922 U0610 電腦視覺
Computer Vision

Homework 9

授課教師： 傅楸善 教授

學生系級： 資工所一年級

學生姓名： 姚嘉昇

學生學號： R06922002

I. INTRODUCTION

1.1. Descriptions of Problem

This homework is to do general edge detection with following rules:

A. Robert's operator with threshold of 12.
B. Prewitt's edge detector with threshold of 24.
C. Sobel's edge detector with threshold of 38.
D. Frei and Chen's gradient operator with threshold of 30.
E. Kirsch's compass operator with threshold of 135.
F. Robinson's compass operator with threshold of 43.
G. Nevatia-Babu 5x5 operator with threshold of 12500.

1.2. Programming Tools

1.2.1. Programming Language: Python3
1.2.2. Programming IDE: Visual Studio Code

II. METHOD

2.1. Algorithms

2.1.1. Robert's operator

-1	
	1

r_1

	-1
1	

r_2

Figure 7.21 Masks used for the Roberts operators.

2.1.2. Prewitt's edge detector

-1	-1	-1
1	1	1

p_1

-1		1
-1		1
-1		1

p_2

Figure 7.22 Prewitt edge detector masks.

2.1.3. Sobel's edge detector

-1	-2	-1
1	2	1

s_1

-1		1
-2		2
-1		1

s_2

Figure 7.23 Sobel edge detector masks.

2.1.4. Frei and Chen's gradient operator

-1	$-\sqrt{2}$	-1
1	$\sqrt{2}$	1

f_1

-1		1
$-\sqrt{2}$		$\sqrt{2}$
-1		1

f_2

Figure 7.24 Frei and Chen gradient masks.

2.1.5. Kirsch's compass operator

-3	-3	5
-3		5
-3	-3	5

k_0

-3	5	5
-3		5
-3	-3	-3

k_1

5	5	5
-3		-3
-3	-3	-3

k_2

5	5	-3
5		-3
-3	-3	-3

k_3

5	-3	-3
5		-3
5	-3	-3

k_4

-3	-3	-3
5		-3
5	5	-3

k_5

-3	-3	-3
-3		-3
5	5	5

k_6

-3	-3	-3
-3		5
-3	5	5

k_7

Figure 7.25 Kirsch compass masks.

2.1.6. Robinson's compass operator

-1		1
-2		2
-1		1

r_0

	1	2
-1		1
-2	-1	

r_1

1	2	1
-1	-2	-1

r_2

2	1	
1		-1
	-1	-2

r_3

1		-1
2		-2
1		-1

r_4

	-1	-2
1		-1
2	1	

r_5

-1	-2	-1
1	2	1

r_6

-2	-1	
-1		1
	1	2

r_7

Figure 7.26 Robinson compass masks.

2.1.7. Nevatia-Babu 5x5 operator

100	100	100	100	100
100	100	100	100	100
0	0	0	0	0
-100	-100	-100	-100	-100
-100	-100	-100	-100	-100

0°

100	100	100	100	100
100	100	100	78	-32
100	92	0	-92	-100
32	-78	-100	-100	-100
-100	-100	-100	-100	-100

30°

100	100	100	32	-100
100	100	92	-78	-100
100	100	0	-100	-100
100	78	-92	-100	-100
100	-32	-100	-100	-100

60°

-100	-100	0	100	100
-100	-100	0	100	100
-100	-100	0	100	100
-100	-100	0	100	100
-100	-100	0	100	100

-90°

-100	32	100	100	100
-100	-78	92	100	100
-100	-100	0	100	100
-100	-100	-92	78	100
-100	-100	-100	-32	100

-60°

100	100	100	100	100
-32	78	100	100	100
-100	-92	0	92	100
-100	-100	-100	-78	32
-100	-100	-100	-100	-100

-30°

Figure 7.27 Nevatia-Babu 5 × 5 compass template masks.

2.2. Code Fragments

2.2.1. Code fragments of this homework

```python
def getRobertsImage(originalImage, threshold):
    """
    :type originalImage: Image (from PIL)
    :type threshold: float
    :return type: Image (from PIL)
    """
    from PIL import Image
    import math
    # New image with the same size and 'binary' format.
    robertsImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Calculate x0, y0, x1, y1 and avoid out of image range.
            x0 = c
            y0 = r
            x1 = min(c + 1, originalImage.size[0] - 1)
            y1 = min(r + 1, originalImage.size[1] - 1)
            # Calculate r1 and r2 of Robert.
            r1 = -originalImage.getpixel((x0, y0)) + originalImage.getpixel((x1, y1))
            r2 = -originalImage.getpixel((x1, y0)) + originalImage.getpixel((x0, y1))
            # Calulate Grandient magnitude.
            magnitude = int(math.sqrt(r1 ** 2 + r2 ** 2))
            # Binarize with threshold.
            if (magnitude >= threshold):
                robertsImage.putpixel((c, r), 0)
            else:
                robertsImage.putpixel((c, r), 1)
    return robertsImage
```

Figure 2.2.1.1. Code of Robert's operator.

```python
def getPrewittImage(originalImage, threshold):
    """
    :type originalImage: Image (from PIL)
    :type threshold: float
    :return type: Image (from PIL)
    """
    from PIL import Image
    import math
    # New image with the same size and 'binary' format.
    prewittImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
            x0 = max(c - 1, 0)
            y0 = max(r - 1, 0)
            x1 = c
            y1 = r
            x2 = min(c + 1, originalImage.size[0] - 1)
            y2 = min(r + 1, originalImage.size[1] - 1)
            # Calculate p1 and p2 of Prewitt.
            p1 = -originalImage.getpixel((x0, y0)) - originalImage.getpixel((x1, y0)) - originalImage.getpixel((x2, y0))\
                + originalImage.getpixel((x0, y2)) + originalImage.getpixel((x1, y2)) + originalImage.getpixel((x2, y2))
            p2 = -originalImage.getpixel((x0, y0)) - originalImage.getpixel((x0, y1)) - originalImage.getpixel((x0, y2))\
                + originalImage.getpixel((x2, y0)) + originalImage.getpixel((x2, y1)) + originalImage.getpixel((x2, y2))
            # Calulate Grandient magnitude.
            magnitude = int(math.sqrt(p1 ** 2 + p2 ** 2))
            # Binarize with threshold.
            if (magnitude >= threshold):
                prewittImage.putpixel((c, r), 0)
            else:
                prewittImage.putpixel((c, r), 1)
    return prewittImage
```

Figure 2.2.1.2. Code of Prewitt's edge detector.

```python
def getSobelImage(originalImage, threshold):
    """
    :type originalImage: Image (from PIL)
    :type threshold: float
    :return type: Image (from PIL)
    """
    from PIL import Image
    import math
    # New image with the same size and 'binary' format.
    sobelImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
            x0 = max(c - 1, 0)
            y0 = max(r - 1, 0)
            x1 = c
            y1 = r
            x2 = min(c + 1, originalImage.size[0] - 1)
            y2 = min(r + 1, originalImage.size[1] - 1)
            # Calculate p1 and p2 of Sobel.
            p1 = -originalImage.getpixel((x0, y0)) - 2 * originalImage.getpixel((x1, y0)) - originalImage.getpixel((x2, y0))\
                 + originalImage.getpixel((x0, y2)) + 2 * originalImage.getpixel((x1, y2)) + originalImage.getpixel((x2, y2))
            p2 = -originalImage.getpixel((x0, y0)) - 2 * originalImage.getpixel((x0, y1)) - originalImage.getpixel((x0, y2))\
                 + originalImage.getpixel((x2, y0)) + 2 * originalImage.getpixel((x2, y1)) + originalImage.getpixel((x2, y2))
            # Calulate Grandient magnitude.
            magnitude = int(math.sqrt(p1 ** 2 + p2 ** 2))
            # Binarize with threshold.
            if (magnitude >= threshold):
                sobelImage.putpixel((c, r), 0)
            else:
                sobelImage.putpixel((c, r), 1)
    return sobelImage
```

Figure 2.2.1.3. Code of Sobel's edge detector.

```python
def getFreiChenImage(originalImage, threshold):
    """
    :type originalImage: Image (from PIL)
    :type threshold: float
    :return type: Image (from PIL)
    """
    from PIL import Image
    import math
    # New image with the same size and 'binary' format.
    FreiChenImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
            x0 = max(c - 1, 0)
            y0 = max(r - 1, 0)
            x1 = c
            y1 = r
            x2 = min(c + 1, originalImage.size[0] - 1)
            y2 = min(r + 1, originalImage.size[1] - 1)
            # Calculate p1 and p2 of FreiChen.
            p1 = -originalImage.getpixel((x0, y0)) - math.sqrt(2) * originalImage.getpixel((x1, y0)) - originalImage.getpixel((x2, y0))\
                 + originalImage.getpixel((x0, y2)) + math.sqrt(2) * originalImage.getpixel((x1, y2)) + originalImage.getpixel((x2, y2))
            p2 = -originalImage.getpixel((x0, y0)) - math.sqrt(2) * originalImage.getpixel((x0, y1)) - originalImage.getpixel((x0, y2))\
                 + originalImage.getpixel((x2, y0)) + math.sqrt(2) * originalImage.getpixel((x2, y1)) + originalImage.getpixel((x2, y2))
            # Calulate Grandient magnitude.
            magnitude = int(math.sqrt(p1 ** 2 + p2 ** 2))
            # Binarize with threshold.
            if (magnitude >= threshold):
                FreiChenImage.putpixel((c, r), 0)
            else:
                FreiChenImage.putpixel((c, r), 1)
    return FreiChenImage
```

Figure 2.2.1.4. Code of Frei and Chen's gradient operator.

```python
def getKirschImage(originalImage, threshold):
    """
    :type originalImage: Image (from PIL)
    :type threshold: float
    :return type: Image (from PIL)
    """
    from PIL import Image
    import numpy as np
    import math
    # New image with the same size and 'binary' format.
    KirschImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
            x0 = max(c - 1, 0)
            y0 = max(r - 1, 0)
            x1 = c
            y1 = r
            x2 = min(c + 1, originalImage.size[0] - 1)
            y2 = min(r + 1, originalImage.size[1] - 1)
            # Calculate k0-k7 of Kirsch.
            k = np.zeros(8)
            k[0] = -3 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) + 5 * originalImage.getpixel((x2, y0))\
                 - 3 * originalImage.getpixel((x0, y1)) + 5 * originalImage.getpixel((x2, y1))\
                 - 3 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) + 5 *originalImage.getpixel((x2, y2))
            k[1] = -3 * originalImage.getpixel((x0, y0)) + 5 * originalImage.getpixel((x1, y0)) + 5 * originalImage.getpixel((x2, y0))\
                 - 3 * originalImage.getpixel((x0, y1)) + 5 * originalImage.getpixel((x2, y1))\
                 - 3 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) - 3 *originalImage.getpixel((x2, y2))
            k[2] = 5 * originalImage.getpixel((x0, y0)) + 5 * originalImage.getpixel((x1, y0)) + 5 * originalImage.getpixel((x2, y0))\
                 - 3 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1))\
                 - 3 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) - 3 *originalImage.getpixel((x2, y2))
            k[3] = 5 * originalImage.getpixel((x0, y0)) + 5 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0))\
                 + 5 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1))\
                 - 3 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) - 3 *originalImage.getpixel((x2, y2))
            k[4] = 5 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0))\
                 + 5 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1))\
                 + 5 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) - 3 *originalImage.getpixel((x2, y2))
            k[5] = -3 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0))\
                 + 5 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1))\
                 + 5 * originalImage.getpixel((x0, y2)) + 5 * originalImage.getpixel((x1, y2)) - 3 *originalImage.getpixel((x2, y2))
            k[6] = -3 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0))\
                 - 3 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1))\
                 + 5 * originalImage.getpixel((x0, y2)) + 5 * originalImage.getpixel((x1, y2)) + 5 *originalImage.getpixel((x2, y2))
            k[7] = -3 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0))\
                 - 3 * originalImage.getpixel((x0, y1)) + 5 * originalImage.getpixel((x2, y1))\
                 - 3 * originalImage.getpixel((x0, y2)) + 5 * originalImage.getpixel((x1, y2)) + 5 *originalImage.getpixel((x2, y2))
            # Calulate Grandient magnitude.
            magnitude = max(k)
            # Binarize with threshold.
            if (magnitude >= threshold):
                KirschImage.putpixel((c, r), 0)
            else:
                KirschImage.putpixel((c, r), 1)
    return KirschImage
```

Figure 2.2.1.5. Code of Kirsch's compass operator.

```python
def getRobinsonImage(originalImage, threshold):
    """
    :type originalImage: Image (from PIL)
    :type threshold: float
    :return type: Image (from PIL)
    """
    from PIL import Image
    import numpy as np
    import math
    # New image with the same size and 'binary' format.
    RobinsonImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
            x0 = max(c - 1, 0)
            y0 = max(r - 1, 0)
            x1 = c
            y1 = r
            x2 = min(c + 1, originalImage.size[0] - 1)
            y2 = min(r + 1, originalImage.size[1] - 1)
            # Calculate r0-r7 of Robinson.
            k = np.zeros(8)
            k[0] = -1 * originalImage.getpixel((x0, y0)) - 2 * originalImage.getpixel((x0, y1)) - 1 * originalImage.getpixel((x0, y2))\
                + 1 * originalImage.getpixel((x2, y0)) + 2 * originalImage.getpixel((x2, y1)) + 1 *originalImage.getpixel((x2, y2))
            k[1] = -1 * originalImage.getpixel((x0, y1)) - 2 * originalImage.getpixel((x0, y2)) - 1 * originalImage.getpixel((x1, y2))\
                + 1 * originalImage.getpixel((x1, y0)) + 2 * originalImage.getpixel((x2, y0)) + 1 *originalImage.getpixel((x2, y1))
            k[2] = -1 * originalImage.getpixel((x0, y2)) - 2 * originalImage.getpixel((x1, y2)) - 1 * originalImage.getpixel((x2, y2))\
                + 1 * originalImage.getpixel((x0, y0)) + 2 * originalImage.getpixel((x1, y0)) + 1 *originalImage.getpixel((x2, y0))
            k[3] = -1 * originalImage.getpixel((x1, y2)) - 2 * originalImage.getpixel((x2, y2)) - 1 * originalImage.getpixel((x2, y1))\
                + 1 * originalImage.getpixel((x0, y1)) + 2 * originalImage.getpixel((x0, y0)) + 1 *originalImage.getpixel((x1, y0))
            k[4] = -1 * originalImage.getpixel((x2, y0)) - 2 * originalImage.getpixel((x2, y1)) - 1 * originalImage.getpixel((x2, y2))\
                + 1 * originalImage.getpixel((x0, y0)) + 2 * originalImage.getpixel((x0, y1)) + 1 *originalImage.getpixel((x0, y2))
            k[5] = -1 * originalImage.getpixel((x1, y0)) - 2 * originalImage.getpixel((x2, y0)) - 1 * originalImage.getpixel((x2, y1))\
                + 1 * originalImage.getpixel((x0, y1)) + 2 * originalImage.getpixel((x0, y2)) + 1 *originalImage.getpixel((x1, y2))
            k[6] = -1 * originalImage.getpixel((x0, y0)) - 2 * originalImage.getpixel((x1, y0)) - 1 * originalImage.getpixel((x2, y0))\
                + 1 * originalImage.getpixel((x0, y2)) + 2 * originalImage.getpixel((x1, y2)) + 1 *originalImage.getpixel((x2, y2))
            k[7] = -1 * originalImage.getpixel((x0, y1)) - 2 * originalImage.getpixel((x0, y0)) - 1 * originalImage.getpixel((x1, y0))\
                + 1 * originalImage.getpixel((x1, y2)) + 2 * originalImage.getpixel((x2, y2)) + 1 *originalImage.getpixel((x2, y1))
            # Calulate Grandient magnitude.
            magnitude = max(k)
            # Binarize with threshold.
            if (magnitude >= threshold):
                RobinsonImage.putpixel((c, r), 0)
            else:
                RobinsonImage.putpixel((c, r), 1)
    return RobinsonImage
```

Figure 2.2.1.6. Code of Robinson’s compass operator.

```python
def getNevatiaBabuImage(originalImage, threshold):
    """
    :type originalImage: Image (from PIL)
    :type threshold: float
    :return type: Image (from PIL)
    """
    from PIL import Image
    import numpy as np
    import math
    # New image with the same size and 'binary' format.
    NevatiaBabuImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Calculate x0, y0, x1, y1, x2, y2, x3, y3, x4, y4 and avoid out of image range.
            x0 = max(c - 2, 0)
            y0 = max(r - 2, 0)
            x1 = max(c - 1, 0)
            y1 = max(r - 1, 0)
            x2 = c
            y2 = r
            x3 = min(c + 1, originalImage.size[0] - 1)
            y3 = min(r + 1, originalImage.size[1] - 1)
            x4 = min(c + 2, originalImage.size[0] - 1)
            y4 = min(r + 2, originalImage.size[1] - 1)
            # Get 5x5 neighbors.
            neighbors = [originalImage.getpixel((x0, y0)), originalImage.getpixel((x1, y0)), originalImage.getpixel((x2, y0)), originalImage.getpixel((x3, y0)), originalImage.getpixel((x4, y0)),
                         originalImage.getpixel((x0, y1)), originalImage.getpixel((x1, y1)), originalImage.getpixel((x2, y1)), originalImage.getpixel((x3, y1)), originalImage.getpixel((x4, y1)),
                         originalImage.getpixel((x0, y2)), originalImage.getpixel((x1, y2)), originalImage.getpixel((x2, y2)), originalImage.getpixel((x3, y2)), originalImage.getpixel((x4, y2)),
                         originalImage.getpixel((x0, y3)), originalImage.getpixel((x1, y3)), originalImage.getpixel((x2, y3)), originalImage.getpixel((x3, y3)), originalImage.getpixel((x4, y3)),
                         originalImage.getpixel((x0, y4)), originalImage.getpixel((x1, y4)), originalImage.getpixel((x2, y4)), originalImage.getpixel((x3, y4)), originalImage.getpixel((x4, y4))]
            # Calculate k0-k5 of NevatiaBabu.
            k = np.zeros(6)
            k[0] = (100) * neighbors[0] + (100) * neighbors[1] + (100) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
                   (100) * neighbors[5] + (100) * neighbors[6] + (100) * neighbors[7] + (100) * neighbors[8] + (100) * neighbors[9] + \
                   (0) * neighbors[10] + (0) * neighbors[11] + (0) * neighbors[12] + (0) * neighbors[13] + (0) * neighbors[14] + \
                   (-100) * neighbors[15] + (-100) * neighbors[16] + (-100) * neighbors[17] + (-100) * neighbors[18] + (-100) * neighbors[19] + \
                   (-100) * neighbors[20] + (-100) * neighbors[21] + (-100) * neighbors[22] + (-100) * neighbors[23] + (-100) * neighbors[24]
            k[1] = (100) * neighbors[0] + (100) * neighbors[1] + (100) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
                   (100) * neighbors[5] + (100) * neighbors[6] + (100) * neighbors[7] + (78) * neighbors[8] + (-32) * neighbors[9] + \
                   (100) * neighbors[10] + (92) * neighbors[11] + (0) * neighbors[12] + (-92) * neighbors[13] + (-100) * neighbors[14] + \
                   (32) * neighbors[15] + (-78) * neighbors[16] + (-100) * neighbors[17] + (-100) * neighbors[18] + (-100) * neighbors[19] + \
                   (-100) * neighbors[20] + (-100) * neighbors[21] + (-100) * neighbors[22] + (-100) * neighbors[23] + (-100) * neighbors[24]
            k[2] = (100) * neighbors[0] + (100) * neighbors[1] + (100) * neighbors[2] + (32) * neighbors[3] + (-100) * neighbors[4] + \
                   (100) * neighbors[5] + (100) * neighbors[6] + (92) * neighbors[7] + (-78) * neighbors[8] + (-100) * neighbors[9] + \
                   (100) * neighbors[10] + (100) * neighbors[11] + (0) * neighbors[12] + (-100) * neighbors[13] + (-100) * neighbors[14] + \
                   (100) * neighbors[15] + (78) * neighbors[16] + (-92) * neighbors[17] + (-100) * neighbors[18] + (-100) * neighbors[19] + \
                   (100) * neighbors[20] + (-32) * neighbors[21] + (-100) * neighbors[22] + (-100) * neighbors[23] + (-100) * neighbors[24]
            k[3] = (-100) * neighbors[0] + (-100) * neighbors[1] + (0) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
                   (-100) * neighbors[5] + (-100) * neighbors[6] + (0) * neighbors[7] + (100) * neighbors[8] + (100) * neighbors[9] + \
                   (-100) * neighbors[10] + (-100) * neighbors[11] + (0) * neighbors[12] + (100) * neighbors[13] + (100) * neighbors[14] + \
                   (-100) * neighbors[15] + (-100) * neighbors[16] + (0) * neighbors[17] + (100) * neighbors[18] + (100) * neighbors[19] + \
                   (-100) * neighbors[20] + (-100) * neighbors[21] + (0) * neighbors[22] + (100) * neighbors[23] + (100) * neighbors[24]
            k[4] = (-100) * neighbors[0] + (32) * neighbors[1] + (100) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
                   (-100) * neighbors[5] + (-78) * neighbors[6] + (92) * neighbors[7] + (100) * neighbors[8] + (100) * neighbors[9] + \
                   (-100) * neighbors[10] + (-100) * neighbors[11] + (0) * neighbors[12] + (100) * neighbors[13] + (100) * neighbors[14] + \
                   (-100) * neighbors[15] + (-100) * neighbors[16] + (-92) * neighbors[17] + (78) * neighbors[18] + (100) * neighbors[19] + \
                   (-100) * neighbors[20] + (-100) * neighbors[21] + (-100) * neighbors[22] + (-32) * neighbors[23] + (100) * neighbors[24]
            k[5] = (100) * neighbors[0] + (100) * neighbors[1] + (100) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
                   (-32) * neighbors[5] + (78) * neighbors[6] + (100) * neighbors[7] + (100) * neighbors[8] + (100) * neighbors[9] + \
                   (-100) * neighbors[10] + (-92) * neighbors[11] + (0) * neighbors[12] + (92) * neighbors[13] + (100) * neighbors[14] + \
                   (-100) * neighbors[15] + (-100) * neighbors[16] + (-100) * neighbors[17] + (-78) * neighbors[18] + (32) * neighbors[19] + \
                   (-100) * neighbors[20] + (-100) * neighbors[21] + (-100) * neighbors[22] + (-100) * neighbors[23] + (-100) * neighbors[24]
            # Calulate Grandient magnitude.
            magnitude = max(k)
            # Binarize with threshold.
            if (magnitude >= threshold):
                NevatiaBabuImage.putpixel((c, r), 0)
            else:
                NevatiaBabuImage.putpixel((c, r), 1)
    return NevatiaBabuImage
```

Figure 2.2.1.7. Code of Nevatia-Babu 5x5 operator.

III. RESULTS

3.1. Original Image

Figure 3.1. Original lena.bmp.

3.2. Results of Robert's operator

Figure 3.2.1. Original lena.bmp.

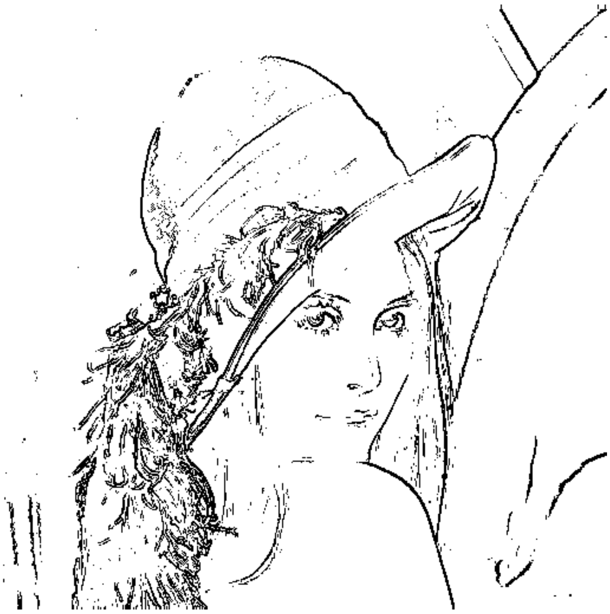

Figure 3.2.2. Robert.bmp.

3.3. Results of Prewitt's edge detector

Figure 3.3.1. Original lena.bmp.

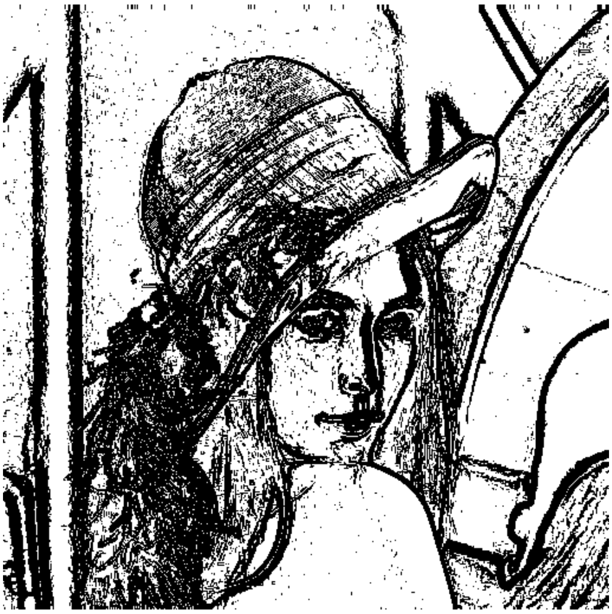

Figure 3.3.2. Prewitt.bmp.

3.4. Results of Sobel's edge detector

Figure 3.4.1. Original lena.bmp.

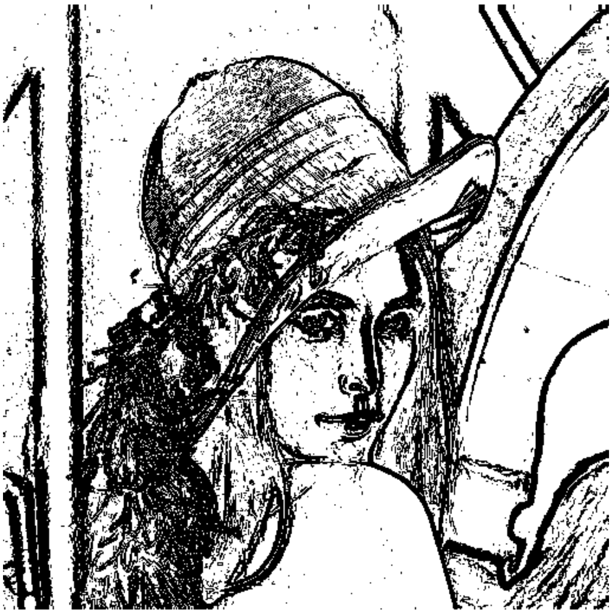

Figure 3.4.2. Sobel.bmp.

3.5. Results of Frei and Chen's gradient operator

Figure 3.5.1. Original lena.bmp.

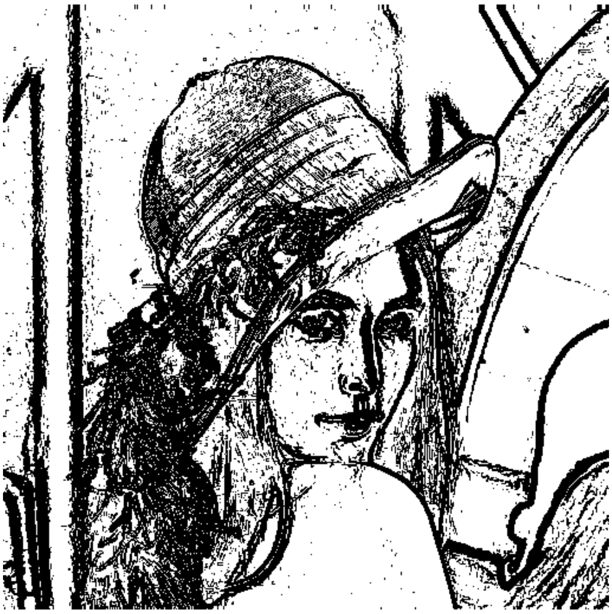

Figure 3.5.2. FreiChen.bmp.

3.6. Results of Kirsch's compass operator

Figure 3.6.1. Original lena.bmp.

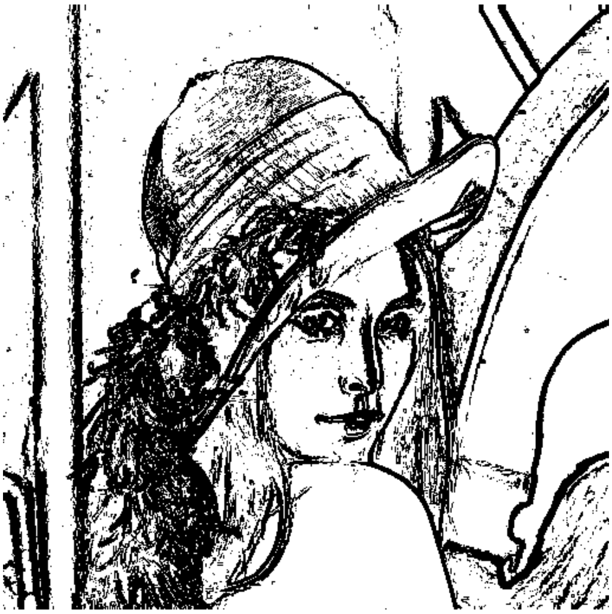

Figure 3.6.2. Kirsch.bmp.

3.7. Results of Robinson's compass operator

Figure 3.7.1. Original lena.bmp.

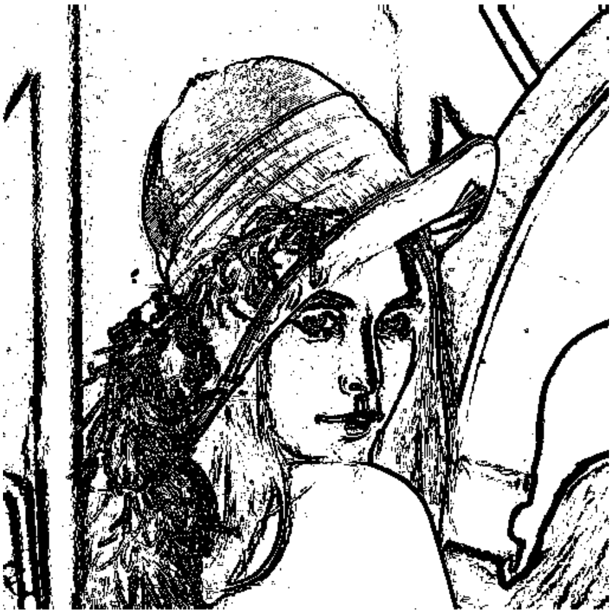

Figure 3.7.2. Robinson.bmp.

3.8. Results of Nevatia-Babu 5x5 operator

Figure 3.8.1. Original lena.bmp.

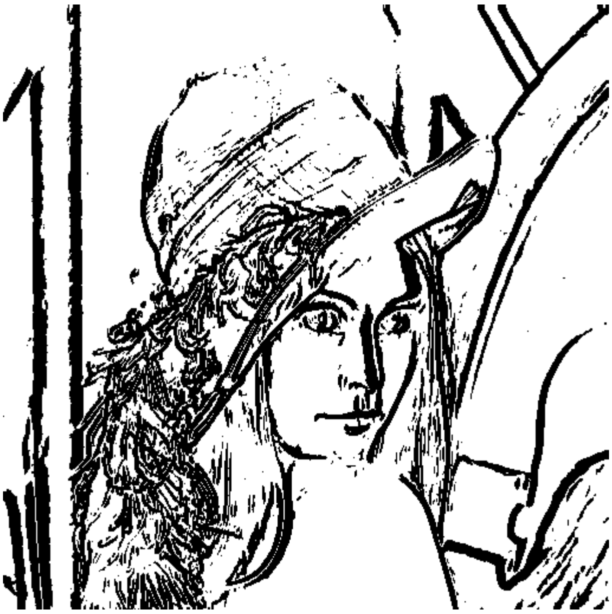

Figure 3.8.2. NevatiaBabu.bmp.