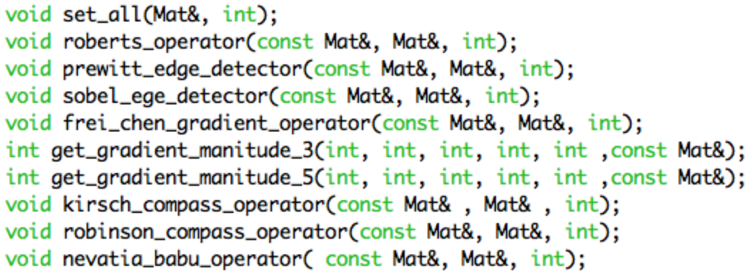
Computer Vision hw_9

By R01922124 許彥彬

1. Intro of this homework:

   This homework is to use different edge detectors with different mask, the following are functions that is used in this homework.



```
void set_all(Mat&, int);
void roberts_operator(const Mat&, Mat&, int);
void prewitt_edge_detector(const Mat&, Mat&, int);
void sobel_ege_detector(const Mat&, Mat&, int);
void frei_chen_gradient_operator(const Mat&, Mat&, int);
int get_gradient_manitude_3(int, int, int, int, int ,const Mat&);
int get_gradient_manitude_5(int, int, int, int, int ,const Mat&);
void kirsch_compass_operator(const Mat& , Mat& , int);
void robinson_compass_operator(const Mat&, Mat&, int);
void nevatia_babu_operator( const Mat&, Mat&, int);
```

2. Edge detectors in this homework

   I. Robert's Edge Detector

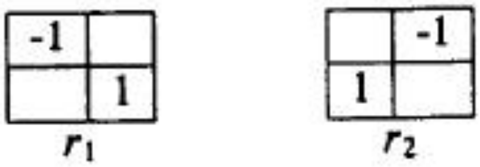
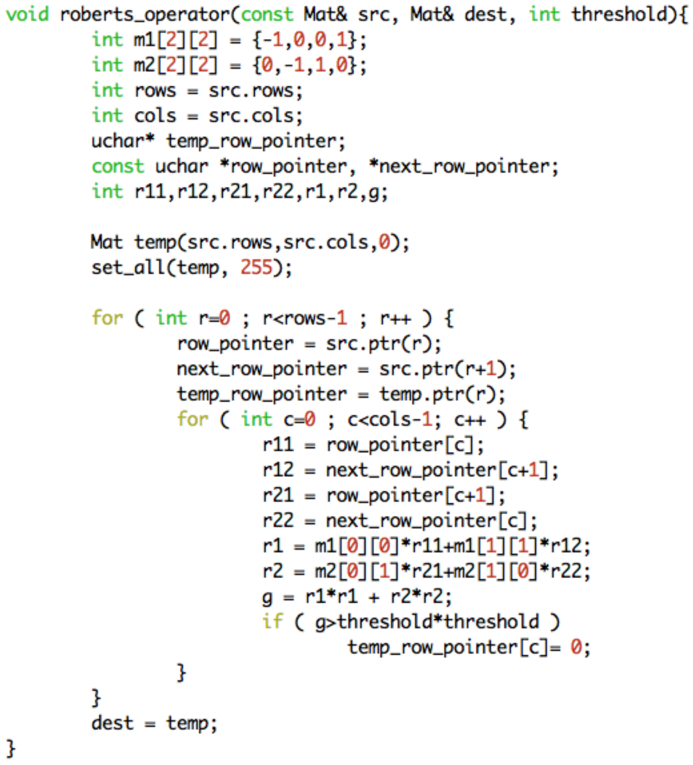


Figure 7.21 Masks used for the Roberts operators.

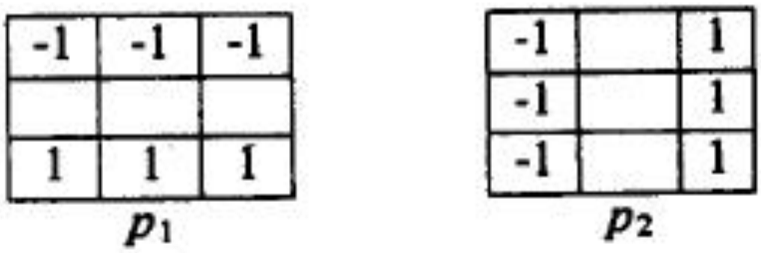


```
void roberts_operator(const Mat& src, Mat& dest, int threshold){
        int m1[2][2] = {-1,0,0,1};
        int m2[2][2] = {0,-1,1,0};
        int rows = src.rows;
        int cols = src.cols;
        uchar* temp_row_pointer;
        const uchar *row_pointer, *next_row_pointer;
        int r11,r12,r21,r22,r1,r2,g;

        Mat temp(src.rows,src.cols,0);
        set_all(temp, 255);

        for ( int r=0 ; r<rows-1 ; r++ ) {
                row_pointer = src.ptr(r);
                next_row_pointer = src.ptr(r+1);
                temp_row_pointer = temp.ptr(r);
                for ( int c=0 ; c<cols-1; c++ ) {
                        r11 = row_pointer[c];
                        r12 = next_row_pointer[c+1];
                        r21 = row_pointer[c+1];
                        r22 = next_row_pointer[c];
                        r1 = m1[0][0]*r11+m1[1][1]*r12;
                        r2 = m2[0][1]*r21+m2[1][0]*r22;
                        g = r1*r1 + r2*r2;
                        if ( g>threshold*threshold )
                                temp_row_pointer[c]= 0;
                }
        }
        dest = temp;
}
```

II. Prewitt’s Edge Detector



-1	-1	-1
1	1	1

p_1

-1		1
-1		1
-1		1

p_2

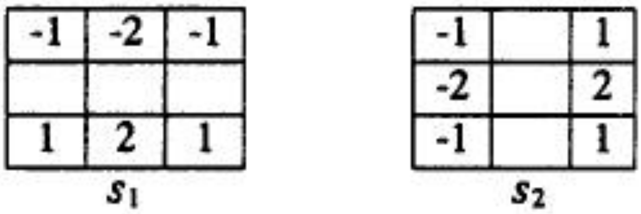
Figure 7.22 Prewitt edge detector masks.

```
void prewitt_edge_detector (const Mat& src, Mat& dest, int threshold ) {
        int m1[3][3] = {-1,-1,-1,0,0,0,1,1,1}, m2[3][3] = {-1,0,1,-1,0,1,-1,0,1};
        int rows = src.rows;
        int cols = src.cols;
        Mat temp(rows,cols,0);
        uchar* temp_row_pointer;
        int cr = 1, cc=1;
        int p1,p2,g ,tr,tc;

        set_all(temp, 255);

        for ( int r = 1 ; r<rows-1 ; r++ ) {
                temp_row_pointer = temp.ptr(r);
                for ( int c=1 ; c<cols-1 ; c++ ){
                //calc p1 and p2;
                        p1 = p2 = 0;
                        for ( int i=0 ; i<3 ; i++ ) {
                                int tr = i-cr + r;
                                const uchar* p = src.ptr(tr);
                                for ( int j=0 ; j<3 ; j++ ) {
                                        int tc = j-cc + c;
                                        p1 += m1[i][j]*p[tc];
                                        p2 += m2[i][j]*p[tc];
                                }
                        }
                        g = p1*p1+p2*p2;
                        if ( g>threshold*threshold )
                                temp_row_pointer[c] = 0;
                }
        }
        dest = temp;
}
```

III. Sobel’s Edge Detector



-1	-2	-1
1	2	1

s_1

-1		1
-2		2
-1		1

s_2

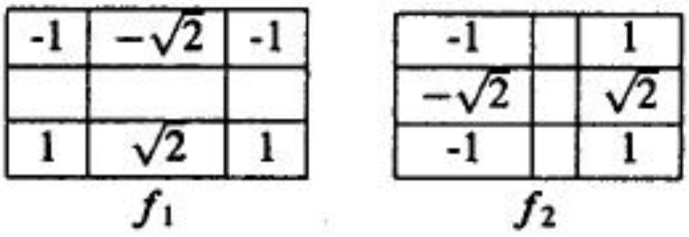
Figure 7.23 Sobel edge detector masks.

```
void sobel_ege_detector ( const Mat& src, Mat& dest, int threshold ){
        int m1[3][3] = {-1,-2,-1,0,0,0,1,2,1}, m2[3][3] = {-1,0,1,-2,0,2,-1,0,1};
        int rows = src.rows;
        int cols = src.cols;
        Mat temp(rows,cols,0);
        const uchar *row_pointer, *next_row_pointer;
        uchar* temp_row_pointer;
        int cr = 1, cc=1;
        int s1,s2,g ,tr,tc;

        set_all(temp, 255);

        for ( int r = 1 ; r<rows-1 ; r++ ) {
                temp_row_pointer = temp.ptr(r);
                for ( int c=1 ; c<cols-1 ; c++ ){
                //calc s1 and s2;
                        s1 = s2 = 0;
                        for ( int i=0 ; i<3 ; i++ ) {
                                int tr = i-cr + r;
                                const uchar* p = src.ptr(tr);
                                for ( int j=0 ; j<3 ; j++ ) {
                                        int tc = j-cc + c;
                                        s1 += m1[i][j]*p[tc];
                                        s2 += m2[i][j]*p[tc];
                                }
                        }
                        g = s1*s1+s2*s2;
                        if ( g>threshold*threshold )
                                temp_row_pointer[c] = 0;
                }
        }
        dest = temp;
}
```

IV. Frei and Chen's Edge Detector



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

f_1

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

f_2

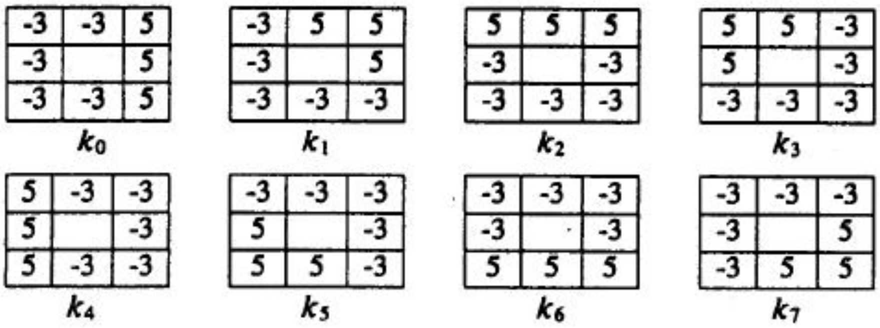
Figure 7.24 Frei and Chen gradient masks.

```
void frei_chen_gradient_operator ( const Mat& src, Mat& dest, int threshold ){
        double m1[3][3] = {-1,-sqrt(2.0),-1,0,0,0,1,sqrt(2.0),1}, m2[3][3] = {-1,0,1,-sqrt(2.0),0,sqrt(2.0),-1,0,1};
        int rows = src.rows;
        int cols = src.cols;
        Mat temp(rows,cols,0);
        const uchar *row_pointer, *next_row_pointer;
        uchar* temp_row_pointer;
        int cr = 1, cc=1,tr,tc;
        double f1,f2,g ;

        set_all(temp, 255);

        for ( int r = 1 ; r<rows-1 ; r++ ) {
                temp_row_pointer = temp.ptr(r);
                for ( int c=1 ; c<cols-1 ; c++ ){
                        //calc f1 and f2;
                        f1 = f2 = 0;
                        for ( int i=0 ; i<3 ; i++ ) {
                                int tr = i-cr + r;
                                const uchar* p = src.ptr(tr);
                                for ( int j=0 ; j<3 ; j++ ) {
                                        int tc = j-cc + c;
                                        f1 += m1[i][j]*p[tc];
                                        f2 += m2[i][j]*p[tc];
                                }
                        }
                        g = f1*f1+f2*f2;
                        if ( g>threshold*threshold )
                                temp_row_pointer[c] = 0;
                }
        }
        dest = temp;
}
```

V. Kirsch’s Edge Detector



-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

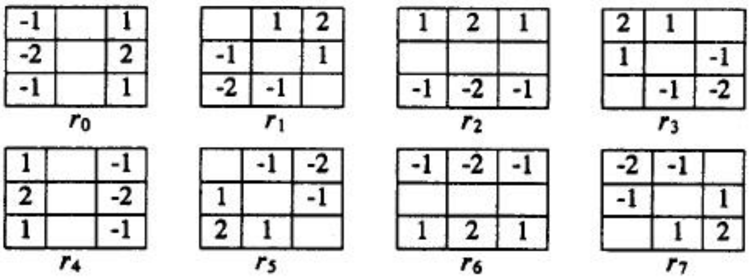
Figure 7.25 Kirsch compass masks.

```
void kirsch_compass_operator ( const Mat& src, Mat& dest, int threshold ){
        int k0[3][3] = {-3,-3,5,-3,0,5,-3,-3,5}, k1[3][3] = {-3,5,5,-3,0,5,-3,-3,-3}, k2[3][3] = {5,5,5,-3,0,-3,-3,-3,-3},\
        k3[3][3] = {5,5,-3,5,0,-3,-3,-3,-3}, k4[3][3] = {5,-3,-3,5,0,-3,5,-3,-3}, k5[3][3] = {-3,-3,-3,5,0,-3,5,5,-3},\
        k6[3][3] = {-3,-3,-3,-3,0,-3,5,5,5}, k7[3][3] = {-3,-3,-3,-3,0,5,-3,5,5};
        int rows = src.rows;
        int cols = src.cols;
        Mat temp(rows,cols,0);
        const uchar *row_pointer, *next_row_pointer;
        uchar* temp_row_pointer;
        int cr = 1, cc=1;
        int g ,tr,tc,tg;

        set_all(temp, 255);

        for ( int r = 1 ; r<rows-1 ; r++ ) {
                temp_row_pointer = temp.ptr(r);
                for ( int c=1 ; c<cols-1 ; c++ ){
                        g = INT_MIN;
                        tg = get_gradient_manitude_3(r,c,cr,cc,k0,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,k1,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,k2,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,k3,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,k4,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,k5,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,k6,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,k7,src); if ( tg>g ) g=tg;
                        if ( g>threshold )
                                temp_row_pointer[c] = 0;
                }
        }
        dest = temp;
}
```

VI. Robinson’s Edge Detector



-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

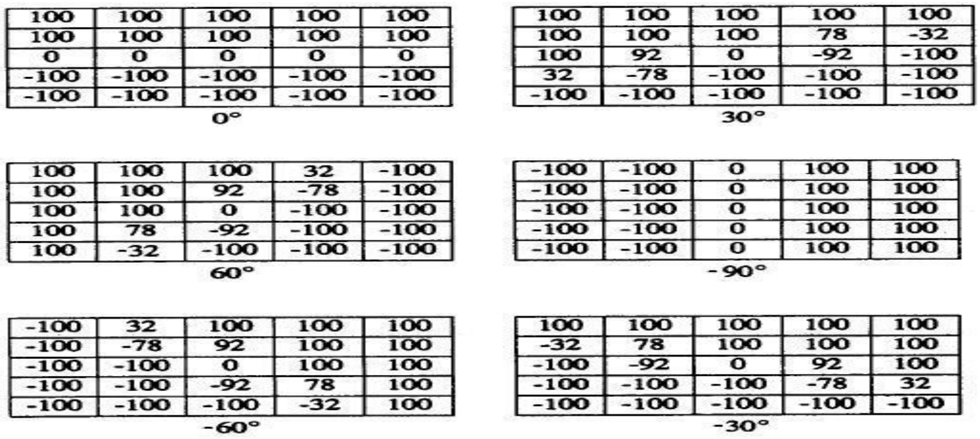
Figure 7.26 Robinson compass masks.

```
void robinson_compass_operator ( const Mat& src, Mat& dest, int threshold ){
        int r0[3][3] = {-1,0,1,-2,0,2,-1,0,1}, r1[3][3] = {0,1,2,-1,0,1,-2,-1,0}, r2[3][3] = {1,2,1,0,0,0,-1,-2,-1},\
        r3[3][3] = {2,1,0,1,0,-1,0,-1,-2}, r4[3][3] = {1,0,-1,2,0,-2,1,0,-1}, r5[3][3] = {0,-1,-2,1,0,-1,2,1,0},\
        r6[3][3] = {-1,-2,-1,0,0,0,1,2,1}, r7[3][3] = {-2,-1,0,-1,0,1,0,1,2};
        int rows = src.rows;
        int cols = src.cols;
        Mat temp(rows,cols,0);
        const uchar *row_pointer, *next_row_pointer;
        uchar* temp_row_pointer;
        int cr = 1, cc=1;
        int g ,tr,tc,tg;

        set_all(temp, 255);

        for ( int r = 1 ; r<rows-1 ; r++ ) {
                row_pointer = src.ptr(r);
                temp_row_pointer = temp.ptr(r);
                for ( int c=1 ; c<cols-1 ; c++ ){
                        g = INT_MIN;
                        tg = get_gradient_manitude_3(r,c,cr,cc,r0,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,r1,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,r2,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_3(r,c,cr,cc,r3,src); if ( tg>g ) g=tg;
                        if ( g>threshold )
                                temp_row_pointer[c] = 0;
                }
        }
        dest = temp;
}
```

VII. Nevatia-Babu 5x5 Edge Detector



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.

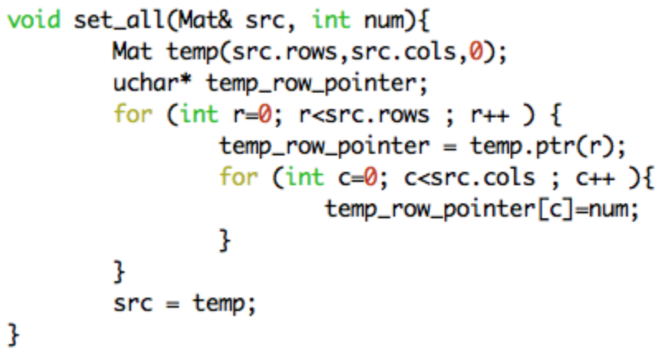
```
void nevatia_babu_operator ( const Mat& src, Mat& dest, int threshold ){
        int r0[5][5] = {100,100,100,100,100,100,100,100,100,100,0,0,0,0,0,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100}, \
        r1[5][5] = {100,100,100,100,100,100,100,100,78,-32,100,92,0,-92,-100,32,-78,-100,-100,-100,-100,-100,-100,-100,-100},\
        r2[5][5] = {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},\
        r3[5][5] = {-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},\
        r4[5][5] = {-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},\
        r5[5][5] = {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};
        int rows = src.rows;
        int cols = src.cols;
        Mat temp(rows,cols,0);
        uchar* temp_row_pointer;
        int cr = 1, cc=1;
        int g ,tr,tc,tg;

        set_all(temp, 255);

        for ( int r = 2 ; r<rows-2 ; r++ ) {
                temp_row_pointer = temp.ptr(r);
                for ( int c=2 ; c<cols-2 ; c++ ){
                        g = INT_MIN;
                        tg = get_gradient_manitude_5(r,c,2,2,r0,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_5(r,c,2,2,r1,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_5(r,c,2,2,r2,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_5(r,c,2,2,r3,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_5(r,c,2,2,r4,src); if ( tg>g ) g=tg;
                        tg = get_gradient_manitude_5(r,c,2,2,r5,src); if ( tg>g ) g=tg;
                        if ( g>threshold )
                                temp_row_pointer[c] = 0;
                }
        }
        dest = temp;
}
```

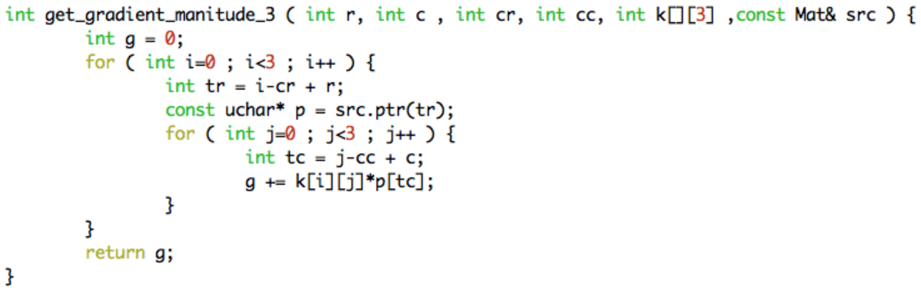
VIII. Other functions

i. set_all: set all pixels of input image to a fixed number

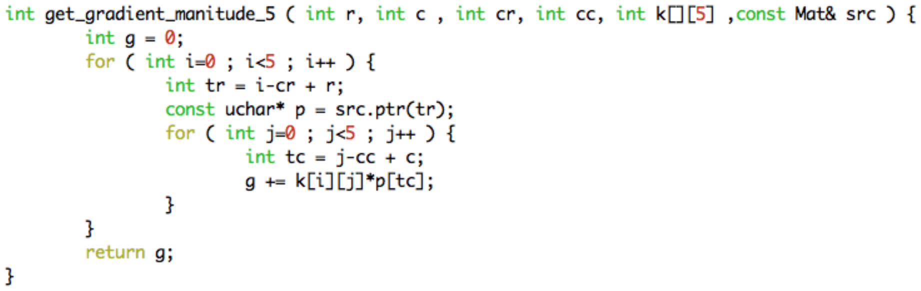


```
void set_all(Mat& src, int num){
        Mat temp(src.rows,src.cols,0);
        uchar* temp_row_pointer;
        for (int r=0; r<src.rows ; r++ ) {
                temp_row_pointer = temp.ptr(r);
                for (int c=0; c<src.cols ; c++ ){
                        temp_row_pointer[c]=num;
                }
        }
        src = temp;
}
```

ii. get_gradient_manitude: get the gradient magnitude of size 3 and 5



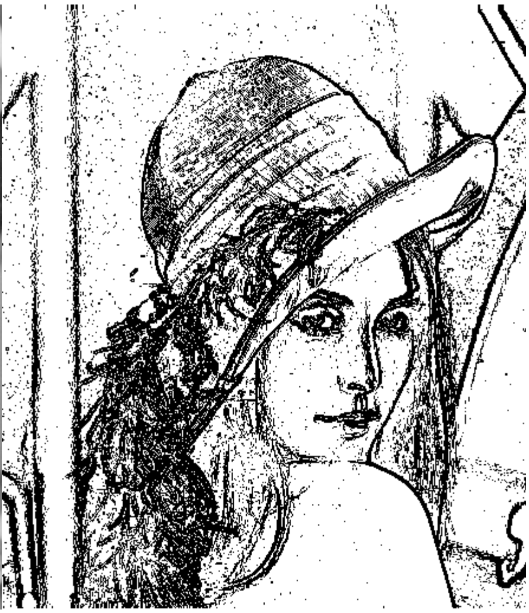
```
int get_gradient_manitude_3 ( int r, int c , int cr, int cc, int k[][3] ,const Mat& src ) {
        int g = 0;
        for ( int i=0 ; i<3 ; i++ ) {
                int tr = i-cr + r;
                const uchar* p = src.ptr(tr);
                for ( int j=0 ; j<3 ; j++ ) {
                        int tc = j-cc + c;
                        g += k[i][j]*p[tc];
                }
        }
        return g;
}
```



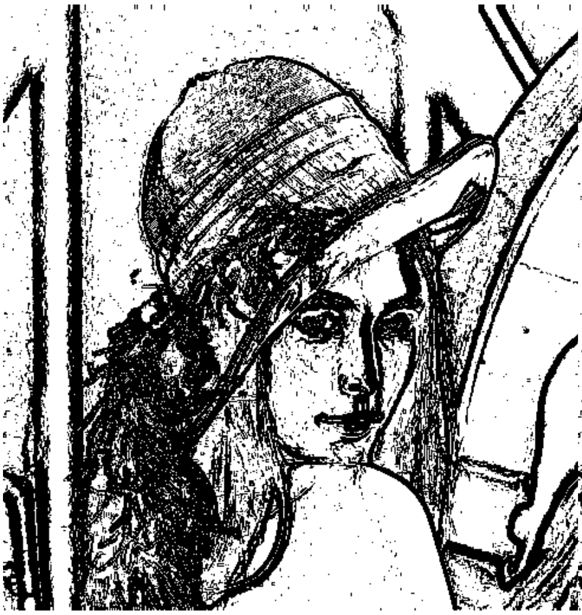
```
int get_gradient_manitude_5 ( int r, int c , int cr, int cc, int k[][5] ,const Mat& src ) {
        int g = 0;
        for ( int i=0 ; i<5 ; i++ ) {
                int tr = i-cr + r;
                const uchar* p = src.ptr(tr);
                for ( int j=0 ; j<5 ; j++ ) {
                        int tc = j-cc + c;
                        g += k[i][j]*p[tc];
                }
        }
        return g;
}
```

3. Result

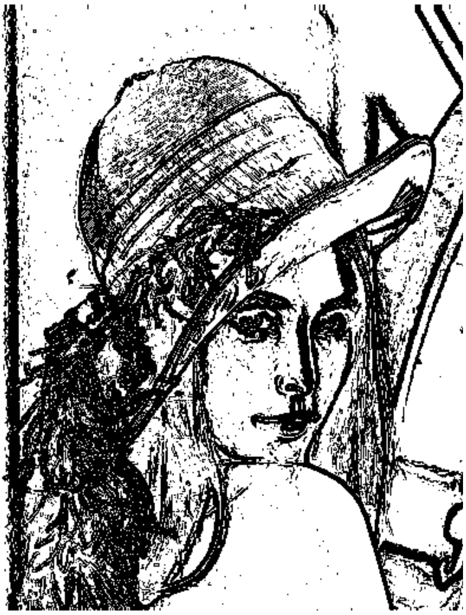
lena.bmp



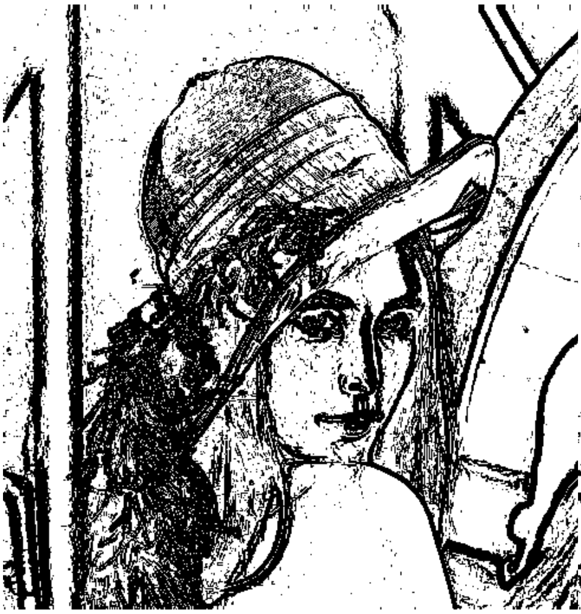
roberts_operator_12.bmp



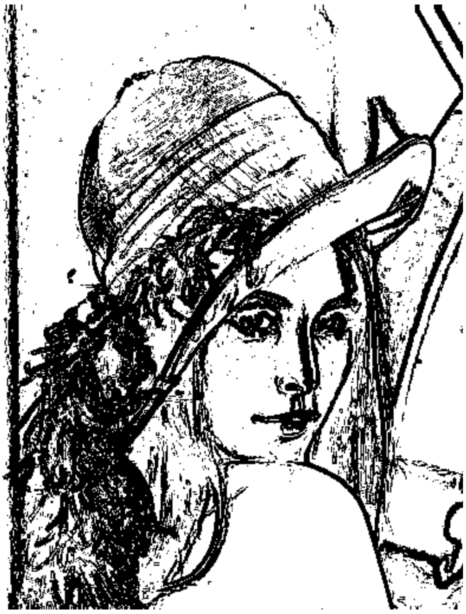
prewitt_edge_detector_24.bmp



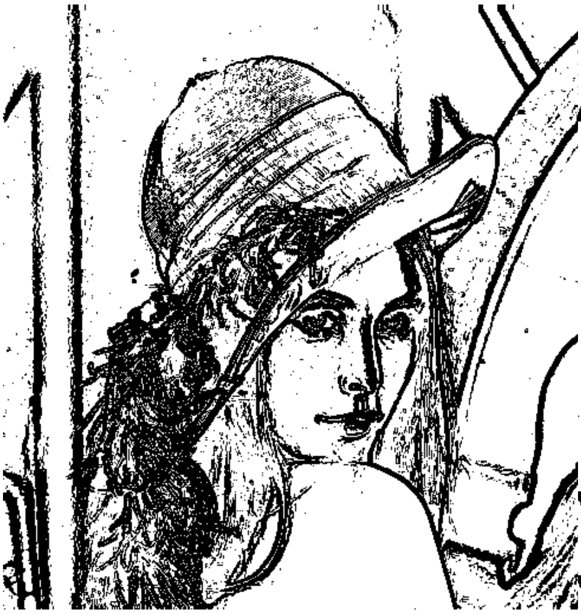
sobel_ege_detector_38.bmp



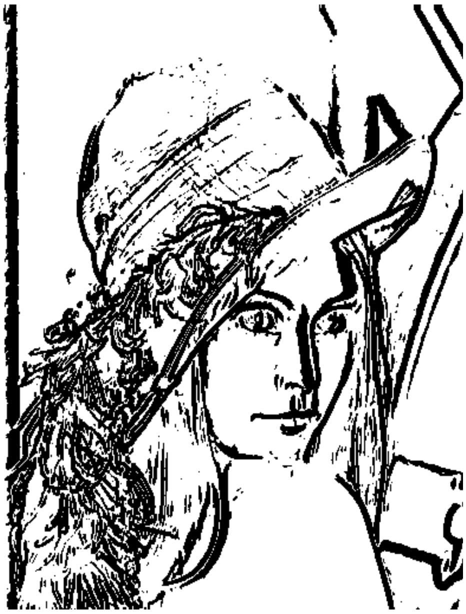
frei_chen_gradient_operator_30.bmp



kirsch_compass_operator_135.bmp



robinson_compass_operator_43.bmp



nevatia_babu_operator_12500.bmp

4. Appendix

I. build_all.sh

“sh build_all.sh” will automatically compile the code in terminal.

II. R01922124_HW9.cpp

source code

III. lena.bmp

original lena image

IV. Many result images

V. R01922124_HW9.pdf

report