Image Processing – HW5

Submitters :
Maya Atwan , ID:314813494
Obaida Khateeb , ID: 201278066

Problem 1 – Template matching

a) Implementation of the function 'scale_down':

We start the implementation by blurring the image to prevent aliasing, as we learned in the class we must blur the image to reduce high frequency that can cause artifacts before scaling down. After that we make sure the image in grayscale and applied the fourier transform to the image and shift the dc to the center of the spectrum. After this preprocessing we crop the frequency domain to the desired size. Last we apply the inverse fourier transform to reconstruct the image in the spatial domain(normalize the image to unit8 before).

b) Implementation of the function 'scale_up':

The implemetion is similar to 'scale_down' but instead of cropping the frequency domain we pad it with zeros to increase the size of the image. Zero padding the image → scale up. (here we don't blurring and all the implementation is the same).

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

We used this formula from the lecture:

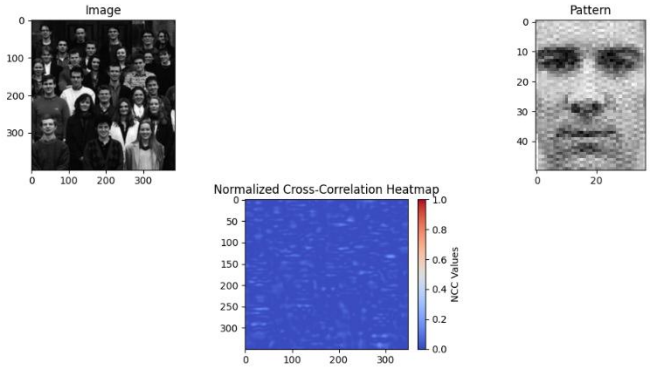
Normalized Correlation

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

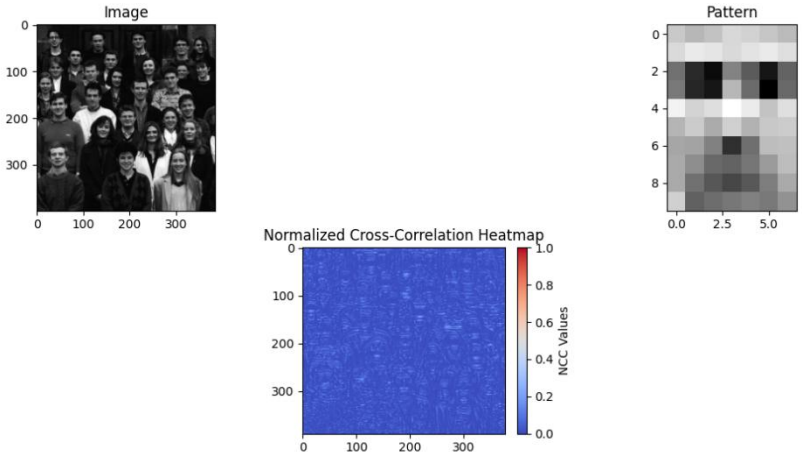
The function moves a window over the image and compares it to the pattern , it calculates the mean for the pattern and window then

computes the NCC value. We used a double for loop to go through the image and check each window. We take care of the case ' If the denominator is 0'.

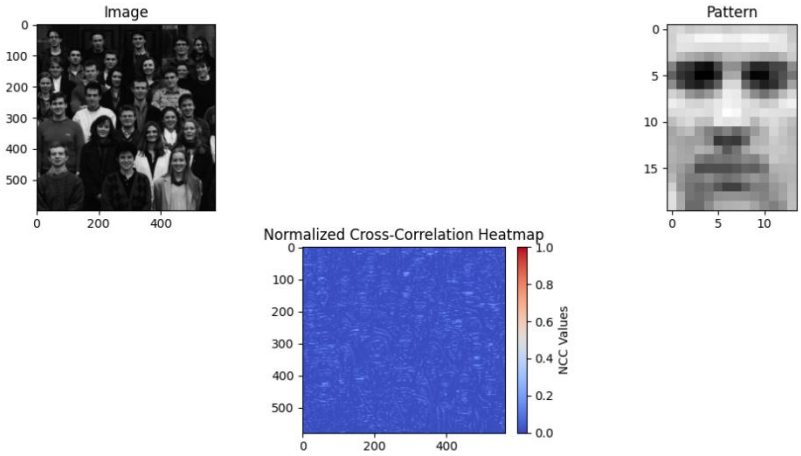
d) We start by NCC heatmap without any scaling to understand better the images:



After that we tried scaling down the pattern 0.2 and the image without any change we got:

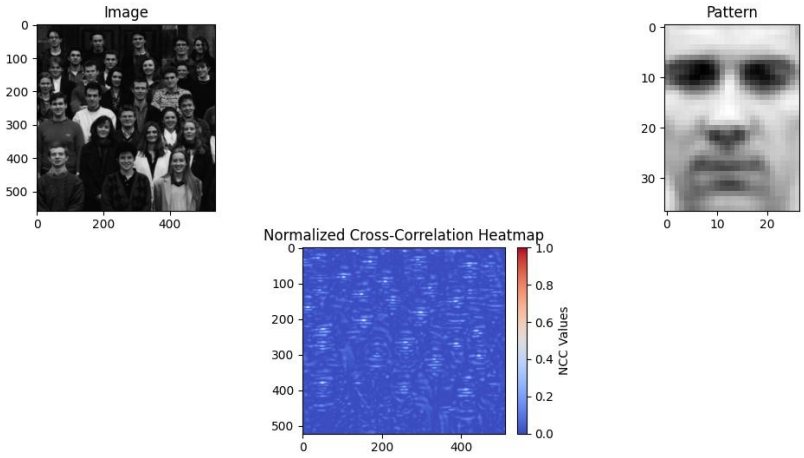


Not a good result so we tried scaling up the image 1.5 and scaling down the pattern but with higher value 0.4 :



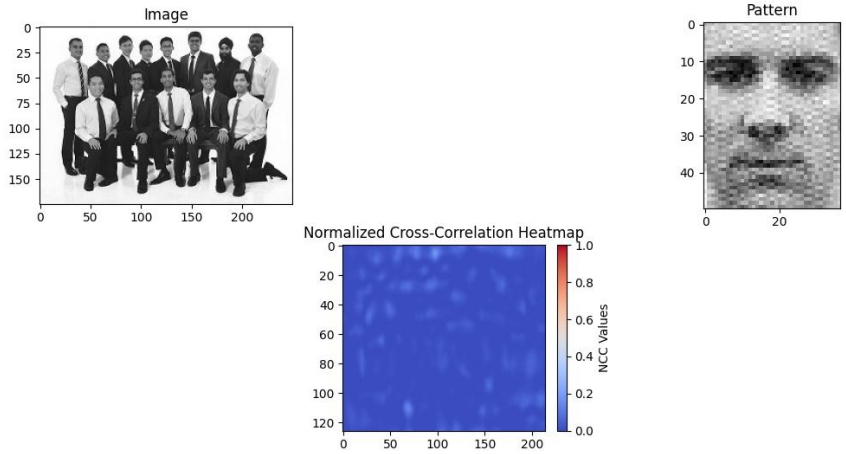
A better we can see a little peaks, so we are in the right crack.

Scale down the pattern 0.75 and scale up the image 1.4:

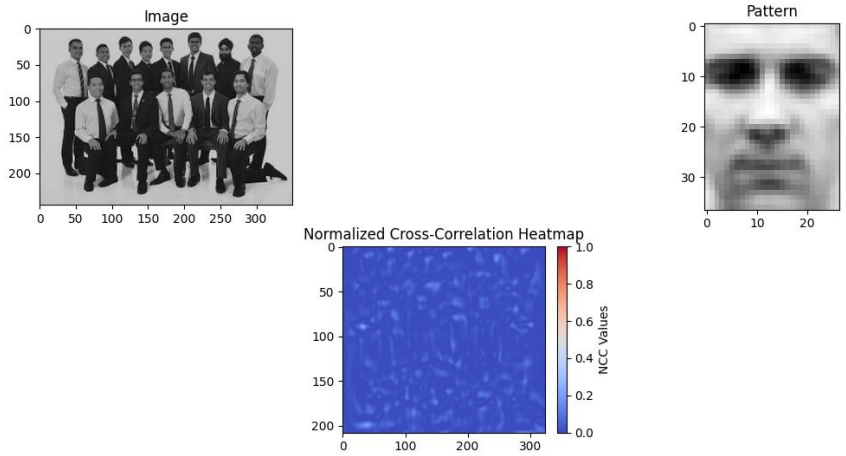


We can see the white peaks where are faces in the image.

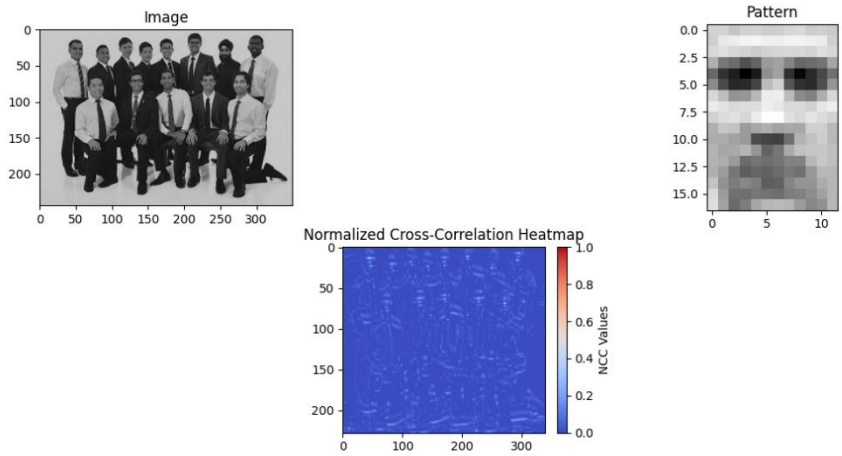
For the second image the crew: we start with no scaling



Then we tried the same as the students image scale up the image 1.4 and scale down the pattern 0.75:



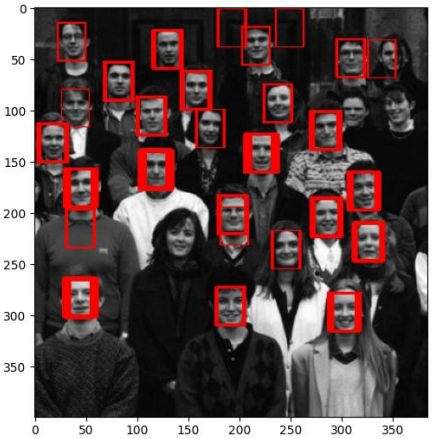
A little better but not that good, so we tried scale up the image with the same as before 1.4 and scale down the pattern to 0.35 and we got:



We can see the white peaks where is the faces.

e) After we now that scale up the image and down the pattern we tried different values for the scale up image , scale down pattern and the threshold.

We got a duplicated rect over the same face like this :

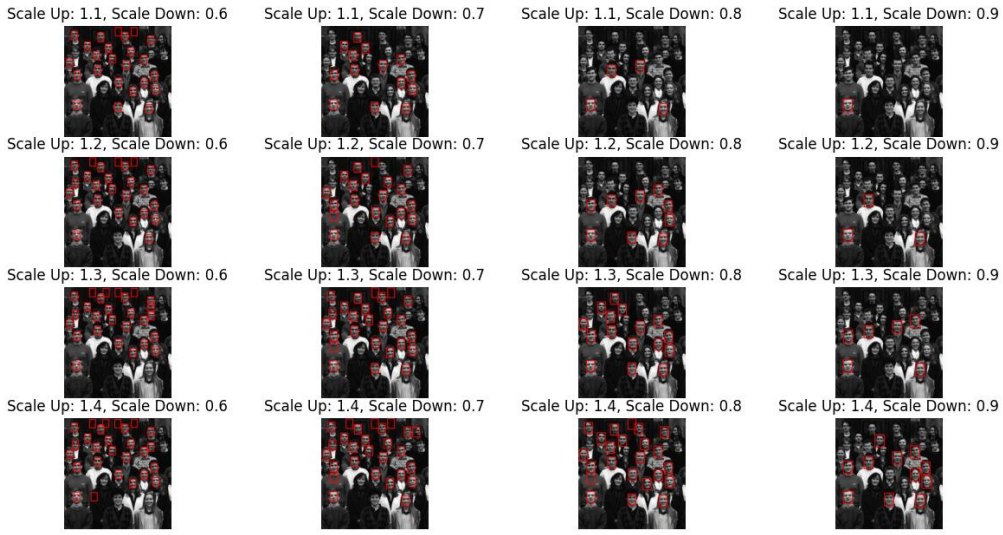


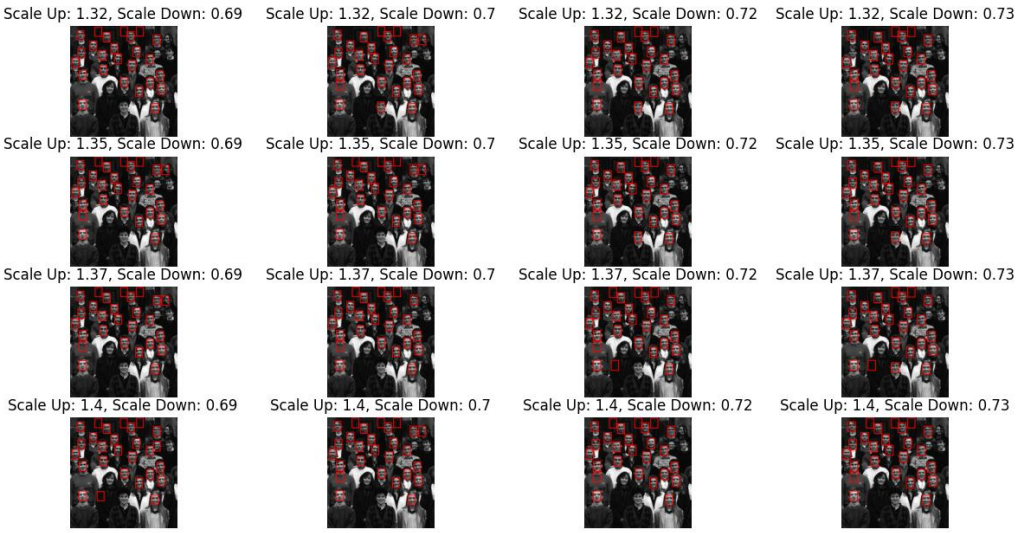
To prevent it we add this line to the code:

' ncc[ncc!= maximum_filter(ncc, size=(40,20))]=0 ' by using ' from scipy.ndimage import maximum_filter'

Choosing threshold for students image:

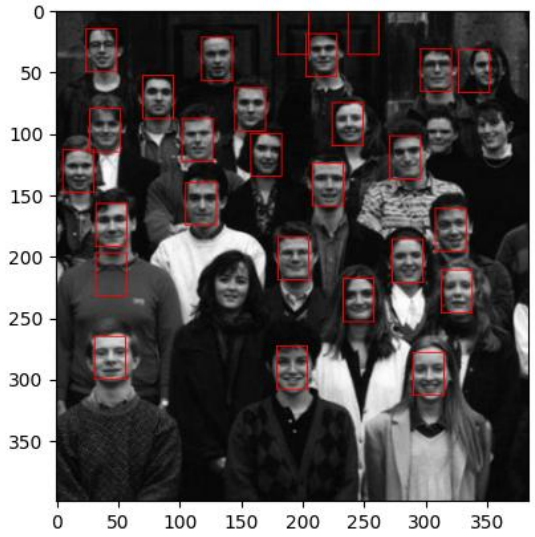
From the result we choose 0.52. and now we tried values for the scale up and scale down:





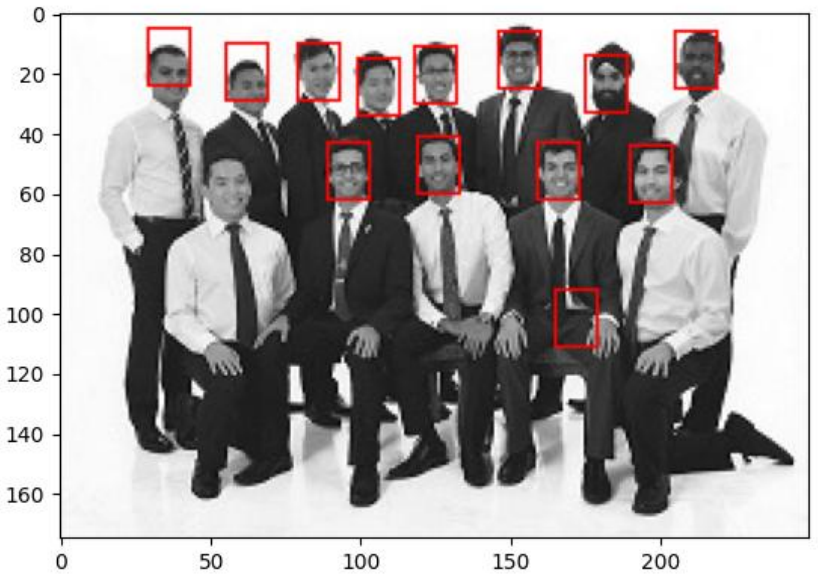
From the result scale up the image 1.32 and scale down the pattern 0.7 got the best detection.

Final result for the students image:



Choosing threshold:

From the result threshold 0.45 is the best so we choose 0.452.
Final result for the crew image:
Scale up the image 1.66 and scale down the pattern 0.38



Problem 2 - Multiband blending

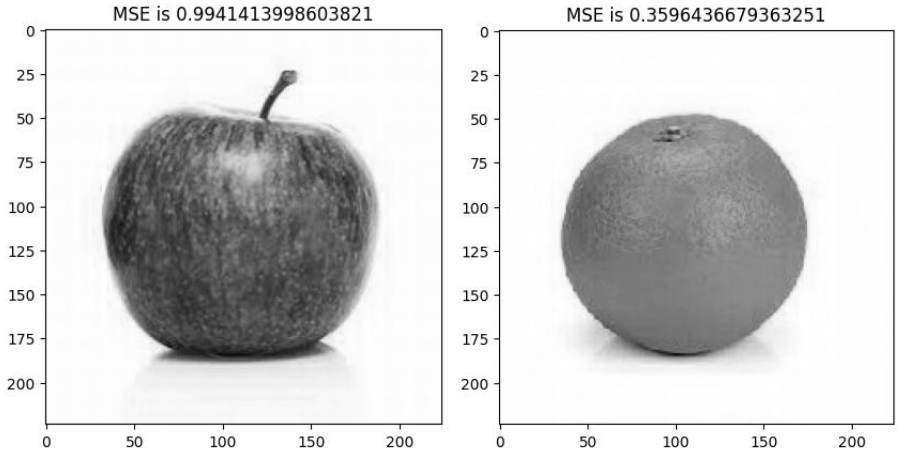
The function 'get_laplacian_pyramid' implemented such that it splits the image into multiple Laplacian layers and returns a list of these layers. A layer is created by subtracting a gaussian blurred version of the image from the original one. After each subtraction, the image is replaced by the blurred one and get downsampled by a given ratio. Finally, the remaining image stored as the last layer.

```
def get_laplacian_pyramid(image, levels, resize_ratio=0.5):
    laplacian_pyr = []
    image = image = image.astype(np.float32)
    for i in range(levels - 1):
        blurred_img = cv2.GaussianBlur(image, (27,27), 0)
        layer = cv2.subtract(image, blurred_img)
        laplacian_pyr.append(layer)
        new_width = max(1, int(image.shape[1] * resize_ratio))
        new_height = max(1, int(image.shape[0] * resize_ratio))
        image = cv2.resize(blurred_img, (new_width, new_height),
                           interpolation=cv2.INTER_CUBIC)
    laplacian_pyr.append(image)
    return laplacian_pyr
```

The function 'restore_from_pyramid' implemented such that it do the inverse operation. It gets a Laplacian pyramid and builds an image out of it. The function initialize the image as the highest layer of the pyramid (the last element of the given list) and iteratively adds the layers from the highest to the lowest, while resizing the image each time to match the next Laplacian layer. The resizing ratio (the difference in size between each two consecutive layers) is given to the function as a parameter.

```
def restore_from_pyramid(pyramidList, resize_ratio=2):
    image = pyramidList[-1]
    for i in range(len(pyramidList) - 2, -1, -1):
        image = cv2.resize(image, (pyramidList[i].shape[1],
                pyramidList[i].shape[0]), interpolation=cv2.INTER_CUBIC)
        image = cv2.add(image, pyramidList[i])
    return image
```

Both functions applied to the two provided images, and the results were close to the original with $MSE < 1$:



The function 'blend_pyramids' was also implemented. It aims to blend the two provided images in a smooth nice looking way, using their Laplacian pyramids. The function loops through the corresponding layers of the two pyramids, and applies a cosine function to define the gradual transition. The size of the transition window is set to 1/7 of the constructed image width. Finally the blending is performed using a cross-dissolve operation with the generated mask.

```
def blend_pyramids(levels):
    global pyr_apple, pyr_orange
    blended_pyr = []
    for apple_layer, orange_layer in zip(pyr_apple, pyr_orange):
        rows, cols = apple_layer.shape
        mask = np.zeros((rows, cols), dtype=np.float32)
        mask[:, :cols // 2] = 1
        blend_width = cols // 7
        for j in range(cols // 2 - blend_width, cols // 2 + blend_width):
            if 0 <= j < cols:
                mask[:, j] = 0.5 + 0.5 * np.cos(np.pi * (j - (cols // 2
                             - blend_width)) / (2 * blend_width))
        blended_layer = mask * orange_layer + (1 - mask) * apple_layer
        blended_pyr.append(blended_layer)
    return blended_pyr
```

The function was tested on the two provided image using both the method mentioned in the PDF and our implemented one, and we preferred the last. In addition, we experimented with different transition window sizes, aiming to

achieve the best transition in terms of smooth transition, uniform background, natural-looking shadow, and minimizing noticeable artifacts, especially around the stem of the apple. Here's the result:

Below are some of the less successful results received:

- The result obtained with the too small window size of 1/10:

- The result obtained using the method mentioned in the PDF:

Problem 3 - Hough transform

The two functions 'parametric_x' and 'parametric_y' were implemented based on the heart's parametric equation mentioned in the instructions document:

```
def parametric_x(t):
    return 14.5 * np.sin(t)** 3

def parametric_y(t):
    return 0.5 * np.cos(4*t) +2 * np.cos(3*t) +
        4* np.cos(2*t) - 13* np.cos(t)
```

The following code was modified so it now generates all possible heart shape candidates using different values of r and θ:

```
shape_candidates = []
for r in rs:
    for theta in thetas:
        shape_candidates.append((r, theta))
```

The implementation of the following piece of code which iterate over all edge points and candidate hearts computing the center of each possible heart was completed. The code in addition ensures that the detected centers are within the image boundaries:

```
edge_points = np.argwhere(edge_image> 0)
for point in edge_points:
    y, x = point
    for r, theta in shape_candidates:
        x_center = int(x - r * sin_thetas[theta // 2])
        y_center = int(y - r * cos_thetas[theta // 2])

        if x_center >= 0 and x_center < img_width
            and y_center >= 0 and y_center < img_height:
            accumulator[(x_center, y_center, r)] += 1
```

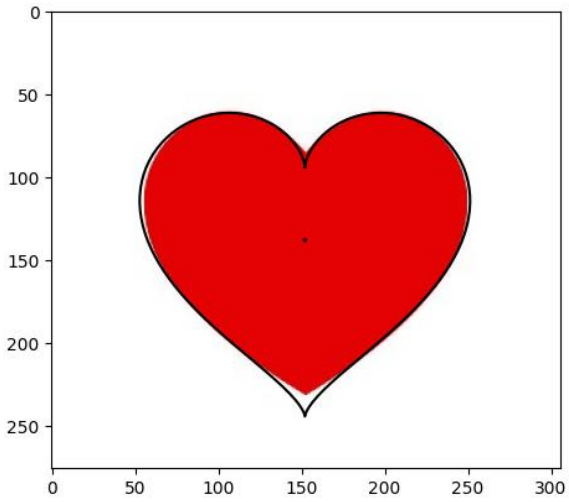
The following code filters weak detection was implemented:

```
if current_vote_percentage > bin_threshold:
    out_shapes.append((x, y, r, current_vote_percentage))
```

Below are the detection output for each image along with the parameter values which provided the best results:

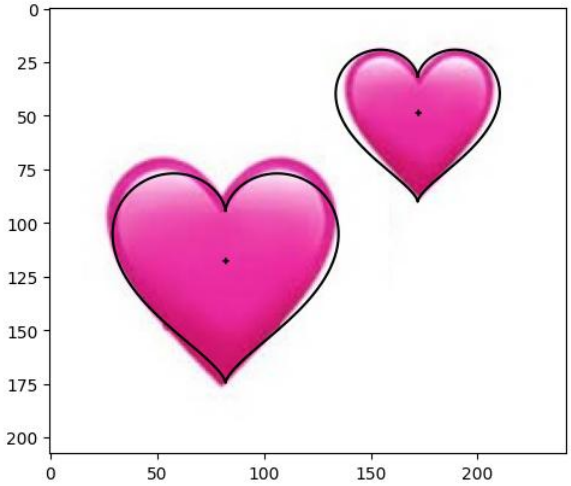
Note: the bin threshold is given as a range of values that all produce the same result. If a single value had to be chosen, we would the midpoint of the range, which balances between avoiding false detections and succussing in detecting hearts when edges are weak.

- Simple:



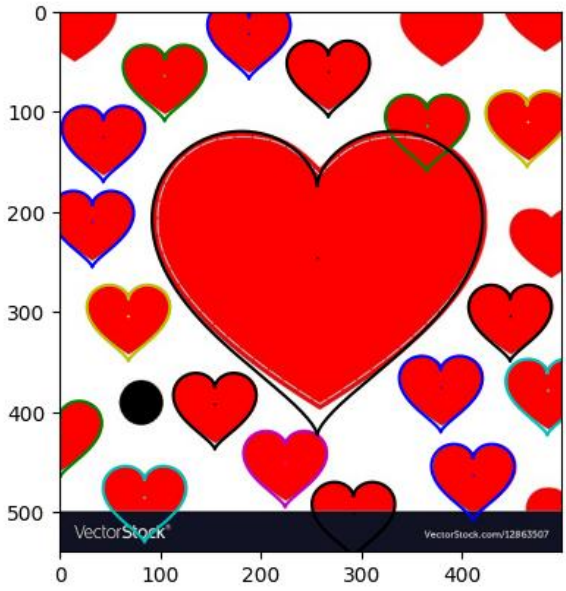
$r\_\min = 6.85\,, r\_\max = 6.95\,, bin_{threshold} = 0.13 - 0.31$

- Med:



$r\_\min = 2.65\,, r\_\max = 3.7\,, bin_{threshold} = 0.23 - 0.3$

- Hard:



$r\_\min = 2.85\,, r\_\max = 11.4\,, bin_{threshold} = 0.24 - 0.28$