Image Processing HW_5

Student -1- : Mostufa Jbareen (212955587) מוסטפא גבארין

Student -2- : Mohammed Egbaria (318710761) מוחמד אגבאריה

Problem 1:

Section a:

Implementing scale_down function by applying our knowledge on Fourier, we know that a scaled down image by ratio is the image smaller in that ratio and has and without the high frequencies in the same ratio, so we take the crop the Fourier transform by the resize_ratio starting from the lower frequencies, and then we inverse the cropped window to get the sacled_down image.

```
# Section a
def scale_down(image, resize_ratio):
    fourier_spectrum = fftshift(fft2(image))
    h, w = image.shape
    new_h, new_w = int(h / resize_ratio), int(w / resize_ratio)

    start_row = (h - new_h) // 2
    start_col = (w - new_w) // 2
    end_row = start_row + new_h
    end_col = start_col + new_w

    cropped_spectrum = fourier_spectrum[start_row:end_row, start_col:end_col]

    return np.abs(ifft2(ifftshift(cropped_spectrum)))
```

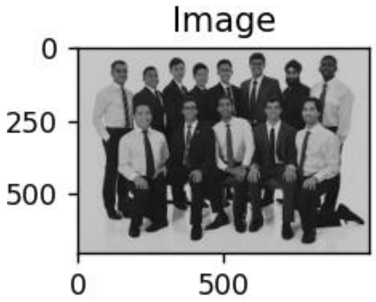
Section b:

The same thing we did in scale_down, butt this time we need to strentgth the freuquencies because now we have a bigger image, and we do that in the last line

```
# Section b
def scale_up(image, resize_ratio):
    fourier_spectrum = fftshift(fft2(image))
    h, w = image.shape
    new_h, new_w = int(h * resize_ratio), int(w * resize_ratio)
    fourier_spectrum_zero_padding = np.zeros((new_h, new_w), dtype=complex)

    start_row = (new_h - h) // 2
    start_col = (new_w - w) // 2
    end_row = start_row + h
    end_col = start_col + w

    fourier_spectrum_zero_padding[start_row:end_row, start_col:end_col] = fourier_spectrum

    return np.abs(ifft2(ifftshift(fourier_spectrum_zero_padding))) * (resize_ratio ** 2)
```

Example:

The crew image scaled up with resize_ratio = 4:



We can see now the image contains more pixels than the original and in the same quality

Section c:

This function calculates the ncc (Nomalized Cross Correlation) between an image and a pattern, using sliding windows, mean and variance. This fuction used in “find the pattern in the image problem”.

The following implementation is based on this formula:

$$\frac{\sum_{x,y\in N}\left[I(u+x,v+y)-\bar{I}_{uv}\right]\left[P(x,y)-\bar{P}\right]}{\left[\sum_{x,y\in N}\left[I(u+x,v+y)-\bar{I}_{uv}\right]^2\sum_{x,y\in N}\left[P(x,y)-\bar{P}\right]^2\right]^{1/2}}$$

Where:
P – pattern
I – Image
$\bar{I}$ -image mean

$\bar{P}$ -pattern mean

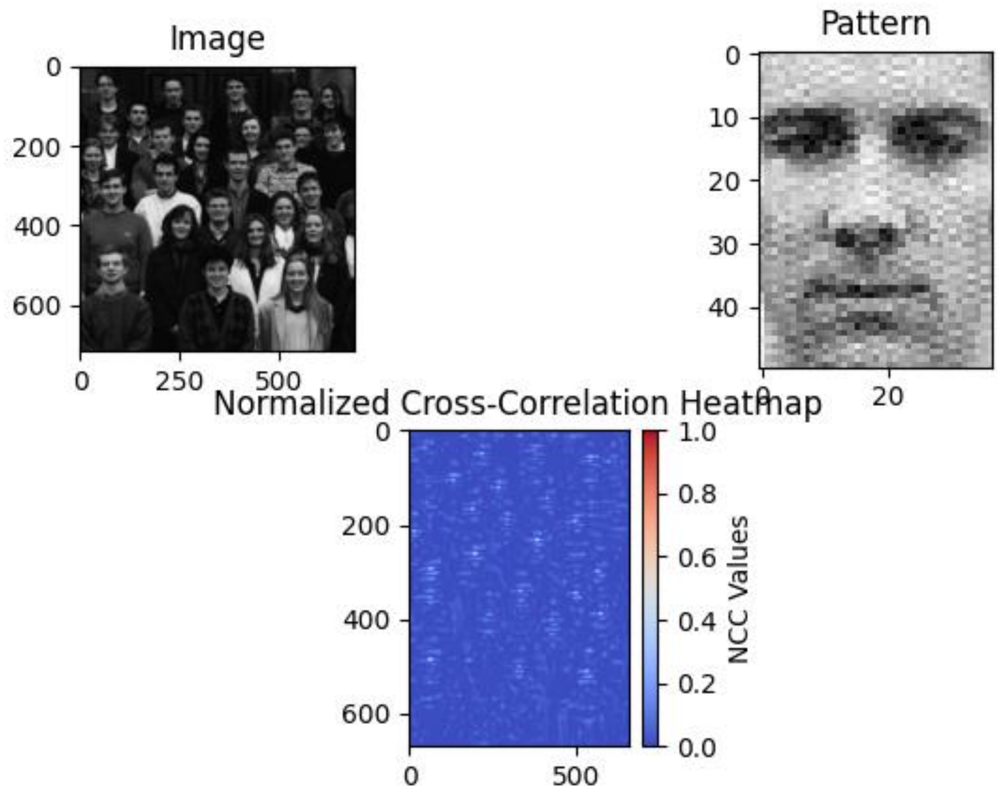
```python
# Section c
def ncc_2d(im, patt):
    windows = np.lib.stride_tricks.sliding_window_view(im, patt.shape)
    ncc_im = np.zeros(windows.shape[:2])
    patt_mean = np.mean(patt)
    patt_var = np.sum((pattern - patt_mean) ** 2)

    # implementing the formula for calculating ncc
    for row in range(len(windows)):
        for col in range(len(windows[0])):
            window_mean = np.mean(windows[row, col])
            window_var = np.sum((windows[row, col] - window_mean) ** 2)
            means_sum = np.sum((windows[row, col] - window_mean) * (patt - patt_mean))
            denominator = np.sqrt(window_var * patt_var)
            if denominator == 0:
                continue
            ncc_im[row, col] = means_sum / denominator

    return ncc_im
```

Section d:

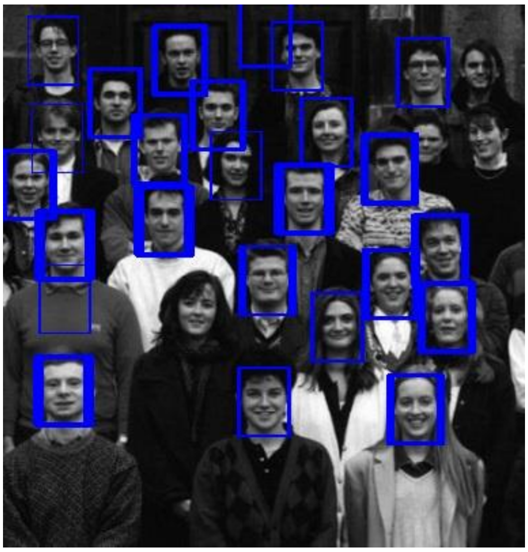
Students image:

Display:

```
image_scale_ratio = 1.8
pattern_scale_ratio = 1
threshold = 0.46
```



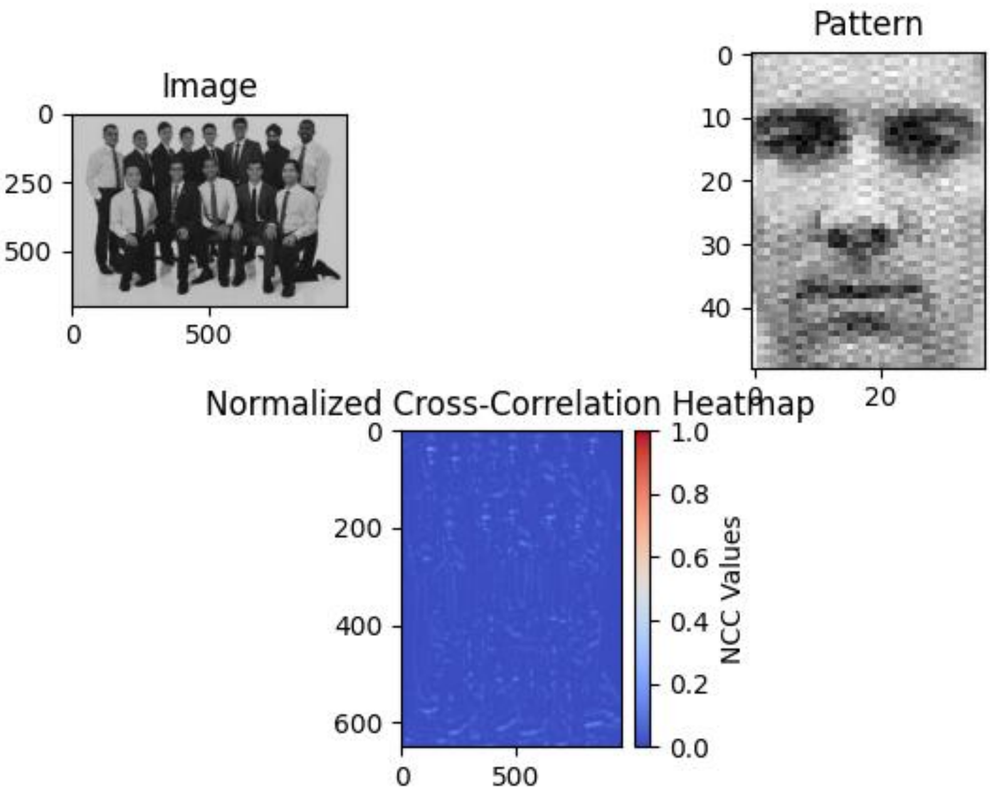
Result with false positives:



Thecrew image:

Display:

```
image_scale_ratio = 4
pattern_scale_ratio = 1
threshold = 0.37
```



Result with false positives:

Section e:

Our best Results

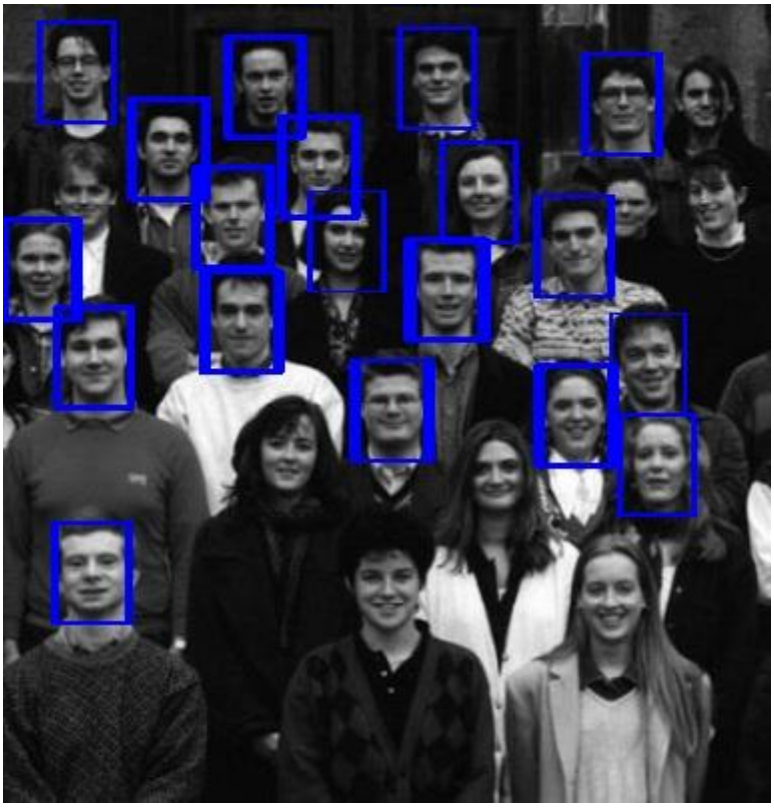
After search and trial and error, here are our best results

Students:

we chose these params:

```
image_scale_ratio = 2
pattern_scale_ratio = 1
threshold = 0.55
```

and the result is:



thecrew:

we chose these params:

```
image_scale_ratio = 4
pattern_scale_ratio = 1
threshold = 0.4
```

Problem 2:

First I want to mention that I implemented these aid functions in addition to the functions required in the question:

```python
def get_gaussian_pyramid(image, levels):
    """
    this function takes an image and returns the gaussain pyramid of the image
    :param image: grey-scale image
    :param levels: the levels of the pyramid
    :return: list contains the layers of the gaussian pyramid of the image
    """
    pyramid = [image]
    current_layer = image
    for _ in range(levels - 1):
        current_layer = cv2.GaussianBlur(current_layer, (7, 7), 0)
        current_layer = current_layer[::2, ::2]
        pyramid.append(current_layer)

    return pyramid
```

And i used scale_down() and scale_up() from Question 1

Section a:

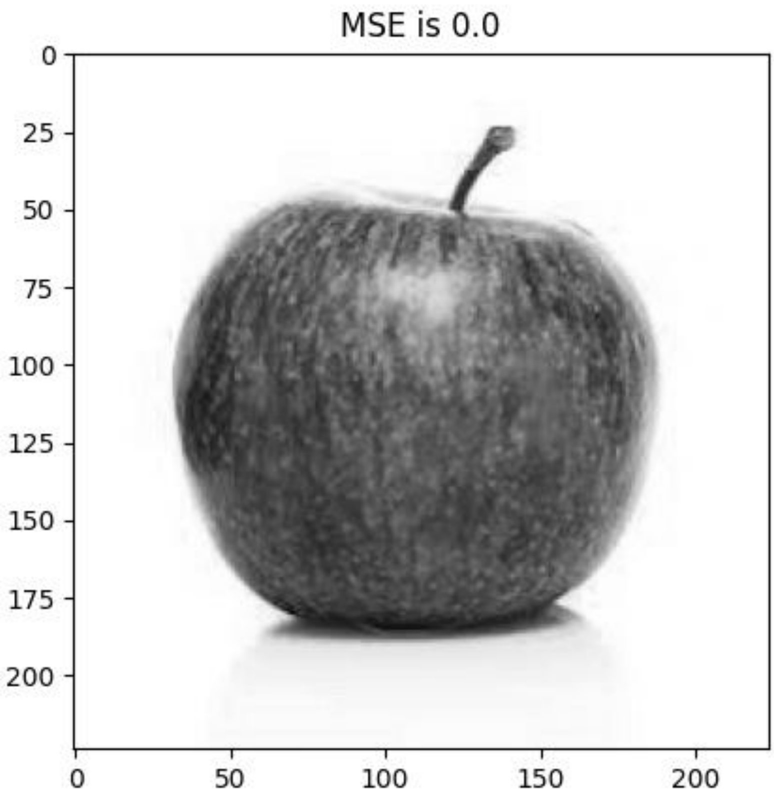
```python
def get_laplacian_pyramid(image, levels):
    """
    this function takes an image and levels, and it returns the laplacian pyramid of the image as a list,
    it does that by applying what we learned in the class, and by using the gaussian pyramid, and a simple formula
    we learned in class
    :param image: the image we want to create the laplacian pyramid for
    :param levels: levels of the pyramid
    :return: the laplacian pyramid of the given image
    """
    gaussian_pyramid = get_gaussian_pyramid(image, levels)
    laplacian_pyramid = []
    # Running on the levels and doing the needed
    for i in range(levels - 1):
        expanded = scale_up(gaussian_pyramid[i + 1], 2)
        laplacian_pyramid.append(gaussian_pyramid[i] - expanded)
    laplacian_pyramid.append(gaussian_pyramid[-1])  # Append the smallest level

    return laplacian_pyramid
```

Section b:

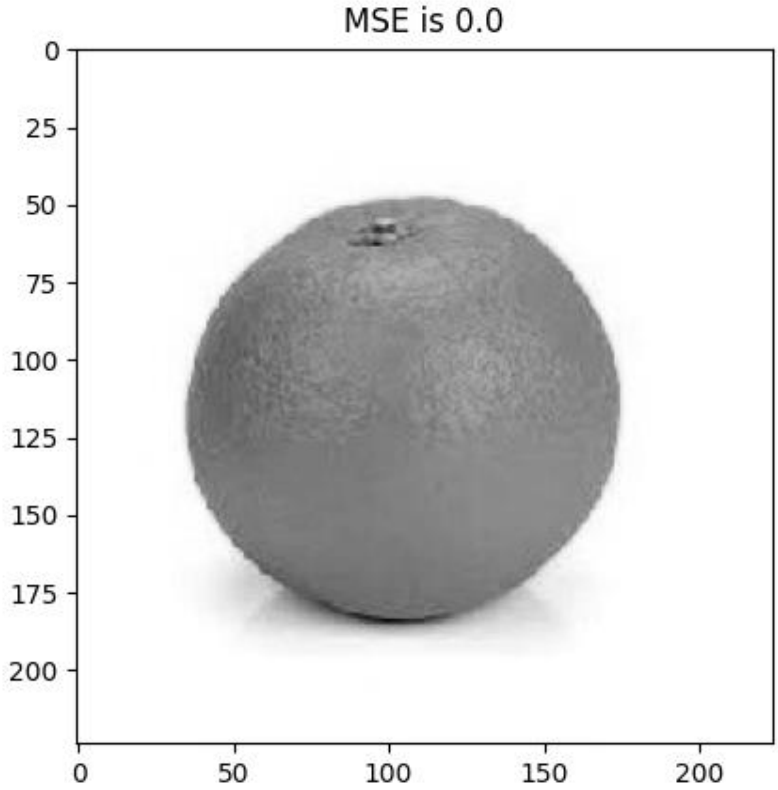
```python
def restore_from_pyramid(pyramidList, resize_ratio=2):
    """
    this function gets a Laplacian pyramid and returns the image by "collapsing" the pyramid as we saw in the class
    :param pyramidList: laplacian pyramid of certain image
    :param resize_ratio: default resize ratio = 2, because each layer is 2x larger than the layer above it
    :return: the image (restored from the laplacian pyramid)
    """
    curr = pyramidList[len(pyramidList) - 1]
    # Running on the layers and "collapsing" the pyramid
    for i in range(len(pyramidList) - 2, -1, -1):
        temp = scale_up(curr, resize_ratio)
        temp += pyramidList[i]
        curr = temp

    return curr
```

By using the given function validate_operation, we created Laplacian pyramids for both orange and apple and we restored it and we got this results:

Apple:



Orange:



Less than 1, so NICE!

Section c:

```python
def blend_pyramids(levels):
    """
    this function uses two images laplacian pyramids in order to blends two images into one image but in smart way,
    it blends each layer in the laplacian pyramid in specific cross dissolve in the middle of the image,
    which give us a smooth transition between the two images
    :param levels: the levels of the pyramids
    :return: laplacian pyramid of one blended image that in left side image1(orange) is dominant ,
    and in left side image2(apple) is dominant with a smooth transition between the two sides in the middle
    """
    blend_pyr = []
    # Running on the levels
    for curr_level in range(levels):
        # creating the mask of the cross-dissolve
        mask = np.zeros(pyr_apple[curr_level].shape)
        width = mask.shape[1]

        # Initialize mask's columns
        mask[:, :int((0.5 * width) - (curr_level + 1))] = 1.0

        # Applying the given cross-dissolve formula
        for i in range(2 * (curr_level + 1)):
            mask[:, (width // 2) - (curr_level + 1) + i] = 0.9 - 0.9 * i / (2 * (curr_level + 1))

        # Adding the layer to the blended image laplacian pyramid
        blend_pyr.append((pyr_orange[curr_level] * mask) + (pyr_apple[curr_level] * (1 - mask)))

    return blend_pyr
```

Section d:

We created Laplacian pyramids for both images Apple and Orange, and then we blended the two images using our blend function:

```
# Creating laplacian pyramids for both images Orange and Apple
pyr_apple = get_laplacian_pyramid(apple, levels)
pyr_orange = get_laplacian_pyramid(orange, levels)

# Blending Pyramids
pyr_result = blend_pyramids(levels)

# Getting and plotting the blended image
final = restore_from_pyramid(pyr_result)
plt.imshow(final, cmap='gray')
plt.show()
```

But before that we needed to choose the best level value that gives the best image we can get from blending the two images, after trying all the possible levels, we found out that level=3 is the perfect match in the trade-off between the smoothens in the cross-dissolve and between the “showing” that this is blended image (there is dominant part in each side has its own colors)

Level = 1:

Level = 2:

Level = 3: our best result

Level = 4:

Level = 5

in levels = 1\ levels = 2, we clearly can see the line in the middle that separates the two images, the transition between the images is not smooth and it is not good.

While in levels = 4\ levels = 5, we can see also the fainting in the colors of each side, which is also not good.

And finally in levels=3 we find the best results, we have a good blending and also we have great colors for each side.

Bonus Question:

Section a:

I completed this section in the code file in many different places

Section b:

I used the following canny implementation to find edges in the image:

```
min_edge_threshold, max_edge_threshold = 100, 200
edge_image = cv2.Canny(edge_image, min_edge_threshold,
                       max_edge_threshold)  # Apply edge detector with
min_edge_threshold, max_edge_threshold
```

Section c:

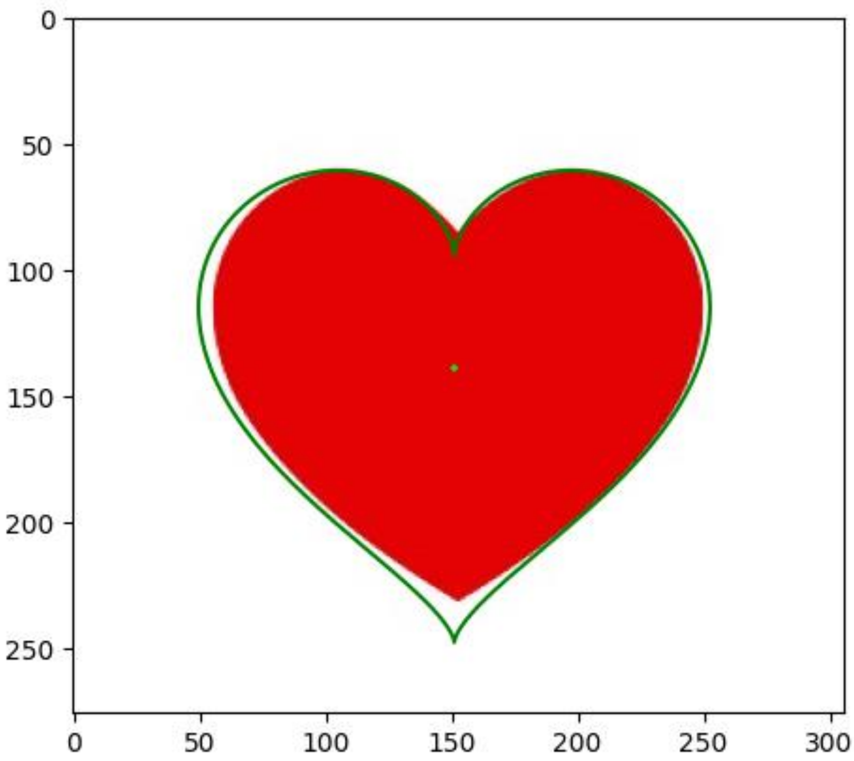
After many trail and error I found these parameters that led me to the best result I had:

simple:

Parmeters:

```
r_min = 6.5
r_max = 7.5
bin_threshold = 0.18
```

Result

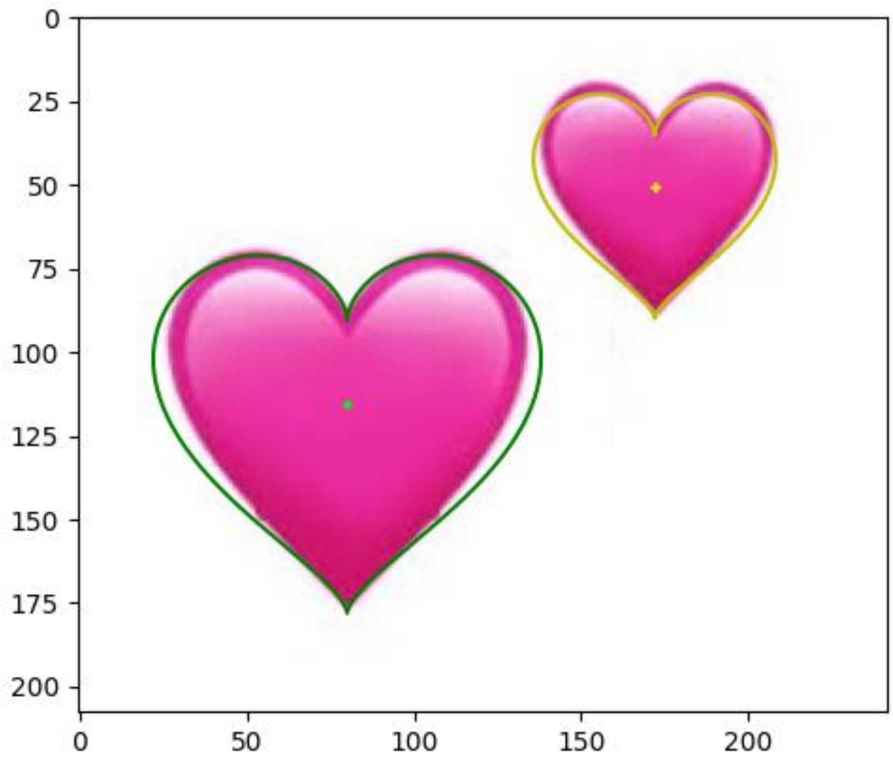


med:

Parameters:

```
r_min = 2
r_max = 5
bin_threshold = 0.3
```

Result:

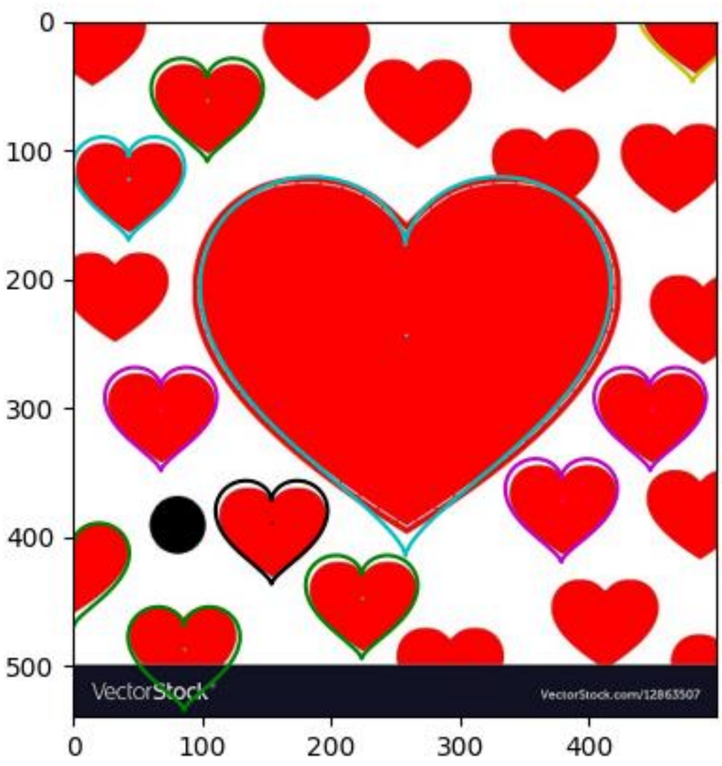


hard:

Parameters:

```
r_min = 3
r_max = 12
bin_threshold = 0.26
```

Result:



We can see that we succeeded to find all the hearts in simple and med image, but we have a problem to do that in the hard image, and that is because we can see that in simple and med image we only have hearts and few of them (simple contains 1 and med contains 2) while in the hard image we not only have a much more hearts but also we have also different shapes in the image itself, like the black circle and the black rectangle below, these factors makes it hard to detect all the hearts in the image.