HW 2 Report

Name: 陳翰雯

ID：B08902092

Problem 1: Edge Detection

(a) Sobel Edge Detection

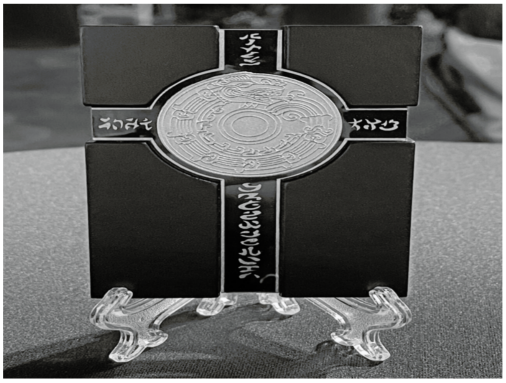

Fig 1. sample1.png

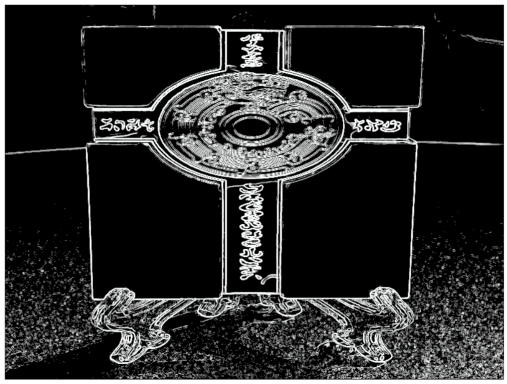

Fig 2. result2.png

The Sobel Mask used:

$$G_R = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad G_C = \frac{1}{4}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

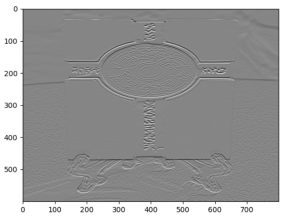

Fig 3. Sobel gradient image
(Row gradient)

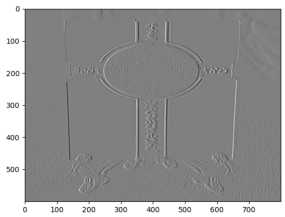

Fig 4. Sobel gradient image
(Column gradient)

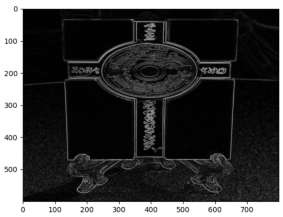

Fig 5. result1.png
(Sobel gradient Image)

We can see that as we increase the threshold value, we could see that there are lesser grey areas seen. Moreover, the image get a lot more accentuated and cleaner too. However, there are few parts that are supposed to be edges that are missing.

We used T=100 because it looks cleaner yet still detailed.

Fig 6. Sobel mask, T=50

Fig 7. Sobel mask, T=75

Fig 8. Sobel mask, T=100

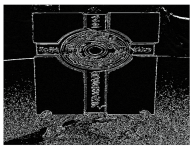

Fig 9. Sobel mask, T=125

Fig 10. Sobel mask, T=150

(b) Canny Edge Detection

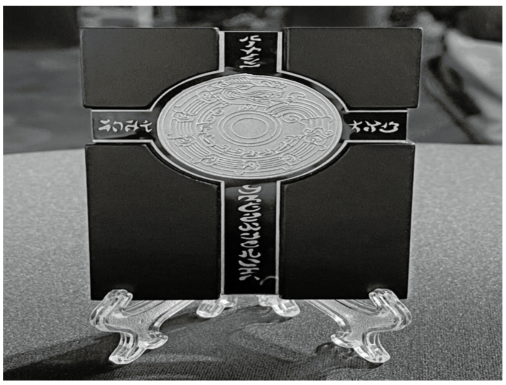

Fig 1. sample1.png

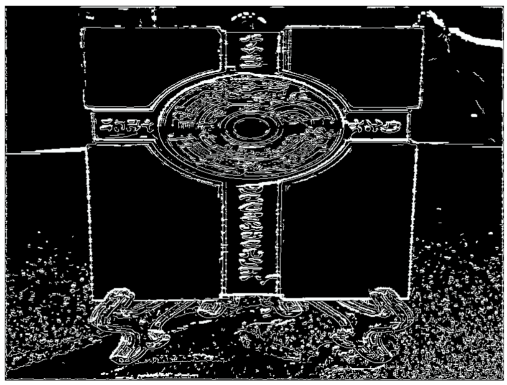

Fig 11. result3.png

The Canny Edge Detection is altogether done in 5 steps:

1. Noise Reduction by Gaussian Filtering

The filter used is: $\frac{1}{273}\begin{bmatrix}1 & 4 & 7 & 4 & 1\\4 & 16 & 26 & 16 & 4\\7 & 26 & 41 & 26 & 7\\4 & 16 & 26 & 16 & 4\\1 & 4 & 7 & 4 & 1\end{bmatrix}$

This overall smoothens the image

2. Compute Gradient and Orientation

This is done using the 1st order detection method. Here we choose to use Sobel Edge Detection method with the following Sobel Masks:

$$G_R = \frac{1}{4}\begin{bmatrix}1 & b & 1\\b & b^2 & b\\1 & b & 1\end{bmatrix} \qquad G_C = \frac{1}{4}\begin{bmatrix}-1 & 0 & 1\\-2 & 0 & 2\\-1 & 0 & 1\end{bmatrix}$$

3. Non-maximal Suppression

Here, the neighbors are divided into 4 direction parts

The code is implemented as follows:

```
for x in range(1,img.shape[0]-1):
    for y in range(1,img.shape[1]-1):
      # angle 0
```

```
        if (0 <= orientation[x,y] < 22.5) or (157.5 <= orientation[x,y] <= 180):
            one = sobel_filtered_img[x, y+1]
            two = sobel_filtered_img[x, y-1]
        # angle 45
        elif (22.5 <= orientation[x,y] < 67.5):
            one = sobel_filtered_img[x+1, y-1]
            two = sobel_filtered_img[x-1, y+1]
        # angle 90
        elif (67.5 <= orientation[x,y] < 112.5):
            one = sobel_filtered_img[x+1, y]
            two = sobel_filtered_img[x-1, y]
        # angle 135
        elif (112.5 <= orientation[x,y] < 157.5):
            one = sobel_filtered_img[x-1, y-1]
            two = sobel_filtered_img[x+1, y+1]

        if (sobel_filtered_img[x,y] >= one) and (sobel_filtered_img[x,y] >= two):
            suppressed_img[x,y] = sobel_filtered_img[x,y]
        else:
            suppressed_img[x,y] = 0
```

4. Hysteretic Thresholding

Each pixels are labelled as Edge pixel, Candidate pixel and Non-edge pixel.

We set the 2 thresholds T_H and T_L as follows:

```
low_threshold_ratio = 0.05
high_threshold_ratio = 0.09

T_h = suppressed_img.max() * high_threshold_ratio
T_l = T_h * low_threshold_ratio
```

5. Connected Component Labeling Method

If pixel is connected to an edge pixel directly or via another candidate pixel then it is declared as an edge pixel

The progress could be seen in the images below:

Fig 12. step 1: after Gaussian filter

Fig 13. step 2: after Sobel mask

Fig 14. step 3: after supression

Fig 15. step 4: after Hysteretic thresholding

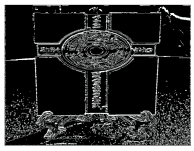

Fig 16. step 5: after connecting and labeling

(c) Laplacian of Gaussian Edge Detection

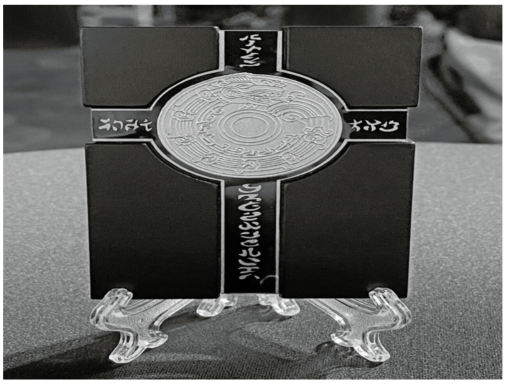

Fig 1. sample1.png

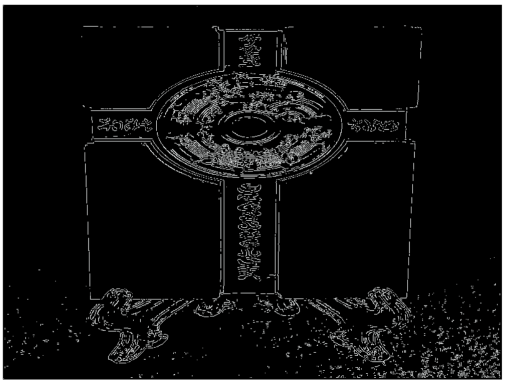

Fig 17. result4.png, T=5

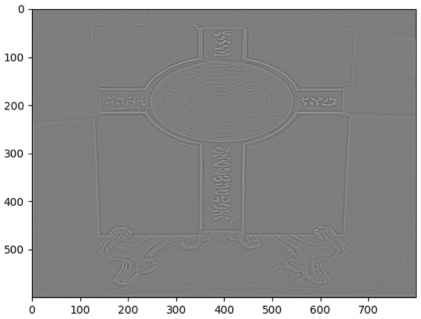

Fig 22. Laplacian gradient image

We first apply the following Gaussian filter:

$$\frac{1}{273}\begin{bmatrix}1 & 4 & 7 & 4 & 1\\ 4 & 16 & 26 & 16 & 4\\ 7 & 26 & 41 & 26 & 7\\ 4 & 16 & 26 & 16 & 4\\ 1 & 4 & 7 & 4 & 1\end{bmatrix}$$

Then, following non-separable eight-neighbor is mask is applied:

$$H = \frac{1}{8}\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix}$$

We then used the threshold T=5 to seperate zero and non-zero. The threshold T is chosen based on the following histogram:

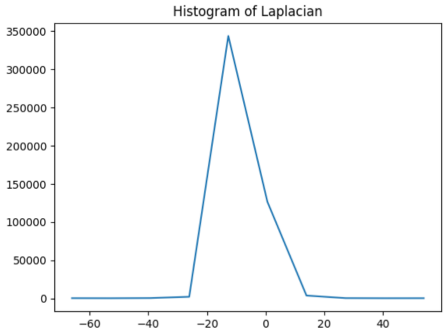

Fig 18. Laplacian Histogram

We then make all the values in the matrix to consist only of -1, 0, 1. This is done so that we can easily do zero-crossing in the next step. This done with the following code: `thresh_img = np.sign(thresh_img)`. `np.sign()` returns -1 when it is a negative value and 1 when it is a positive value.

Lastly, we then decide where it is a zero-crossing point for every $G'(j,k) = 0$

This is implemented with the following code:

```
if thresh_img[x,y] == 0:
      window = thresh_img[x-1:x+2, y-1:y+2]
      cross = np.unique(window)

      if (-1 in cross) and (1 in cross):
          edge_img[x,y] = 255
      else:
          edge_img[x,y] = 0
```

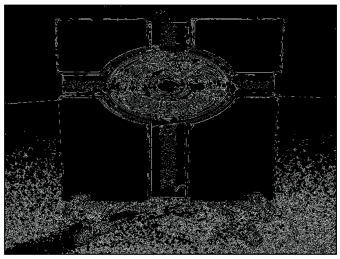

Fig 19. Laplacian, T=2

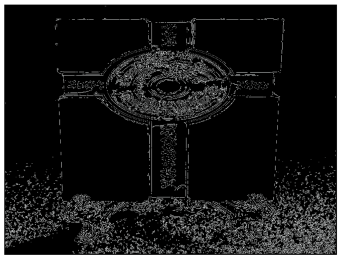

Fig 20. Laplacian, T=3

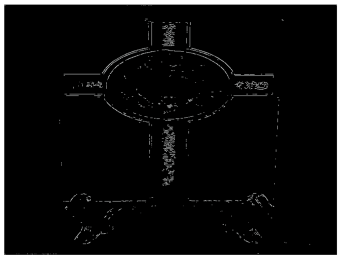

Fig 21. Laplacian, T=10)

Comparison:

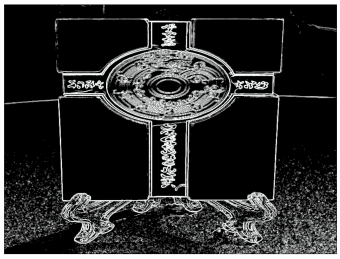

Fig 2. result2.png

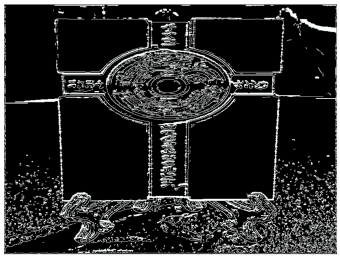

Fig 11. result3.png

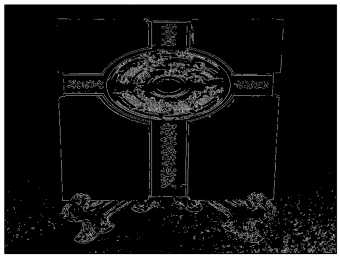

Fig 17. result4.png

From the 3 images above, we could see that Sobel Edge Detection without thresholding has the most grey areas however, it showed better details overall. In the case of Canny Edge Detection, there are still many noises that are regarded as edges. Lastly, in the case of Laplacian Edge Detection, the image is overall a lot cleaner, the lines are not too thick but it has fewer details shown in comparison to the other 2 methods. Moreover, another downside is that some of the edges are seem to be missing.

(d) Edge Crispening

Fig 23. sample2.png

Fig 24. result5.png

The image after edge crispening is somehow clearer and more accentuated.

To do this, we first apply the following low-pass filter:

$H = \frac{1}{(b+2)^2}\begin{bmatrix} 1 & b & 1 \\ b & b^2 & b \\ 1 & b & 1 \end{bmatrix}$ with b = 2

and then use the following Unsharp Masking method:

$G(j,k) = \frac{c}{2c-1}F(j,k) - \frac{1-c}{2c-1}F_L(j,k)$ where $c = \frac{3}{5}$

Fig 24. Unsharp Masking, c=3/5

Fig 25. Unsharp Masking, c=20/30

Fig 26. Unsharp Maskin, c=23/30)

Problem 2: Geometrical Modification

(a) Image Improvement

Fig 27. sample3.png

Fig 28. sample3.png
after improvement

Here there are a total of 5 steps done:

First, we decrease the brightness of some parts of the image. We can see that the cat's leg is a lot brighter than the other parts of the body. Hence, we decrease the brightness especially on the cat's leg with the following algorithm:

```
for round in range(6,14):
        for i in range(int(np.round(round*h/24)), h):
            for j in range(w):
                if img[i,j]!=0:
                    img[i,j] = np.floor(img[i,j]/1.2) if img[i,j]/1.2 >0 else 0
```

I imagined the image as being divided into 24 parts. From 6th part until the 14th part of the whole image, we decrease the brightness of each parts by 1.2 recursively.

Second, we increase the brightness of the whole image by 2.5.

Third, I performed edge crispening with unsharp masking method on the image to produce a more detailed and accentuated image.

Fourth, as we can see, there are visible white edges around the cat. Hence, to remove the white edges, I used the following code:

```
for i in range(h):
        for j in range(w):
            if final[i,j] >= 220:
                final[i,j] = 0
```

Lastly, I performed outlier detection which is also a method to remove noise in the image. This is to remove the remaining white spots around the cat. We do a total of 3 rounds with ε = 80, 70, 60 respectively.

The progress could be seen in the images below:

Fig 29. step 1: after decreasing brightness

Fig 30. step 2: after increasing brightness

Fig 31. step 3: after edge crispening

Fig 32. step 4: after white removal

Fig 33. step 5: after outlier detection

(b) Make sample3.png like sample4.png

Fig 27. sample3.png

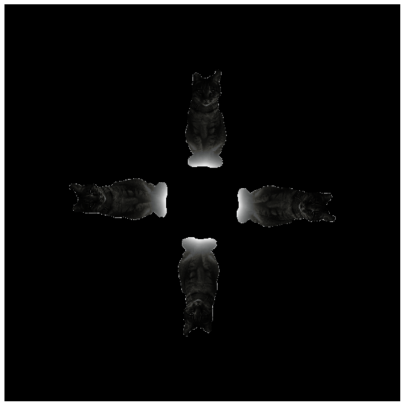

Fig 34. result7.png

To make sample3.png like sample4.png we do multiple scaling, rotation and translation in a for loop for a total of 4 rounds. (`for steps in range(4):..`)

In all of the rounds, the images are rotated to 0°, 90°, 180° and 270° respectively.

The images are then scaled with a scale factor of 0.4

The images are then translated to its respective positions with the following code: (`x_tr` and `y_tr` are the translation factors in the x and y direction respectively)

```
x_tr = 0
y_tr = 0
if steps == 0:
    x_tr = (img.shape[0]/2) - (scaled_img.shape[0]*scale_factor)
    y_tr = (img.shape[1]/2) - (scaled_img.shape[1]*scale_factor/2)
```

```
elif steps == 1:
    x_tr = (img.shape[0]/2) - (scaled_img.shape[0]*scale_factor/2)
    y_tr = (img.shape[1]/2) - (scaled_img.shape[1]*scale_factor)
elif steps == 2:
    x_tr = img.shape[0]/2
    y_tr = (img.shape[1]/2) - (scaled_img.shape[1]*scale_factor/2)
elif steps == 3:
    x_tr = (img.shape[0]/2) - (scaled_img.shape[0]*scale_factor/2)
    y_tr = img.shape[1]/2
```

The progress could be seen in the images below:

Fig 35. rotation: 0 degrees

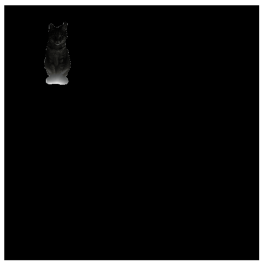

Fig 36. scaling with factor: 0.4

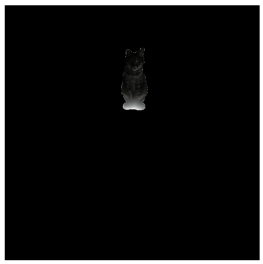

Fig 37. translation at step=0

Fig 38. rotation: 90 degrees

Fig 39. scaling with factor: 0.4

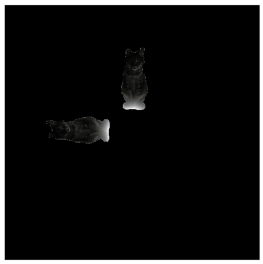

Fig 40. translation at step = 1

Fig 41. rotation: 180 degrees

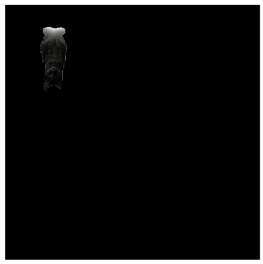

Fig 42. scaling with factor: 0.4

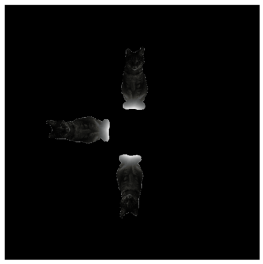

Fig 43. translation at step =2

Fig 44. rotation: 270 degrees

Fig 45. scaling with factor: 0.4

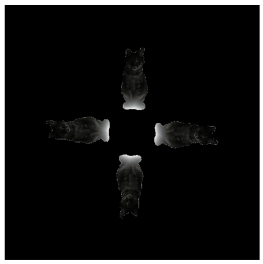

Fig 46. translation at step =3

(c) Make sample5.png like sample6.png

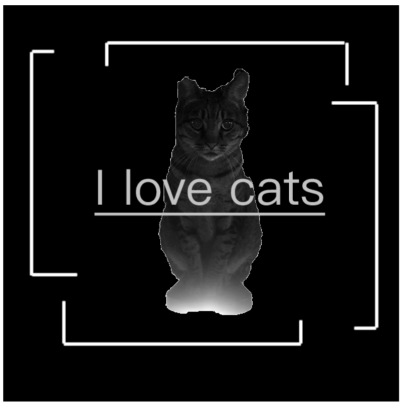

Fig 47. sample5.png

Fig 48. result8.png

The x',y' position of the image are calculated with the following formula:

$x' = x \times 20 \times sin(\frac{8\pi y}{h})$, where h is the height of the image

$y' = y \times 20 \times sin(\frac{8\pi y}{w})$, where w is the height of the image

The code is implemented as follows:

```
for x in range(size):
        for y in range(size):

            x_ = x + 20 * np.sin((8*np.pi * y/size))
            x_pos = int(np.round(x_))

            y_ = y + 20 * np.sin((8*np.pi * x/size))
            y_pos = int(np.round(y_))

            if x_pos>=0 and x_pos<size and y_pos>=0 and y_pos<size:
                final[x_pos,y_pos] = filtered_img[x,y]
```

Here some other experiments with different formula used:

warped1.png: $x' = x \times 5 \times sin(\frac{8\pi y}{h}), \quad y' = y \times 5 \times sin(\frac{8\pi y}{w})$

warped2.png: $x' = x \times 35 \times sin(\frac{8\pi y}{h}), \quad y' = y \times 35 \times sin(\frac{8\pi y}{w})$

warped3.png: $x' = x \times 20 \times sin(\frac{4\pi y}{h}), \quad y' = y \times 20 \times sin(\frac{4\pi y}{w})$

warped4.png: $x' = x \times 20 \times sin(\frac{12\pi y}{h}), \quad y' = y \times 20 \times sin(\frac{12\pi y}{w})$

Fig 48. result8.png

Fig 49. warped1.png

Fig 50. warped2.png

Fig 51. warped3.png

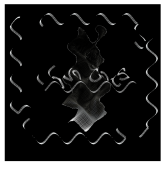

Fig 51. warped4.png