第四章作业 黄海浪 9181040G0818

1. 通过分析说明边缘宽度与模板尺寸间的关系。
答：当边模版尺寸为 2，边缘宽度为 1；模版尺寸为 3，边缘宽度为 2；模版尺寸为 4，边缘宽度为 3；由图也可得到，边缘宽度 = 模版尺寸 - 1

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	100	100	100	100	1	1	1
1	1	100	100	100	100	1	1	1
1	1	100	100	100	100	1	1	1
1	1	100	100	100	100	1	1	1
1	1	100	100	100	100	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

2. 图像平滑模板的各系数之和 1 且无负数，边缘检测算子的模板有什么特点？边缘锐化算子的模板有什么特点？
答：边缘检测：各个系数之和为 0，有负数。
边缘锐化：各个系数之和大于 0（一般为 1），有负数。

3. 梯度算子中$\sqrt{\Delta_x^2+\Delta_y^2}$的最大值是多少？如何使用查找表来替代开方运算？

图像中绝大多数像素的梯度值是非常小的，如何根据这个特点来设计局部查表法？

答：$\sqrt{\Delta_x^2+\Delta_y^2}$的最大值是$\sqrt{2}\max\{\Delta\}=255\sqrt{2}$ 。

由于Δ的取值大部分集中在一个很小的范围（一副正常的图像，两个相邻像素之间差别大部分不太大），在这个小范围内进行查表，稍微大一点的范围进行计算。

多幅图像进行实验，将Δ的取值范围测试出来后，假设Δ在 0-10 之间，则设计相应的 0-200 的一维表。如果$\Delta_x^2+\Delta_y^2$计算出来在 0-200，那么进行查表，否则进行计算。

4. Prewitt，Robinson，Kirsch 算子计算得到的边缘强度值，哪个更大？哪个更小？

答：Prewitt 最小，Kirsch 最大。

Prewitt：边缘强度最小

$$P_x=\begin{bmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{bmatrix},P_y=\begin{bmatrix}1 & 1 & 1\\ 0 & 0 & 0\\ -1 & -1 & -1\end{bmatrix}$$

Robinson

$$R_N=\begin{bmatrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & 1\end{bmatrix},R_{NE}=\begin{bmatrix}0 & 1 & 2\\ -1 & 0 & 1\\ -2 & -1 & 0\end{bmatrix}$$

$$R_E=\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix},R_{SE}=\begin{bmatrix}-2 & -1 & 0\\ -1 & 0 & 1\\ 0 & 1 & 2\end{bmatrix}$$

$$R_S=\begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\,1\\ 1 & 2 & 5\end{bmatrix},R_{SW}=\begin{bmatrix}0 & -1 & -2\\ 1 & 0 & -1\\ 2 & 1 & 0\end{bmatrix}$$

$$R_W=\begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix},R_{NW}=\begin{bmatrix}2 & 1 & 0\\ 1 & 0 & -1\\ 0 & -1 & -2\end{bmatrix}$$

Kirsch：边缘强度最大

$$K_N=\begin{bmatrix}5 & 5 & 5\\ -3 & 0 & 3\\ -3 & -3 & -3\end{bmatrix},K_{NE}=\begin{bmatrix}-3 & 5 & 5\\ -3 & 0 & 5\\ -3 & -3 & -3\end{bmatrix}$$

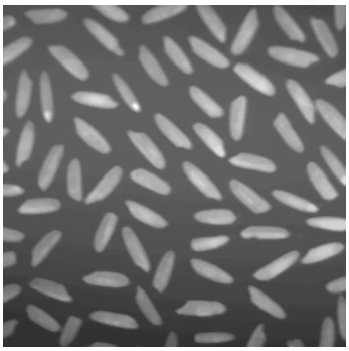
$$K_E=\begin{bmatrix}-3 & -3 & 5\\ -3 & 0 & 5\\ -3 & -3 & 5\end{bmatrix},K_{SE}=\begin{bmatrix}-3 & -3 & -3\\ -3 & 0 & 5\\ -3 & 5 & 5\end{bmatrix}$$

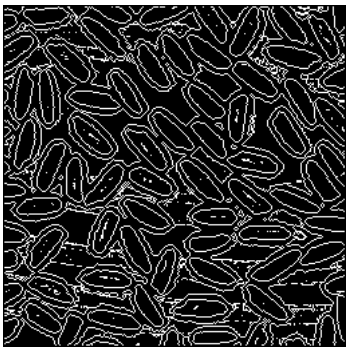
$$K_S=\begin{bmatrix}-3 & -3 & -3\\ -3 & 0 & -3\\ 5 & 5 & 5\end{bmatrix},K_{SW}=\begin{bmatrix}-3 & -3 & -3\\ 5 & 0 & -3\\ 5 & 5 & -3\end{bmatrix}$$

$$K_W=\begin{bmatrix}5 & -3 & -3\\ 5 & 0 & -3\\ 5 & -3 & -3\end{bmatrix},K_{NW}=\begin{bmatrix}5 & 5 & -3\\ 5 & 0 & -3\\ -3 & -3 & -3\end{bmatrix}$$

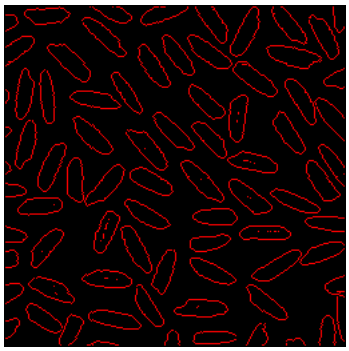
5. 用 C/C++编程实现 H0401Gry.bmp 中的米粒边缘检测，一定要采用一阶微分算子和二阶微分算子-沈俊算子结合的方法。

答：原图、sobel、沈俊算子（0.5）处理结果分别如下：





给一个阈值 64，对于 sobel 处理后的边缘强度图像，值大于 64 认为是边缘。和沈算子进行结合，最终得到图像如下。



沈算子主要是边缘的精细表达，有是/不是而没有大小，sobel 算子有大小而不是那么精确，将两者结合可以认为是各取优点。

主要代码：

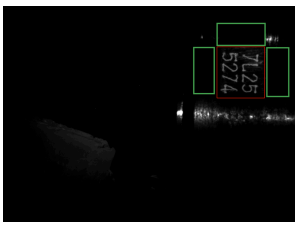
```cpp
/// 例子 米检测
void BmpFile::riceEdgeDetector(double const &alpha, int const &grd) {
    // 备份原来的data
    auto *srcData = new unsigned char[infoHeader.image_size];
    memcpy(srcData, data, infoHeader.image_size);
    // sobel处理
    sobelProcess();
    // 拿回原来的data
    auto *sobelData = data;
    data = srcData;
    // 沈俊处理
    ShenJunProcess(alpha);

    // 融合
    for (int i = 0; i < infoHeader.image_size; i++) {
        data[i] = (data[i] && (sobelData[i] > grd)) ? 0xff : 0x0;
    }
    delete[] sobelData;

    // 伪彩色
    palette[255].red = 255;
    palette[255].blue = 0;
    palette[255].green = 0;
}
```

```cpp
/// 一阶微分算子 sobel
bool BmpFile::sobelProcess() {
    if (isRgb) {
        return false;
    }
    auto *resData = new unsigned char[infoHeader.image_size];
    memset(resData,  c: 0, infoHeader.image_size);
    const unsigned &width = infoHeader.bitmap_width;
    const unsigned &height = infoHeader.bitmap_height;
    auto *pRes = resData + width;
    for (auto *pCur = data + width, *pEnd = data + width * (height - 1); pCur < pEnd;) {
        ++pCur, ++pRes;
        for (auto *pWidthEnd = pCur + width - 2; pCur < pWidthEnd; pCur++) {
            auto lastRow = pCur - width;
            auto nextRow = pCur + width;
            int dx = *(lastRow - 1) + *(pCur - 1) * 2 + *(nextRow - 1);
            dx -= *(lastRow + 1) + *(pCur + 1) * 2 + *(nextRow + 1);
            int dy = *(lastRow - 1) + *(lastRow) * 2 + *(lastRow + 1);
            dy -= *(nextRow - 1) + *(nextRow) * 2 + *(nextRow + 1);
            *(pRes++) = rmin( lcpp_x: 255,  lcpp_y: abs(dx) + abs(dy));
        }
        ++pCur, ++pRes;
    }
    delete[] data;
    data = resData;
    return true;
}
```

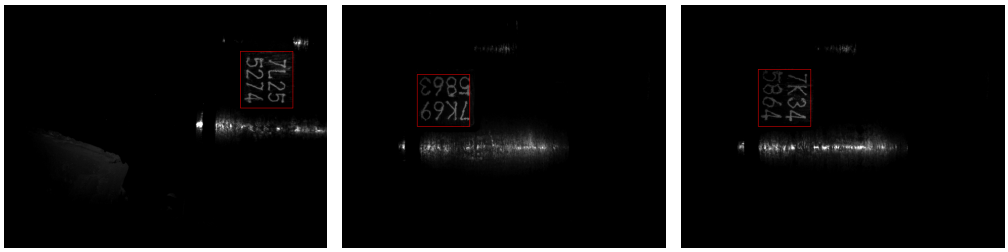
```cpp
///沈俊算子
bool BmpFile::ShenJunProcess(const double &alpha) {
    assert(0.0 < alpha && alpha < 1.0);
    if (isRgb) {
        return false;
    }
    const unsigned &width = infoHeader.bitmap_width;
    const unsigned &height = infoHeader.bitmap_height;

    // 查找表
    auto *T_LUT = new int[1 << 9], *LUT = T_LUT + 256;
    *LUT = 0;
    for (int i = 1; i < 256; ++i) {
        LUT[i] = int(i * alpha + 0.5);
        LUT[-i] = -LUT[i];
    }
    // 递推滤波
    int pre;
    auto *resData = new unsigned char[infoHeader.image_size], *pRes = resData;
    // 按行
    for (auto *pCur = data, *pEnd = data + infoHeader.image_size; pCur < pEnd;) {
        *(pRes++) = pre = *(pCur++);
        // 从左到右 p1(y,x)=p1(y,x-1)+a*[p(y,x)-p1(y,x-1)]
        for (auto *pRowEnd = pCur + width - 1; pCur < pRowEnd; ++pCur) {
            *(pRes++) = pre = pre + LUT[(*pCur) - pre];
        }
        pRes -= 2;
        // 从右到左 p2(y,x)=p2(y,x+1)-a*[p1(y,x)-p2(y,x+1)]
        for (auto *pRowEnd = pRes - width + 1; pRes > pRowEnd;) {
            *(pRes--) = pre = pre + LUT[*(pRes) - pre];
        }
        pRes += width + 1;
    }
    // 按列
    pRes = resData;
    for (unsigned x = 0; x < width; ++x, pRes = resData + x) {
        pre = *pRes;
        //从上向下 p3(y,x)=p3(y-1,x)+a*[p2(y,x)-p3(y-1,x)]
        pRes += width;
        for (auto *pColEnd = pRes + width * (height - 1); pRes < pColEnd; pRes += width) {
            *pRes = pre = pre + LUT[*(pRes) - pre];
        }
        pRes -= width << 1;
        for (auto *pColEnd = pRes - width * (height - 1); pRes > pColEnd; pRes -= width) {
            *pRes = pre = pre + LUT[*(pRes) - pre];
        }
    }

    // 计算导数 大于才为1 否则为0
    pRes = resData;
    for (auto *pCur = data, *pEnd = data + infoHeader.image_size; pCur < pEnd; ++pCur, ++pRes) {
        *pRes = *pRes > *pCur;
    }
    // 过零点检测
    auto *tmpData = resData;
    resData = new unsigned char[infoHeader.image_size];
    memset(resData,  c: 0, infoHeader.image_size);
    pRes = resData + width + 1;
    for (auto *pCur = tmpData + width + 1, *pEnd = tmpData + width * (height - 1); pCur < pEnd;) {
        for (auto *pRowEnd = pCur + width - 2; pCur < pRowEnd; ++pCur, ++pRes) {
            if (*pCur) {
                // 上下左右 有小于等于0的
                if (!*(pCur - 1) || !*(pCur + 1) || !*(pCur - width) || !*(pCur + width)) {
                    *pRes = 0xff;
                }
            }
        }
        pCur += 2, pRes += 2;
    }

    delete[] tmpData;
    delete[] data;
    data = resData;
    return true;
}
```

6. 用 C/C++编程实现 H0402Gry.bmp,H0403Gry.bmp，H0404Gry.bmp 中的文本定位，一定要使用变分辨率、边缘强度、积分图。

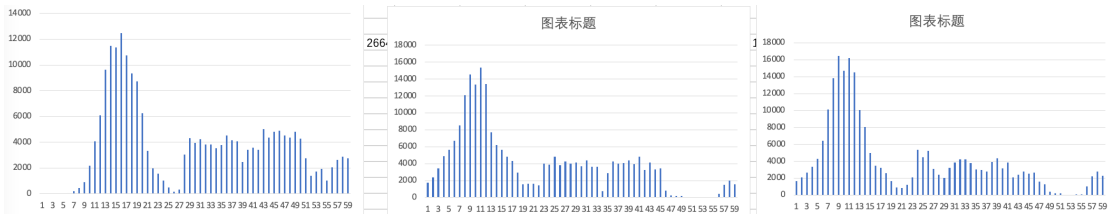
答：首先对图像进行了长宽均缩小 4 倍的压缩处理，然后进行 sobel 处理计算得到边缘强度图，对得到的边缘强度图计算积分图，然后采用特殊的算子进行处理，寻找区域最大的积分。最后获得一个点坐标（长宽都是自己给的）。见下图，绿色积分区域取负，红色为正，最后框出来为红色区域。(trick 处理？为了画出红色方框，把原来图像像素值为 255 的修改为 254，然后将图像框像素值改为 255，最后使用假彩色，将最后一个调色板改为 r=255，g=b=0。)



最终结果如下：

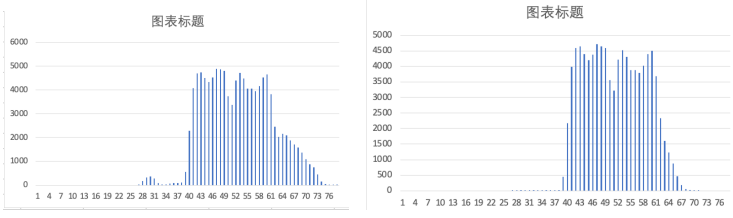


方法 2（更快的方法）：更快，意味着可能会失去部分精度。首先按照上述方法 1 压缩图像，然后对图像进行分析，得到图像的行积分，如下图：



然后对行积分进行平滑，得到如下波形：

找波谷然后进行切割，最后会得到类似下面的图 1，然后又进行平滑，得到图 2：



最终进行切割，得到最后的图像。
对于原始图像，第一次切割结果如下：

第二次切割后，得到如下图像：

多次测试后取平均，时间对比如下：

	debug	release
第一种方法	1.573ms	0.260ms
第二种方法	0.514ms	0.127ms

可以看到第二种方法速度快得多（因为减少了暴力搜索，仅进行行/列积分计算和一维滤波），但是波峰波谷找起来会降低代码的鲁棒性。第二种方法代码较多，故这里贴出第一种方法的代码，第一种方法，代码获取窗口的区域依然能够优化，但是会降低可读性和可维护性：

```cpp
/// 文本定位
void BmpFile::textPositioningByWindow(const int &o_w, const int &o_h) {
    clock_t t_start = clock();
    const int w = o_w >> 2;
    const int h = o_h >> 2;
    // 首先保存原始数据图像
    auto *orgImgBak = new unsigned char[infoHeader.image_size];
    memcpy(orgImgBak, data, infoHeader.image_size);
    auto orgWidth = infoHeader.bitmap_width;
    auto orgHeight = infoHeader.bitmap_height;
    auto orgImgSize = infoHeader.image_size;
    // 图片压缩
    compressImg( widthSlope: 4,  heightSlope: 4);
    // 使用sobel算子计算边缘强度 用边缘强度去寻找 而非原始图像的亮度
    sobelProcess();
    // 计算得到的边缘图 的 积分图
    getSumGryGraphBySumCol();
    const int &width = infoHeader.bitmap_width;
    const int &height = infoHeader.bitmap_height;
    int maxNum = 0, max_x = 0, max_y = 0;
    int w_2 = w / 2;
    int h_2 = h / 2;
//    for(int times = 0;times<1000;++times){
    for (int x = w_2, end_x = width - w_2 * 3, end_y = height - h_2 * 3; x < end_x; ++x) {
        for (int y = 0; y < end_y; ++y) {
            int tmpSum = getWindowsSumGraph(x, y, w, h) - getWindowsSumGraph( s_x: x + w, y, w_2, h) -
                         getWindowsSumGraph( s_x: x - w_2, y, w_2, h) - getWindowsSumGraph(x,  s_y: y + h, w, h_2);
            if (maxNum < tmpSum) {
                maxNum = tmpSum;
                max_x = x;
                max_y = y;
            }
        }
    }
//    }
    max_x <<= 2;
    max_y <<= 2;
    delete[] data;
    data = orgImgBak;
    infoHeader.bitmap_width = orgWidth;
    infoHeader.bitmap_height = orgHeight;
    infoHeader.image_size = orgImgSize;
    fileHeader.file_size = fileHeader.offset_bits + orgImgSize;
    drawImg(max_y, max_x, o_h, o_w);
    clock_t t_end = clock();
    std::cout << (t_end - t_start) / 1000.0;
}
```

7. 学习并编程实现 Canny 算子，并用该算子检测 H0401Gry.bmp 中的米粒边缘

答：canny 算子步骤

(1)高斯滤波平滑图像，去除噪声

(2)寻找图像梯度方向和大小

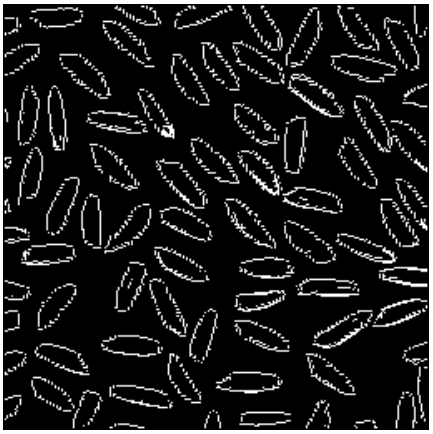
(3)非最大抑制消除边缘误检

(4)双阈值决定可能的边界并应用滞后技术跟踪边界

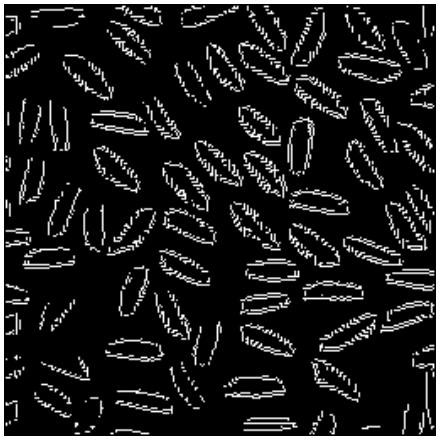
(3)：计算得到梯度值和梯度方向后，对图片进行全面的扫描，以去除不构成边缘的无关像素点。对于每个像素，检查其是否是在梯度方向中其临近像素点中的局部最大值。在梯度方向上，边缘点若是梯度方向上的像素点局部最大值，则不变，否则置为 0（抑制）。

(4)：双阈值，需要设定两个阈值，minVal 和 maxVal。任何边缘的强度梯度大于 maxVal 的确定为边缘；而小于 minVal 的确定为非边缘，并置为 0。位于 maxVal 和 minVal 阈值间的边缘为待分类边缘，或非边缘，基于连续性进行判断。如果边缘像素连接着“确定边缘”像素，则认为该边缘属于真正边缘的一部分；否则，置为 0。

实现后效果：



无高斯滤波,maxVal=150,minVal=100



sigma=2,maxVal=120,minVal=80

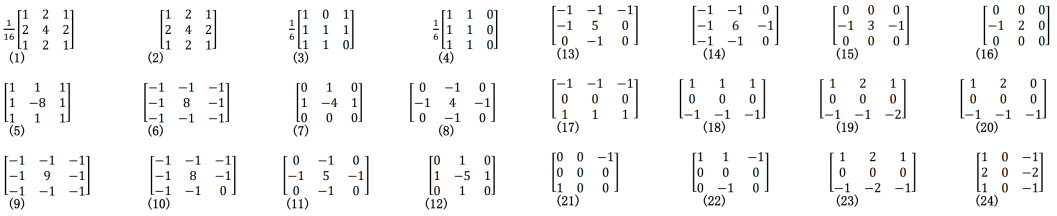
opencv 自带的 canny 没有进行滤波处理，处理结果和 opencv 都有点区别，程序检查了几遍感觉也没问题，等老师您讲解后我再改一下试试。

主要代码：

```
//非最大值抑制 NMS
auto *pCurPos = resPos + width;
for (auto *pCur = resData + width, *pEnd = resData + width * (height - 1); pCur < pEnd;) {
    ++pCur, ++pCurPos;
    for (auto *pWidthEnd = pCur + width - 2; pCur < pWidthEnd; ++pCur, ++pCurPos) {
        if (*pCurPos == 0) {
            // '-'
            if (*pCur < *(pCur - 1) || *pCur < *(pCur + 1)) {
                *pCur = 0x0;
            }
        } else if (*pCurPos == 1) {
            // '|'
            if (*pCur < *(pCur - width) || *pCur < *(pCur + width)) {
                *pCur = 0x0;
            }
        } else if (*pCurPos == 2) {
            // '/'
            if (*pCur < *(pCur - width + 1) || *pCur < *(pCur + width - 1)) {
                *pCur = 0x0;
            }
        } else if (*pCurPos == 3) {
            // '\'
            if (*pCur < *(pCur - width - 1) || *pCur < *(pCur + width + 1)) {
                *pCur = 0x0;
            }
        }else{
            *pCur = 0x0;
        }
    }
    ++pCur, ++pCurPos;
}
```

```
    //双阈值
    for (auto *pCur = resData + width, *pEnd = resData + width * (height - 1); pCur < pEnd;) {
        ++pCur;
        for (auto *pWidthEnd = pCur + width - 2; pCur < pWidthEnd; ++pCur) {
            if (*pCur <= minVal) {
                *pCur = 0x0;
                continue;
            }
            if (minVal < *pCur && *pCur < maxVal) {
                auto *pLastRowCur = pCur - width;
                auto *pNextRowCur = pCur + width;
                if (*(pCur - 1) == 0 && *(pCur + 1) == 0 && *(pLastRowCur - 1) == 0 && *(pLastRowCur + 1) == 0 &&
                    *(pNextRowCur - 1) == 0 && *(pNextRowCur + 1) == 0 && *pLastRowCur == 0 && *pNextRowCur == 0) {
                    *pCur = 0x0;
                } else {
                    *pCur = 0xff;
                }
            } else if (*pCur >= maxVal) {
                *pCur = 0xff;
            }
        }
        ++pCur;
    }
```

8. 看到模板的结构就能知道模板的效果和性能，是图像处理研究人员的基本素质。若使用下列模板分别对一幅灰度图像进行卷积，会达到什么样的效果？请在模板的系数之和、系数的正负号等方面进行区分。注意，(17)到(24)带有绝对值。可以自己编个小程序测试，也可以使用 Photoshop 验证。



(1) 全为+，和为 1，高斯平滑，图像变模糊，去噪（可以用移位做乘除）
(2) 全为+，和为 16，简单加权，调高亮度，基本全白。
(3) 全为+，和为 7/6，个性化算子，用周围除了上和右下进行平滑并调高亮度
(4) 全为+，和为 1，局部滤波，带方向平滑，图像轻微的一个像素地右移
(5) 有+有-，和为 0，拉普拉斯边缘检测
(6) 有+有-，和为 0，拉普拉斯边缘检测，和 5 效果一样
(7) 有+有-，和为-1，拉普拉斯滤波结果的减法，滤掉了部分拉普拉斯结果，并且图像变暗，只剩下部分边缘点、边界线、轮廓，甚至全黑
(8) 有+有-，和为 0，拉普拉斯边缘检测，四个领域，少了边角处理
(9) 有+有-，和为 1，拉普拉斯算子边缘锐化
(10)有+有-，和为 1，拉普拉斯算子锐化，少了右下角处理，图像比 9 轻微移动
(11)有+有-，和为 1，拉普拉斯算子锐化，少边角处理
(12)有+有-，和为-1，除非边缘特征特别强烈，否则基本全黑
(13)有+有-，和为 0，边缘检测，主要是左上角的边缘
(14)有+有-，和为 0，边缘检测，主要是左边边缘
(15)有+有-，和为 1，边缘锐化，左右锐化
(16)有+有-，和为 1，边缘锐化，左边缘锐化
(17)有+有-，和为 0，边缘检测，上下边缘，上移
(18)有+有-，和为 0，边缘检测，上下边缘，下移
(19)有+有-，和为 0，边缘检测，上下边缘，下移并左移
(20)有+有-，和为 0，边缘检测，上下边缘
(21)有+有-，和为 0，边缘检测，斜边缘 ‘\’
(22)有+有-，和为 0，边缘检测
(23)有+有-，和为 0，边缘检测，上下边缘，边缘信息更强（上下中间 2）
(24)有+有-，和为 0，边缘检测，左右边缘，边缘信息更强（左右中间 2）

图像移动，如果邻域内的像素，有一个方向是空的，那么图像就会朝着这个方向移动，说的粗糙点就是中心点把其他像素都综合过来了，也就是空方向的反方向的像素跑中间来了，对于每一个像素都这样，那么图像就看起来移动了。