

Homework 9

R13525009 羅筠笙

a. Robert's Operator: 12

Description: Uses a 2x2 kernel to compute the gradient magnitude for edge detection by applying convolution to the input image.



```
1. def robert(img_arr, threshold):
2.     res_arr = np.zeros([img_size0, img_size1])
3.     for i in range(img_size0-1):
4.         for j in range(img_size1-1):
5.             '''
6.             r1 = [-1 0
7.                  0 1]
8.             r2 = [[0 -1
9.                  1 0]
10.            '''
11.             r1 = - int(img_arr[i][j]) + int(img_arr[i+1][j+1])
12.             r2 = - int(img_arr[i][j+1]) + int(img_arr[i+1][j])
13.
14.             grad = np.sqrt(r1**2 + r2**2)
15.             res_arr[i, j] = 255 if grad < threshold else 0
16.
17.     return res_arr
```

b. Prewitt's Edge Detector: 24

Description: Applies 3x3 kernels for horizontal and vertical gradients using convolution.



```
1. def prewitt_edge(img_arr, threshold):
2.     img_arr = expand_with_replicate(img_arr, 1)
3.     m, n = img_arr.shape
4.     res_arr = np.zeros([img_size0, img_size1])
5.
6.     for i in range(img_size0):
7.         for j in range(img_size1):
8.             '''
9.             p1 = [-1 -1 -1
10.                  0 0 0
11.                  1 1 1]
12.             p2 = [-1 0 1
13.                  -1 0 1
14.                  -1 0 1]
15.             '''
16.             p1 = - int(img_arr[i, j]) - int(img_arr[i, j+1]) - int(img_arr[i, j+2]) + int(img_arr[i+2, j]) + int(img_arr[i+2, j+1]) + int(img_arr[i+2, j+2])
17.             p2 = - int(img_arr[i, j]) - int(img_arr[i+1, j]) - int(img_arr[i+2, j]) + int(img_arr[i, j+2]) + int(img_arr[i+1, j+2]) + int(img_arr[i+2, j+2])
18.
19.             grad = np.sqrt(p1**2 + p2**2)
```

```

20.         res_arr[i, j] = 255 if grad < threshold else 0
21.
22.     return res_arr

```

c. Sobel's Edge Detector: 38

Description: Implements Sobel's operator by convolving the image with 3x3 kernels that emphasize the center pixels.



```

1. def sobel_edge(img_arr, threshold):
2.     img_arr = expand_with_replicate(img_arr, 1)
3.     m, n = img_arr.shape
4.     res_arr = np.zeros([img_size0, img_size1])
5.
6.     for i in range(img_size0):
7.         for j in range(img_size1):
8.             '''
9.             s1 = [-1 -2 -1
10.                  0 0 0
11.                  1 2 1]
12.             s2 = [-1 0 1
13.                  -2 0 2
14.                  -1 0 1]
15.             '''
16.             s1 = - int(img_arr[i, j]) - 2 * int(img_arr[i, j+1]) - int(img_arr
[i, j+2]) + int(img_arr[i+2, j]) + 2 * int(img_arr[i+2, j+1]) + int(img_arr[i
+2, j+2])

```

```

17.         s2 = - int(img_arr[i, j]) - 2 * int(img_arr[i+1, j]) - int(img_arr
    [i+2, j]) + int(img_arr[i, j+2]) + 2 * int(img_arr[i+1, j+2]) + int(img_arr[i
    +2, j+2])
18.
19.         grad = np.sqrt(s1**2 + s2**2)
20.         res_arr[i, j] = 255 if grad < threshold else 0
21.
22.     return res_arr

```

d. Frei and Chen's Gradient Operator: 30

Description: Uses modified Sobel-like 3x3 kernels, applying specific weights to capture diagonal and straight-edge gradients



```

1. def frei_and_chen_grad(img_arr, threshold):
2.     img_arr = expand_with_replicate(img_arr, 1)
3.     m, n = img_arr.shape
4.     res_arr = np.zeros([img_size0, img_size1])
5.
6.     for i in range(img_size0):
7.         for j in range(img_size1):
8.             '''
9.             f1 = [-1 sqrt(-2)  -1
10.                  0      0  0
11.                  1 sqrt(2)  1]
12.             f2 = [-1      0  1
13.                   sqrt(-2)  0 sqrt(2)
14.                   -1      0  1]

```

```

15.         """
16.         f1 = - int(img_arr[i, j]) - np.sqrt(2) * int(img_arr[i, j+1]) - int
            (img_arr[i, j+2]) + int(img_arr[i+2, j]) + np.sqrt(2) * int(img_arr[i+2, j+1])
            + int(img_arr[i+2, j+2])
17.         f2 = - int(img_arr[i, j]) - np.sqrt(2) * int(img_arr[i+1, j]) - int
            (img_arr[i+2, j]) + int(img_arr[i, j+2]) + np.sqrt(2) * int(img_arr[i+1, j+2])
            + int(img_arr[i+2, j+2])
18.
19.         grad = np.sqrt(f1**2 + f2**2)
20.         res_arr[i, j] = 255 if grad < threshold else 0
21.
22.     return res_arr

```

e. Kirsch's Compass Operator: 135

Description: Applies eight 3x3 directional masks (compass masks) to calculate gradient magnitudes in all compass directions.



```

1. def kirsch_compass(img_arr, threshold):
2.     img_arr = expand_with_replicate(img_arr, 1)
3.     m, n = img_arr.shape
4.     res_arr = np.zeros([img_size0, img_size1])
5.
6.     for i in range(img_size0):
7.         for j in range(img_size1):
8.             """
9.             k0 = [-3  -3   5
10.                  -3   0   5

```

```

11.         -3  -3  5]
12.         k1 = [-3  5  5
13.             -3  0  5
14.             -3  -3  -3]
15.         k2 = [5  5  5
16.             -3  0  -3
17.             -3  -3  -3]
18.         k3 = [5  5  -3
19.             5  0  -3
20.             -3  -3  -3]
21.         k4 = [5  -3  -3
22.             5  0  -3
23.             5  -3  -3]
24.         k5 = [-3  -3  -3
25.             5  0  -3
26.             5  5  -3]
27.         k6 = [-3  -3  -3
28.             -3  0  -3
29.             5  5  5]
30.         k7 = [-3  -3  -3
31.             -3  0  5
32.             -3  5  5]
33.         ""
34.         k0 = - 3 * int(img_arr[i, j]) - 3 * int(img_arr[i, j+1]) + 5 *
int(img_arr[i, j+2]) \
35.             - 3 * int(img_arr[i+1, j]) + 5 * int(img_arr[i+1, j+2]) \
36.             - 3 * int(img_arr[i+2, j]) - 3 * int(img_arr[i+2, j+1]) + 5 *
int(img_arr[i+2, j+2])
37.         k1 = - 3 * int(img_arr[i, j]) + 5 * int(img_arr[i, j+1]) + 5 *
int(img_arr[i, j+2]) \
38.             - 3 * int(img_arr[i+1, j]) + 5 * int(img_arr[i+1, j+2]) \
39.             - 3 * int(img_arr[i+2, j]) - 3 * int(img_arr[i+2, j+1]) - 3 *
int(img_arr[i+2, j+2])
40.         k2 = 5 * int(img_arr[i, j]) + 5 * int(img_arr[i, j+1]) + 5 *
int(img_arr[i, j+2]) \
41.             - 3 * int(img_arr[i+1, j]) - 3 * int(img_arr[i+1, j+2]) \
42.             - 3 * int(img_arr[i+2, j]) - 3 * int(img_arr[i+2, j+1]) - 3 *
int(img_arr[i+2, j+2])

```

```

43.         k3 = 5 * int(img_arr[i, j]) + 5 * int(img_arr[i, j+1]) - 3 *
            int(img_arr[i, j+2]) \
44.             + 5 * int(img_arr[i+1, j]) - 3 * int(img_arr[i+1, j+2]) \
45.             - 3 * int(img_arr[i+2, j]) - 3 * int(img_arr[i+2, j+1]) - 3 *
            int(img_arr[i+2, j+2])
46.         k4 = 5 * int(img_arr[i, j]) - 3 * int(img_arr[i, j+1]) - 3 *
            int(img_arr[i, j+2]) \
47.             + 5 * int(img_arr[i+1, j]) - 3 * int(img_arr[i+1, j+2]) \
48.             + 5 * int(img_arr[i+2, j]) - 3 * int(img_arr[i+2, j+1]) - 3 *
            int(img_arr[i+2, j+2])
49.         k5 = - 3 * int(img_arr[i, j]) - 3 * int(img_arr[i, j+1]) - 3 *
            int(img_arr[i, j+2]) \
50.             + 5 * int(img_arr[i+1, j]) - 3 * int(img_arr[i+1, j+2]) \
51.             + 5 * int(img_arr[i+2, j]) + 5 * int(img_arr[i+2, j+1]) - 3 *
            int(img_arr[i+2, j+2])
52.         k6 = - 3 * int(img_arr[i, j]) - 3 * int(img_arr[i, j+1]) - 3 *
            int(img_arr[i, j+2]) \
53.             - 3 * int(img_arr[i+1, j]) - 3 * int(img_arr[i+1, j+2]) \
54.             + 5 * int(img_arr[i+2, j]) + 5 * int(img_arr[i+2, j+1]) + 5 *
            int(img_arr[i+2, j+2])
55.         k7 = - 3 * int(img_arr[i, j]) - 3 * int(img_arr[i, j+1]) - 3 *
            int(img_arr[i, j+2]) \
56.             - 3 * int(img_arr[i+1, j]) + 5 * int(img_arr[i+1, j+2]) \
57.             - 3 * int(img_arr[i+2, j]) + 5 * int(img_arr[i+2, j+1]) + 5 *
            int(img_arr[i+2, j+2])
58.
59.         k_list = np.array([k0, k1, k2, k3, k4, k5, k6, k7])
60.         grad = np.max(k_list)
61.
62.         res_arr[i, j] = 255 if grad < threshold else 0
63.
64.     return res_arr

```

f. **Robinson's Compass Operator: 43**

Description: Implements eight directional 3x3 masks similar to Kirsch but uses simpler calculations.



```
1. def robinson_compass(img_arr, threshold):
2.     img_arr = expand_with_replicate(img_arr, 1)
3.     m, n = img_arr.shape
4.     res_arr = np.zeros([img_size0, img_size1])
5.
6.     for i in range(img_size0):
7.         for j in range(img_size1):
8.             '''
9.                 r0 = [-1    0    1
10.                    -2    0    2
11.                    -1    0    1]
12.                 r1 = [0     1    2
13.                    -1    0    1
14.                    -2   -1    0]
15.                 r2 = [1     2    1
16.                     0     0    0
17.                    -1   -2   -1]
18.                 r3 = [2     1    0
19.                     1     0   -1
20.                     0    -1   -2]
21.                 r4 = [1     0   -1
22.                     2     0   -2
```



```

23.         1    0   -1]
24.     r5 = [0    -1   -2
25.           1    0   -1
26.           2    1    0]
27.     r6 = [-1   -2   -1
28.           0    0    0
29.           1    2    1]
30.     r7 = [-2   -1    0
31.           -1    0    1
32.           0    1    2]
33.     ""
34.     r0 = - int(img_arr[i, j]) + int(img_arr[i, j+2]) \
35.           - 2 * int(img_arr[i+1, j]) + 2 * int(img_arr[i+1, j+2]) \
36.           - int(img_arr[i+2, j]) + int(img_arr[i+2, j+2])
37.     r1 = int(img_arr[i, j+1]) + 2 * int(img_arr[i, j+2]) \
38.           - int(img_arr[i+1, j]) + int(img_arr[i+1, j+2]) \
39.           - 2 * int(img_arr[i+2, j]) - int(img_arr[i+2, j+1])
40.     r2 = int(img_arr[i, j]) + 2 * int(img_arr[i, j+1]) + int(img_arr[i,
41.           j+2]) \
42.           - int(img_arr[i+2, j]) - 2 * int(img_arr[i+2, j+1]) -
43.           int(img_arr[i+2, j+2])
44.     r3 = 2 * int(img_arr[i, j]) + int(img_arr[i, j+1]) \
45.           + int(img_arr[i+1, j]) - int(img_arr[i+1, j+2]) \
46.           - int(img_arr[i+2, j+1]) - 2 * int(img_arr[i+2, j+2])
47.     r4 = int(img_arr[i, j]) - int(img_arr[i, j+2]) \
48.           + 2 * int(img_arr[i+1, j]) - 2 * int(img_arr[i+1, j+2]) \
49.           + int(img_arr[i+2, j]) - int(img_arr[i+2, j+2])
50.     r5 = - int(img_arr[i, j+1]) - 2 * int(img_arr[i, j+2]) \
51.           + int(img_arr[i+1, j]) - int(img_arr[i+1, j+2]) \
52.           + 2 * int(img_arr[i+2, j]) + int(img_arr[i+2, j+1])
53.     r6 = - int(img_arr[i, j]) - 2 * int(img_arr[i, j+1]) - int(img_arr[i,
54.           j+2]) \
55.           + int(img_arr[i+2, j]) + 2 * int(img_arr[i+2, j+1]) +
56.           int(img_arr[i+2, j+2])
57.     r7 = - 2 * int(img_arr[i, j]) - int(img_arr[i, j+1]) \
58.           - int(img_arr[i+1, j]) + int(img_arr[i+1, j+2]) \
59.           + int(img_arr[i+2, j+1]) + 2 * int(img_arr[i+2, j+2])
60.
61.     r_list = np.array([r0, r1, r2, r3, r4, r5, r6, r7])

```

```

58.         grad = np.max(r_list)
59.
60.         res_arr[i, j] = 255 if grad < threshold else 0
61.
62.     return res_arr

```

g. Nevatia-Babu 5x5 Operator: 12500

Description: Applies multiple 5x5 kernels to analyze high-resolution images, detecting fine and directional edges.



```

1. def nevatia_babu_5x5(img_arr, threshold):
2.     img_arr = expand_with_replicate(img_arr, 2)
3.     m, n = img_arr.shape
4.     res_arr = np.zeros([img_size0, img_size1])
5.
6.     for i in range(img_size0):
7.         for j in range(img_size1):
8.             '''
9.                 n0 = [100  100  100  100  100
10.                    100  100  100  100  100
11.                     0   0   0   0   0
12.                    -100 -100 -100 -100 -100
13.                    -100 -100 -100 -100 -100]
14.                 n1 = [100  100  100  100  100
15.                     100  100  100  78  -32
16.                     100  92   0  -92 -100
17.                     32  -78 -100 -100 -100]

```

```

18.         -100 -100 -100 -100 -100]
19.         n2 = [100  100  100  32 -100
20.             100  100  92  -78 -100
21.             100  100   0 -100 -100
22.             100  78  -92 -100 -100
23.             100 -32 -100 -100 -100]
24.         n3 = [-100 -100   0  100  100
25.             -100 -100   0  100  100
26.             -100 -100   0  100  100
27.             -100 -100   0  100  100
28.             -100 -100   0  100  100]
29.         n4 = [-100  32  100  100  100
30.             -100 -78  92  100  100
31.             -100 -100   0  100  100
32.             -100 -100 -92  78  100
33.             -100 -100 -100 -32  100]
34.         n5 = [100  100  100  100  100
35.             -32  78  100  100  100
36.             -100 -92   0  92  100
37.             -100 -100 -100 -78  32
38.             -100 -100 -100 -100 -100]
39.         ""
40.         n0 = 100 * int(img_arr[i, j]) + 100 * int(img_arr[i, j+1]) + 100 *
41.             int(img_arr[i, j+2]) + 100 * int(img_arr[i, j+3]) + 100 * int(img_arr[i, j+4]) \
42.             + 100 * int(img_arr[i+1, j]) + 100 * int(img_arr[i+1, j+1]) +
43.             100 * int(img_arr[i+1, j+2]) + 100 * int(img_arr[i+1, j+3]) + 100 *
44.             int(img_arr[i+1, j+4]) \
45.             - 100 * int(img_arr[i+3, j]) - 100 * int(img_arr[i+3, j+1]) -
46.             100 * int(img_arr[i+3, j+2]) - 100 * int(img_arr[i+3, j+3]) - 100 *
47.             int(img_arr[i+3, j+4]) \
48.             - 100 * int(img_arr[i+4, j]) - 100 * int(img_arr[i+4, j+1]) -
49.             100 * int(img_arr[i+4, j+2]) - 100 * int(img_arr[i+4, j+3]) - 100 *
50.             int(img_arr[i+4, j+4])
51.         n1 = 100 * int(img_arr[i, j]) + 100 * int(img_arr[i, j+1]) + 100 *
52.             int(img_arr[i, j+2]) + 100 * int(img_arr[i, j+3]) + 100 * int(img_arr[i, j+4]) \
53.             + 100 * int(img_arr[i+1, j]) + 100 * int(img_arr[i+1, j+1]) +
54.             100 * int(img_arr[i+1, j+2]) + 78 * int(img_arr[i+1, j+3]) - 32 *
55.             int(img_arr[i+1, j+4]) \

```

```

46.          + 100 * int(img_arr[i+2, j]) + 92 * int(img_arr[i+2, j+1]) - 92
          * int(img_arr[i+2, j+3]) - 100 * int(img_arr[i+2, j+4]) \
47.          + 32 * int(img_arr[i+3, j]) - 78 * int(img_arr[i+3, j+1]) - 100
          * int(img_arr[i+3, j+2]) - 100 * int(img_arr[i+3, j+3]) - 100 * int(img_arr[i+3,
          j+4]) \
48.          - 100 * int(img_arr[i+4, j]) - 100 * int(img_arr[i+4, j+1]) -
          100 * int(img_arr[i+4, j+2]) - 100 * int(img_arr[i+4, j+3]) - 100 *
          int(img_arr[i+4, j+4])
49.          n2 = 100 * int(img_arr[i, j]) + 100 * int(img_arr[i, j+1]) + 100 *
          int(img_arr[i, j+2]) + 32 * int(img_arr[i, j+3]) - 100 * int(img_arr[i, j+4]) \
50.          + 100 * int(img_arr[i+1, j]) + 100 * int(img_arr[i+1, j+1]) +
          92 * int(img_arr[i+1, j+2]) - 78 * int(img_arr[i+1, j+3]) - 100 *
          int(img_arr[i+1, j+4]) \
51.          + 100 * int(img_arr[i+2, j]) + 100 * int(img_arr[i+2, j+1]) -
          100 * int(img_arr[i+2, j+3]) - 100 * int(img_arr[i+2, j+4]) \
52.          + 100 * int(img_arr[i+3, j]) + 78 * int(img_arr[i+3, j+1]) - 92
          * int(img_arr[i+3, j+2]) - 100 * int(img_arr[i+3, j+3]) - 100 * int(img_arr[i+3,
          j+4]) \
53.          + 100 * int(img_arr[i+4, j]) - 32 * int(img_arr[i+4, j+1]) -
          100 * int(img_arr[i+4, j+2]) - 100 * int(img_arr[i+4, j+3]) - 100 *
          int(img_arr[i+4, j+4])
54.          n3 = - 100 * int(img_arr[i, j]) - 100 * int(img_arr[i, j+1]) + 100 *
          int(img_arr[i, j+3]) + 100 * int(img_arr[i, j+4]) \
55.          - 100 * int(img_arr[i+1, j]) - 100 * int(img_arr[i+1, j+1]) +
          100 * int(img_arr[i+1, j+3]) + 100 * int(img_arr[i+1, j+4]) \
56.          - 100 * int(img_arr[i+2, j]) - 100 * int(img_arr[i+2, j+1]) +
          100 * int(img_arr[i+2, j+3]) + 100 * int(img_arr[i+2, j+4]) \
57.          - 100 * int(img_arr[i+3, j]) - 100 * int(img_arr[i+3, j+1]) +
          100 * int(img_arr[i+3, j+3]) + 100 * int(img_arr[i+3, j+4]) \
58.          - 100 * int(img_arr[i+4, j]) - 100 * int(img_arr[i+4, j+1]) +
          100 * int(img_arr[i+4, j+3]) + 100 * int(img_arr[i+4, j+4])
59.          n4 = - 100 * int(img_arr[i, j]) + 32 * int(img_arr[i, j+1]) + 100 *
          int(img_arr[i, j+2]) + 100 * int(img_arr[i, j+3]) + 100 * int(img_arr[i, j+4]) \
60.          - 100 * int(img_arr[i+1, j]) - 78 * int(img_arr[i+1, j+1]) + 92
          * int(img_arr[i+1, j+2]) + 100 * int(img_arr[i+1, j+3]) + 100 *
          int(img_arr[i+1, j+4]) \
61.          - 100 * int(img_arr[i+2, j]) - 100 * int(img_arr[i+2, j+1]) +
          100 * int(img_arr[i+2, j+3]) + 100 * int(img_arr[i+2, j+4]) \

```

```

62.         - 100 * int(img_arr[i+3, j]) - 100 * int(img_arr[i+3, j+1]) - 92
          * int(img_arr[i+3, j+2]) + 78 * int(img_arr[i+3, j+3]) + 100 * int(img_arr[i+3,
          j+4]) \
63.         - 100 * int(img_arr[i+4, j]) - 100 * int(img_arr[i+4, j+1]) -
          100 * int(img_arr[i+4, j+2]) - 32 * int(img_arr[i+4, j+3]) + 100 *
          int(img_arr[i+4, j+4])
64.         n5 = 100 * int(img_arr[i, j]) + 100 * int(img_arr[i, j+1]) + 100 *
          int(img_arr[i, j+2]) + 100 * int(img_arr[i, j+3]) + 100 * int(img_arr[i, j+4]) \
65.         - 32 * int(img_arr[i+1, j]) + 78 * int(img_arr[i+1, j+1]) + 100
          * int(img_arr[i+1, j+2]) + 100 * int(img_arr[i+1, j+3]) + 100 *
          int(img_arr[i+1, j+4]) \
66.         - 100 * int(img_arr[i+2, j]) - 92 * int(img_arr[i+2, j+1]) + 92
          * int(img_arr[i+2, j+3]) + 100 * int(img_arr[i+2, j+4]) \
67.         - 100 * int(img_arr[i+3, j]) - 100 * int(img_arr[i+3, j+1]) -
          100 * int(img_arr[i+3, j+2]) - 78 * int(img_arr[i+3, j+3]) + 32 *
          int(img_arr[i+3, j+4]) \
68.         - 100 * int(img_arr[i+4, j]) - 100 * int(img_arr[i+4, j+1]) -
          100 * int(img_arr[i+4, j+2]) - 100 * int(img_arr[i+4, j+3]) - 100 *
          int(img_arr[i+4, j+4])
69.
70.         n_list = np.array([n0, n1, n2, n3, n4, n5])
71.         grad = np.max(n_list)
72.
73.         res_arr[i, j] = 255 if grad < threshold else 0
74.
75.     return res_arr

```