Image Processing – HW5

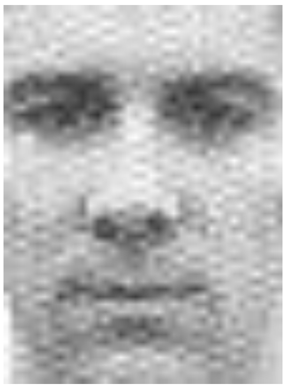
Problem 1 – Match Faces

The target is to locate as many faces as possible on two separate images using a given pattern

Students image

Crew image

Pattern



Our strategy was to search for faces with multiple pattern sizes. Meaning, we've scaled down the pattern and defined a threshold for correlation results for each size. This allowed us to locate faces of different size. We did not change the image size as we've preferred to keep the resolution, at the expense of having a longer testing cycle

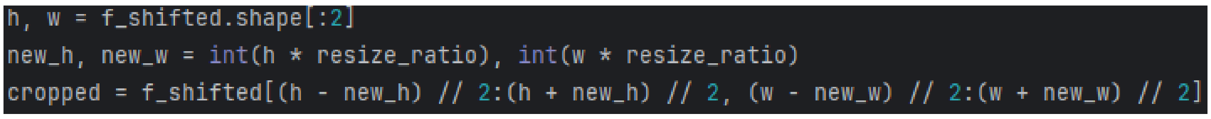
Used Methods

Method: scale_down

Input: the image to be scaled (np array), resize ratio which is between 0 and 1

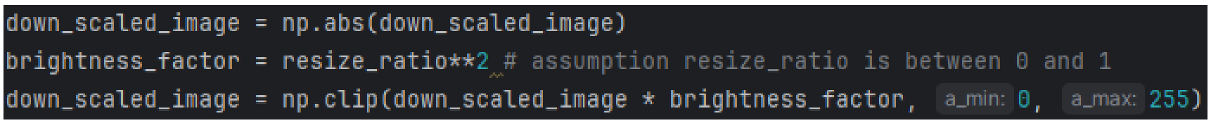
Output: a down scaled image in as np array

We've used FFT to scale down the image. Once the image was transformed and shifted, we've calculated by how much we need to down size it, and cropped the center, deleting the high frequency area



```
h, w = f_shifted.shape[:2]
new_h, new_w = int(h * resize_ratio), int(w * resize_ratio)
cropped = f_shifted[(h - new_h) // 2:(h + new_h) // 2, (w - new_w) // 2:(w + new_w) // 2]
```

Before returning the image we've also adjusted the brightness since we've removed some of the frequencies



```
down_scaled_image = np.abs(down_scaled_image)
brightness_factor = resize_ratio**2 # assumption resize_ratio is between 0 and 1
down_scaled_image = np.clip(down_scaled_image * brightness_factor, a_min: 0, a_max: 255)
```

Method: scale_up

Input: the image to be scaled (np array), resize ratio which is 1 or higher

Output: an up scaled image in as np array

Very similar to scale down method, and the Zebra question in HW4. We've increased the image in the frequency domain by padding it with zeroes

```
scaled_f_shifted = np.zeros( shape: (new_h, new_w), dtype=complex)

# Calculate the center position and copy the original image to the center of the padded array
start_h, start_w = (new_h - h) // 2, (new_w - w) // 2
scaled_f_shifted[start_h:start_h + h, start_w:start_w + w] = f_shifted
```

Before returning it we've following the same brightness method described earlier

```
scaled_image = np.abs(scaled_image)
brightness_factor = resize_ratio**2 # assumption resize_ratio is at least 1
scaled_image = np.clip(scaled_image * brightness_factor,  a_min: 0,  a_max: 255)
```

Method: ncc_2d

Input: the image to checked for patters (np array), a pattern image (np array)

Output: array containing the correlation results of each sliding window with the pattern

This method calculates the NCC equation below

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

We use sliding window technique to extract all possible windows of the given pattern

```
windows = np.lib.stride_tricks.sliding_window_view(image, pattern.shape)
```

To avoid division by zero we validate and set the denominator to one

```
denominator = np.where(denominator == 0, 1, denominator)
```

Method: check_pattern

Input: tuple of scale ratios and thresholds, the image to checked for patters (np array), a pattern image (np array)

Output: list of points which are the center x, y coordinates of a matched face + the pattern size used to detect it

This method was adjusted from what was given to us. On top of calculating the NCC and keeping all matches above a defined threshold. We run a loop on all pattern size and threshold we've predefined, and keep all matches found

```
for match in current_matches:
    matches.append((match, pattern_scaled.shape)) # keep all matches and its respected pattern size
```

Example on how this method is called

```
scale_and_threshold = [(0.6, 0.5), (0.5, 0.55), (0.45, 0.6), (0.4, 0.6)]

# Keep array of all matches + the pattern size they were found
matches = check_pattern(scale_and_threshold, image, pattern)
```

Method: remove_duplicates

Input: sorted list of points which are the center x, y coordinates of a matched face + the pattern size used to detect it

Output: same list without duplicate coordinates, where duplicate means either (i) center coordinate closer than 10 pixels away or (ii) same center coordinate which was in more than one pattern size

The list which we pass to this method is sorted

```
matches.sort(key=lambda item: (-np.sqrt(item[1][0]**2 + item[1][1]**2), item[0][0]))
```

For each center point of a face we check if we already have it in our unique array, or if it we have another center point up to 10 pixels away. The 10 pixels was a value we've adjusted for good filtering of overlapping detections. Since the largest pattern size is sorted first, we'll be able to see what was the first window the face was detected

```
for item in matches:
    point = item[0]  # get the x,y coordinates from the numpy array
    # check if the center point is closer than 10 pixels, or if i've already saved the same center point of the face
    if not any(are_close(point, seen) for seen in seen_points):
        unique.append(item)
        seen_points.append(point)
```

Method: are_close

Input: two tuples of x, y coordinates

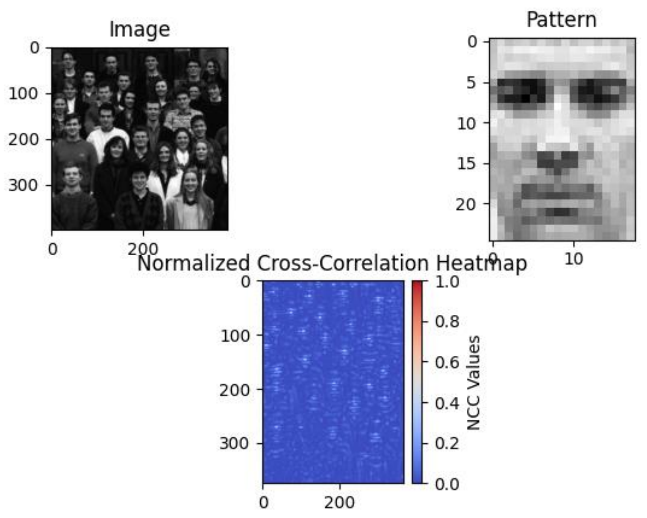
Output: Boolean, true if the coordinates are below 10 pixels distance. False otherwise

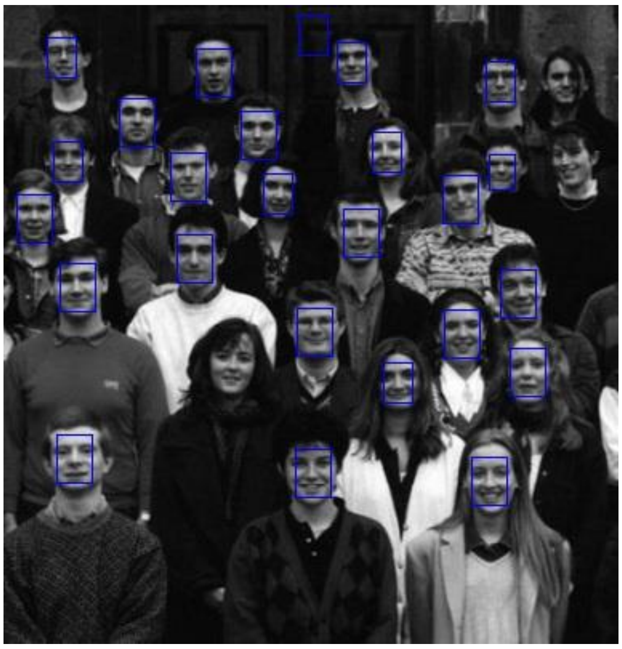
```
return np.abs(point1[0] - point2[0]) <= 10 and np.abs(point1[1] - point2[1]) <= 10
```

<u>Results</u>

Students image:

We were able to locate all but three faces, having one false detection. Reducing the pattern size by half resulted with the most detected faces. **Trying to locate additional faces resulted with more false positives than added true positives**.

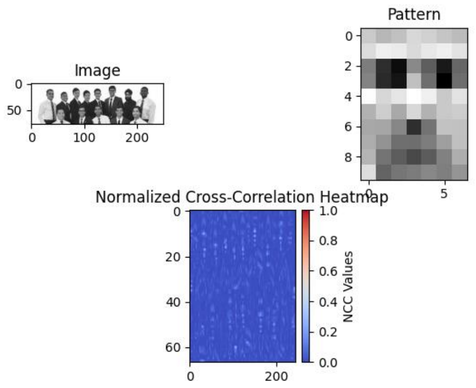


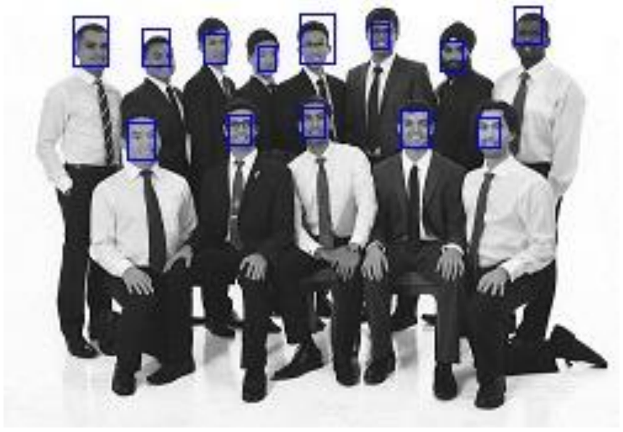


Crew image:

We were able to locate all faces. Though, to help with run time and false detection, we've passed only part of the picture and left the bottom part (which has no faces) out of the search area. Scaling by 80% found the most faces, although some were located prior to that

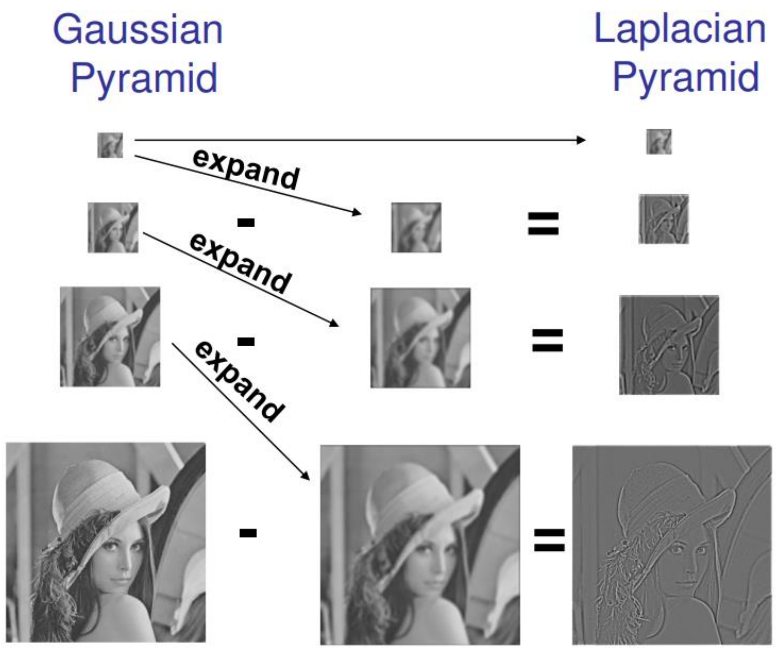
```
matches = check_pattern(scale_and_threshold, image[:76, :], pattern)
```





Problem 2 – Blending Fruits

We were given two images, an apple and an orange. The target is to blend both images so the right half will be apple and the left half will be apple. To do this we've built a Laplacian pyramid of each image (both containing the same number of levels) and blended them together using a mask. Finally, the reconstructed image from the blended pyramid resulted in a blended image. The system to build the pyramid was to blur the image with an optimal gaussian filter and subtract it from the original image. To downsize the blurred image, we've followed the instructions from the lecture of taking every second pixel thus reducing the image by half



Used Methods:

Method: get_laplacian_pyramid

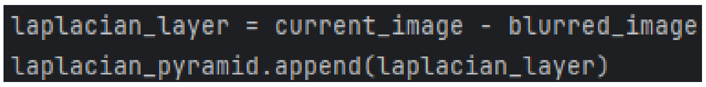
Input: image (np array), number of levels to create (int)

Output: laplacian pyramid (array of images)

As presented above, we first blur the image (apply_gaussian_blur is explained later)

```
blurred_image = apply_gaussian_blur(current_image)
```

Subtract and save the result as Laplacian level

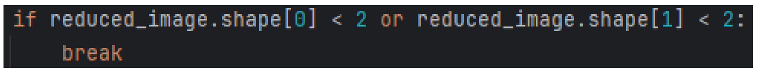


```
laplacian_layer = current_image - blurred_image
laplacian_pyramid.append(laplacian_layer)
```

Reduce by taking every second pixel

```
reduced_image = blurred_image[::2, ::2]
```

In case the image is too small to dowscale we exit the loop



```
if reduced_image.shape[0] < 2 or reduced_image.shape[1] < 2:
    break
```

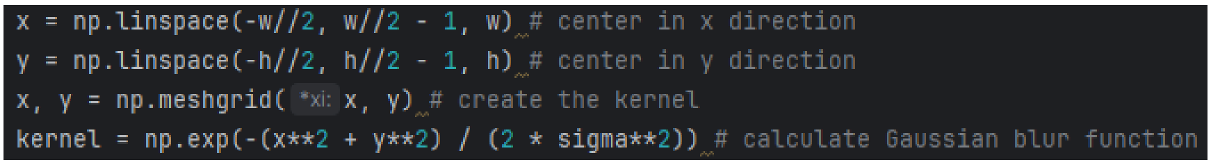
Using this code yielded **MSE of 0.0004** on the apple and **0.0005** on the orange

Method: create_gaussian_kernel

Input: shape (tuple), sigma (float)

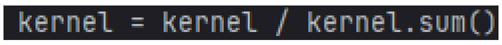
Output: a normalized Gaussian Kernal in the frequency domain

We prepare a kernel or the image size



```
x = np.linspace(-w//2, w//2 - 1, w) # center in x direction
y = np.linspace(-h//2, h//2 - 1, h) # center in y direction
x, y = np.meshgrid( *xi: x, y) # create the kernel
kernel = np.exp(-(x**2 + y**2) / (2 * sigma**2)) # calculate Gaussian blur function
```

And normalize it



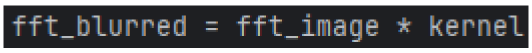
```
kernel = kernel / kernel.sum()
```

Method: apply_gaussian_blur

Input: image (np array), sigma (float)

Output: a blurred image in the spatial domain (np array)

The Gaussian blur is applied in the frequency domain, which means we could multiply



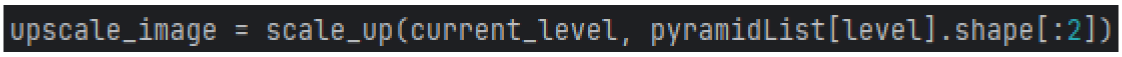
```
fft_blurred = fft_image * kernel
```

Method: restore_from_pyramid

Input: laplacian pyramid (array of images)

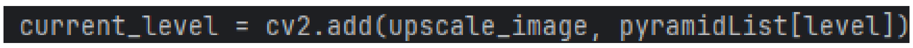
Output: restored image (np array)

We iterate from the top layer to bottom, expand the layer using scale_up method from question 1



```
upscale_image = scale_up(current_level, pyramidList[level].shape[:2])
```

And add it to the next layer



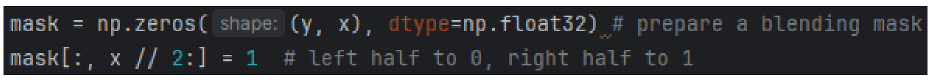
```
current_level = cv2.add(upscale_image, pyramidList[level])
```

Method: blend_pyramids

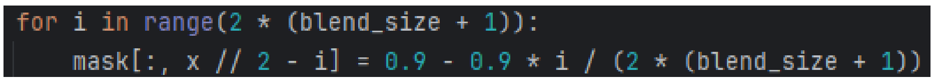
Input: two laplacian pyramids (array of images)

Output: blended laplacian pyramid (array of images)

The first to blend the pyramid are to make a mask where half is apple and half is orange with a predefined area for the transition (a gradual change from zero to one)

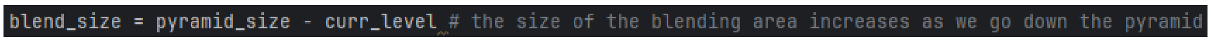


```
mask = np.zeros( shape: (y, x), dtype=np.float32) # prepare a blending mask
mask[:, x // 2:] = 1  # left half to 0, right half to 1
```



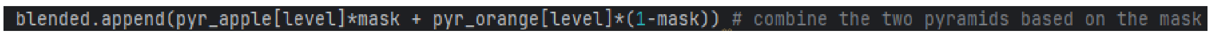
```
for i in range(2 * (blend_size + 1)):
    mask[:, x // 2 - i] = 0.9 - 0.9 * i / (2 * (blend_size + 1))
```

The blend size increases as we go down the pyramid



```
blend_size = pyramid_size - curr_level # the size of the blending area increases as we go down the pyramid
```

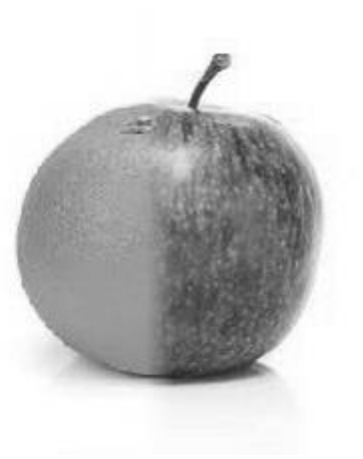
We then combine the two pyramids using the mask



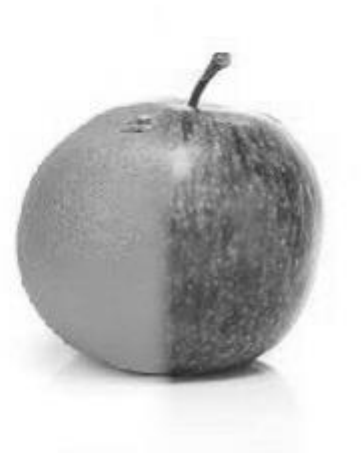
```
blended.append(pyr_apple[level]*mask + pyr_orange[level]*(1-mask)) # combine the two pyramids based on the mask
```

<u>Results</u>

The suggested algorithm for blending yielded the below result

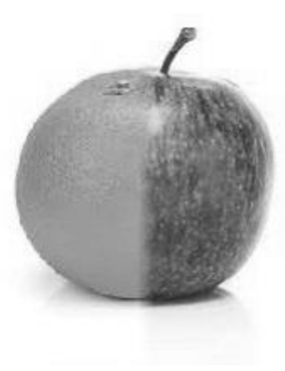


We’ve modified this blend in two ways. The first is to make the blend exponential rather than linear. The blend width is smaller, but the transition looks smoother

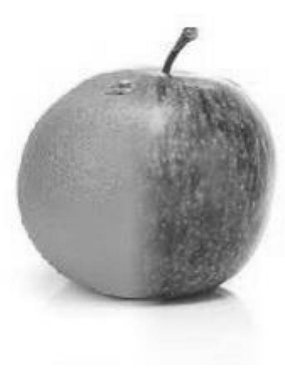


Another thing is we’ve tried different widths for the transition – we believe the 10% transition width is the overall best

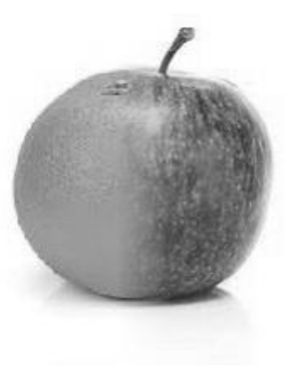
5% transition width



10% transition width



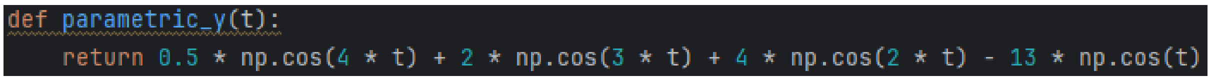
15% transition width



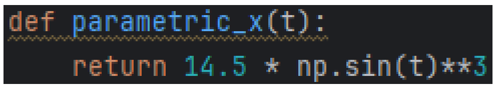
Bonus – Hough Transfrom

We've implemented the skeleton code given by Alon

Parametric_x and y were the heart shape x and y geometrical equation

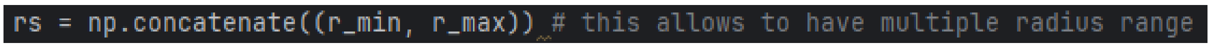


```
def parametric_y(t):
    return 0.5 * np.cos(4 * t) + 2 * np.cos(3 * t) + 4 * np.cos(2 * t) - 13 * np.cos(t)
```



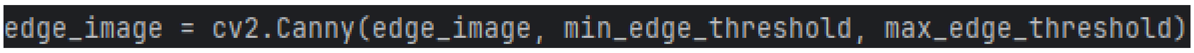
```
def parametric_x(t):
    return 14.5 * np.sin(t)**3
```

To test multiple radius ranges (need for the hard image) we've added two ranges instead of r_min to r_max. Now both contain range and not a single value



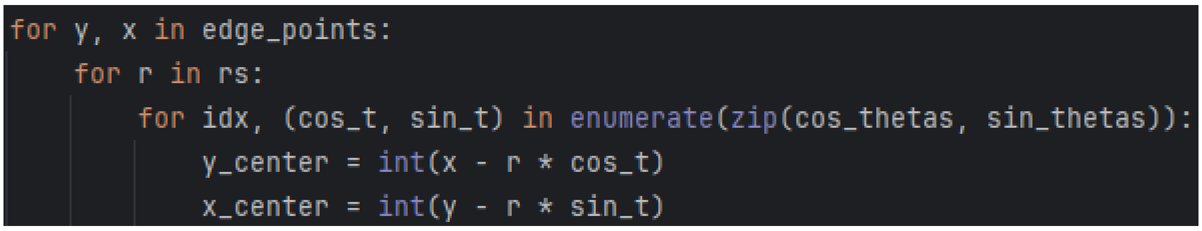
```
rs = np.concatenate((r_min, r_max)) # this allows to have multiple radius range
```

After using Canny edge detection



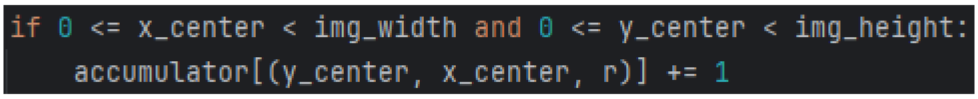
```
edge_image = cv2.Canny(edge_image, min_edge_threshold, max_edge_threshold)
```

We ran a loop on all those points and checked for center heart candidates by subtracting all possible radius and thetas from that edge



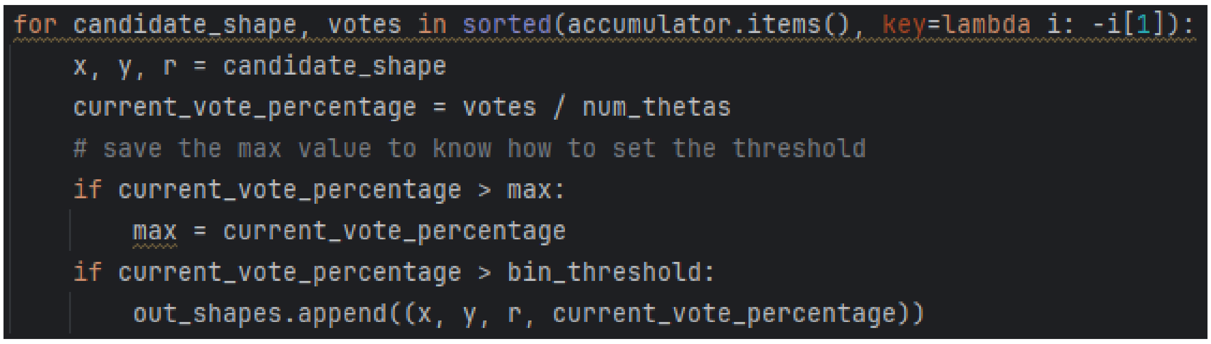
```
for y, x in edge_points:
    for r in rs:
        for idx, (cos_t, sin_t) in enumerate(zip(cos_thetas, sin_thetas)):
            y_center = int(x - r * cos_t)
            x_center = int(y - r * sin_t)
```

To help with false points, we've also added a check that the center is not out of the image



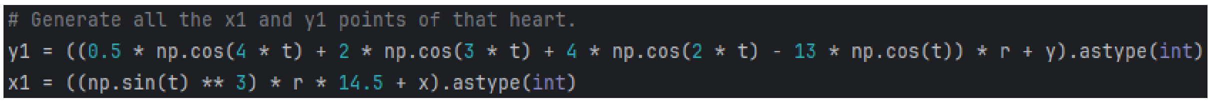
```
if 0 <= x_center < img_width and 0 <= y_center < img_height:
    accumulator[(y_center, x_center, r)] += 1
```

Next was to vote for all center candidates and keep the max current_vote_percentage to help us determine which threshold to choose



```
for candidate_shape, votes in sorted(accumulator.items(), key=lambda i: -i[1]):
    x, y, r = candidate_shape
    current_vote_percentage = votes / num_thetas
    # save the max value to know how to set the threshold
    if current_vote_percentage > max:
        max = current_vote_percentage
    if current_vote_percentage > bin_threshold:
        out_shapes.append((x, y, r, current_vote_percentage))
```

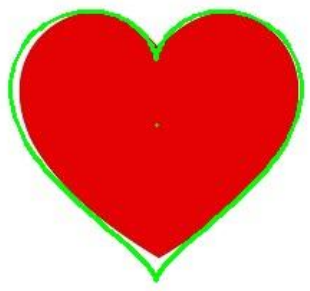
Once all center points were mapped and votes we took only those above the threshold and drew them on the image



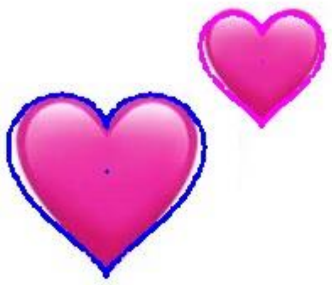
```
# Generate all the x1 and y1 points of that heart.
y1 = ((0.5 * np.cos(4 * t) + 2 * np.cos(3 * t) + 4 * np.cos(2 * t) - 13 * np.cos(t)) * r + y).astype(int)
x1 = ((np.sin(t) ** 3) * r * 14.5 + x).astype(int)
```

Results

Simple image – radius 6-7.5, threshold 0.036

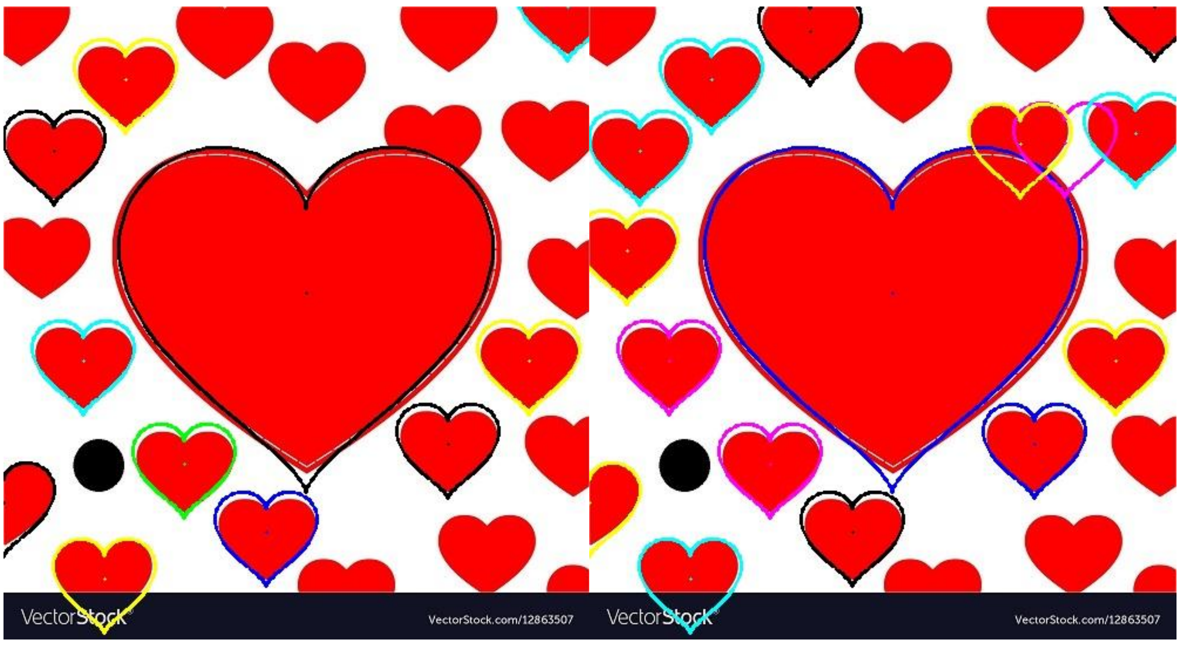


Med image – radius 2-5.5, threshold 0.033



Hard image – radius 3-3.5 and 11-11.5, threshold 0.065 or 0.056

For the hard image decreasing the threshold resulted in both true and false positives. This is the best we could come up with without any false positives (11 found total)



With one false positive we were able to add 3 more hearts (to 14) but with one false positive