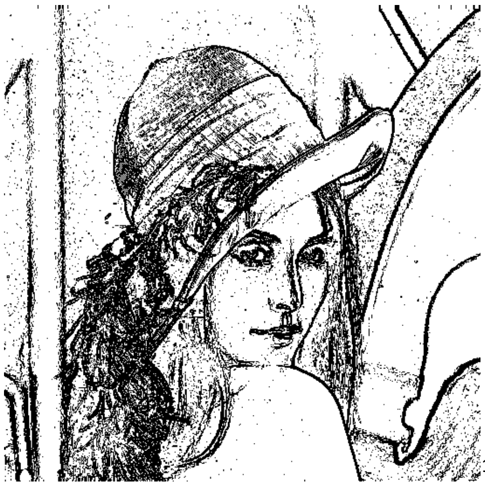
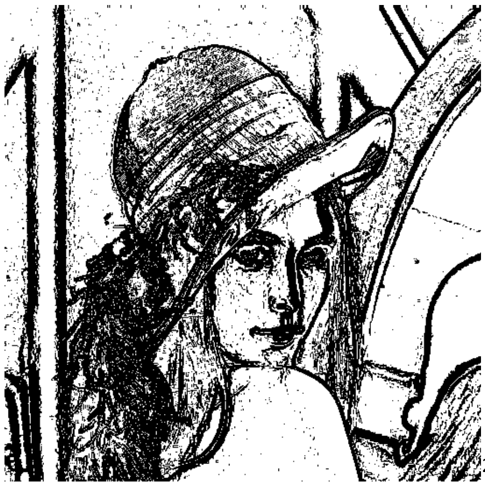
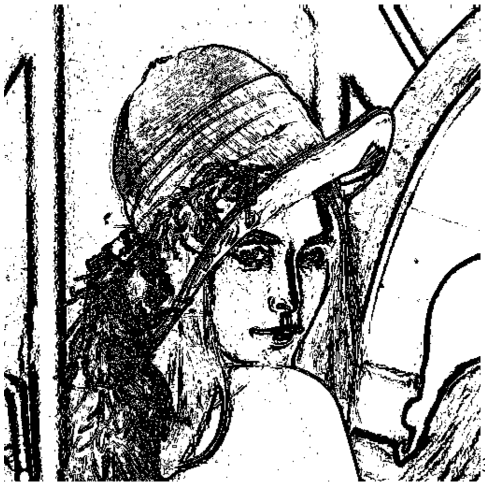
R13922136 周洛君

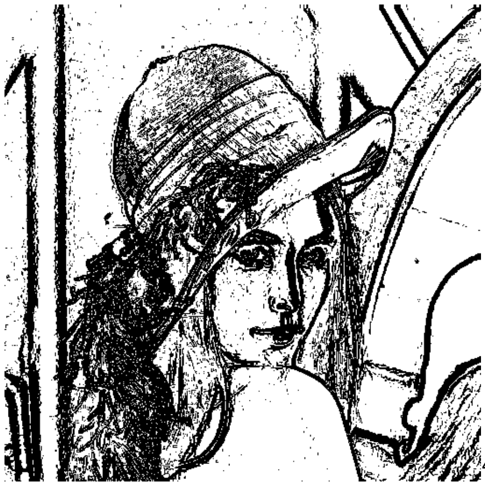
Computer Vision HW 9: Report

The resulting image are as below:

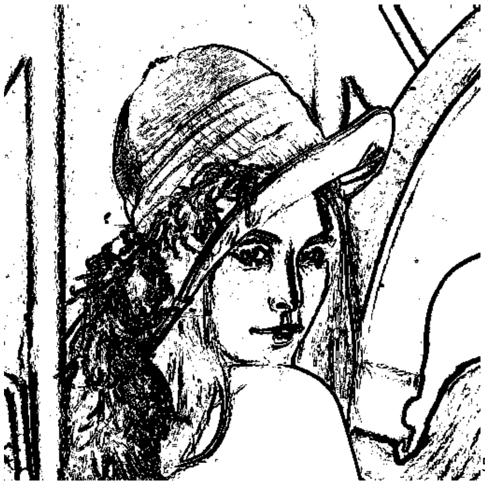

[1]

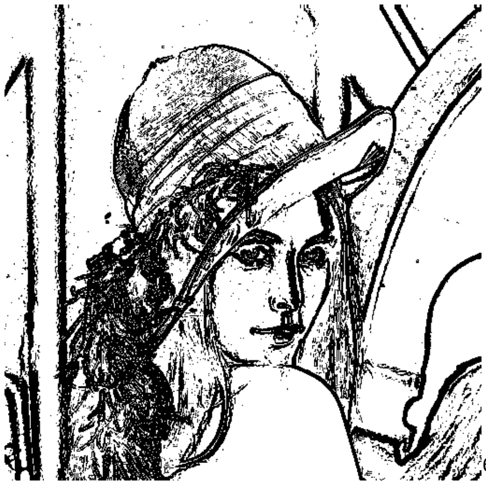

[2]

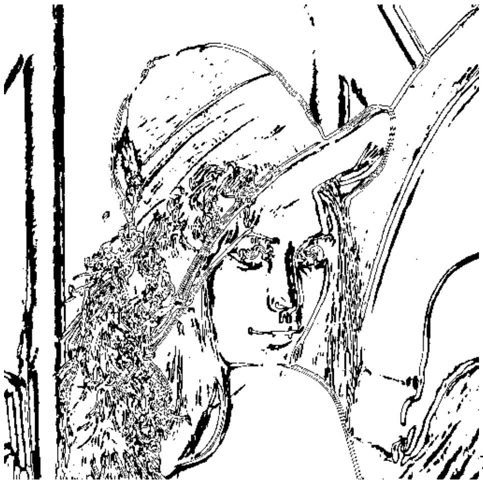

[3]


[4]

[1] (a) Roberts, threshold = 12
[2] (b) Prewitt, threshold = 24
[3] (c) Sobel, threshold = 38
[4] (d) Frei and Chen, threshold = 30


[5]


[6]

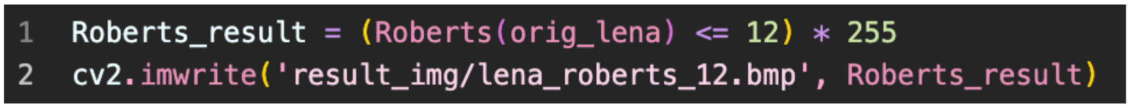

[7]

Code Explanation:

In the source file, it consists of:

- a convolution function
- functions for each operator / edge detector (correspond to (a)~(g))

Each operator / edge detector function is followed by a code block of this form:



```
Roberts_result = (Roberts(orig_lena) <= 12) * 255
cv2.imwrite('result_img/lena_roberts_12.bmp', Roberts_result)
```

Which executes the function (Roberts() in this example), and performs the required thresholding, then multiply by 255, and saving the result using cv2.imwrite() as usual.

---

[5] (e) Kirsch, threshold = 135
[6] (f) Robinson, threshold = 43
[7] (g) Nevatia Babu, threshold = 12500

The convolution function is called in each of the functions (a)~(g) (we would explain later):

```python
def convolution(neighborhood, convolution_mask):
    value = 0
    for i in range(neighborhood.shape[0]):
        for j in range(neighborhood.shape[1]):
            value += (neighborhood[i, j] * convolution_mask[convolution_mask.shape[0] - i - 1, convolution_mask.shape[1] - j - 1])
    return value
```

For the functions corresponding to (a)~(g), they actually have similar structure. Take Robert's for example, the first step is to convert our image (the original lena.bmp) into a ndarray:

```python
def Roberts(img):
    img = np.asarray(img, np.int16)
```

Then we define the masks as in the textbook, I wrote brief descriptions in the corresponding markdown cell:

(a) Robert's Operator: 12

> ppt p.166

Use 2x2 mask to calculate gradient, across two diagonal directions.

$$r_1 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad r_2 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

- r_1: value calculated from the first mask
- r_2: value calculated from the second mask

$\rightarrow$ gradient magnitude g:

$$g = \sqrt{r_1^2 + r_2^2}$$

```python
    k1 = np.array([
        [1, 0],
        [0, -1]
    ])
    k2 = np.array([
        [0, 1],
        [-1, 0]
    ])
```

To calculate the G_x, G_y as defined in the ppt, we first initialize an empty ndarray for each G_x, G_y:

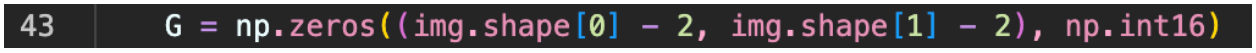
$$\begin{aligned} G_x &= f(x, y+1) - f(x+1, y), \\ G_y &= f(x, y) - f(x+1, y+1) \end{aligned} \qquad (8)$$

```python
    G_x = np.zeros((img.shape[0] - 1, img.shape[1] - 1), np.int16)
    G_y = np.zeros((img.shape[0] - 1, img.shape[1] - 1), np.int16)
```

If we look at the dimension settings, we can see that both the height and width are subtracted by 1 since our masks are 2x2. If our mask is 3x3, then we subtract both dimensions by 2 like below:

```python
    G = np.zeros((img.shape[0] - 2, img.shape[1] - 2), np.int16)
```

Also, for (a)~(d), we define G_x, G_y, but for some edge detectors, we calculate the result by choosing the maximum, under such cases, we only define one G (problem (e)~(g)).

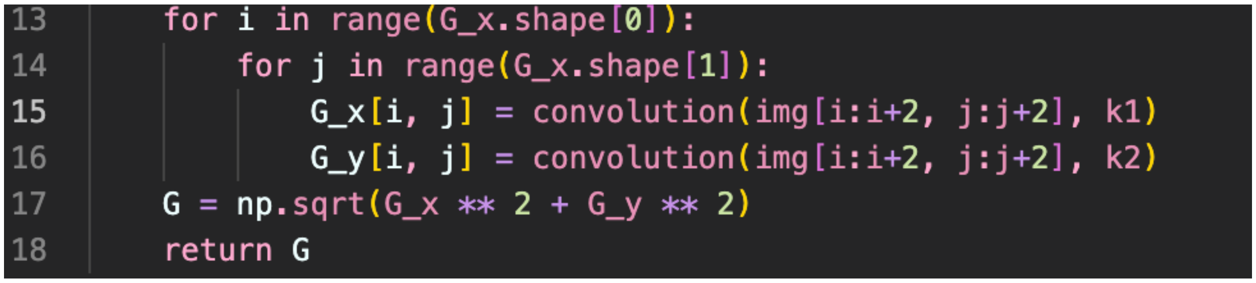


```python
    G = np.zeros((img.shape[0] - 2, img.shape[1] - 2), np.int16)
```

The last thing to mention that we need to set np.int16 instead of np.int8 (which will cause wrong results for some problems.)

Going back to our example Roberts function, we loop through each pixel (as the same size of G_x). For each pixel in the resulting array, in order to calculate ${r_1}^2$, ${r_2}^2$ in the gradient magnitude formula, we calculate r_1, r_2 by calling the convolution function respectively. In the convolution function, we deal with a 2x2 neighborhood each time, thus we use the index img[i:i+2, j:j+2] as the first input. The second input is the corresponding mask k_1 in order to calculate r_1.

After these calculations are all done for the resulting picture, we then take the squares for each r_1, r_2 and the square root for the whole result to get our result G.



```python
    for i in range(G_x.shape[0]):
        for j in range(G_x.shape[1]):
            G_x[i, j] = convolution(img[i:i+2, j:j+2], k1)
            G_y[i, j] = convolution(img[i:i+2, j:j+2], k2)
    G = np.sqrt(G_x ** 2 + G_y ** 2)
    return G
```