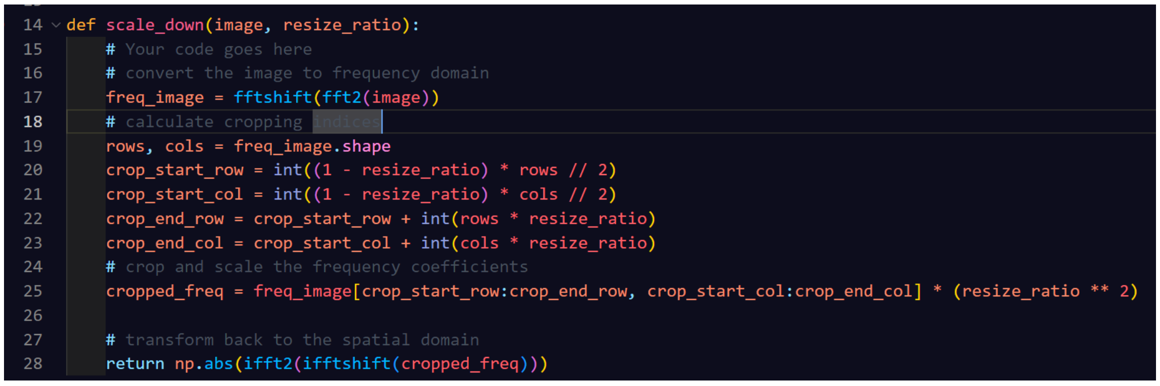
Names:

Raneem Ibraheem (212920896),

Selan Abu Saleh (212111439)

Problem 1 – Template matching

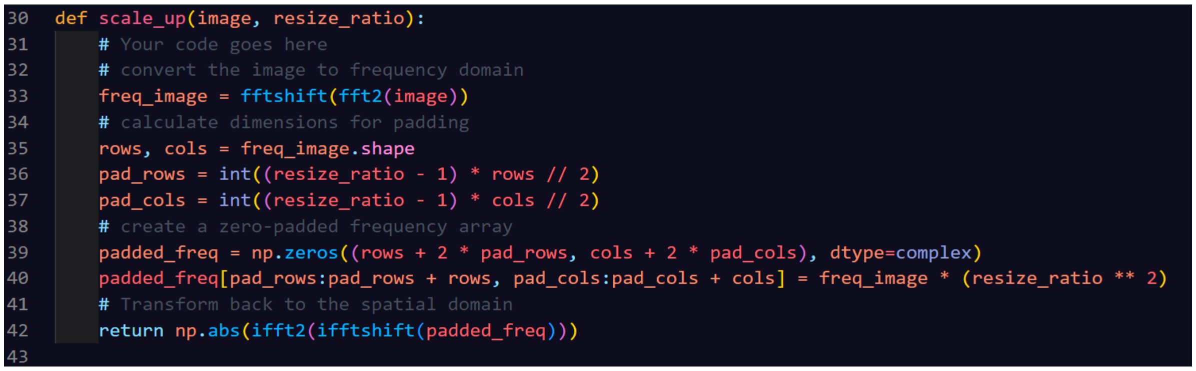
A. First of all, we were asked to implement the scale_down function that takes an image and a resize_ratio and scales the image down by that ratio, hence we started by converting the image to the frequency representation, because when we want to scale down the image, the low frequencies that preserve the important information of the picture will be in the center, so we want to group them in the center and then start cropping the image according to the ratio and keep the middle part that we cropped. After preserving the middle part, we use the inverse transformation to go back to the spatial domain and get the scaled down image.



```python
def scale_down(image, resize_ratio):
    # Your code goes here
    # convert the image to frequency domain
    freq_image = fftshift(fft2(image))
    # calculate cropping indices
    rows, cols = freq_image.shape
    crop_start_row = int((1 - resize_ratio) * rows // 2)
    crop_start_col = int((1 - resize_ratio) * cols // 2)
    crop_end_row = crop_start_row + int(rows * resize_ratio)
    crop_end_col = crop_start_col + int(cols * resize_ratio)
    # crop and scale the frequency coefficients
    cropped_freq = freq_image[crop_start_row:crop_end_row, crop_start_col:crop_end_col] * (resize_ratio ** 2)

    # transform back to the spatial domain
    return np.abs(ifft2(ifftshift(cropped_freq)))
```

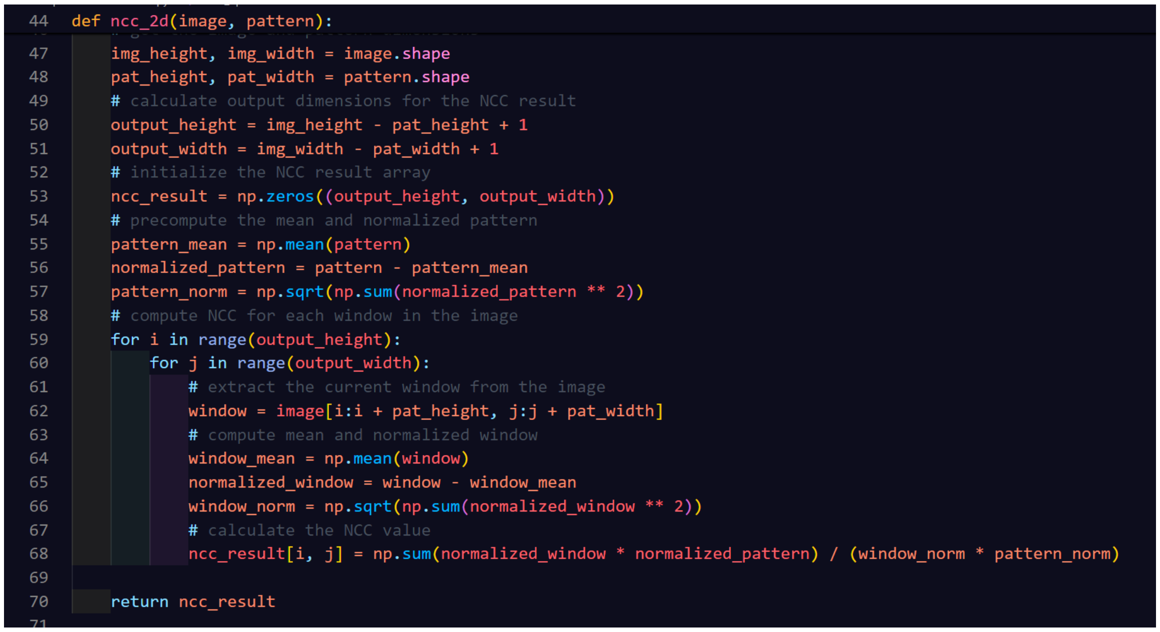
B. As for the scale_up image, a similar approach to the scale_down function is taken, but instead we create an image that is all zeros, and then we put our image that is in

the frequency domain in the middle of the zeros image, so what happens is that we get the frequency representation of our original image padded with zeros on all the sides, of course all of these happen in relation to the ratio size. Then we do the inverse transform to go back to the spatial domain and get our scaled up image.



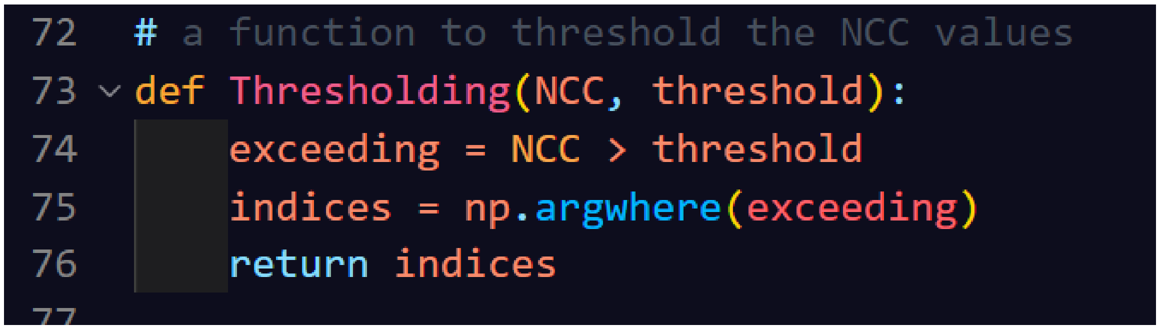
```python
def scale_up(image, resize_ratio):
    # Your code goes here
    # convert the image to frequency domain
    freq_image = fftshift(fft2(image))
    # calculate dimensions for padding
    rows, cols = freq_image.shape
    pad_rows = int((resize_ratio - 1) * rows // 2)
    pad_cols = int((resize_ratio - 1) * cols // 2)
    # create a zero-padded frequency array
    padded_freq = np.zeros((rows + 2 * pad_rows, cols + 2 * pad_cols), dtype=complex)
    padded_freq[pad_rows:pad_rows + rows, pad_cols:pad_cols + cols] = freq_image * (resize_ratio ** 2)
    # Transform back to the spatial domain
    return np.abs(ifft2(ifftshift(padded_freq)))

```

C. In this section we were requested to implement the NCC function which takes the image and the pattern, and then slide the pattern across every possible position. We calculate the mean and the standard deviation for each overlapping window and normalize both to ensure the scale is similar. Then we use the formula of the NCC value at each position to get the 2d array of the NCC result where each value is the similarity between the pattern and the position in the image.



```python
def ncc_2d(image, pattern):
    img_height, img_width = image.shape
    pat_height, pat_width = pattern.shape
    # calculate output dimensions for the NCC result
    output_height = img_height - pat_height + 1
    output_width = img_width - pat_width + 1
    # initialize the NCC result array
    ncc_result = np.zeros((output_height, output_width))
    # precompute the mean and normalized pattern
    pattern_mean = np.mean(pattern)
    normalized_pattern = pattern - pattern_mean
    pattern_norm = np.sqrt(np.sum(normalized_pattern ** 2))
    # compute NCC for each window in the image
    for i in range(output_height):
        for j in range(output_width):
            # extract the current window from the image
            window = image[i:i + pat_height, j:j + pat_width]
            # compute mean and normalized window
            window_mean = np.mean(window)
            normalized_window = window - window_mean
            window_norm = np.sqrt(np.sum(normalized_window ** 2))
            # calculate the NCC value
            ncc_result[i, j] = np.sum(normalized_window * normalized_pattern) / (window_norm * pattern_norm)

    return ncc_result
```

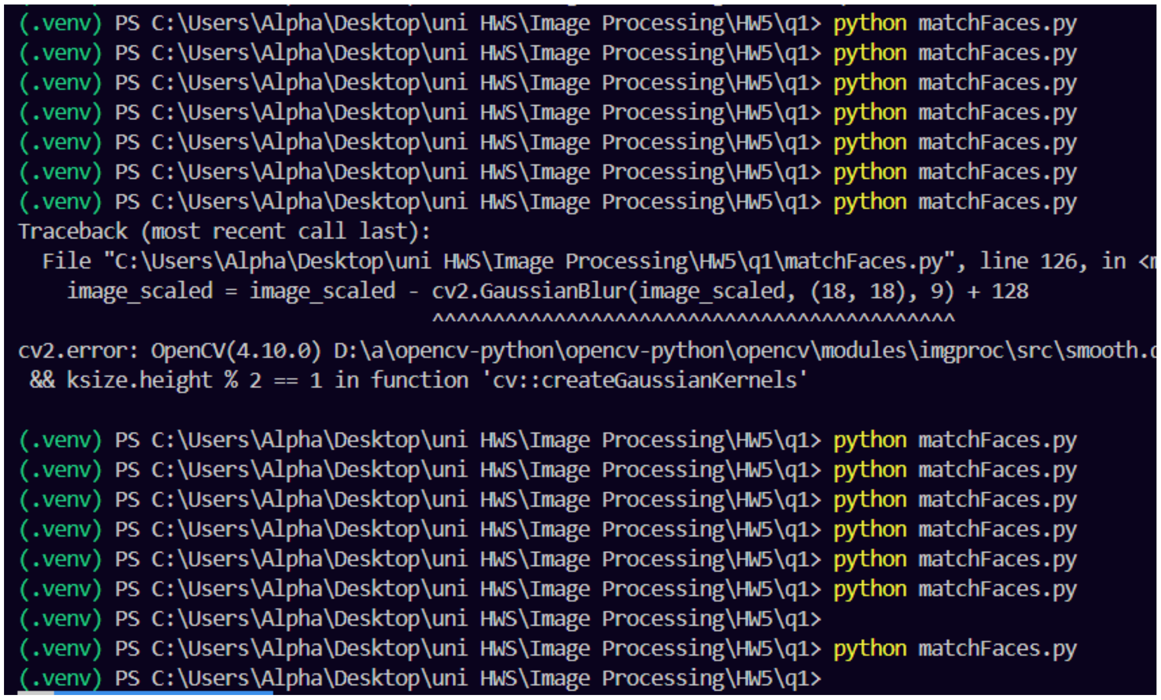
D. **Technically in here we only had to implement the Thresholding function which takes an NCC value and a threshold and filters out the values that don't exceed the threshold.**



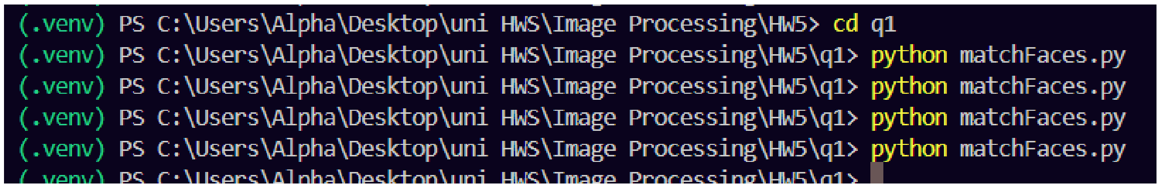
```python
# a function to threshold the NCC values
def Thresholding(NCC, threshold):
    exceeding = NCC > threshold
    indices = np.argwhere(exceeding)
    return indices
```

E. **For this section we had to scale up the images, and scale down the filters to an appropriate size in order to find the matching pattern, and we had use a gaussian blur for both the first image and the pattern to reduce the noise and smooth the high frequencies, and this helps to get a more accurate pattern matching. As for the second image,**

blurring wasn't necessary because the facial features were more defined and blurring in this case might damage those features and reduce the accuracy.
The parameters choice in this section was through a trial-and-error process to find the best befitting parameters:

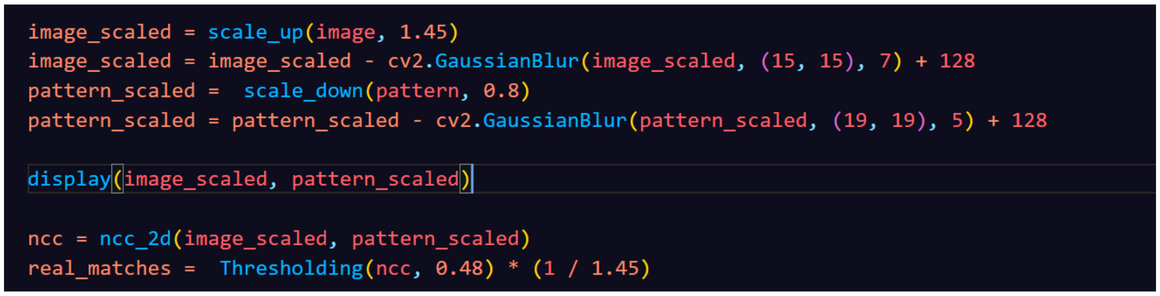


```
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
Traceback (most recent call last):
  File "C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1\matchFaces.py", line 126, in <m
    image_scaled = image_scaled - cv2.GaussianBlur(image_scaled, (18, 18), 9) + 128
                                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
cv2.error: OpenCV(4.10.0) D:\a\opencv-python\opencv-python\opencv\modules\imgproc\src\smooth.c
 && ksize.height % 2 == 1 in function 'cv::createGaussianKernels'

(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1>
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1>
```

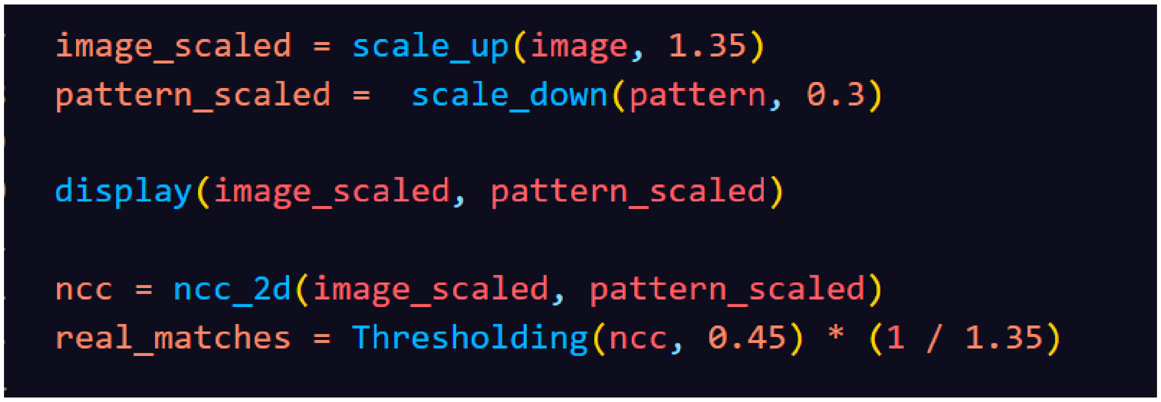


```
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5> cd q1
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1> python matchFaces.py
(.venv) PS C:\Users\Alpha\Desktop\uni HWS\Image Processing\HW5\q1>
```

Note: more trials have been conducted and there could have been better parameters to choose, but the current parameters are performing well in this case.

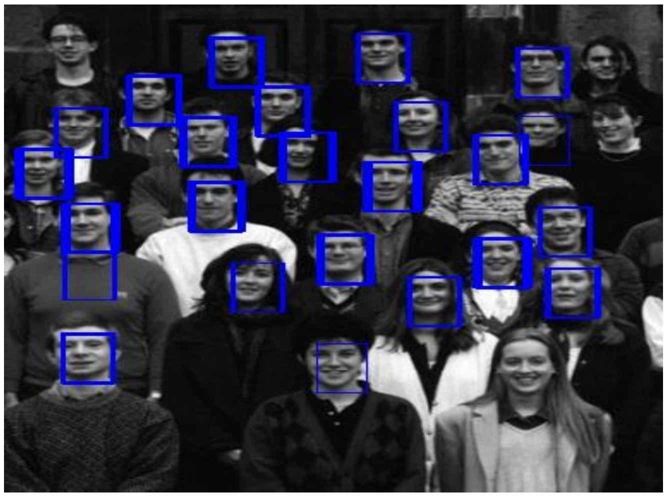


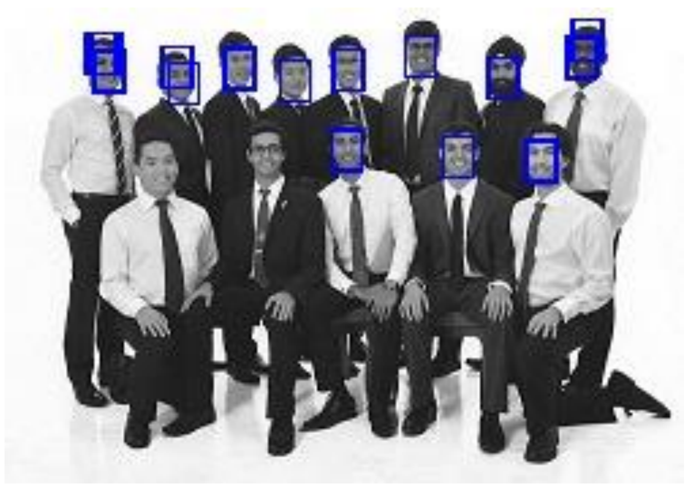
```python
image_scaled = scale_up(image, 1.45)
image_scaled = image_scaled - cv2.GaussianBlur(image_scaled, (15, 15), 7) + 128
pattern_scaled =  scale_down(pattern, 0.8)
pattern_scaled = pattern_scaled - cv2.GaussianBlur(pattern_scaled, (19, 19), 5) + 128

display(image_scaled, pattern_scaled)

ncc = ncc_2d(image_scaled, pattern_scaled)
real_matches =  Thresholding(ncc, 0.48) * (1 / 1.45)
```



```
image_scaled = scale_up(image, 1.35)
pattern_scaled =  scale_down(pattern, 0.3)

display(image_scaled, pattern_scaled)

ncc = ncc_2d(image_scaled, pattern_scaled)
real_matches = Thresholding(ncc, 0.45) * (1 / 1.35)
```

Results for the last section:





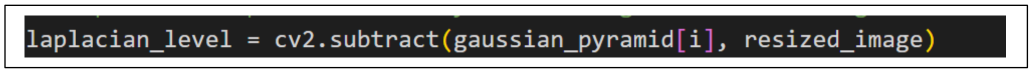
Problem 2 – Multiband blending

Blending two images using Laplacian pyramids is a technique that combines aspects of multi-resolution analysis and spatial blending. It enables smooth transitions between two images by mixing them at different levels of detail.

a:

We have to follow these steps:

- **Build the Gaussian pyramids for apple image and orange image**
- **Build Laplacian pyramids to both images using Gaussian pyramids**
- **Li= Gi - expand (Gi+1)**

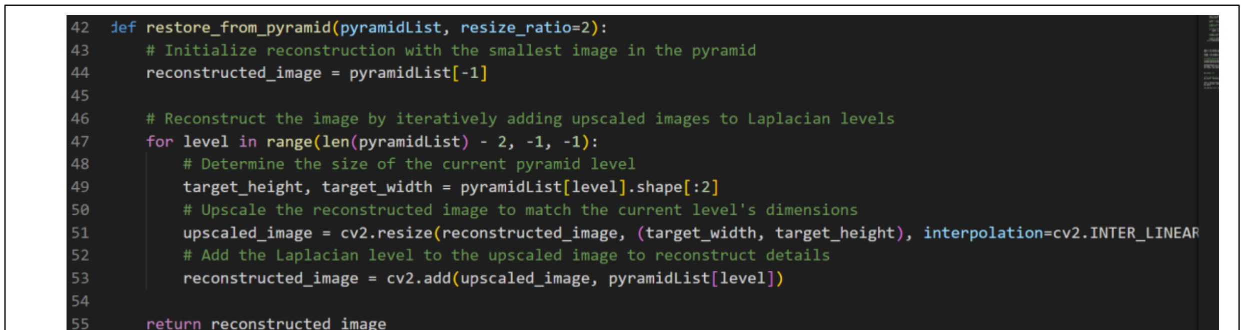


```
laplacian_level = cv2.subtract(gaussian_pyramid[i], resized_image)
```

I use np.float 32 to convert the input `image` to the `float32` data type to get

b:

This is how we can restore an image from a Laplacian pyramid, we begin with the smallest image which is the bottom of the pyramid and then iteratively upscale it to the size of the next larger pyramid level and add back the Laplacian detail which restores finer features, this process continues until we reach the original image size.

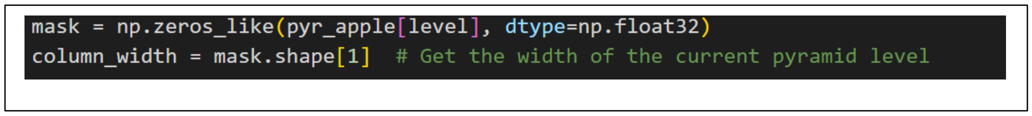


```
def restore_from_pyramid(pyramidList, resize_ratio=2):
    # Initialize reconstruction with the smallest image in the pyramid
    reconstructed_image = pyramidList[-1]

    # Reconstruct the image by iteratively adding upscaled images to Laplacian levels
    for level in range(len(pyramidList) - 2, -1, -1):
        # Determine the size of the current pyramid level
        target_height, target_width = pyramidList[level].shape[:2]
        # Upscale the reconstructed image to match the current level's dimensions
        upscaled_image = cv2.resize(reconstructed_image, (target_width, target_height), interpolation=cv2.INTER_LINEAR
        # Add the Laplacian level to the upscaled image to reconstruct details
        reconstructed_image = cv2.add(upscaled_image, pyramidList[level])

    return reconstructed_image
```

C:

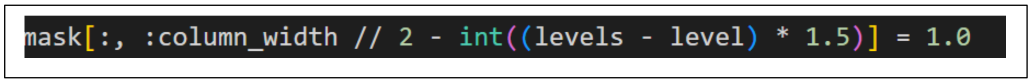
To blend pyramids we have to:

.1. Create an empty mask whit the same shape as the current pyramid level



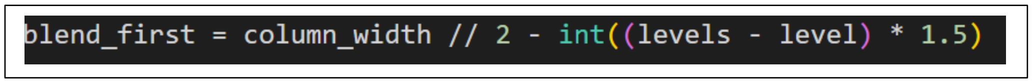
```python
mask = np.zeros_like(pyr_apple[level], dtype=np.float32)
column_width = mask.shape[1]  # Get the width of the current pyramid level
```

2. set the left half of the mask to 1



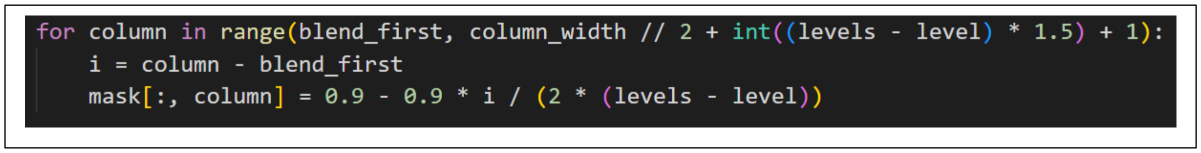
```python
mask[:, :column_width // 2 - int((levels - level) * 1.5)] = 1.0
```

3.define the start point for the blend area



```python
blend_first = column_width // 2 - int((levels - level) * 1.5)
```

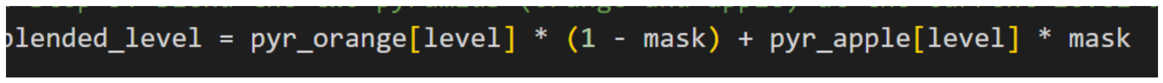
4.gradually create the blend by filling the transition area with a gradient:



```python
for column in range(blend_first, column_width // 2 + int((levels - level) * 1.5) + 1):
    i = column - blend_first
    mask[:, column] = 0.9 - 0.9 * i / (2 * (levels - level))
```

5. blend the two pyramids at the current level using mask:

Lblend = maski * L1i + (1 – maski) L2i

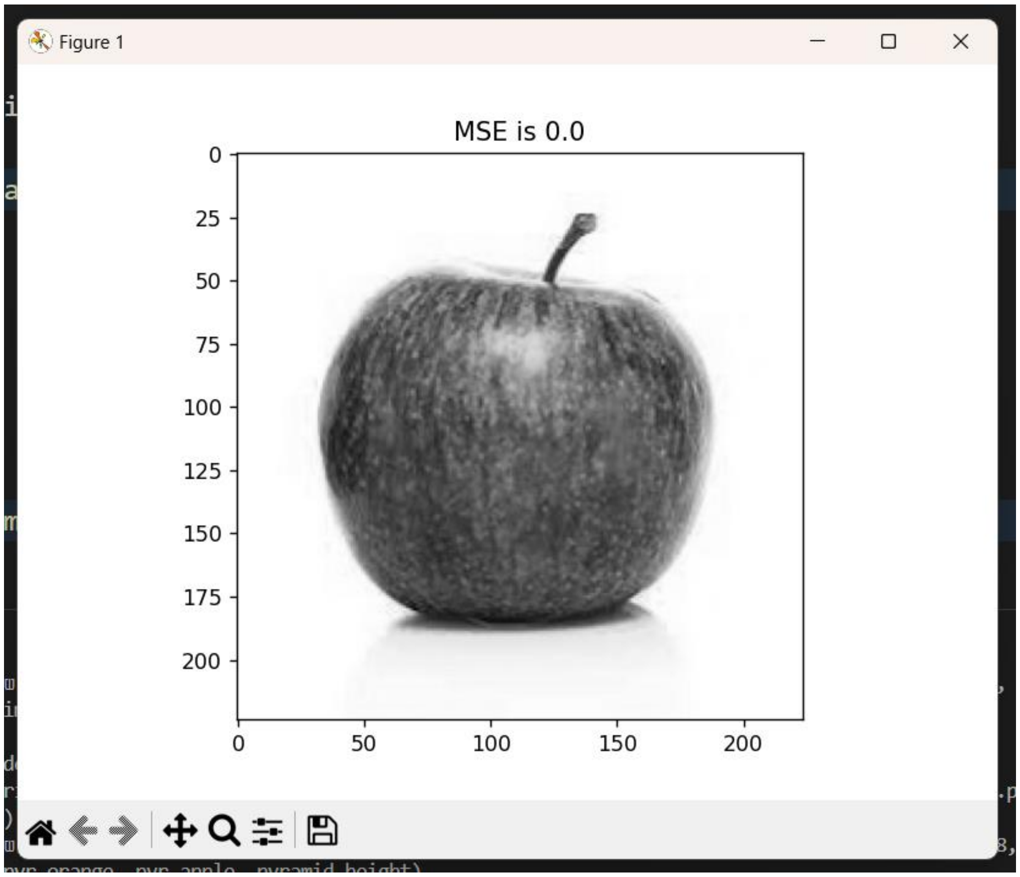


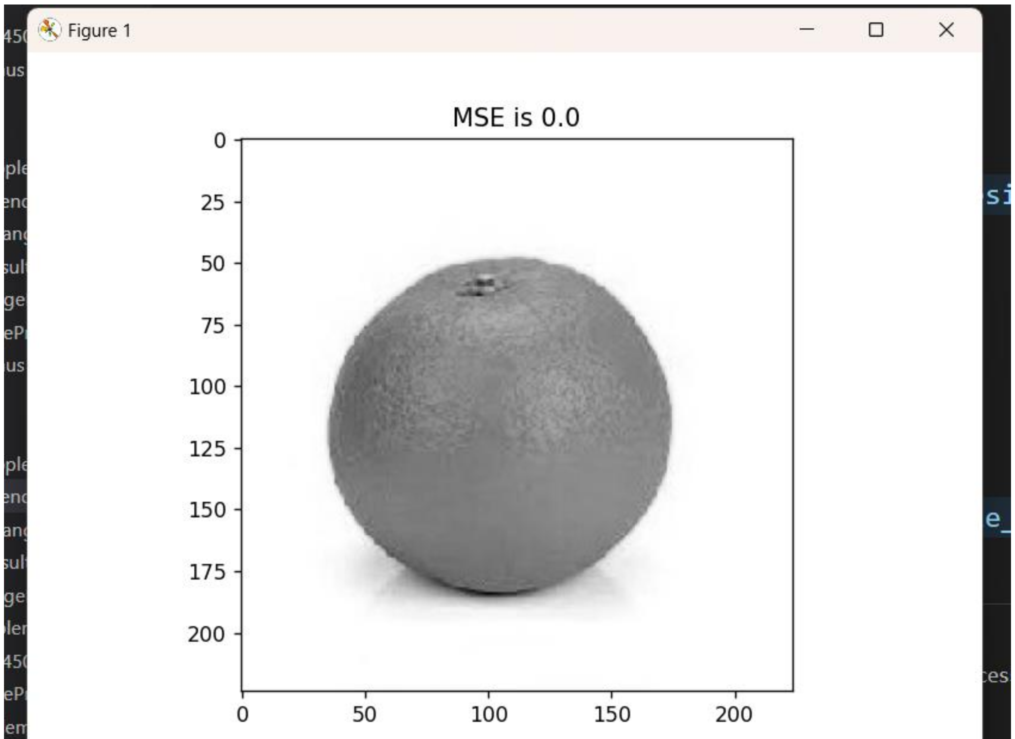
```
blended_level = pyr_orange[level] * (1 - mask) + pyr_apple[level] * mask
```

We started by defining a mask that matches the size of the current pyramid level. This mask is essentially a 2D matrix, with the same dimensions as the image at the current pyramid level, for the mask's columns, we initialized them from the first column up to 0.5 * width - current level, and we set these values to 1.0, ensuring that the initial part of the mask is fully set to one image, allowing for a smooth transition to the next stage of blending.

I chose 7 levels for the pyramid to ensure more detailed and accurate blending, especially for images with a resolution of 224x224. Each level represents a progressively smaller version of the original image, capturing different levels of detail.

224*224 -> 112*112 -> 56*56 -> 28*28 -> 14*14 -> 7*7 -> 3*3





the result: