Ilan Sapoznikov

Liad Arvatz

Image Processing: Assignment #5

Problem 1 – Template matching:

Code explanation:

scale_down(image, resize_ratio):

This function reduces the size of an image by a specified ratio using the Fourier transform. It involves applying the Fourier transform, cropping the frequency domain, and then performing the inverse Fourier transform.

scale_up(image, resize_ratio):

This function increases the size of an image by a specified ratio using a similar method to scale_down, but with padding in the frequency domain instead of cropping.

ncc_2d(image, pattern):

This function computes the normalized cross-correlation (NCC) between an image and a pattern by first calculating the mean of the pattern and each potential matching window in the image. It normalizes the pattern and each window by subtracting their respective means. Then, it multiplies the normalized pattern with each normalized window element-wise, and sums the products. This sum is divided by the product of their standard deviations to normalize the correlation, ensuring it is scaled such that 1 indicates a perfect match and 0 indicates no correlation. The function handles cases where a window's standard deviation is zero by skipping the division to avoid mathematical errors, effectively managing uniform areas within the image.

cluster_matches(matches, eps, min_samples):

This is a function that we added to deal with multiple detections near the same object (in our case people faces). The cluster_matches function is designed to process a list of detected pattern matches in an image and cluster them to identify unique occurrences of the pattern. This is important because the normalized cross-correlation (NCC) used for template matching can often detect multiple closely situated high-response areas for the same pattern, especially if the pattern or parts of it appear multiple times near each other.

*The **display(image, pattern)** and **draw_matches(image, matches, pattern_size)** functions were left as they were provided in the assignment because we wanted to challenge ourselves by dealing with the general structure of the script without changing its logical structure. The same is true for the main part of the script, towards the end of the code, where we only added lines responsible for filtering out overlapping detected faces using the **cluster_matches** function.

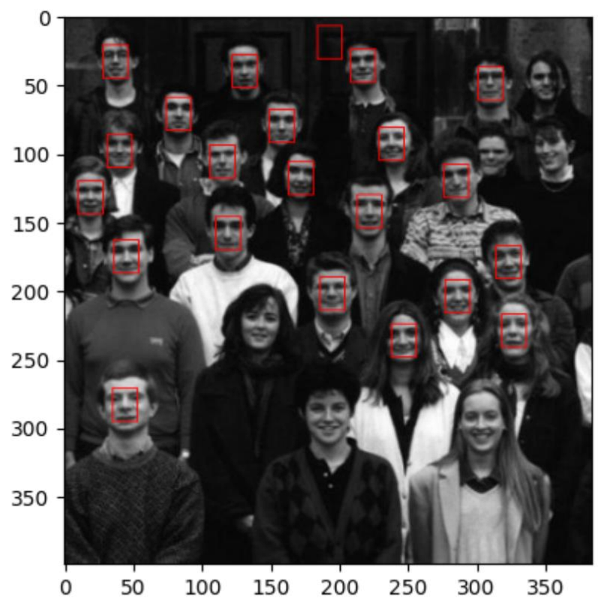
Now let's walk through the flow and functionality of this script:

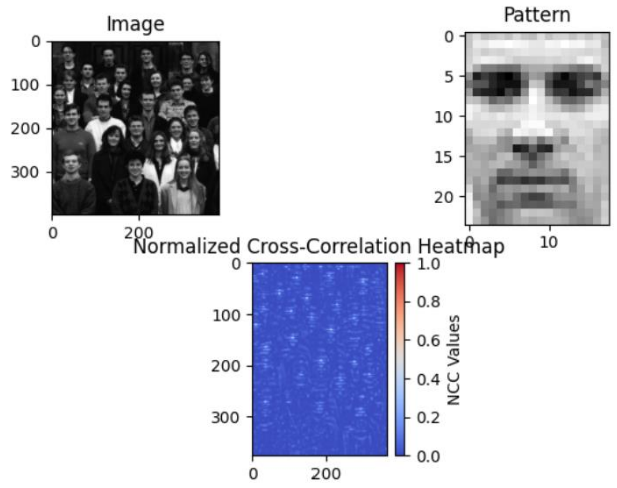
- For each image (“students” and “thecrew”), the images are loaded and converted to grayscale, which simplifies the processing by reducing the dimensionality of the data. And also a pattern image is loaded similarly and is intended to be detected within the main images.
- The pattern is scaled down to adjust for size discrepancies, improving detection accuracy.
  For both of the images the pattern in scaled down by 50% (resize_ratio=0.5). On the other hand, unlike the "students" image that is not scaled, the "thecrew" image is scaled up by 200% (resize_ratio=2) using the scale_up function. This scaling up is performed to enhance small details in the image, possibly making it easier to match the pattern. Of course it took a lot of trial and error to eventually get this parameters.
- The ncc_2d function is used to compute the NCC between the scaled image (which is actually the original grayscale image in this instance, as the image itself isn't resized) and the scaled-down pattern. This computation generates a map showing how well the pattern matches different parts of the image.
  For the “students” image a threshold of 0.55 is applied to the NCC results to filter out less significant matches. And for the “thecrew” image we used a lower threshold (0.44). Only regions with a correlation above this threshold are considered real matches.
- The coordinates of the matches are adjusted to account for the center of the pattern. This ensures that detected points refer to the center of the matched pattern in the image. Since the “thecrew” image was scaled up for matching, the detected coordinates are adjusted by dividing by 2 to map back to the original image scale.
- For both pictures, the cluster_matches function applies clustering to the adjusted match points to group closely located detections into single detections. This step reduces redundancy by ensuring that each detected pattern is marked only once.

The Results:

"students.jpg":

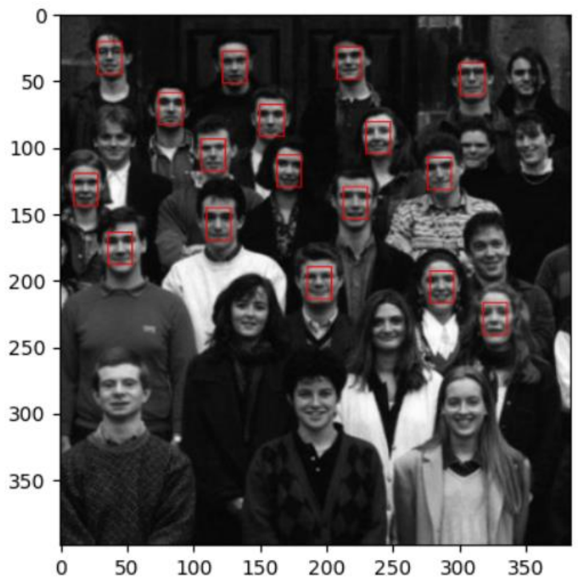
<u>Option #1:</u>





- Scale down pattern by 0.5
- Without scaling the image
- Set threshold to **0.55**

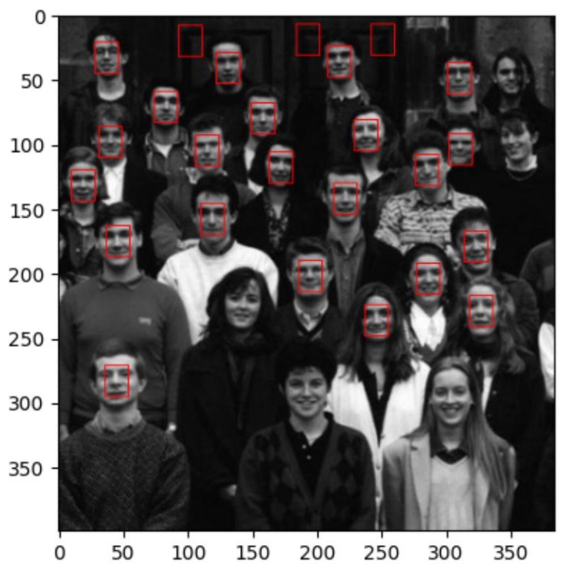
<u>Option #2:</u>



- Scale down pattern by 0.5
- Without scaling the image
- Set threshold to 0.6

In this option, by increasing the threshold from 0.5 to 0.6, we eliminated the only false positive detection from the previous option. However, this change also resulted in one fewer face being detected compared to before, though most faces are still identified.

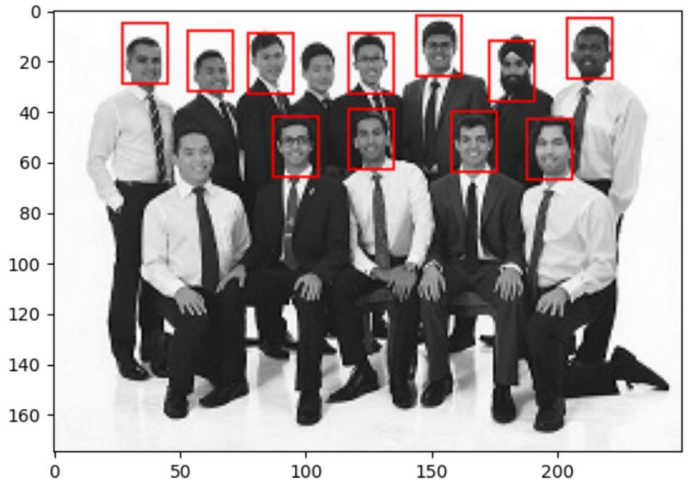
<u>Option #3:</u>

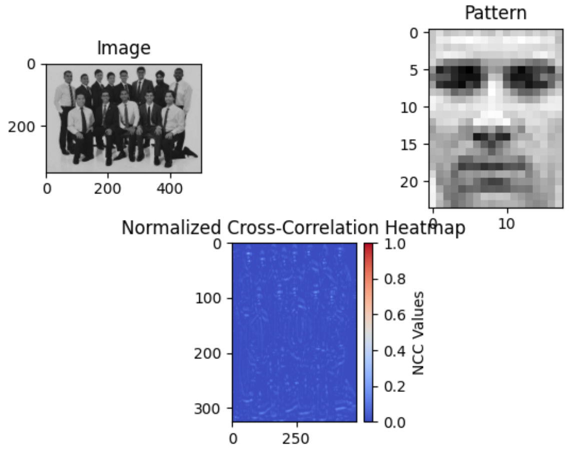


- Scale down pattern by 0.5
- Without scaling the image
- Set threshold to 0.45

As expected, decreasing the threshold allows us to recognize more faces, or even all of them, however, this comes with the trade-off of more false positives. Option #3 is an example of it. Buy decreasing the threshold to 0.4 we managed to detect even more face, but with additional false positives.

“thecrew.jpg”:





- Scale down pattern by 0.5
- Scale up image by 2
- Set threshold to 0.44

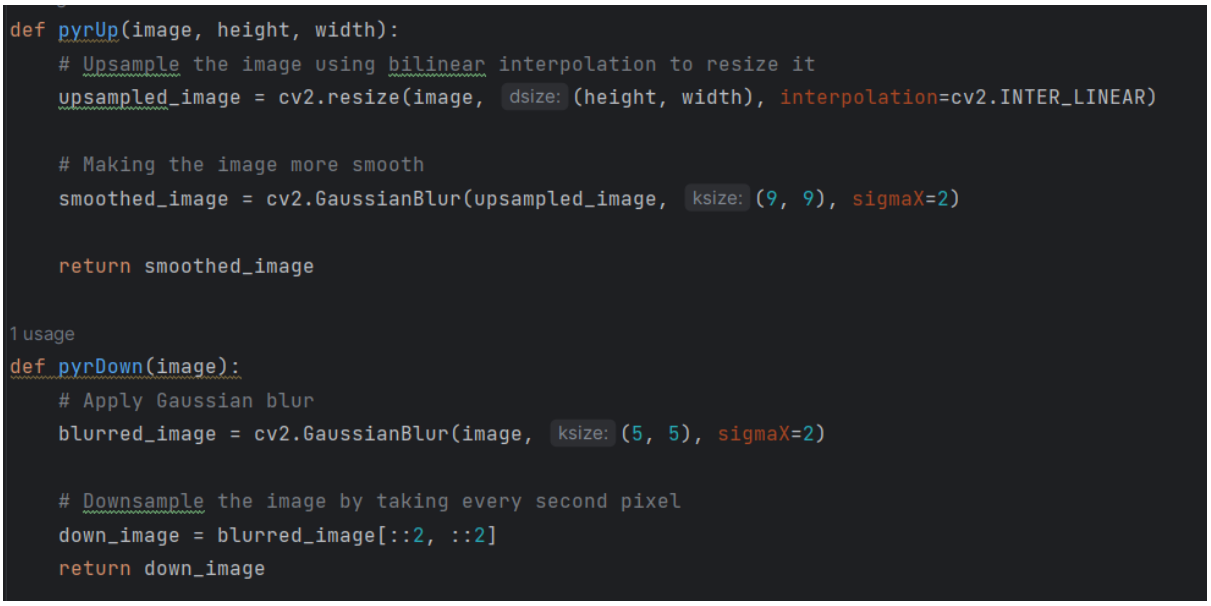
With this set of parameters, we achieved the best result compared to the other combinations of values we tried, detecting most of the faces without any false positives.

Problem 2 – Multiband blending:

Code explanation:

pyrUp(image, height, width):
The function gets an image and two dimensions size (height and width). Then the function resizes the image to the dimension it gets and smooths the image by Gaussian blur (because the image is blurring).

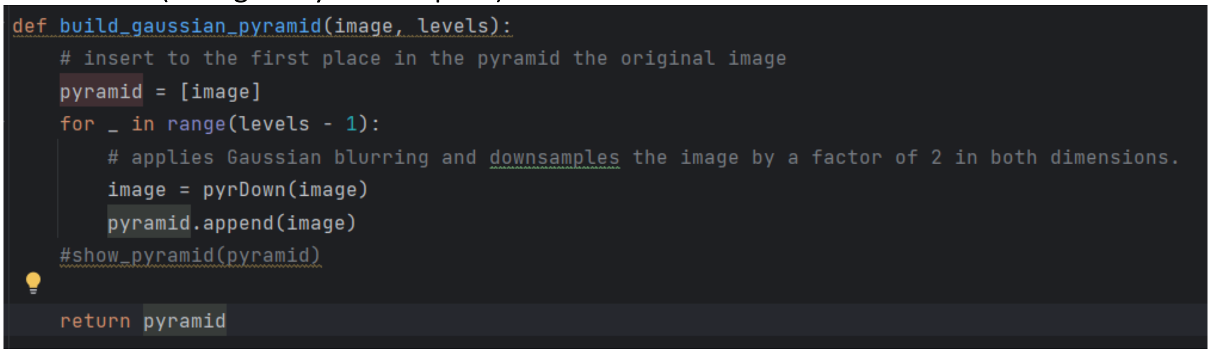
pyrDown(image):
The image gets an image, blurring it image by Gaussian blur, and then decreasing image's size by taking every second pixel.



```python
def pyrUp(image, height, width):
    # Upsample the image using bilinear interpolation to resize it
    upsampled_image = cv2.resize(image, dsize: (height, width), interpolation=cv2.INTER_LINEAR)

    # Making the image more smooth
    smoothed_image = cv2.GaussianBlur(upsampled_image, ksize: (9, 9), sigmaX=2)

    return smoothed_image

1 usage
def pyrDown(image):
    # Apply Gaussian blur
    blurred_image = cv2.GaussianBlur(image, ksize: (5, 5), sigmaX=2)

    # Downsample the image by taking every second pixel
    down_image = blurred_image[::2, ::2]
    return down_image
```

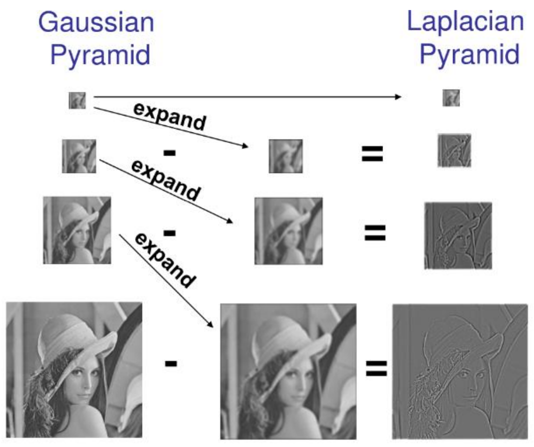
pyrUp size kernel:9x9*
pyrDown size kernel: 5X5*

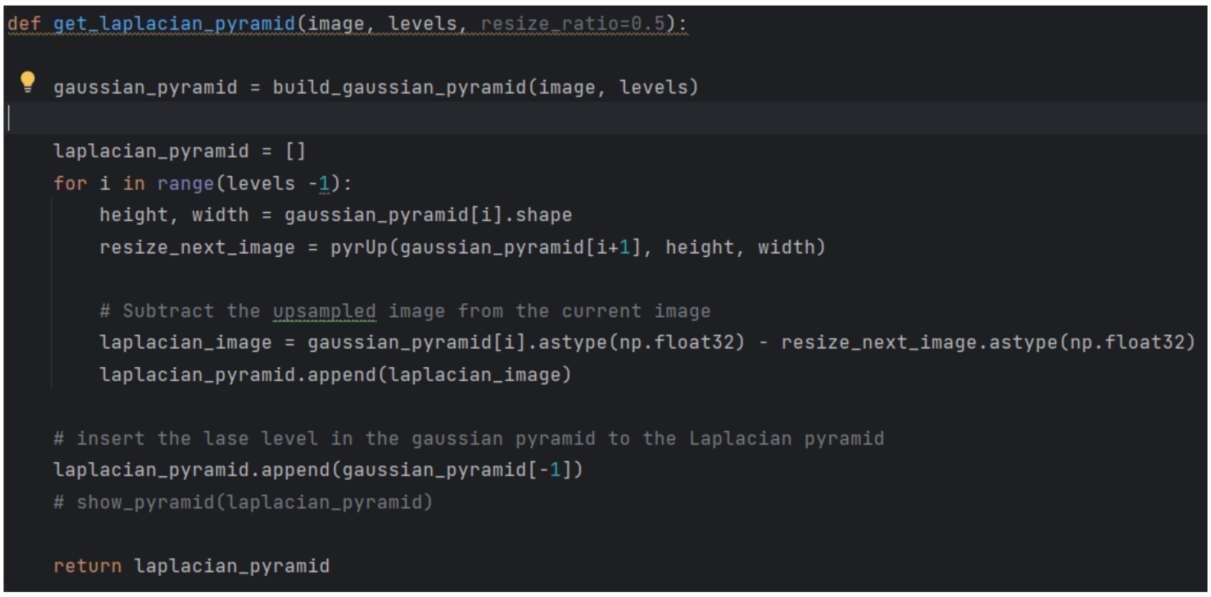
build_gaussian_pyramid(image, levels):
The function gets the original image and number of levels we want for our Laplacian pyramid, it blurs the image in every step and decreases its size by factor two in both dimensions (taking every second pixel).



```python
def build_gaussian_pyramid(image, levels):
    # insert to the first place in the pyramid the original image
    pyramid = [image]
    for _ in range(levels - 1):
        # applies Gaussian blurring and downsamples the image by a factor of 2 in both dimensions.
        image = pyrDown(image)
        pyramid.append(image)
    #show_pyramid(pyramid)

    return pyramid
```

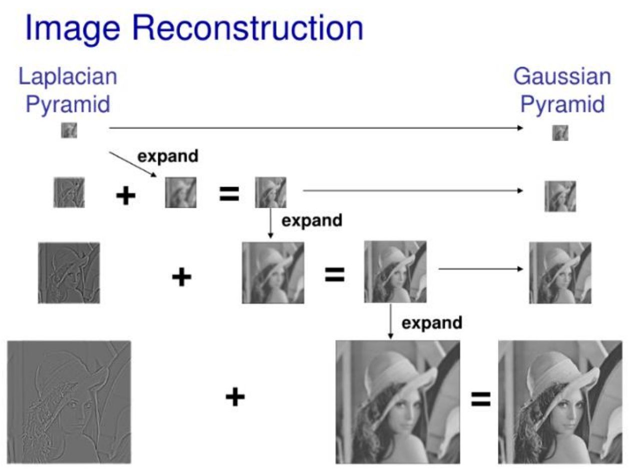
get_laplacian_pyramid(image, levels, resize_ratio):
The function creates the Laplacian pyramid. The function gets the original image and the number of levels we want. First of all it creates the gaussian pyramid. Then the function runs over the images in the gaussian pyramid. For every image it takes it's sharp and resizes the next image in the pyramid (which should be smaller). pyrUp method smoothes the image and then we subtract gaussianPyramid[i]-resize_next_image. The algorithm is similar to the one we can see here:

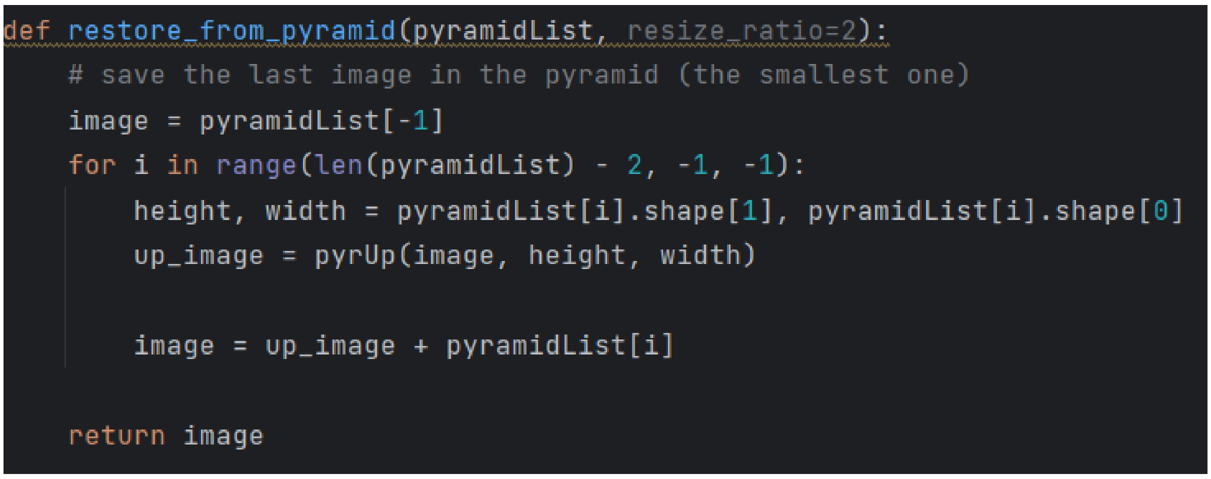




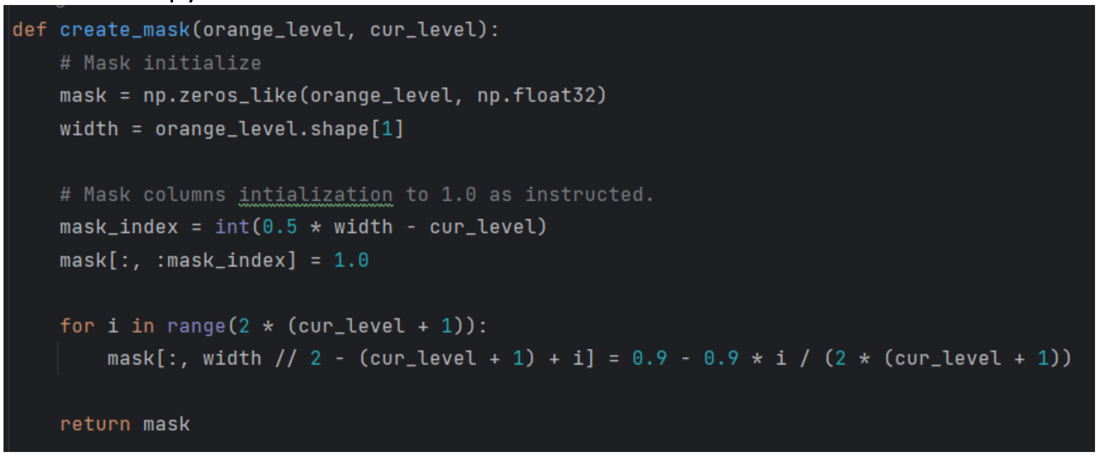
```
def get_laplacian_pyramid(image, levels, resize_ratio=0.5):

    gaussian_pyramid = build_gaussian_pyramid(image, levels)

    laplacian_pyramid = []
    for i in range(levels -1):
        height, width = gaussian_pyramid[i].shape
        resize_next_image = pyrUp(gaussian_pyramid[i+1], height, width)

        # Subtract the upsampled image from the current image
        laplacian_image = gaussian_pyramid[i].astype(np.float32) - resize_next_image.astype(np.float32)
        laplacian_pyramid.append(laplacian_image)

    # insert the lase level in the gaussian pyramid to the Laplacian pyramid
    laplacian_pyramid.append(gaussian_pyramid[-1])
    # show_pyramid(laplacian_pyramid)

    return laplacian_pyramid
```

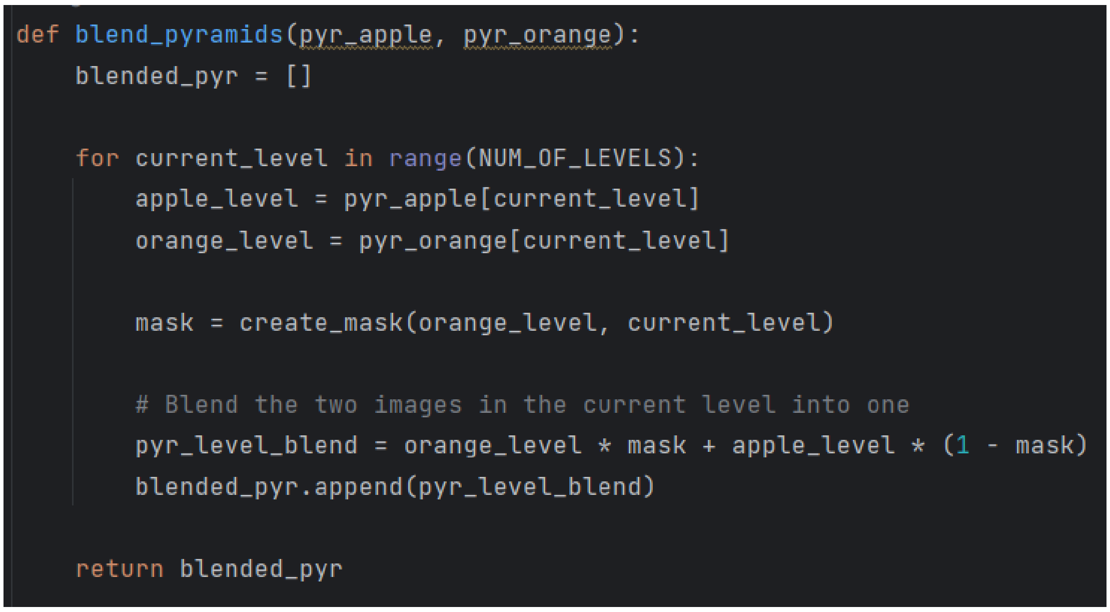
restore_from_pyramid(pyramidList, resize_ratio):
The function restores the original image from the Laplacian pyramid using the following algorithm:
save the last (and smallest image) in the Laplacian image.
we resize the image in the current level to be in size of the previous level, then we summarize the resize_image + previous_level_image (we say previous level because we run over the pyramid images in reverse)




```
def restore_from_pyramid(pyramidList, resize_ratio=2):
    # save the last image in the pyramid (the smallest one)
    image = pyramidList[-1]
    for i in range(len(pyramidList) - 2, -1, -1):
        height, width = pyramidList[i].shape[1], pyramidList[i].shape[0]
        up_image = pyrUp(image, height, width)

        image = up_image + pyramidList[i]

    return image
```

blend_pyramid+create_mask:
The function gets two images and creates a blended pyramid.
For every level in range between [0,NUM_OF_LEVELS), we create a mask for the current level gradually transitions from one pyramid to the other pyramid. It assigns weights to each pixel, by using the formula we were given. At last, the blending process combines the two pyramids based on the calculated mask, and returns the blended Laplacian pyramid.
The last operation we do to get the final result is to call restore_from_pyramid method with the blended pyramid.

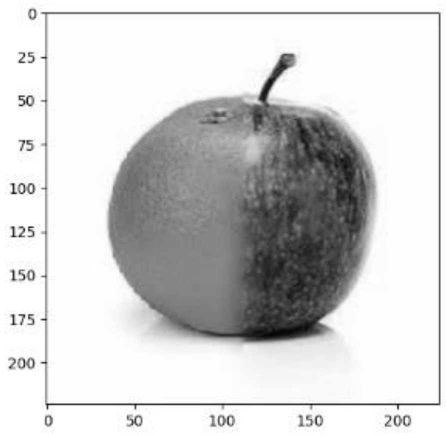

```
def create_mask(orange_level, cur_level):
    # Mask initialize
    mask = np.zeros_like(orange_level, np.float32)
    width = orange_level.shape[1]

    # Mask columns intialization to 1.0 as instructed.
    mask_index = int(0.5 * width - cur_level)
    mask[:, :mask_index] = 1.0

    for i in range(2 * (cur_level + 1)):
        mask[:, width // 2 - (cur_level + 1) + i] = 0.9 - 0.9 * i / (2 * (cur_level + 1))

    return mask
```



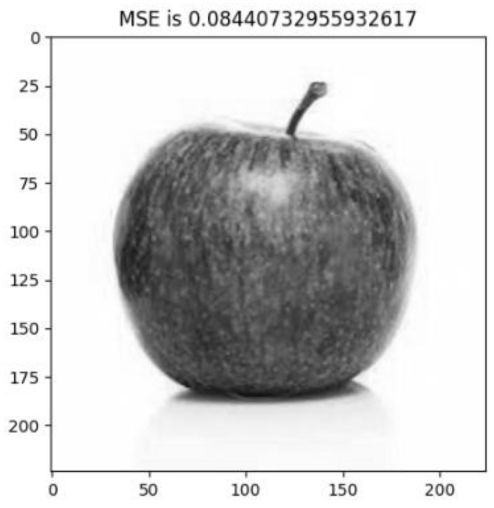
```
def blend_pyramids(pyr_apple, pyr_orange):
    blended_pyr = []

    for current_level in range(NUM_OF_LEVELS):
        apple_level = pyr_apple[current_level]
        orange_level = pyr_orange[current_level]

        mask = create_mask(orange_level, current_level)

        # Blend the two images in the current level into one
        pyr_level_blend = orange_level * mask + apple_level * (1 - mask)
        blended_pyr.append(pyr_level_blend)

    return blended_pyr
```

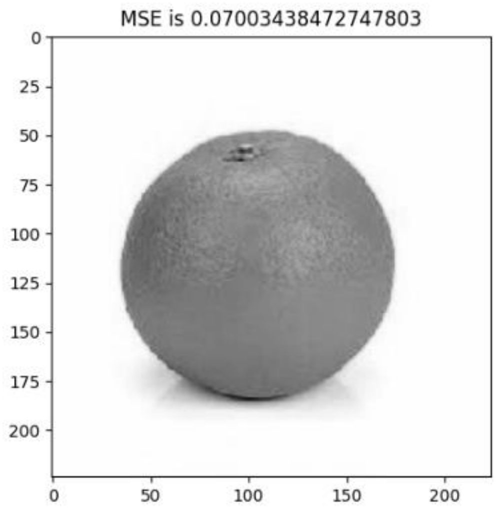
The Results:

The blended image:



Validation function results:





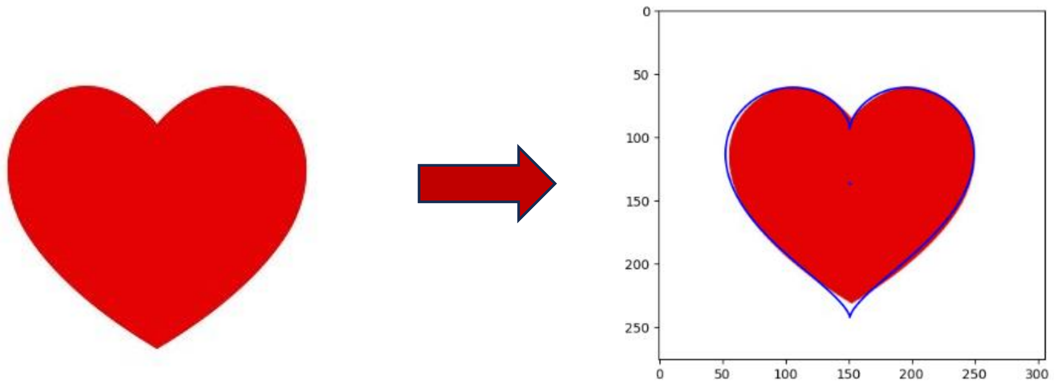
Problem 3 – Hough transform:

The code implements a specialized version of the Hough Transform to detect heart-shaped patterns in images using parametric equations. It reads an image, applies edge detection, and then uses the Hough Transform logic to identify potential heart centers by accumulating votes for parameters defining the heart shape based on the parametric definitions. Detected shapes are filtered based on a voting threshold and proximity to other shapes to avoid duplicates. The identified hearts are then drawn onto the original image, and the results are displayed and saved.

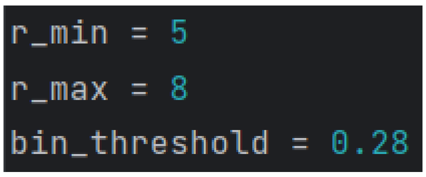
* In addition to the lines of code that we had to change or add in this section of the assignment, we also adjusted the last argument in this line:
`rs = np.arange(r_min, r_max, 0.5)` , for each one of the images. This change caused the plotted hearts in the output image to be more accurate.

The Results:

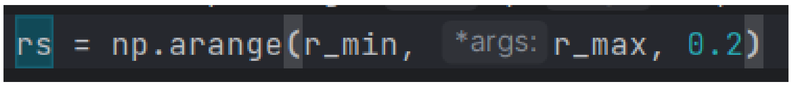
Simple:



Our parameters:

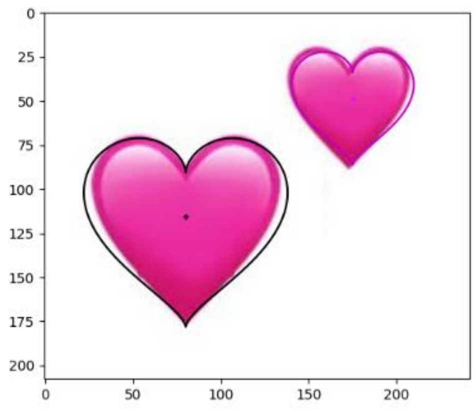


```
r_min = 5
r_max = 8
bin_threshold = 0.28
```

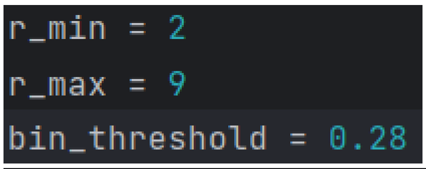


```
rs = np.arange(r_min, *args: r_max, 0.2)
```

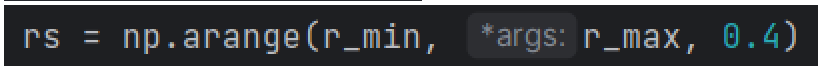
Med:



Our parameters:



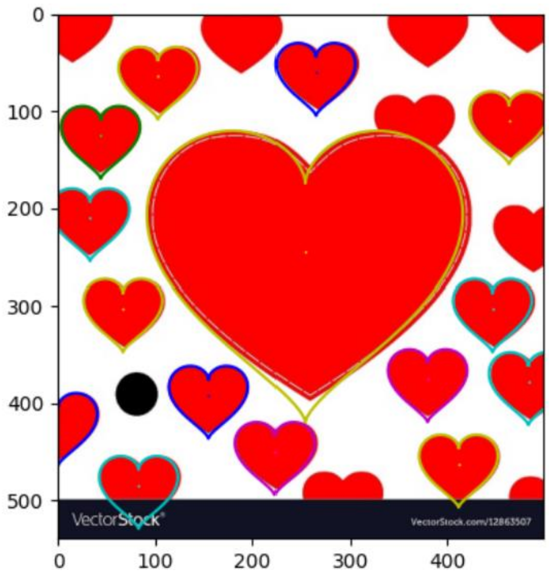
```
r_min = 2
r_max = 9
bin_threshold = 0.28
```



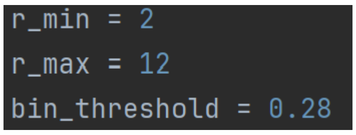
```
rs = np.arange(r_min, *args: r_max, 0.4)
```

Hard:

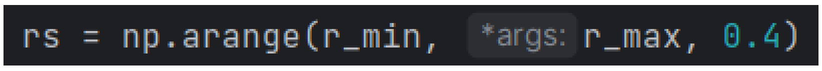
VectorStock®
VectorStock.com/12863507



Our parameters:



```
r_min = 2
r_max = 12
bin_threshold = 0.28
```



```
rs = np.arange(r_min, *args: r_max, 0.4)
```