

# 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

<table><tr><td>-3</td><td>-3</td><td>5</td></tr><tr><td>-3</td><td></td><td>5</td></tr><tr><td>-3</td><td>-3</td><td>5</td></tr></table> $k_0$	-3	-3	5	-3		5	-3	-3	5	<table><tr><td>-3</td><td>5</td><td>5</td></tr><tr><td>-3</td><td></td><td>5</td></tr><tr><td>-3</td><td>-3</td><td>-3</td></tr></table> $k_1$	-3	5	5	-3		5	-3	-3	-3	<table><tr><td>5</td><td>5</td><td>5</td></tr><tr><td>-3</td><td></td><td>-3</td></tr><tr><td>-3</td><td>-3</td><td>-3</td></tr></table> $k_2$	5	5	5	-3		-3	-3	-3	-3	<table><tr><td>5</td><td>5</td><td>-3</td></tr><tr><td>5</td><td></td><td>-3</td></tr><tr><td>-3</td><td>-3</td><td>-3</td></tr></table> $k_3$	5	5	-3	5		-3	-3	-3	-3
-3	-3	5																																					
-3		5																																					
-3	-3	5																																					
-3	5	5																																					
-3		5																																					
-3	-3	-3																																					
5	5	5																																					
-3		-3																																					
-3	-3	-3																																					
5	5	-3																																					
5		-3																																					
-3	-3	-3																																					
<table><tr><td>5</td><td>-3</td><td>-3</td></tr><tr><td>5</td><td></td><td>-3</td></tr><tr><td>5</td><td>-3</td><td>-3</td></tr></table> $k_4$	5	-3	-3	5		-3	5	-3	-3	<table><tr><td>-3</td><td>-3</td><td>-3</td></tr><tr><td>5</td><td></td><td>-3</td></tr><tr><td>5</td><td>5</td><td>-3</td></tr></table> $k_5$	-3	-3	-3	5		-3	5	5	-3	<table><tr><td>-3</td><td>-3</td><td>-3</td></tr><tr><td>-3</td><td></td><td>-3</td></tr><tr><td>5</td><td>5</td><td>5</td></tr></table> $k_6$	-3	-3	-3	-3		-3	5	5	5	<table><tr><td>-3</td><td>-3</td><td>-3</td></tr><tr><td>-3</td><td></td><td>5</td></tr><tr><td>-3</td><td>5</td><td>5</td></tr></table> $k_7$	-3	-3	-3	-3		5	-3	5	5
5	-3	-3																																					
5		-3																																					
5	-3	-3																																					
-3	-3	-3																																					
5		-3																																					
5	5	-3																																					
-3	-3	-3																																					
-3		-3																																					
5	5	5																																					
-3	-3	-3																																					
-3		5																																					
-3	5	5																																					

Figure 7.25 Kirsch compass masks.

## 2.1.6. Robinson's compass operator

<table><tr><td>-1</td><td></td><td>1</td></tr><tr><td>-2</td><td></td><td>2</td></tr><tr><td>-1</td><td></td><td>1</td></tr></table> $r_0$	-1		1	-2		2	-1		1	<table><tr><td></td><td>1</td><td>2</td></tr><tr><td>-1</td><td></td><td>1</td></tr><tr><td>-2</td><td>-1</td><td></td></tr></table> $r_1$		1	2	-1		1	-2	-1		<table><tr><td>1</td><td>2</td><td>1</td></tr><tr><td></td><td></td><td></td></tr><tr><td>-1</td><td>-2</td><td>-1</td></tr></table> $r_2$	1	2	1				-1	-2	-1	<table><tr><td>2</td><td>1</td><td></td></tr><tr><td>1</td><td></td><td>-1</td></tr><tr><td></td><td>-1</td><td>-2</td></tr></table> $r_3$	2	1		1		-1		-1	-2
-1		1																																					
-2		2																																					
-1		1																																					
	1	2																																					
-1		1																																					
-2	-1																																						
1	2	1																																					
-1	-2	-1																																					
2	1																																						
1		-1																																					
	-1	-2																																					
<table><tr><td>1</td><td></td><td>-1</td></tr><tr><td>2</td><td></td><td>-2</td></tr><tr><td>1</td><td></td><td>-1</td></tr></table> $r_4$	1		-1	2		-2	1		-1	<table><tr><td></td><td>-1</td><td>-2</td></tr><tr><td>1</td><td></td><td>-1</td></tr><tr><td>2</td><td>1</td><td></td></tr></table> $r_5$		-1	-2	1		-1	2	1		<table><tr><td>-1</td><td>-2</td><td>-1</td></tr><tr><td></td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>1</td></tr></table> $r_6$	-1	-2	-1				1	2	1	<table><tr><td>-2</td><td>-1</td><td></td></tr><tr><td>-1</td><td></td><td>1</td></tr><tr><td></td><td>1</td><td>2</td></tr></table> $r_7$	-2	-1		-1		1		1	2
1		-1																																					
2		-2																																					
1		-1																																					
	-1	-2																																					
1		-1																																					
2	1																																						
-1	-2	-1																																					
1	2	1																																					
-2	-1																																						
-1		1																																					
	1	2																																					

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

60°

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

30°

-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	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

```

35 def getRobertsImage(originalImage, threshold):
36     """
37     :type originalImage: Image (from PIL)
38     :type threshold: float
39     :return type: Image (from PIL)
40     """
41     from PIL import Image
42     import math
43     # New image with the same size and 'binary' format.
44     robertsImage = Image.new('1', originalImage.size)
45     # Scan each column in original image.
46     for c in range(originalImage.size[0]):
47         # Scan each row in original image.
48         for r in range(originalImage.size[1]):
49             # Calculate x0, y0, x1, y1 and avoid out of image range.
50             x0 = c
51             y0 = r
52             x1 = min(c + 1, originalImage.size[0] - 1)
53             y1 = min(r + 1, originalImage.size[1] - 1)
54             # Calculate r1 and r2 of Robert.
55             r1 = -originalImage.getpixel((x0, y0)) + originalImage.getpixel((x1, y1))
56             r2 = -originalImage.getpixel((x1, y0)) + originalImage.getpixel((x0, y1))
57             # Calculate Gradient magnitude.
58             magnitude = int(math.sqrt(r1 ** 2 + r2 ** 2))
59             # Binarize with threshold.
60             if (magnitude >= threshold):
61                 robertsImage.putpixel((c, r), 0)
62             else:
63                 robertsImage.putpixel((c, r), 1)
64     return robertsImage

```

Figure 2.2.1.1. Code of Robert's operator.

```

66 def getPrewittImage(originalImage, threshold):
67     """
68     :type originalImage: Image (from PIL)
69     :type threshold: float
70     :return type: Image (from PIL)
71     """
72     from PIL import Image
73     import math
74     # New image with the same size and 'binary' format.
75     prewittImage = Image.new('1', originalImage.size)
76     # Scan each column in original image.
77     for c in range(originalImage.size[0]):
78         # Scan each row in original image.
79         for r in range(originalImage.size[1]):
80             # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
81             x0 = max(c - 1, 0)
82             y0 = max(r - 1, 0)
83             x1 = c
84             y1 = r
85             x2 = min(c + 1, originalImage.size[0] - 1)
86             y2 = min(r + 1, originalImage.size[1] - 1)
87             # Calculate p1 and p2 of Prewitt.
88             p1 = -originalImage.getpixel((x0, y0)) - originalImage.getpixel((x1, y0)) - originalImage.getpixel((x2, y0))\
89                 + originalImage.getpixel((x0, y2)) + originalImage.getpixel((x1, y2)) + originalImage.getpixel((x2, y2))
90             p2 = -originalImage.getpixel((x0, y0)) - originalImage.getpixel((x0, y1)) - originalImage.getpixel((x0, y2))\
91                 + originalImage.getpixel((x2, y0)) + originalImage.getpixel((x2, y1)) + originalImage.getpixel((x2, y2))
92             # Calculate Gradient magnitude.
93             magnitude = int(math.sqrt(p1 ** 2 + p2 ** 2))
94             # Binarize with threshold.
95             if (magnitude >= threshold):
96                 prewittImage.putpixel((c, r), 0)
97             else:
98                 prewittImage.putpixel((c, r), 1)
99     return prewittImage

```

Figure 2.2.1.2. Code of Prewitt's edge detector.

```

101 def getSobelImage(originalImage, threshold):
102     """
103     :type originalImage: Image (from PIL)
104     :type threshold: float
105     :return type: Image (from PIL)
106     """
107     from PIL import Image
108     import math
109     # New image with the same size and 'binary' format.
110     sobelImage = Image.new('1', originalImage.size)
111     # Scan each column in original image.
112     for c in range(originalImage.size[0]):
113         # Scan each row in original image.
114         for r in range(originalImage.size[1]):
115             # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
116             x0 = max(c - 1, 0)
117             y0 = max(r - 1, 0)
118             x1 = c
119             y1 = r
120             x2 = min(c + 1, originalImage.size[0] - 1)
121             y2 = min(r + 1, originalImage.size[1] - 1)
122             # Calculate p1 and p2 of Sobel.
123             p1 = -originalImage.getpixel((x0, y0)) - 2 * originalImage.getpixel((x1, y0)) - originalImage.getpixel((x2, y0)) \
124                 + originalImage.getpixel((x0, y2)) + 2 * originalImage.getpixel((x1, y2)) + originalImage.getpixel((x2, y2))
125             p2 = -originalImage.getpixel((x0, y0)) - 2 * originalImage.getpixel((x0, y1)) - originalImage.getpixel((x0, y2)) \
126                 + originalImage.getpixel((x2, y0)) + 2 * originalImage.getpixel((x2, y1)) + originalImage.getpixel((x2, y2))
127             # Calculate Gradient magnitude.
128             magnitude = int(math.sqrt(p1 ** 2 + p2 ** 2))
129             # Binarize with threshold.
130             if (magnitude >= threshold):
131                 sobelImage.putpixel((c, r), 0)
132             else:
133                 sobelImage.putpixel((c, r), 1)
134     return sobelImage

```

Figure 2.2.1.3. Code of Sobel's edge detector.

```

136 def getFreiChenImage(originalImage, threshold):
137     """
138     :type originalImage: Image (from PIL)
139     :type threshold: float
140     :return type: Image (from PIL)
141     """
142     from PIL import Image
143     import math
144     # New image with the same size and 'binary' format.
145     FreiChenImage = Image.new('1', originalImage.size)
146     # Scan each column in original image.
147     for c in range(originalImage.size[0]):
148         # Scan each row in original image.
149         for r in range(originalImage.size[1]):
150             # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
151             x0 = max(c - 1, 0)
152             y0 = max(r - 1, 0)
153             x1 = c
154             y1 = r
155             x2 = min(c + 1, originalImage.size[0] - 1)
156             y2 = min(r + 1, originalImage.size[1] - 1)
157             # Calculate p1 and p2 of FreiChen.
158             p1 = -originalImage.getpixel((x0, y0)) - math.sqrt(2) * originalImage.getpixel((x1, y0)) - originalImage.getpixel((x2, y0)) \
159                 + originalImage.getpixel((x0, y2)) + math.sqrt(2) * originalImage.getpixel((x1, y2)) + originalImage.getpixel((x2, y2))
160             p2 = -originalImage.getpixel((x0, y0)) - math.sqrt(2) * originalImage.getpixel((x0, y1)) - originalImage.getpixel((x0, y2)) \
161                 + originalImage.getpixel((x2, y0)) + math.sqrt(2) * originalImage.getpixel((x2, y1)) + originalImage.getpixel((x2, y2))
162             # Calculate Gradient magnitude.
163             magnitude = int(math.sqrt(p1 ** 2 + p2 ** 2))
164             # Binarize with threshold.
165             if (magnitude >= threshold):
166                 FreiChenImage.putpixel((c, r), 0)
167             else:
168                 FreiChenImage.putpixel((c, r), 1)
169     return FreiChenImage

```

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

```

171 def getKirschImage(originalImage, threshold):
172     """
173     :type originalImage: Image (from PIL)
174     :type threshold: float
175     :return type: Image (from PIL)
176     """
177     from PIL import Image
178     import numpy as np
179     import math
180     # New image with the same size and 'binary' format.
181     KirschImage = Image.new('1', originalImage.size)
182     # Scan each column in original image.
183     for c in range(originalImage.size[0]):
184         # Scan each row in original image.
185         for r in range(originalImage.size[1]):
186             # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
187             x0 = max(c - 1, 0)
188             y0 = max(r - 1, 0)
189             x1 = c
190             y1 = r
191             x2 = min(c + 1, originalImage.size[0] - 1)
192             y2 = min(r + 1, originalImage.size[1] - 1)
193             # Calculate k0-k7 of Kirsch.
194             k = np.zeros(8)
195             k[0] = -3 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) + 5 * originalImage.getpixel((x2, y0)) \
196                 - 3 * originalImage.getpixel((x0, y1)) + 5 * originalImage.getpixel((x2, y1)) \
197                 - 3 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) + 5 * originalImage.getpixel((x2, y2))
198             k[1] = -3 * originalImage.getpixel((x0, y0)) + 5 * originalImage.getpixel((x1, y0)) + 5 * originalImage.getpixel((x2, y0)) \
199                 - 3 * originalImage.getpixel((x0, y1)) + 5 * originalImage.getpixel((x2, y1)) \
200                 - 3 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) - 3 * originalImage.getpixel((x2, y2))
201             k[2] = 5 * originalImage.getpixel((x0, y0)) + 5 * originalImage.getpixel((x1, y0)) + 5 * originalImage.getpixel((x2, y0)) \
202                 - 3 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1)) \
203                 - 3 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) - 3 * originalImage.getpixel((x2, y2))
204             k[3] = 5 * originalImage.getpixel((x0, y0)) + 5 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0)) \
205                 + 5 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1)) \
206                 - 3 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) - 3 * originalImage.getpixel((x2, y2))
207             k[4] = 5 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0)) \
208                 + 5 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1)) \
209                 + 5 * originalImage.getpixel((x0, y2)) - 3 * originalImage.getpixel((x1, y2)) - 3 * originalImage.getpixel((x2, y2))
210             k[5] = -3 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0)) \
211                 + 5 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1)) \
212                 + 5 * originalImage.getpixel((x0, y2)) + 5 * originalImage.getpixel((x1, y2)) - 3 * originalImage.getpixel((x2, y2))
213             k[6] = -3 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0)) \
214                 - 3 * originalImage.getpixel((x0, y1)) - 3 * originalImage.getpixel((x2, y1)) \
215                 + 5 * originalImage.getpixel((x0, y2)) + 5 * originalImage.getpixel((x1, y2)) + 5 * originalImage.getpixel((x2, y2))
216             k[7] = -3 * originalImage.getpixel((x0, y0)) - 3 * originalImage.getpixel((x1, y0)) - 3 * originalImage.getpixel((x2, y0)) \
217                 - 3 * originalImage.getpixel((x0, y1)) + 5 * originalImage.getpixel((x2, y1)) \
218                 - 3 * originalImage.getpixel((x0, y2)) + 5 * originalImage.getpixel((x1, y2)) + 5 * originalImage.getpixel((x2, y2))
219             # Calculate Gradient magnitude.
220             magnitude = max(k)
221             # Binarize with threshold.
222             if (magnitude >= threshold):
223                 KirschImage.putpixel((c, r), 0)
224             else:
225                 KirschImage.putpixel((c, r), 1)
226     return KirschImage

```

Figure 2.2.1.5. Code of Kirsch's compass operator.

```

228 def getRobinsonImage(originalImage, threshold):
229     """
230     :type originalImage: Image (from PIL)
231     :type threshold: float
232     :return type: Image (from PIL)
233     """
234     from PIL import Image
235     import numpy as np
236     import math
237     # New image with the same size and 'binary' format.
238     RobinsonImage = Image.new('1', originalImage.size)
239     # Scan each column in original image.
240     for c in range(originalImage.size[0]):
241         # Scan each row in original image.
242         for r in range(originalImage.size[1]):
243             # Calculate x0, y0, x1, y1, x2, y2 and avoid out of image range.
244             x0 = max(c - 1, 0)
245             y0 = max(r - 1, 0)
246             x1 = c
247             y1 = r
248             x2 = min(c + 1, originalImage.size[0] - 1)
249             y2 = min(r + 1, originalImage.size[1] - 1)
250             # Calculate r0-r7 of Robinson.
251             k = np.zeros(8)
252             k[0] = -1 * originalImage.getpixel((x0, y0)) - 2 * originalImage.getpixel((x0, y1)) - 1 * originalImage.getpixel((x0, y2))\
253                 + 1 * originalImage.getpixel((x2, y0)) + 2 * originalImage.getpixel((x2, y1)) + 1 * originalImage.getpixel((x2, y2))
254             k[1] = -1 * originalImage.getpixel((x0, y1)) - 2 * originalImage.getpixel((x0, y2)) - 1 * originalImage.getpixel((x1, y2))\
255                 + 1 * originalImage.getpixel((x1, y0)) + 2 * originalImage.getpixel((x2, y0)) + 1 * originalImage.getpixel((x2, y1))
256             k[2] = -1 * originalImage.getpixel((x0, y2)) - 2 * originalImage.getpixel((x1, y2)) - 1 * originalImage.getpixel((x2, y2))\
257                 + 1 * originalImage.getpixel((x0, y0)) + 2 * originalImage.getpixel((x1, y0)) + 1 * originalImage.getpixel((x2, y0))
258             k[3] = -1 * originalImage.getpixel((x1, y2)) - 2 * originalImage.getpixel((x2, y2)) - 1 * originalImage.getpixel((x2, y1))\
259                 + 1 * originalImage.getpixel((x0, y1)) + 2 * originalImage.getpixel((x0, y0)) + 1 * originalImage.getpixel((x1, y0))
260             k[4] = -1 * originalImage.getpixel((x2, y0)) - 2 * originalImage.getpixel((x2, y1)) - 1 * originalImage.getpixel((x2, y2))\
261                 + 1 * originalImage.getpixel((x0, y0)) + 2 * originalImage.getpixel((x0, y1)) + 1 * originalImage.getpixel((x0, y2))
262             k[5] = -1 * originalImage.getpixel((x1, y0)) - 2 * originalImage.getpixel((x2, y0)) - 1 * originalImage.getpixel((x2, y1))\
263                 + 1 * originalImage.getpixel((x0, y1)) + 2 * originalImage.getpixel((x0, y2)) + 1 * originalImage.getpixel((x1, y2))
264             k[6] = -1 * originalImage.getpixel((x0, y0)) - 2 * originalImage.getpixel((x1, y0)) - 1 * originalImage.getpixel((x2, y0))\
265                 + 1 * originalImage.getpixel((x0, y2)) + 2 * originalImage.getpixel((x1, y2)) + 1 * originalImage.getpixel((x2, y2))
266             k[7] = -1 * originalImage.getpixel((x0, y1)) - 2 * originalImage.getpixel((x0, y0)) - 1 * originalImage.getpixel((x1, y0))\
267                 + 1 * originalImage.getpixel((x1, y2)) + 2 * originalImage.getpixel((x2, y2)) + 1 * originalImage.getpixel((x2, y1))
268             # Calculate Gradient magnitude.
269             magnitude = max(k)
270             # Binarize with threshold.
271             if (magnitude >= threshold):
272                 RobinsonImage.putpixel((c, r), 0)
273             else:
274                 RobinsonImage.putpixel((c, r), 1)
275     return RobinsonImage

```

Figure 2.2.1.6. Code of Robinson's compass operator.



```

277 def getNevatiaBabuImage(originalImage, threshold):
278     """
279     :type originalImage: Image (from PIL)
280     :type threshold: float
281     :return type: Image (from PIL)
282     """
283     from PIL import Image
284     import numpy as np
285     import math
286     # New image with the same size and 'binary' format.
287     NevatiaBabuImage = Image.new('1', originalImage.size)
288     # Scan each column in original image.
289     for c in range(originalImage.size[0]):
290         # Scan each row in original image.
291         for r in range(originalImage.size[1]):
292             # Calculate x0, y0, x1, y1, x2, y2, x3, y3, x4, y4 and avoid out of image range.
293             x0 = max(c - 2, 0)
294             y0 = max(r - 2, 0)
295             x1 = max(c - 1, 0)
296             y1 = max(r - 1, 0)
297             x2 = c
298             y2 = r
299             x3 = min(c + 1, originalImage.size[0] - 1)
300             y3 = min(r + 1, originalImage.size[1] - 1)
301             x4 = min(c + 2, originalImage.size[0] - 1)
302             y4 = min(r + 2, originalImage.size[1] - 1)
303             # Get 5x5 neighbors.
304             neighbors = [originalImage.getpixel((x0, y0)), originalImage.getpixel((x1, y0)), originalImage.getpixel((x2, y0)), originalImage.getpixel((x3, y0)), originalImage.getpixel((x4, y0)),
305                         originalImage.getpixel((x0, y1)), originalImage.getpixel((x1, y1)), originalImage.getpixel((x2, y1)), originalImage.getpixel((x3, y1)), originalImage.getpixel((x4, y1)),
306                         originalImage.getpixel((x0, y2)), originalImage.getpixel((x1, y2)), originalImage.getpixel((x2, y2)), originalImage.getpixel((x3, y2)), originalImage.getpixel((x4, y2)),
307                         originalImage.getpixel((x0, y3)), originalImage.getpixel((x1, y3)), originalImage.getpixel((x2, y3)), originalImage.getpixel((x3, y3)), originalImage.getpixel((x4, y3)),
308                         originalImage.getpixel((x0, y4)), originalImage.getpixel((x1, y4)), originalImage.getpixel((x2, y4)), originalImage.getpixel((x3, y4)), originalImage.getpixel((x4, y4))]
309             # Calculate k0-k5 of NevatiaBabu.
310             k = np.zeros(6)
311             k[0] = (100) * neighbors[0] + (100) * neighbors[1] + (100) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
312                   (100) * neighbors[5] + (100) * neighbors[6] + (100) * neighbors[7] + (100) * neighbors[8] + (100) * neighbors[9] + \
313                   (0) * neighbors[10] + (0) * neighbors[11] + (0) * neighbors[12] + (0) * neighbors[13] + (0) * neighbors[14] + \
314                   (-100) * neighbors[15] + (-100) * neighbors[16] + (-100) * neighbors[17] + (-100) * neighbors[18] + (-100) * neighbors[19] + \
315                   (-100) * neighbors[20] + (-100) * neighbors[21] + (-100) * neighbors[22] + (-100) * neighbors[23] + (-100) * neighbors[24]
316             k[1] = (100) * neighbors[0] + (100) * neighbors[1] + (100) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
317                   (100) * neighbors[5] + (100) * neighbors[6] + (100) * neighbors[7] + (78) * neighbors[8] + (-32) * neighbors[9] + \
318                   (100) * neighbors[10] + (92) * neighbors[11] + (0) * neighbors[12] + (-92) * neighbors[13] + (-100) * neighbors[14] + \
319                   (32) * neighbors[15] + (-78) * neighbors[16] + (-100) * neighbors[17] + (-100) * neighbors[18] + (-100) * neighbors[19] + \
320                   (-100) * neighbors[20] + (-100) * neighbors[21] + (-100) * neighbors[22] + (-100) * neighbors[23] + (-100) * neighbors[24]
321             k[2] = (100) * neighbors[0] + (100) * neighbors[1] + (100) * neighbors[2] + (32) * neighbors[3] + (-100) * neighbors[4] + \
322                   (100) * neighbors[5] + (100) * neighbors[6] + (92) * neighbors[7] + (-78) * neighbors[8] + (-100) * neighbors[9] + \
323                   (100) * neighbors[10] + (100) * neighbors[11] + (0) * neighbors[12] + (-100) * neighbors[13] + (-100) * neighbors[14] + \
324                   (100) * neighbors[15] + (78) * neighbors[16] + (-92) * neighbors[17] + (-100) * neighbors[18] + (-100) * neighbors[19] + \
325                   (100) * neighbors[20] + (-32) * neighbors[21] + (-100) * neighbors[22] + (-100) * neighbors[23] + (-100) * neighbors[24]
326             k[3] = (-100) * neighbors[0] + (-100) * neighbors[1] + (0) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
327                   (-100) * neighbors[5] + (-100) * neighbors[6] + (0) * neighbors[7] + (100) * neighbors[8] + (100) * neighbors[9] + \
328                   (-100) * neighbors[10] + (-100) * neighbors[11] + (0) * neighbors[12] + (100) * neighbors[13] + (100) * neighbors[14] + \
329                   (-100) * neighbors[15] + (-100) * neighbors[16] + (0) * neighbors[17] + (100) * neighbors[18] + (100) * neighbors[19] + \
330                   (-100) * neighbors[20] + (-100) * neighbors[21] + (0) * neighbors[22] + (100) * neighbors[23] + (100) * neighbors[24]
331             k[4] = (-100) * neighbors[0] + (32) * neighbors[1] + (100) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
332                   (-100) * neighbors[5] + (-78) * neighbors[6] + (92) * neighbors[7] + (100) * neighbors[8] + (100) * neighbors[9] + \
333                   (-100) * neighbors[10] + (-100) * neighbors[11] + (0) * neighbors[12] + (100) * neighbors[13] + (100) * neighbors[14] + \
334                   (-100) * neighbors[15] + (-100) * neighbors[16] + (-92) * neighbors[17] + (78) * neighbors[18] + (100) * neighbors[19] + \
335                   (-100) * neighbors[20] + (-100) * neighbors[21] + (-100) * neighbors[22] + (-32) * neighbors[23] + (100) * neighbors[24]
336             k[5] = (100) * neighbors[0] + (100) * neighbors[1] + (100) * neighbors[2] + (100) * neighbors[3] + (100) * neighbors[4] + \
337                   (-32) * neighbors[5] + (78) * neighbors[6] + (100) * neighbors[7] + (100) * neighbors[8] + (100) * neighbors[9] + \
338                   (-100) * neighbors[10] + (-92) * neighbors[11] + (0) * neighbors[12] + (92) * neighbors[13] + (100) * neighbors[14] + \
339                   (-100) * neighbors[15] + (-100) * neighbors[16] + (-100) * neighbors[17] + (-78) * neighbors[18] + (32) * neighbors[19] + \
340                   (-100) * neighbors[20] + (-100) * neighbors[21] + (-100) * neighbors[22] + (-100) * neighbors[23] + (-100) * neighbors[24]
341             # Calculate Grandient magnitude.
342             magnitude = max(k)
343             # Binarize with threshold.
344             if (magnitude >= threshold):
345                 NevatiaBabuImage.putpixel((c, r), 0)
346             else:
347                 NevatiaBabuImage.putpixel((c, r), 1)
348     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.