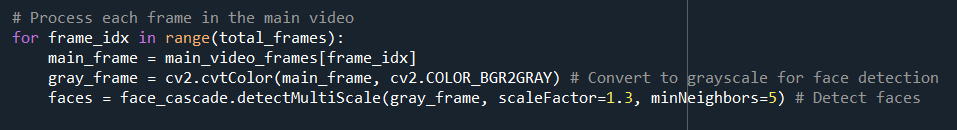
    frames.append(frame)

vid.release()

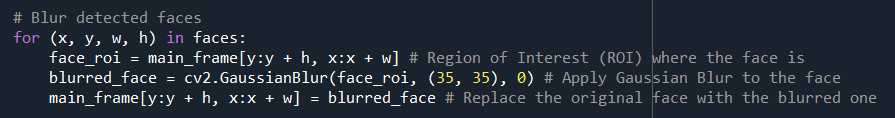
return frames

**Face blurring:**

A Haar Cascade Classifier was used to detect faces in the main video. The **cv2.CascadeClassifier()** function was used to initialise the classifier, and the relevant Haar Cascade file was loaded. The initial step in processing each frame of the main video was to convert it to greyscale using **cv2.cvtColor (main\_frame, cv2, COLOR\_BGR2GRAY)**. By lowering the computational complexity, this step made the face detection procedure simpler. With **scaleFactor = 1.3** and **minNeighbours = 5**, the **detectMultiScale()** function was applied to the grayscale frame to ensure precise facial detection while removing false positives.

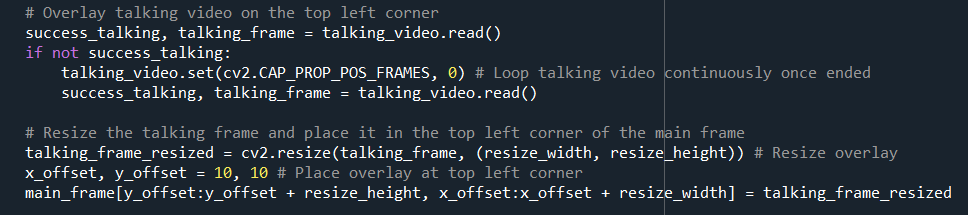


Each region of interest (ROI) that corresponded to a face was then extracted from the frame after the faces were detected. The **cv2.GaussianBlur()** method was then used to apply a Gaussian Blur on these areas, with a kernel size of **(35, 35)** to adequately anonymise the facial details. After that, the blurred ROI was reincorporated into the original frame, essentially replacing the blurred version of the facial characteristics for the real ones. This procedure was carried out again for each identified face in a frame.

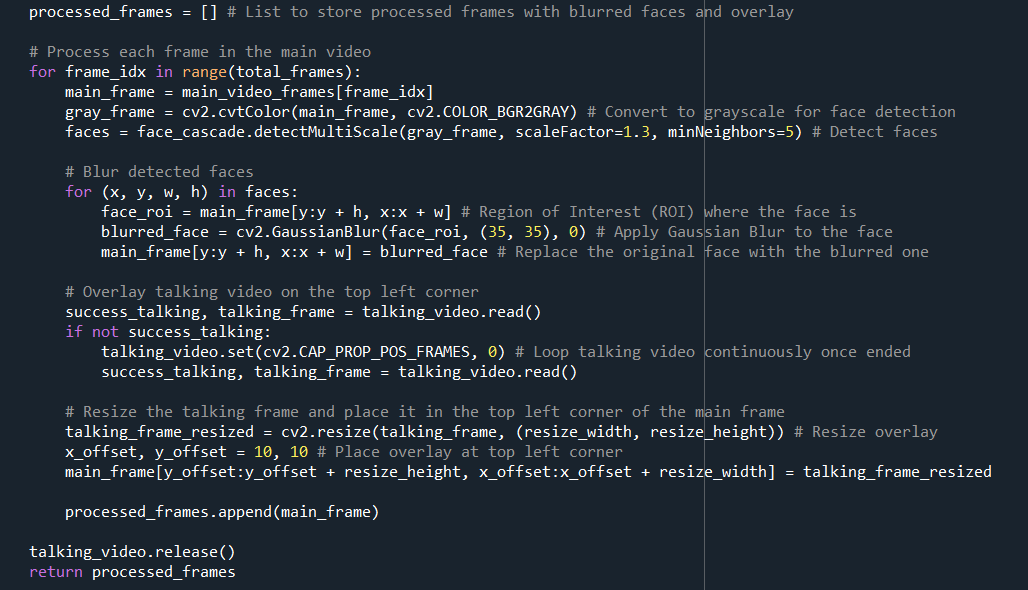


**Overlaying talking video:**

The **cv2.VideoCapture()** function was used to load talking.mp4, a secondary video, for use as the overlay. The **cv2.resize()** function was used to resize each frame of the talking video to 30% of the original video’s dimensions. To prevent the overlay from obscuring crucial elements of the main video, this scaling was required. Each main video frame had the resized overlay placed in the upper-left corner, with 10 pixels away from the top and left margins. Using **cv2.CAP\_PROP\_POS\_FRAMES**, the video playback was looped back to the start if the talking video reached the end of its frames. This made sure the overlay played continuously for the whole main video.

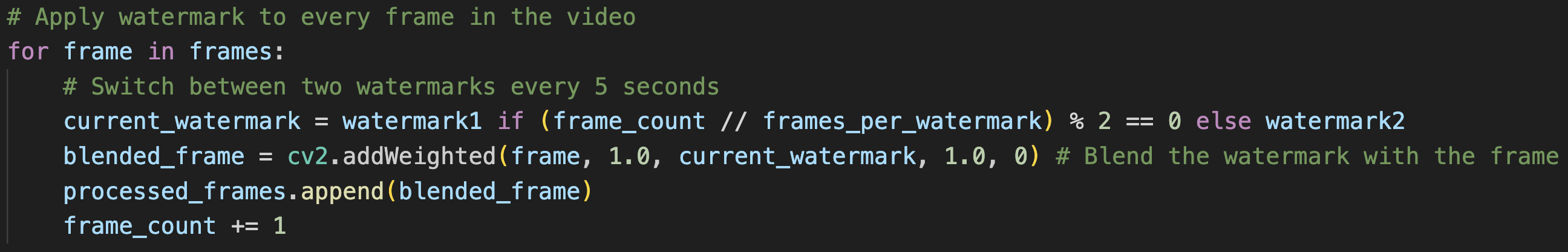


The recognised faces in each frame of the main video were blurred before the resized talking video was overlaid. All the altered frames were appended in a list called **processed\_frames**, to which these processed frames were inserted. Following the completion of the changes, the list of processed frames was either returned combined into a final output video. The video processing loop iterated over every frame in the main video.



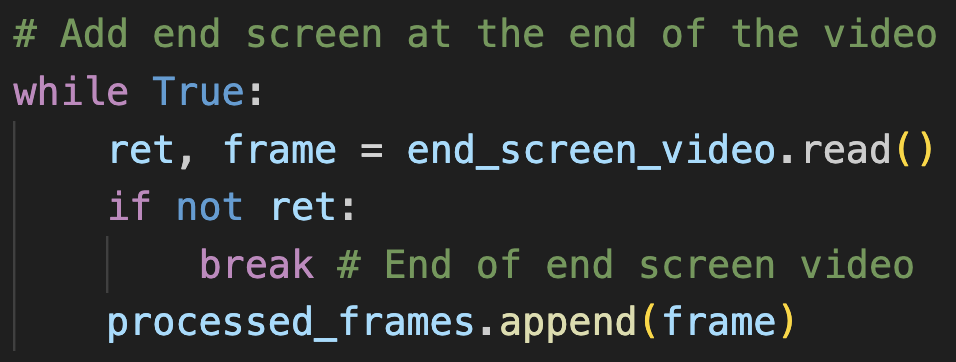
**Watermarks:**

Moving on to the fourth question, we intend to implement two different watermarks every 5 seconds throughout the whole video. This is accomplished by calculating the current time in the video using the frame count divided by the number of frames per second. Based on such, an if-else statement is used to perform toggling between the two watermarks, making sure they change every five seconds. The selected watermark is blended with each frame using OpenCV’s **cv2.addWeighted** function, and the processed frame is stored in a list for further use.

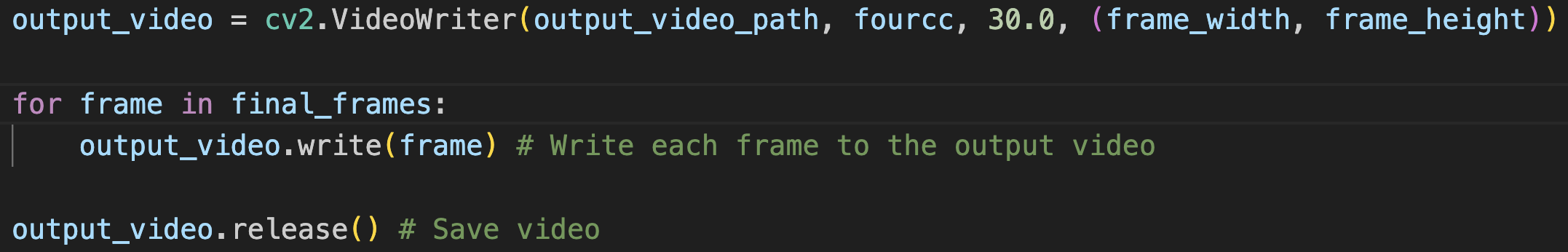


**End Screen:**

After the watermarking is done, we move to the last question and that is adding end screen video. A while loop that reads frames from the end screen video file handles this. The process continues until no more frames can be read (indicated by a False return value), in which case the loop breaks. Each successfully read frame is otherwise appended to the list of processed frames.



Lastly, the processed frames (including brightness adjustment, blurred faces, overlayed talking video, water marked, and end screen) are written to a single output video file. Moreover, to ensure effective management of resources, each video file has been properly closed.



## Result and Discussion

**Increase brightness for nighttime videos:**

After processing, only two nighttime videos were detected: singapore.mp4 and traffic.mp4. The comparisons before and after processing were shown in the table below:

|  |  |
| --- | --- |
| **singapore.mp4** | |
| **Before** | **After** |
|  |  |

Table 1. Comparison of singapore.mp4 after increasing brightness

|  |  |
| --- | --- |
| **traffic.mp4** | |
| **Before** | **After** |
|  |  |

Table 2. Comparison of traffic.mp4 after increasing brightness

**Faces Detection and Blurring:**

|  |  |  |  |
| --- | --- | --- | --- |
| **singapore.mp4** | **office.mp4** | **alley.mp4** | **traffic.mp4** |
|  |  |  |  |

Table 3. Output of face detection and blurring

**Overlaying Talking Video:**

|  |  |  |  |
| --- | --- | --- | --- |
| **singapore.mp4** | **office.mp4** | **alley.mp4** | **traffic.mp4** |
|  |  |  |  |

Table 4. Output of overlaying talking video

**Watermark:**

|  |  |
| --- | --- |
| **First Watermark** | **Second watermark** |
|  |  |

Table 5. Output of both watermarks added in alley.mp4

As shown in the output above, first watermark and second watermark (labelled in red box) alternate every 5 seconds throughout the duration of the video, same goes to office.mp4, singapore.mp4 and traffic.mp4.

**End Screen:**



Figure 1. Output of end screen video concatenated to the end of video

The end screen video is successfully added to the end of all four videos (singapore.mp4, office.mp4, alley.mp4, traffic.mp4).

# 

# Task B

## Proposed Approach

The proposed approach for paragraph extraction involves first identifying and extracting columns, which are stored separately. Then, each extracted column is analysed to determine if it meets the criteria for a paragraph. Valid paragraphs and non-paragraph content are stored separately.

The process starts by identifying columns in the image. This is done by converting the image into a binary format, where pixel values are either 0 or 255.

*def* binarize\_image(*self*):

        '''Convert image to binary'''

        \_, self.binary\_image = cv2.threshold(self.image, self.white\_threshold, 255, cv2.THRESH\_BINARY)

        return self.binary\_image

Next, white vertical columns in the binary image are identified and stored in an array by analysing the vertical pixel intensity sums, where high values indicate regions with mostly white pixels.

*def* find\_white\_col(*self*, *hist\_proj*=False, *threshold*=0.95):

        '''Identify white columns based on vertical projections'''

        vertical\_projection = np.sum(self.binary\_image, *axis*=0)

        peak\_threshold = threshold \* np.max(vertical\_projection)

        self.white\_col = np.where(vertical\_projection > peak\_threshold)[0]

        # Plot the histogram projection if required

        if hist\_proj:

            self.plot\_hist\_proj(vertical\_projection, peak\_threshold)

The start and end points of each column are identified by checking the gaps between consecutive white column indices. Columns that are wider than the predefined minimum column width are considered valid and stored.

*def* find\_col(*self*):

        '''Identify start and end of each column & filter valid regions'''

        start = None

        for i in range(1, len(self.white\_col)):

            if self.white\_col[i] - self.white\_col[i - 1] != 1: # Detect column boundaries.

                if start is None:

                    start = self.white\_col[i - 1]

                if (self.white\_col[i] - start) > self.min\_col\_width: # Filters valid regions.

                    self.column\_start\_end.append((start, self.white\_col[i]))

                start = None  # Reset start

After extracting the columns and saving them in “Columns” folder, the program identifies rows containing text by checking for non-white pixels.

*def* find\_rows(*self*):

        '''Identify rows containing text by checking pixel intensity'''

        self.rows = [row for row in range(self.nrow) if any(pixel\_value < self.white\_threshold for pixel\_value in self.binary\_image[row, :])]

Rows are then grouped into paragraphs based on the gaps between consecutive rows. If a gap between rows exceeds a defined threshold, it indicates the end of a paragraph.

*def* find\_paragraphs(*self*):

        '''Identify start and end points of paragraphs based on row gaps'''

        start = None

        for i in range(1, len(self.rows)):

            # Check if the rows are consecutive (no gap)

            if self.rows[i] - self.rows[i-1] != 1:

                # If the gap is larger than the threshold, consider it as the end of the paragraph

                if start is not None and self.rows[i] - self.rows[i - 1] > self.gap\_threshold:

                    end = self.rows[i - 1]                          # Last non-white row before the gap

                    self.paragraph\_start\_end.append((start, end))   # Store paragraph range

                    start = self.rows[i]                            # Start a new paragraph after the gap

            # If there is no gap, continue the current paragraph

            else:

                if start is None:

                    start = self.rows[i-1]

        # Handle the last paragraph, from the last non-white row to the end of the image

        if start is not None:

            last\_non\_white\_row = self.rows[-1]

            end = last\_non\_white\_row

            while end < self.nrow and all(pixel\_value >= self.white\_threshold for pixel\_value in self.binary\_image[end, :]):

                end += 1

            self.paragraph\_start\_end.append((start, end))

After extracting the paragraphs into the “*Paragraphs*” folder, each one is validated to ensure it contains enough rows and columns and has intermediate white lines as gaps. Paragraphs that do not meet these criteria are moved to the “*Not Paragraphs*” folder.

*def* is\_paragraph(*self*, *white\_threshold*=250, *line\_gap\_threshold*=1, *row\_threshold*=3, *col\_threshold*=5):

        '''Verify if the input image meets paragraph requirements'''

        # Binarize the image

        \_, binary\_image = cv2.threshold(self.gray\_image, white\_threshold, 255, cv2.THRESH\_BINARY\_INV)

        # Compute row and column densities

        row\_density = np.sum(binary\_image, *axis*=1) > 0

        col\_density = np.sum(binary\_image, *axis*=0) > 0

        # Count non-empty rows and columns

        num\_rows = np.sum(row\_density)

        num\_cols = np.sum(col\_density)

        # Check for intermediate white lines

        white\_line\_indices = np.where(~row\_density)[0]  # Rows with no text

        line\_gaps = np.diff(white\_line\_indices)  # Gaps between white lines

        has\_intermediate\_white\_lines = np.any(line\_gaps >= line\_gap\_threshold)

        # Final conditions: must meet row and column thresholds and have intermediate white lines

        return num\_rows >= row\_threshold and num\_cols >= col\_threshold and has\_intermediate\_white\_lines

## Result and Discussion

The final output is saved in the “*Paragraphs*” folder, while column images extracted during processing are stored in the “*Columns*” folder. The “*Not* *Paragraphs*” folder contains images that do not meet the criteria for paragraphs (e.g. tables and photos).

The naming convention for images follows a structured format. It starts with the base name of the original image, followed by an underscore and C*x*, where *x* represents the column number. For paragraphs extracted from a column, an additional underscore and P*x* are appended, where *x* indicates the paragraph number.

For example, processing 007.png is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | | **Extracted Columns**: C1 and C2 | |
| A page of a newspaper  Description automatically generated | | A close up of a text  Description automatically generatedA close-up of a text  Description automatically generated | |
| **Final Output**: | | | |
| Paragraphs from C1 | Paragraphs from C2 | | Non-Paragraph Content |
|  | A close up of a text  Description automatically generatedA text on a white background  Description automatically generatedA close up of a text  Description automatically generated | | A grass field with white flowers  Description automatically generated |

Table 6. Result of 007.png after processed