Problem 1 – Template matching (50 points):

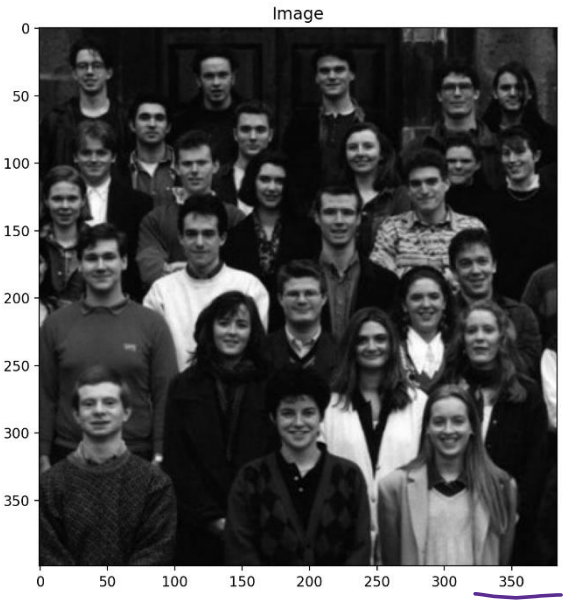
a) Implement the function 'scale_down(image, resize_ratio)'.

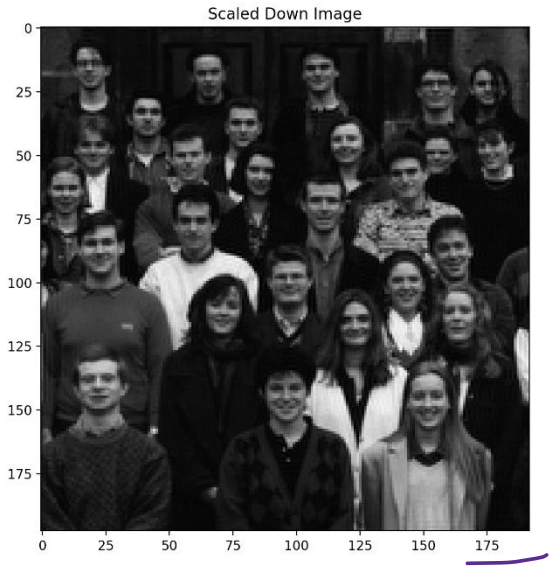
we use the Fourier Transform to scale down an image while preserving essential details. The idea is to transform the image into the frequency domain, crop the high frequencies to retain only the most important components, and then apply the inverse transform to reconstruct the resized image.

The process follows these steps:

1. Convert the image to frequency space using FFT.
2. Crop the frequency domain based on the given resize_ratio, keeping only the low frequencies.
3. Apply the Inverse Fourier Transform (IFFT) to get the scaled-down image.
4. Normalize the output to ensure proper visualization.

We tested the function with an original image of size 350x350 pixels. After applying downsampling with a ratio of 0.5, the resulting image has a size of 175x175 pixels.





b) Implement the function 'scale_up(image, resize_ratio)'.

The scale_up function increases the size of an image using the **Fourier Transform** by expanding its frequency representation. First, we apply FFT to convert the image into the frequency domain, where important structural details are preserved. We then **pad the frequency domain** with zeros to match the new size, ensuring that the low-frequency components remain centered, which helps maintain image clarity. After padding, we apply the **inverse FFT** to reconstruct the upscaled image in the spatial domain. Finally, the image is **normalized** for proper visualization. This method effectively enlarges an image while preserving essential details, making it useful for tasks such as **template matching and feature extraction**.

```python
def pad_frequency_domain(f_transform, new_h, new_w):
    h, w = f_transform.shape
    pad_h, pad_w = (new_h - h) // 2, (new_w - w) // 2
    padded_f_transform = np.zeros((new_h, new_w), dtype=complex)
    padded_f_transform[pad_h:pad_h + h, pad_w:pad_w + w] = f_transform
    return padded_f_transform

def scale_up(image, resize_ratio):
    image = image.astype(np.float32)

    f_transform = np.fft.fftshift(np.fft.fft2(image))
    h, w = image.shape
    new_h, new_w = int(h * resize_ratio), int(w * resize_ratio)

    padded_f_transform = pad_frequency_domain(f_transform, new_h, new_w)

    shifted_f = np.fft.ifftshift(padded_f_transform)
    resized_image = np.abs(np.fft.ifft2(shifted_f))

    return np.uint8(cv2.normalize(resized_image, None, 0, 255, cv2.NORM_MINMAX))
```

c) Implement the function 'ncc_2d(image, pattern)'.

The ncc_2d function calculates the **NCC** between an image and a pattern to measure their similarity.

Precompute Pattern Statistics:

- Computes the **mean** and **normalized version** of the pattern.
- Calculates the **norm** of the centered pattern for normalization.

Extract Image Windows Efficiently:

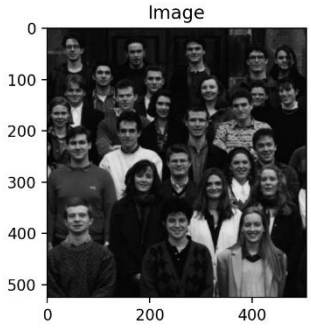
- Uses sliding_window_view() to extract **all possible windows** of the image that match the pattern's size.
- Computes the **mean and centered version** of each image window.

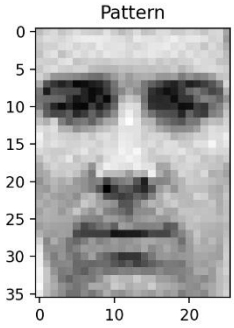
Compute NCC Score:

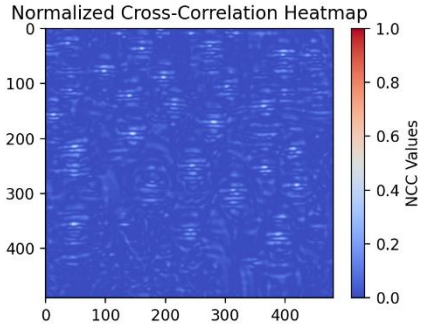
- Computes the **dot product** between the centered image windows and pattern.
- Normalizes using the **product of norms** to ensure values range from **-1 to 1**.
- Handles cases where **variance is zero** to prevent division errors.

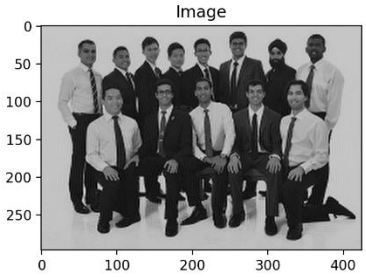
```python
def ncc_2d(image, pattern):
    pattern_h, pattern_w = pattern.shape
    # Compute pattern mean and normalize it
    pattern_mean = np.mean(pattern)
    pattern_centered = pattern - pattern_mean
    pattern_norm = np.linalg.norm(pattern_centered)
    image_windows = sliding_window_view(image, (pattern_h, pattern_w))

    # Compute mean and normalize image windows
    windows_mean = np.mean(image_windows, axis=(2, 3), keepdims=True)
    windows_centered = image_windows - windows_mean
    windows_norm = np.linalg.norm(windows_centered, axis=(2, 3))

    # Compute NCC
    denominator = pattern_norm * windows_norm
    denominator[denominator == 0] = 1  # Prevent division by zero
    ncc_map = np.sum(windows_centered * pattern_centered, axis=(2, 3)) / denominator

    return ncc_map
```

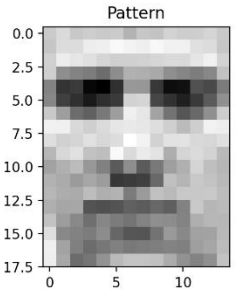
d) The function 'display' gets an image and a pattern, and displays both, along with their NCC heatmap.

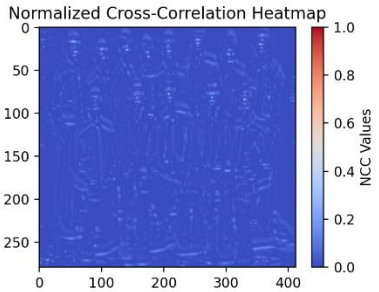








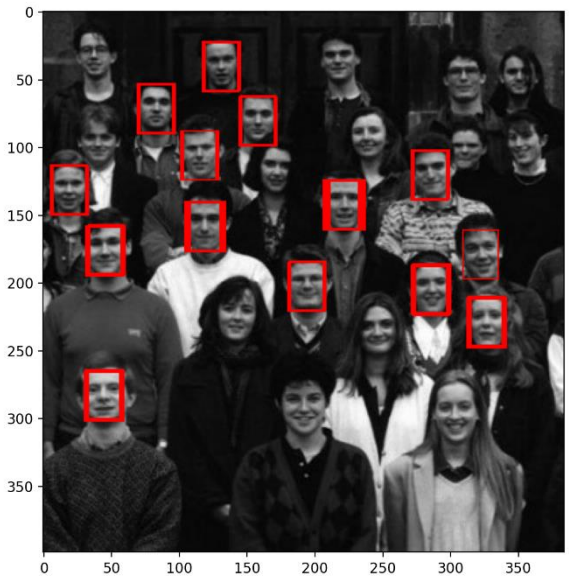




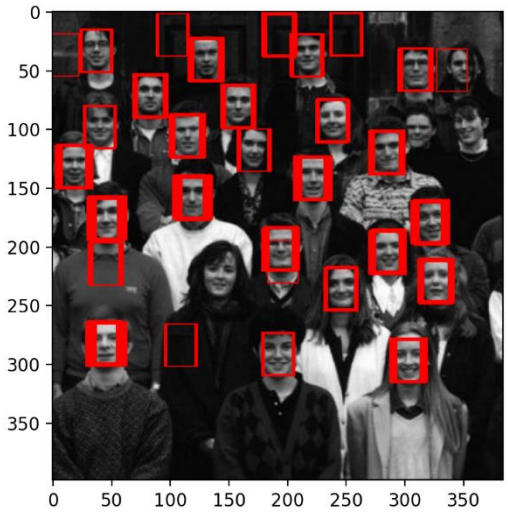
e) Using the filtered matches you found, call the function 'draw_matches' with the original image.

To finalize the detection process, we applied the **draw_matches** function to overlay red rectangles around the detected faces in the original image. Since we scaled the image during processing, we first **adjusted the matched coordinates** back to the original scale to ensure accurate placement of the bounding boxes.

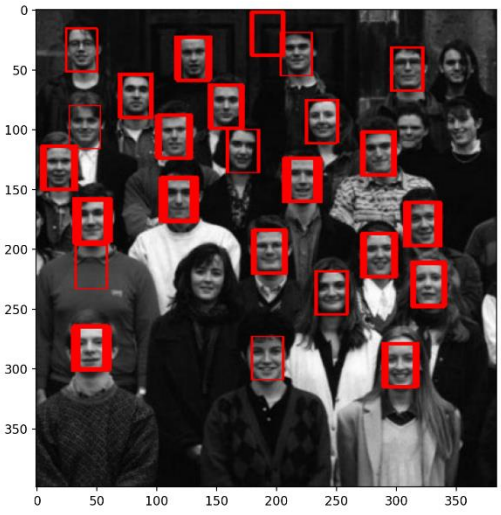
We started with threshold = 0.6 but noticed that not all the faces were detected, as some correct matches were missing. This threshold was too strict and filtered out valid face detections.



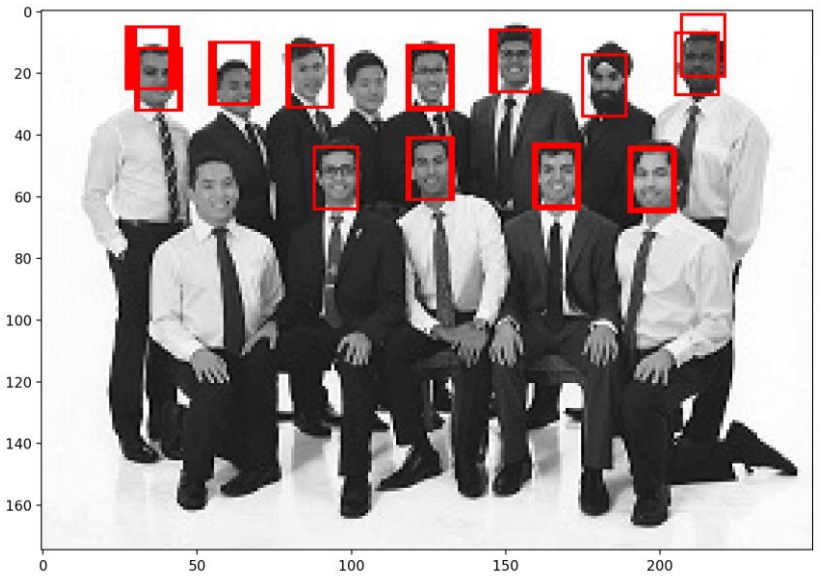
To improve the results, we applied a **threshold = 0.455**, which successfully detected more faces compared to **threshold = 0.6** while maintaining a reasonable level of accuracy. However, we also noticed that some false positives appeared, such as incorrectly placed bounding boxes in areas that do not contain faces.



To refine the detection, we applied a **threshold = 0.488**, which provided a better balance between detecting all faces and minimizing false positives. This adjustment improved the accuracy by reducing unwanted detections while ensuring that most actual faces were correctly identified.



For the crew image, we applied a **threshold = 0.45**, which provided the best balance between detecting all faces and minimizing false positives. After testing multiple threshold values, **0.45** was found to be the most effective, ensuring that most of the faces were correctly identified while reducing incorrect detections in non-face areas.



Problem 2 – Multiband blending (50 points)

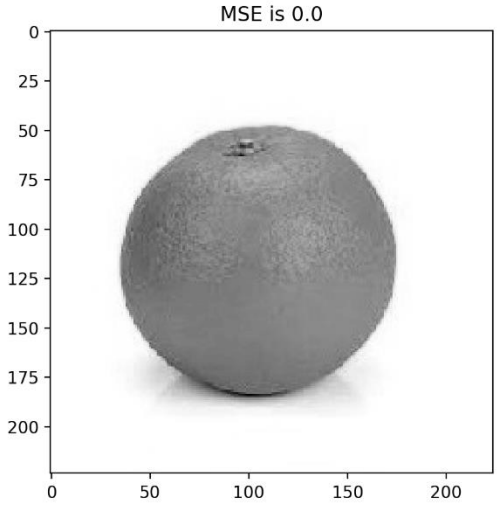
a) Implement the function 'get_laplacian_pyramid'. It gets an image and returns a Laplacian pyramid with the specified levels number. Each additional level is half-size per-axis compared to the previous one.

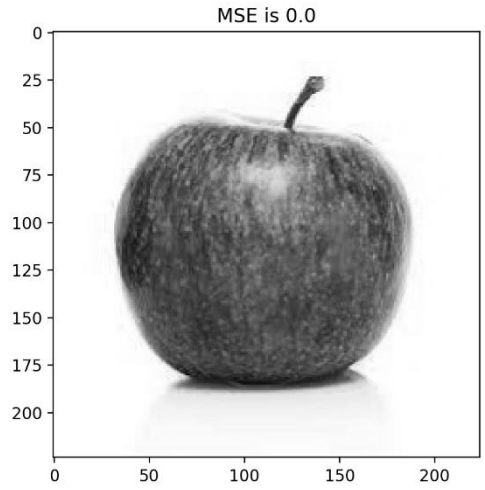
The get_laplacian_pyramid function begins by converting the input image to float32 for numerical precision. It then calls get_gaussian_pyramid to generate a multi-scale representation of the image with progressively smaller versions. To compute the Laplacian pyramid, it precomputes upscaled versions of the next Gaussian levels using a dictionary, ensuring efficient access. Each Laplacian level is obtained by subtracting the upscaled next level from the current Gaussian level, preserving the image details lost during downsampling. Finally, the smallest Gaussian image is appended to the Laplacian pyramid as the last level.

```python
def get_laplacian_pyramid(image, levels, resize_ratio=0.5):
    image = np.float32(image.copy())
    # Generate the Gaussian pyramid
    gaussian_pyr = get_gaussian_pyramid(image, levels, resize_ratio)
    # Precompute upscaled images
    upscaled_images = {
        i: cv2.resize(gaussian_pyr[i + 1], (gaussian_pyr[i].shape[1], gaussian_pyr[i].shape[0]), interpolation=cv2.INTER_LINEAR)
        for i in range(levels - 1)
    }
    # Compute Laplacian pyramid
    laplacian_pyr = [
        cv2.subtract(gaussian_pyr[i], upscaled_images[i]) for i in range(levels - 1)
    ]
    laplacian_pyr.append(gaussian_pyr[-1])
    return laplacian_pyr
```

b) Implement the function 'restore_from_pyramid'. It gets a Laplacian pyramid and returns the image (Remember “collapsing” a Laplacian Pyramid?).

The restore_from_pyramid function reconstructs the original image by **collapsing** the Laplacian pyramid. It starts with the smallest image in the pyramid and progressively **upsamples** it using cv2.resize(), restoring the lost details by **adding back** each Laplacian level. This process continues until reaching the original image size. The function ensures pixel values remain valid using np.clip() and converts the final result to uint8 for proper display. The validation function validate_operation(img) confirms the accuracy of the reconstruction, and achieving an **MSE of 0** indicates a perfect restoration, meaning the original and reconstructed images are identical.





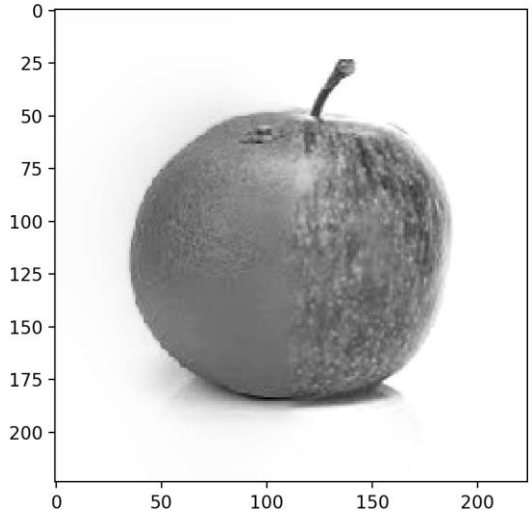
c) Implement the function ‘blend_pyramids’.

The blend_pyramids function merges two Laplacian pyramids by applying a **gradual blending mask** at each level, ensuring a smooth transition between the images. A **mask is created** with the same size as the current pyramid level, initially set to **zero**. The left side of the mask, up to (0.5 * width - curr_level), is set to **1.0**, fully preserving the first image. The **transition region**, spanning from (0.5 * width - curr_level) to (0.5 * width + curr_level), is assigned values according to the formula 0.9 - 0.9 * i / (2 * curr_level), gradually fading from **0.9 to 0** as required. To ensure smooth blending, a **Gaussian blur with a kernel size of (11,11)** is applied to the mask. The final blended pyramid level is computed using the cross-dissolve formula:

$$blended[i] = orange[i] \cdot mask + apple[i] \cdot (1 - mask)$$

This approach **strictly follows the given instructions**, ensuring that the transition **widens** as the pyramid level increases, making the blending **more natural** at lower frequencies.

This is the resulting blended image, created using the Laplacian pyramid method:



d) The new image is the blending between ‘orange.jpg’ and ‘apple.jpg’. Create a Laplacian Pyramid for each of the two images, blend those two pyramids

per-level and then restore the blended image from the result pyramid (do all of this using the functions you've implemented).

We perform the entire blending process in this section. We start by loading and converting the images to grayscale, then validate the Laplacian pyramid reconstruction. Next, we generate the Laplacian pyramids for both images and blend them at each level using a transition mask. Finally, we reconstruct the blended image from the pyramid, display it, and save the final result.

Challenge

We improved the blending function by widening the transition region using 6 * (curr_level + 1), ensuring a more gradual blending between the images. Additionally, we forced the transition width to be odd, which helps maintain symmetry and enhances the effect of Gaussian blur. Instead of a rigid transition, we applied np.linspace(1.0, 0.0), creating a smoother and more continuous gradient, making the blend appear more natural.

These modifications collectively make the blending **smoother, more seamless, and visually appealing**.