数字图像处理报告

09021220 周天逸

形态学实验

1. 实验内容

    1. 对指纹图片进行形态学滤波
    2. 对于结果的断裂处，恢复

2. 实验思路

    1. 将形态学滤波的先开后闭操作依次分开，通过基本的腐蚀和膨胀操作完成。拆分为 腐蚀-膨胀-腐蚀-膨胀
    2. 对于图像因为腐蚀而产生的断裂，我们采用骨架提取和复原的操作进行恢复。
    3. 恢复步骤
        1. 细化是一种形态学操作，将对象的形状减小到其基本结构。该代码使用侵蚀、膨胀和减法等操作，迭代地细化输入图像，直到不能再细化为止。
        2. 细化恢复是细化的逆过程，旨在从其细化中重建原始对象。该代码迭代地膨胀骨架，并取膨胀图像与原始骨架的元素最小值，直到达到指定的迭代次数或不再发生更改。 3.实验结果展示

p2:通过细化后的图片 p3：对于p2的结果膨胀操作可以将指纹线条变粗 p4：对p3进一步膨胀，修复部分因为腐蚀而断裂的部分。 p5：对p4的进行腐蚀恢复原来的图像指纹粗细大小 p6：通过骨架细化和细化恢复操作，连接指纹的断裂部分。

3. 实验思考

    1. 形态学滤波器各个步骤的作用
        1. 腐蚀操作：背景噪声在开操作的腐蚀部分就完全消除因为这种情况下，噪声分量都比结构元小。
        2. 开操作的膨胀：用于消除指纹中的噪声元素。

3. 闭操作的膨胀：用于消除指纹纹路间产生的新的断裂。
4. 闭操作的腐蚀：因为膨胀虽然恢复了大部分断裂，但是纹路变粗了，所以用腐蚀来弥补。

2. 代码实现

```
    erode(img, img, kernel);
    dilate(img,img,kernel);
    dilate(img,img,kernel);
    erode(img, img, kernel);
```

3. 细化和细化恢复
    1. 骨架化是一种形态学运算，将对象的形状减小到其基本结构。该代码使用侵蚀、膨胀和减法操作的组合，迭代地将输入图像变细，直到无法进一步细化。
    2. 骨架重建是骨架化的逆过程。它旨在从其骨架中重建原始对象。该代码迭代地扩张骨架，并取扩张图像与原始骨架之间的逐元素最小值，直到达到指定的迭代次数或不再发生更改。
4. 代码

```
    Mat skeletonize(const cv::Mat &img) {
cv::Mat skel(img.size(), CV_8UC1, cv::Scalar(0));
cv::Mat temp;
cv::Mat eroded;

cv::Mat element = cv::getStructuringElement(cv::MORPH_CROSS, cv::Size(3,
3));

bool done;
do {
    cv::erode(img, eroded, element);
    cv::dilate(eroded, temp, element); // temp = open(img)
    cv::subtract(img, temp, temp);
    cv::bitwise_or(skel, temp, skel);
    eroded.copyTo(img);

    done = (cv::countNonZero(img) == 0);
} while (!done);

return skel;
}
```

图像分割

1. 实验内容
    1. 对比全局阈值和OTSU阈值法对于图像的分割技术。并对于结果分析两者区别与各自好处，优点
2. 实验内容

1. 全局阈值
    1. 通过初始化一个初始阈值T和范围因子ΔT
    2. 每次通过T将直方图分成两部分，分别求两部分的恢复均值。
    3. 通过更新$T' = \frac{1}{2}(m1 + m2)$,知道差值小于$\Delta T$停止。
2. OTSU
    1. 求出图像的最大和最小像素值，确定阈值区间
    2. 遍历每一个像素点，计算类间方差。计算每个阈值下的类间方差并保存到var中
    3. 找到方差的最大值对应的灰度值，就是需要找到的阈值
    4. 通过阈值进行灰度分割

3. 实验关键代码

    1. 全局阈值

```
do
{
    Tp = T;
    for (int i = 0; i < img.rows; i++)
    {
        for (int j = 0; j < img.cols; j++)
        {
            if (img.at<uchar>(i, j) < N)
            {
                p = p + img.at<uchar>(i, j);
                r++;
            }
            else
            {
                q = q + img.at<uchar>(i, j);
                t++;
            }
        }
    }
    T = (p / r + q / t) / 2;
    r = 0;
    t = 0;
    p = 0;
    q = 0;
} while (abs(T - Tp) > 1);
```

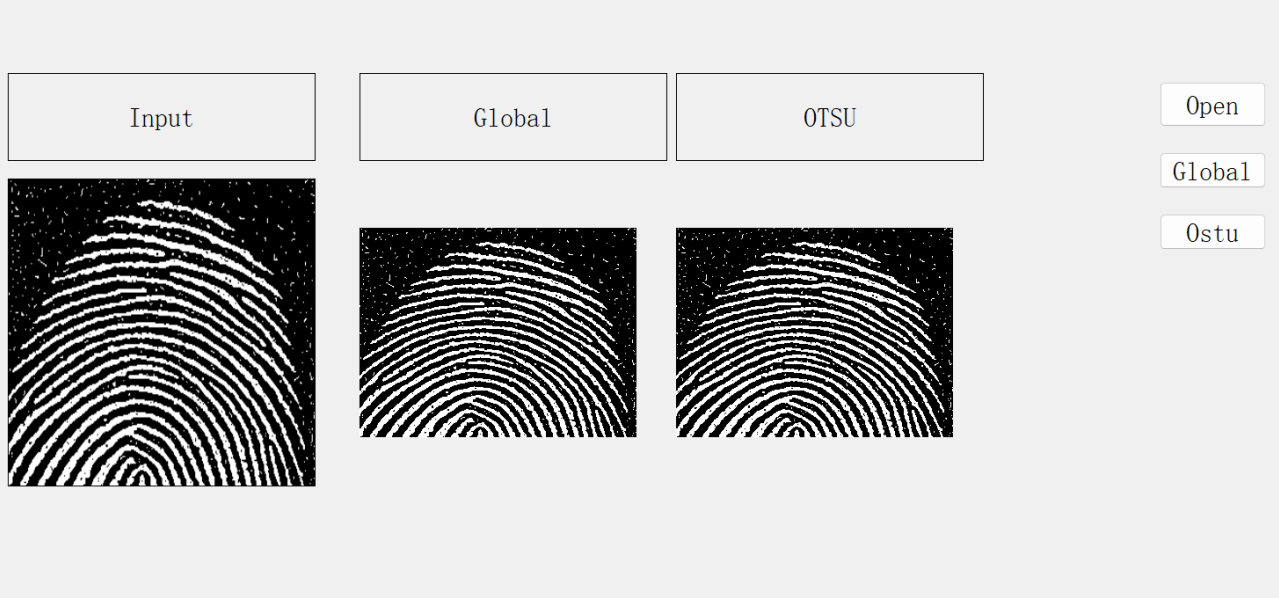
    2. OTSU

```
for (thresh_value = min_value_mat; thresh_value < max_value_mat;
thresh_value++)
{
    double sum = mat.rows*mat.cols;      //图像像素点总数
    double sum_aim = 0, sum_bg = 0;      //目标和背景像素点总数
    double sum_vaule_aim = 0, sum_vaule_bg = 0;      //目标和背景像素点的总灰度
    for (int i = 0; i < mat.rows; i++)
```

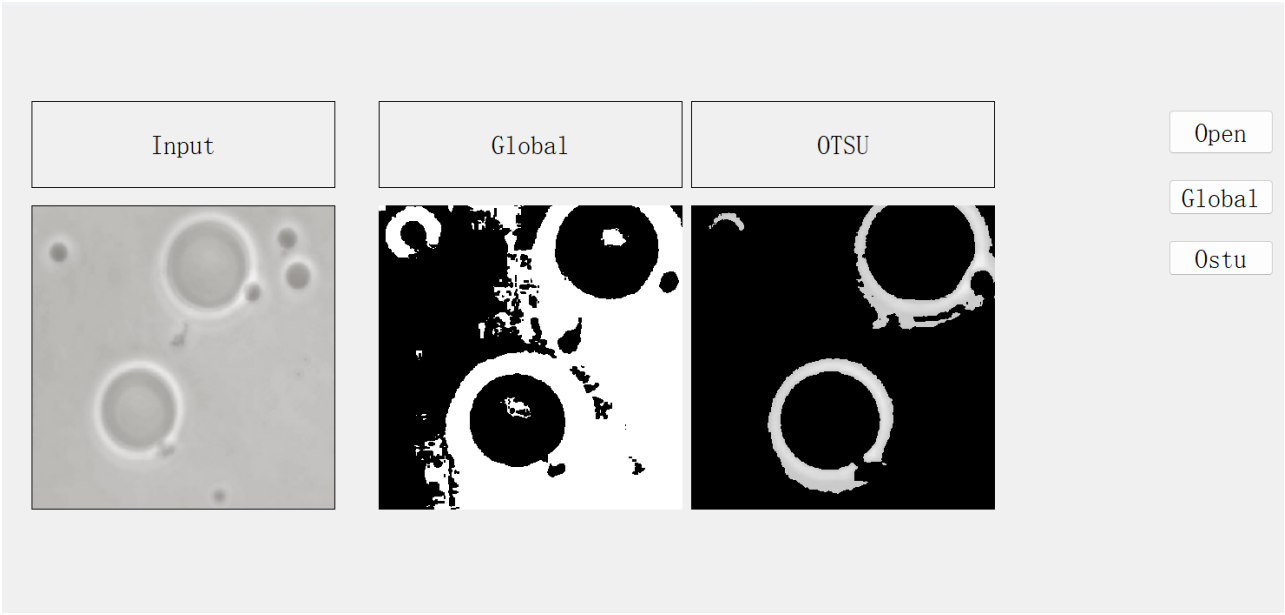
```
            for (int j = 0; j < mat.cols; j++)
            {
                int vaule = mat.at<uchar>(i, j);
                if (vaule < thresh_value)     //背景
                {
                    sum_bg += 1;
                    sum_vaule_bg += vaule;
                }
                else       //目标
                {
                    sum_aim += 1;
                    sum_vaule_aim += vaule;
                }
            }
        double ratio_aim = sum_aim / sum;    //目标像素点所占比例
        double ratio_bg = sum_bg / sum;      //背景像素点所占比例
        double aver_vaule_aim = sum_vaule_aim / sum_aim;      //目标像素点的平均灰
度
        double aver_vaule_bg = sum_vaule_bg / sum_bg;         //背景像素点的平均灰
度
        double aver_vaule_mat = ratio_aim * aver_vaule_aim + ratio_bg *
aver_vaule_bg;     //图像总平均灰度

        //计算每个阈值下的类间方差并保存到var中
        var[m] = ratio_aim * (aver_vaule_aim - aver_vaule_mat)*(aver_vaule_aim -
aver_vaule_mat) +
                 ratio_bg * (aver_vaule_bg - aver_vaule_mat)*(aver_vaule_bg -
aver_vaule_mat);
        m += 1;
    }
```

4. 实验结果展示



展示对于指纹图片的图片分割，基本和原图差别不大。因为原图基本上已经将指纹纹路和背景完全划分开



通过使用课本上的图篇背景和需要分割的图片灰度值没有明显的波谷差别。所以可以很好的分析全局阈值和OTSU算法的区别。全局方法会将背景和需要识别的物体混在一起

5. 实验思考

    1. 全局阈值
        1. 优点：
            1. 当图像直方图中具有明显的波谷或者说波峰之间的分界十分明显时，采用基本的全局阈值处理往往可以很好的解决问题
            2. 简单，易于理解和实现。
        2. 缺点
            1. 效果受噪声和光照变化的影响较大，不适用于所有图像，因为一个固定的阈值可能无法适应图像的变化。
            2. 在原图的直方图分布并没有显著的波谷，或者说仅存在一波峰甚至没有波峰时，采用上述的方法进行迭代分类效果并不理想
    2. OTSU方法
        1. 优点
            1. 在原图的直方图分布并没有显著的波谷，或者说仅存在一波峰甚至没有波峰时，采用上述的方法进行迭代分类效果并不理想，这个时候我们可以采用Otsu方法进行最佳全局阈值处理。Otsu方法采用了类间方差最大化的思想，也就是说方差越大，越接近于正确分割图像的阈值，而这种最佳测度完全基于直接由图像直方图得到的参数。
        2. 缺点
            1. 当图像收到光照或者噪声干扰的时候直方图没有明显波峰波谷时，OTSU也不能很好的分割图片

Canny边缘检测

1. 实验内容

    1. 使用Canny算子检测边界，可调用函数，分析每一步操作的作用，思考为什么会比Sobel方法更好

2. 实验思路

    1. 自己实现Canny算法

    1. 一维高斯卷积，对每行进行高斯卷积。通过高斯滤波器，首先对于图片进行高斯滤波，消除噪声。高斯滤波器，利用3*3的高斯模版进行高斯卷积
    2. 用一阶偏导有限差分计算梯度幅值和方向，将计算所得的梯度和梯度方向，存放在对应的不同的数组中。
    3. NMS非极大值抑制：对于每一个点寻找最近的梯度方向的方向d_k。对比该点梯度大小，如果至少小于沿d_k的两个邻居之一，则令$g_N(x,y)=0$,否则$g_N(x,y)=M(x,y)$.
    4. 双阈值算法检测和连接边缘。对于大于最大阈值的点，将其标记为边缘，小于最小阈值的点，不是边界。对于在中间的点，通过弱边缘点补充连接强边缘点，也就是判断连通性。是否和已经被判断为边缘的点相连，如果连接就是边缘。否则中间部分的点就不是边缘。
2. 调用opencv的库函数
    1. 通过使用opencv库函数canny以及设置参数来对比和自己实现的函数区别。
3. 使用Sobel算子进行图片检测
    1. 首先先使用高斯模糊去噪声
    2. 计算梯度x方向和y方向
    3. 合并梯度，并返回计算后的图像。

3. 实验代码

    1. 调用Canny

```
blur(res, res, Size(3,3));
Canny(res, edge, high, low, 3);
```

    2. 实现Sobel算子边缘检测

```
// 使用高斯模糊去噪声
cv::GaussianBlur(grayImage, grayImage, cv::Size(3, 3), 0, 0);
// 计算梯度
cv::Sobel(grayImage, grad_x, CV_16S, 1, 0, 3, scale, delta,
cv::BORDER_DEFAULT);
cv::Sobel(grayImage, grad_y, CV_16S, 0, 1, 3, scale, delta,
cv::BORDER_DEFAULT);
// 合并梯度（近似）
cv::addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0, edges);
```

    3. 自己实现

```
 gaussianFilter(img, gauss_img); //高斯滤波器

// 用一阶偏导有限差分计算梯度幅值和方向
Mat gradXY, theta;
getGrandient(gauss_img, gradXY, theta);

// 局部非极大值抑制
Mat local_img;
nonLocalMaxValue(gradXY, theta, local_img);
```

```
// 用双阈值算法检测和连接边缘
Mat dst;
doubleThreshold(low, high, local_img, dst);
```

计算梯度

```
void getGrandient(Mat& img, Mat& gradXY, Mat& theta)
{
    gradXY = Mat::zeros(img.size(), CV_8U);
    theta = Mat::zeros(img.size(), CV_8U);

    for (int j = 1; j < img.rows - 1; j++)
    {
        for (int i = 1; i < img.cols - 1; i++)
        {
            double gradY = double(img.ptr<uchar>(j - 1)[i - 1] + 2 *
img.ptr<uchar>(j - 1)[i] + img.ptr<uchar>(j - 1)[i + 1] - img.ptr<uchar>(j +
1)[i - 1] - 2 * img.ptr<uchar>(j + 1)[i] - img.ptr<uchar>(j + 1)[i + 1]);
            double gradX = double(img.ptr<uchar>(j - 1)[i + 1] + 2 *
img.ptr<uchar>(j)[i + 1] + img.ptr<uchar>(j + 1)[i + 1] - img.ptr<uchar>(j -
1)[i - 1] - 2 * img.ptr<uchar>(j)[i - 1] - img.ptr<uchar>(j + 1)[i - 1]);

            gradXY.ptr<uchar>(j)[i] = sqrt(gradX * gradX + gradY * gradY);
//计算梯度
            theta.ptr<uchar>(j)[i] = atan(gradY / gradX);
//计算梯度方向
        }
    }
}
```

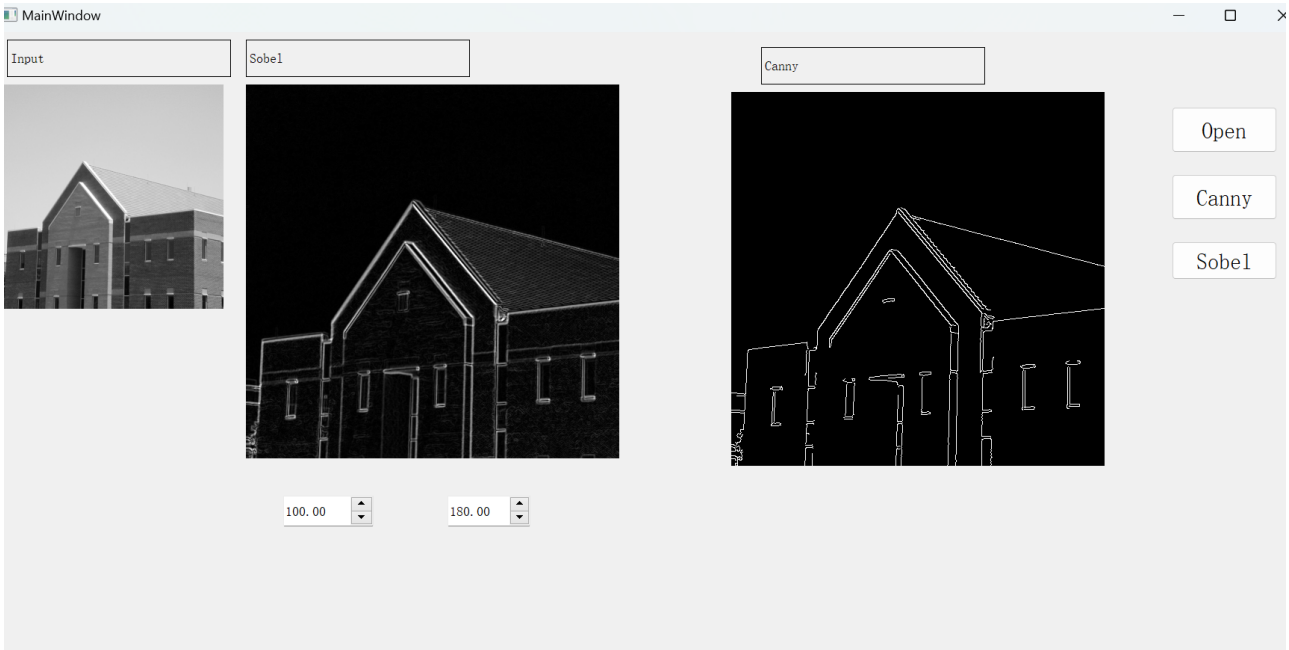
双阈值检测+判断连通性

```
void doubleThresholdLink(Mat& img)
{
    // 循环找到强边缘点，把其领域内的弱边缘点变为强边缘点
    for (int j = 1; j < img.rows - 2; j++)
    {
        for (int i = 1; i < img.cols - 2; i++)
        {
            // 如果该点是强边缘点
            if (img.ptr<uchar>(j)[i] == 255)
            {
                // 遍历该强边缘点领域
                for (int m = -1; m < 1; m++)
                {
                    for (int n = -1; n < 1; n++)
                    {
```

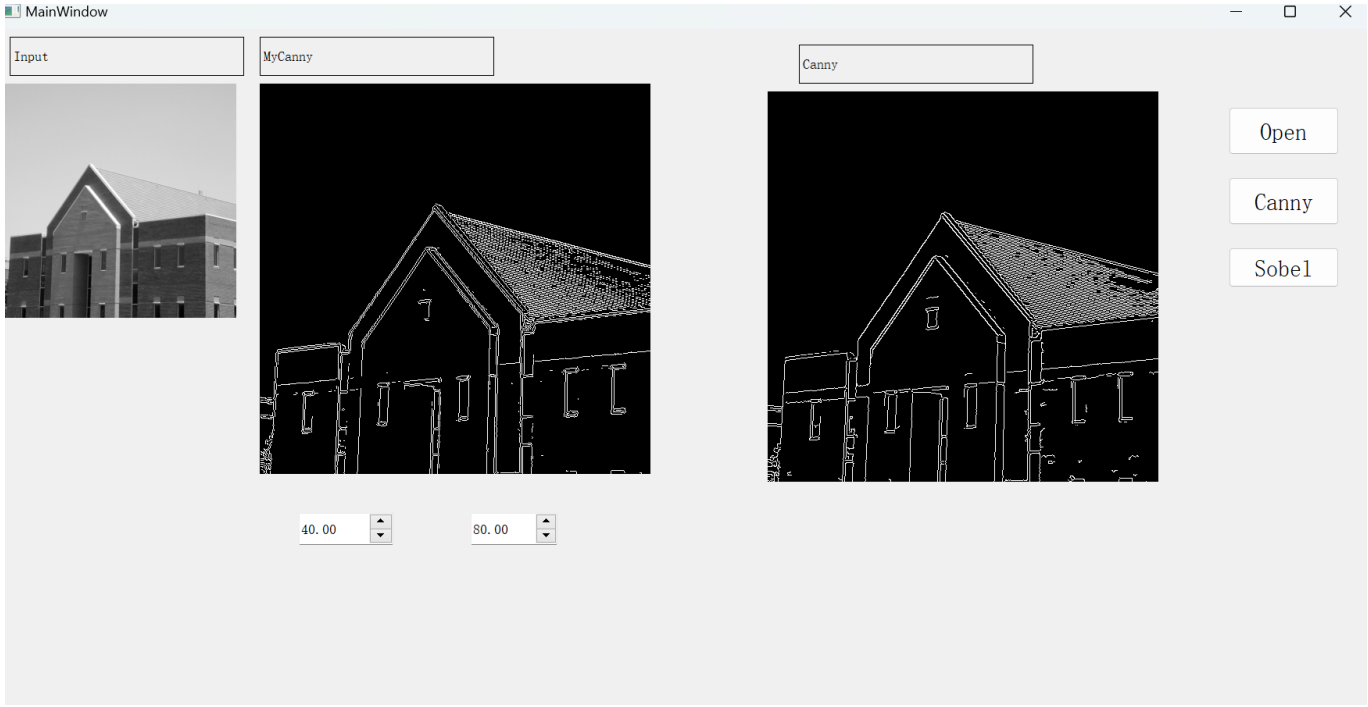
```
                        // 该点为弱边缘点（不是强边缘点，也不是被抑制的0点）
                        if (img.ptr<uchar>(j + m)[i + n] != 0 &&
img.ptr<uchar>(j + m)[i + n] != 255)
                        {
                            img.ptr<uchar>(j + m)[i + n] = 255; //该弱边缘点
补充为强边缘点
                        }
                    }
                }
            }
        }

        for (int j = 0; j < img.rows - 1; j++)
        {
            for (int i = 0; i < img.cols - 1; i++)
            {
                // 如果该点依旧是弱边缘点，及此点是孤立边缘点
                if (img.ptr<uchar>(j)[i] != 255 && img.ptr<uchar>(j)[i] != 255)
                {
                    img.ptr<uchar>(j)[i] = 0; //该孤立弱边缘点抑制
                }
            }
        }
    }
```

4. 实验结果



通过调节双阈值的大小，使得Canny结果和Sobel的结果大致相同



对比自己实现的Canny和opencv的Canny，可以调节双阈值的参数，来动态的变化图像处理的结果

5. 实验思考
    1. Canny算法每一步的作用
        1. 高斯滤波：边缘属于图像中的高频信号，但噪声也属于高频信号，因此可以先对图像做一次高斯模糊，抑制噪声，虽然边缘也会被抑制，但后期连接弱边缘时，被抑制的边缘还可能被重置为强边缘，而被抑制的噪声因为孤立而不被认为强边缘。
        2. 计算梯度并合并：Sobel 算子已经能较好的检测处边缘。但这不符合最优边缘检测的准则之一
        3. 非最大化抑制：细化边缘。在边缘检测的第一步中，使用梯度信息找到了潜在的边缘点。然而，由于像素值是连续的，实际上形成的边缘可能会很宽。非最大值抑制通过在梯度方向上对像素进行比较，只保留梯度方向上的局部最大值，从而细化了检测到的边缘，使其更符合实际边缘的位置。
        4. 双阈值：减少伪边缘点。但许多边缘之间是断断续续的，因此对上一步结果中的每一个弱边缘判断，如果周围 8 个邻接像素有一个是强边缘，则认为该弱边缘也是强边缘，否则视作孤立点抛弃（噪声就是这么被抛弃的）。
    2. 和Sobel算子区别
        1. Sobel算子的边缘检测，只是Canny的前两个步骤。区别和重点在于Canny的后两部。Sobel算子已经能较好的检测处边缘。但这不符合最优边缘检测的准则之一
        2. Canny引入了非最大值抑制，该步骤通过在梯度方向上选择局部最大值来细化边缘，使得检测到的边缘更加细致和准确。
        3. Canny使用了双阈值边缘跟踪，根据像素的强度值将边缘分为强边缘、弱边缘和非边缘。这有助于减少误检率，并在一定程度上提高了抗噪声性能。
        4. Canny具有手动调节性，可以根据不同的图像调节双阈值的大小得到不同的边缘处理效果。
        5. 由于Canny包括高斯滤波和非最大值抑制等步骤，它对噪声的抵抗能力更强，相对于简单的Sobel算子更适用于实际应用中的图像。