实验名称：用 CImg 重写、封装给定的 Canny 代码，并测试

实验要求：

1. 重写Code2，并使用测试数据测试。
2. 所有的图像读写、数据处理只能用CImg库。

实验步骤：

重写部分：

1. 将原始图片转为灰度图的形式。
2. 对上一步的结果进行高斯模糊。
3. 使用Sobel算法处理上一步的结果。
4. 对上一步的结果进行Non-Maxima Suppression（非极大值抑制）。
5. 对上一步的结果进行双阈值处理（Double Threshold）

增加函数：

1. 把相邻的边缘连成长的线条。
2. 删除长度小于20的边缘。

实验过程：

首先，在所给的code2中，它使用了opencv库的cv::Mat结构来存储图像，实际上就是一个矩阵。可以看到它有这样一些声明：

```
grayscaled = Mat(img.rows, img.cols, CV_8UC1)
angles = Mat(gFiltered.rows - 2*size, gFiltered.cols - 2*size, CV_32FC1)
```

在上面的代码中，声明传入的参数不止有矩阵（图像）的长、宽，还传入了一个参数，它的取值有：CV_8UC1、CV_32FC1等，这个参数描述的是存储图像的值的类型、色道数等信息，如CV_8UC1，8U指的是unsigned 8bits，而1就代表单通道。

对应到CImg中，我们直接用CImg()来存储图像，它的声明类似这样：

```
CImg<unsigned char> gray(img.width(), img.height(), 1, 1, 0);
```

最后三个参数分别代表图像深度、色道数、初始化值。

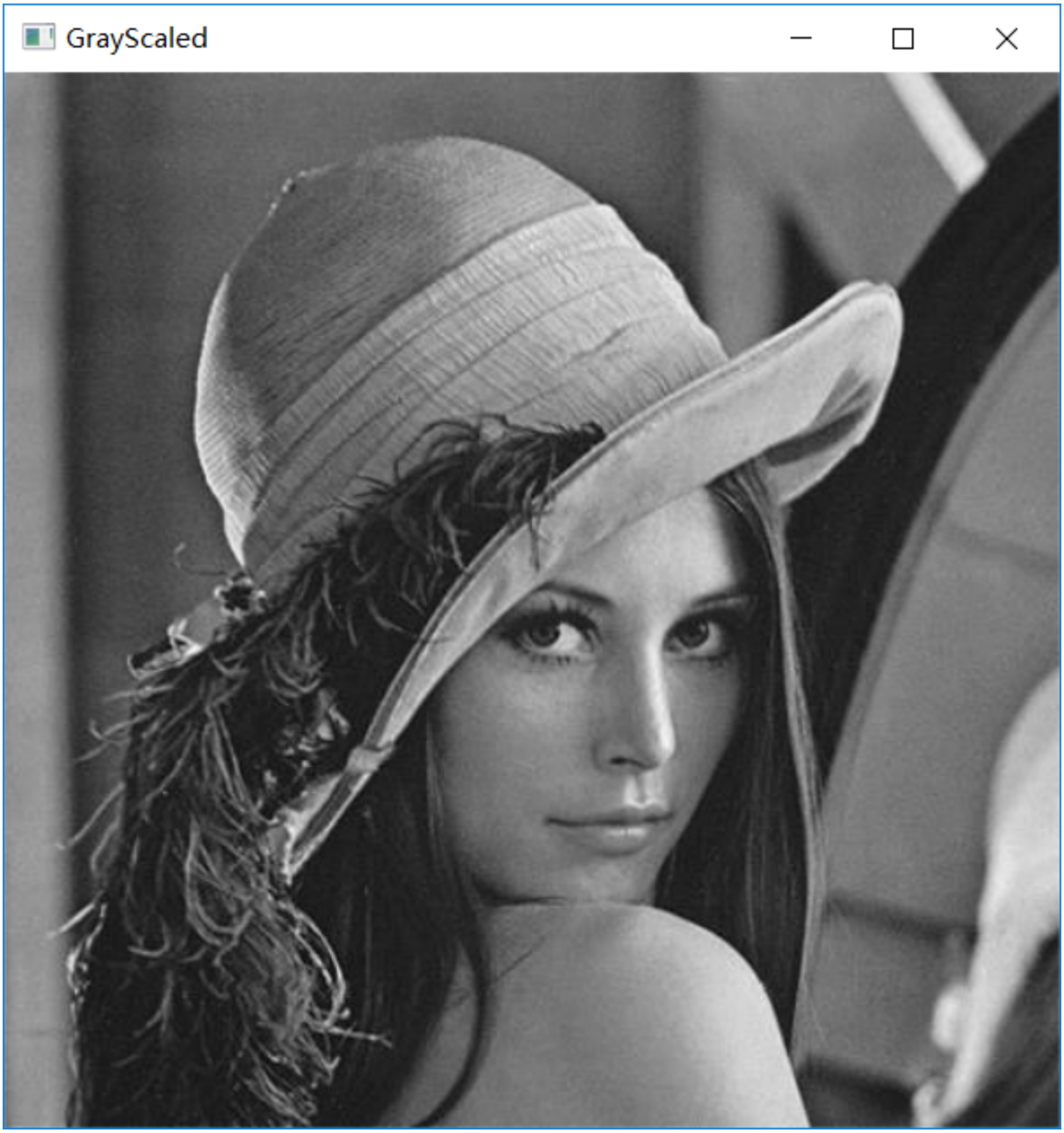
还要注意的一点是opencv与CImg的坐标问题，它们使用的坐标系并不同。当在opencv中获取 `mat.at<T>(i, j)` 时，获取的是竖直方向上坐标为i，水平方向上坐标为j的像素，CImg中则刚好相反。

重写部分

1. 将原始图片转为灰度图的形式。

   这一步就是通过循环，根据原始图片每一像素的值来计算相应的灰度值，填充到灰度图中，其中在使用的计算公式方面可以改进。其原本的计算公式为 `gray = (r * 0.2126 + g * 0.7152 + b * 0.0722)`，这里进行了低速的浮点运算。可以把它转为整数乘除，进而把除法改为移位运算，得到最终的计算公式：`gray = (r * 19595 + g * 38469 + b * 7472) >> 16`。

   实现效果：

2. 对上一步的结果进行高斯模糊。

   首先创建一个filter，它接下来被用来以正态分布计算图像中每个像素的变换，为此我们需要两个参数：模糊半径r与模糊参数σ。这两个参数对后续的图像处理有很大的影响。实现上，对filter的每个元素，在其相应的模糊半径范围内先计算加权正态平均：

$$G(r) = \frac{1}{\sqrt{2\pi\sigma^2}^N} e^{-r^2/(2\sigma^2)}$$

再除以权重之和就得到了filter。

之后就要使用它来做高斯模糊了，这里还存在一个边界处理的问题，也就是如何对边界上的像素进行模糊处理（可能取到越过边界的值）。在code2中，它直接忽略了这些像素，也就导致了处理过后图像会缩小。这种做法在模糊半径小的情况下还可以接受，但若是模糊半径设置的大些，显然会丢失过多像素，不可行。在这里我们可以做一次reflect的操作，也就是在对那些处于边界上的像素做模糊处理时，若要获取超出边界的值，取它对于处理中像素的相反方向的值，这样就避免了丢失像素。举个例子，若`i = 0`，此时要对元素`i - 1`进行计算，显然会出错，此时就计算`i + abs(i - (i - 1))`的值，即元素`i + 1`的值。

3. 使用Sobel算法处理上一步的结果。

这一步需要两个filter，分别从横向与纵向与图像做平面卷积。需要注意的是在这一步中的计算有一些分歧。如果选择的filter矩阵为：

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

此时应采用Hadamard product，也就是左上乘左上，依序一一对应相乘。而若是选择filter矩阵为：

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

此时需要采用kernel convolution，也就是右下乘左上，如图：

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right)[2,2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

同时，在这一步中计算像素的梯度方向：

$$\mathbf{\Theta} = \arctan\left(\frac{\mathbf{G_y}}{\mathbf{G_x}}\right)$$

在重写过程中我发现原代码在这里是这样处理梯度方向的：

```
if(sumx==0) //Arctan Fix
    angles.at<float>(i-size, j-size) = 90;
else
    angles.at<float>(i-size, j-size) = atan(sumy/sumx);
```

因为atan函数得到的是弧度，而在后面进行非极大值抑制的函数中，它又是对角度进行判断，所以这里应该修改为：

```
if(sumx==0) //Arctan Fix
    angles(i, j) = 90;
else
    angles(i, j) = atan(sumy / sumx) * (180.0 / M_PI);
```

以及，这一步也用到了reflect。

4. 对上一步的结果进行Non-Maxima Suppression（非极大值抑制）。

   所谓极大值抑制即是用来压缩边缘的，它可以使得经过Sobel算法处理得到的粗略边缘变得更窄，也就是更加贴近边缘。

   在这里要用到上一步Sobel算法计算得到的梯度方向。注意现在已处理的图像是只有单色道灰度值的，所以梯度方向指向的是亮度变化最大的方向。举个例子，如果某个像素点角度为0°或180°，则说明该点处的法向为水平（即如果有边缘，在该点的切线方向为纵向）。所以此时我们应该把该点与其水平方向的邻居比较，若其邻居有比它大的，则将其设为弱点（非极大值抑制），这样一条粗略边缘就会被缩减。

5. 对上一步的结果进行双阈值处理（Double Threshold）

   双阈值处理对于图像的细节有很好的过滤效果，一般来说低阈值为高阈值的二分之一，若阈值设置的高，则图像会省去许多细节；而若是阈值设置的比较低，则图像会多很多不必要的细节。

   我的做法是，第一遍判断时，若像素的值大于高阈值则设为强点（255），低于低阈值则设为弱点（0），位于阈值之间的普通点则先看它们8-邻阈中有无强点，有则将其设为强点。若无强点而有普通点，则再往外扩一圈，若这一圈（16-邻阈）中有强点，则将其设为强点，否则结束。这样做对普通点有较好的保留效果，避免删除过多（设置过多弱点）。

   在重写过程中，我发现原代码是这样进行边界判断的：

```
if(x <= 0 || y <= 0 || EdgeMat.rows || y > EdgeMat.cols) //Out of bounds
    continue;
...
if(x < 0 || y < 0 || x > EdgeMat.rows || y > EdgeMat.cols) //Out of bounds
    continue;
```

   显然应该改为：

```
if(x < 0 || y < 0 || x >= thres.width() || y >= thres.height()){
    continue;
```

增加函数

1. 把相邻的边缘连成长的线条。

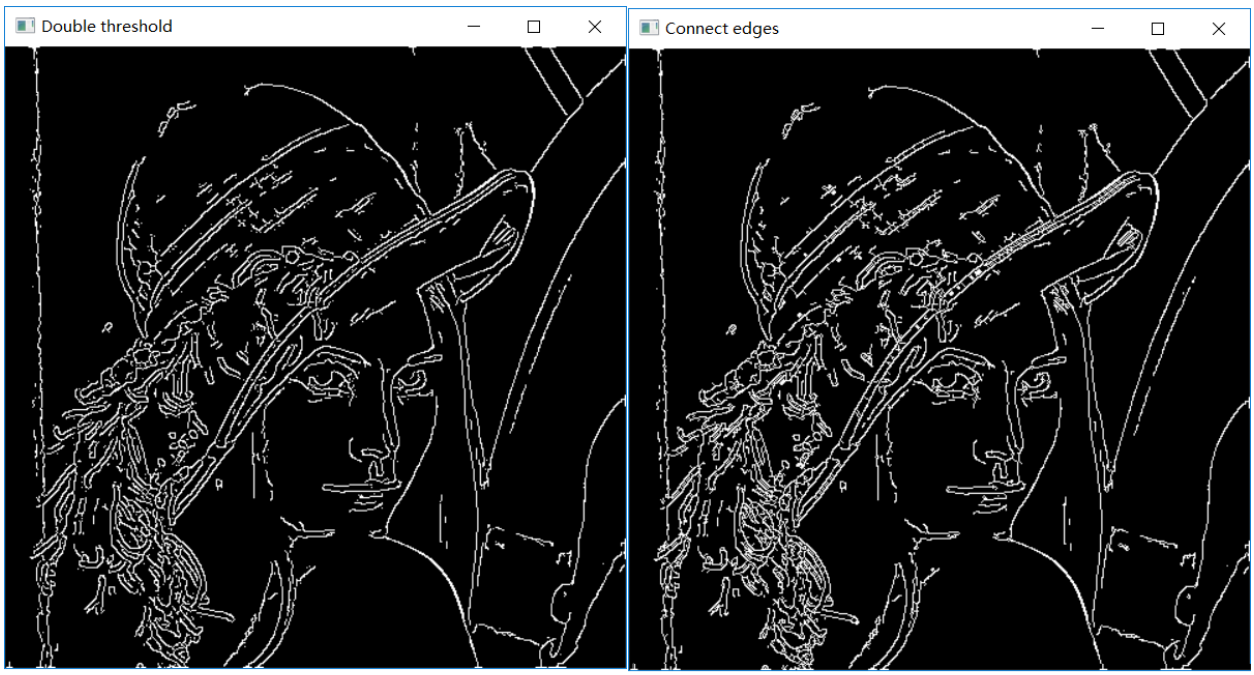
   这个问题的重点在于如何判断相邻的边缘，前面双阈值处理的方法启发了我：对每一个像素点，检查其8-邻阈内有无不相邻的多个强点（相邻则不连接），若有，则将这一像素点设为强点。这样做可以避免连接一大块密集的强点，同时对真正的相邻边缘也能进行连接。

   检查8-邻阈的实现部分：

```
if(conn(i, j) == 0){    //此点为弱点
    vector<pair<int, int>> store;   //存储八邻阈有无两条边
    for(int x = i - 1; x <= i + 1; x++){
        for(int y = j - 1; y <= j + 1; y++){
            if(x < 0 || y < 0 || x >= thres.width() || y >= thres.height()){
```

```
                continue;
            }
            if(thres(x, y) > 0){
                store.push_back(make_pair(x, y));
            }
        }
    }
    if(store.size() < 2){   //无相邻的边
        continue;
    }
    bool flag = true;
    for(int a = 0; a < store.size() - 1; a++){
        for(int b = a + 1; b < store.size(); b++){
            if(abs(store[a].first - store[b].first) + abs(store[a].second -
store[b].second) == 1){
                flag = false;   //强点相邻，不符合要求
            }
        }
    }
    if(flag){
        conn(i, j) = 255;
    }
}
```

结果（r为gaussian filter的半径，实际矩阵size为1 + 2 * r）：



r=2, σ=2, low threshold = 20, high threshold = 40

2. 删除长度小于20的边缘。

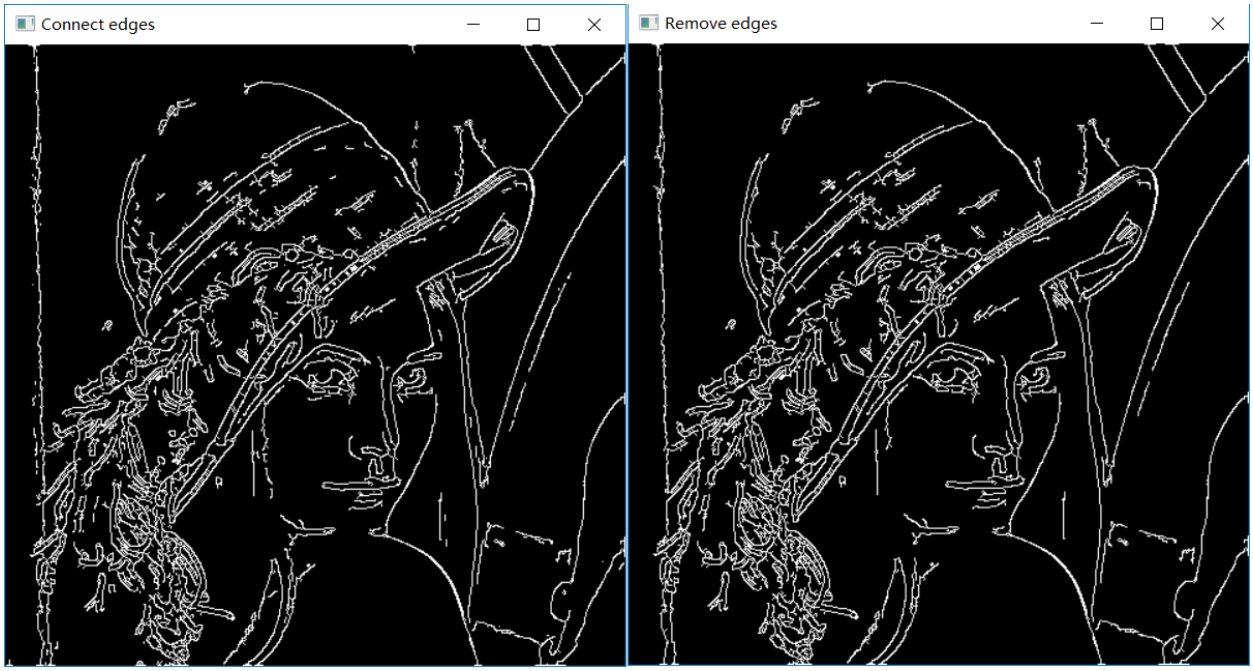
这个问题的难点就在于如何判断是否为一条边缘的同时记录长度了。我的做法是记录下每个像素是否访问过，对于图像的每一个未访问过的像素，从它开始做bfs，将其8-邻阈中存在的相邻点加入队列，并标记为已访问过。同时维护一个vector来记录bfs经过的像素数，bfs退出后，检查vector中元素个数是否超过20。此次bfs过的点下次不会再进行bfs。

这样做的前提是默认一些靠的较近的离散点是属于一条边缘的，所以对于一些比较不友好的网格类型的图或者说毛刺比较多的图的删除效果就不会太好。

对每个像素：

```
if(visited(i, j)){  //已访问过
    continue;
}
mqueue.push(make_pair(i, j));
visited(i, j) = 1;
vector<pair<int, int>> path;
while(!mqueue.empty()){
    int head_i = mqueue.front().first, head_j = mqueue.front().second;
    path.push_back(make_pair(head_i, head_j));
    mqueue.pop();
    for(int x = head_i - 1; x <= head_i + 1; x++){
        for(int y = head_j - 1; y <= head_j + 1; y++){
            if(x < 0 || y < 0 || x >= imgleft.width() || y >= imgleft.height()){
                continue;
            }
            else{
                if(imgleft(x, y) > 0 && !visited(x, y)){
                    mqueue.push(make_pair(x, y));
                    visited(x, y) = 1;
                }
            }
        }
    }
}
if(path.size() < 20){
    for(int x = 0; x < path.size(); x++){
        imgleft(path[x].first, path[x].second) = 0;
    }
}
```

结果：

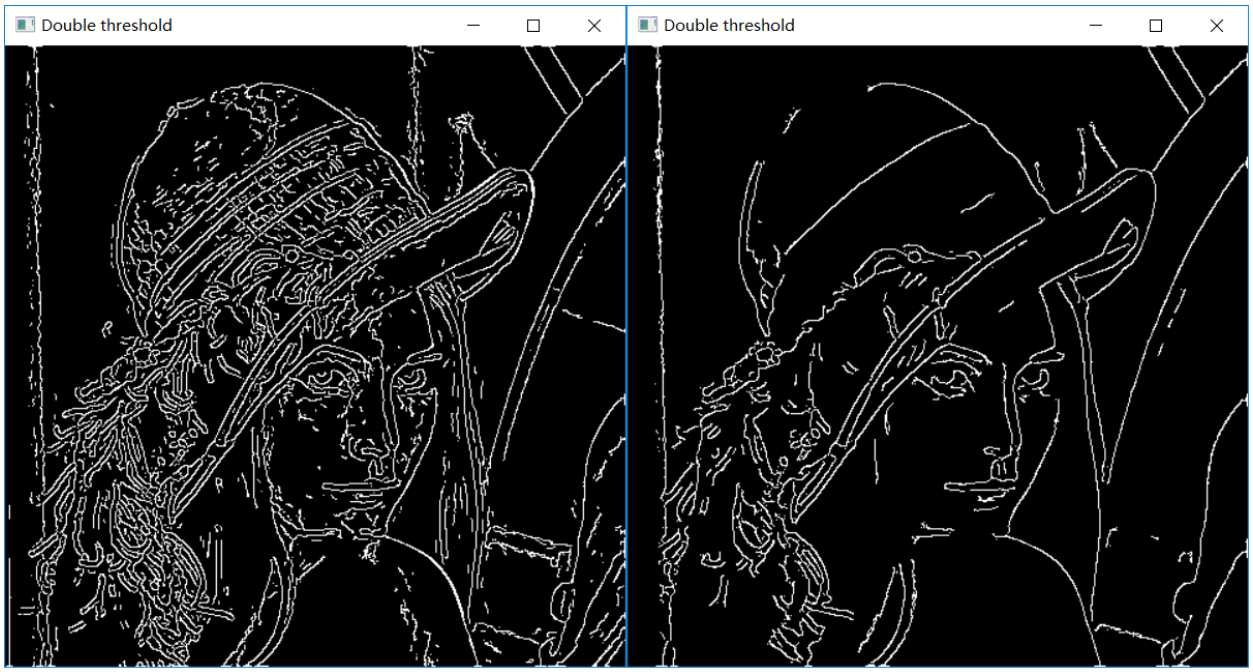


r=2, σ=2, low threshold=20, high threshold=40

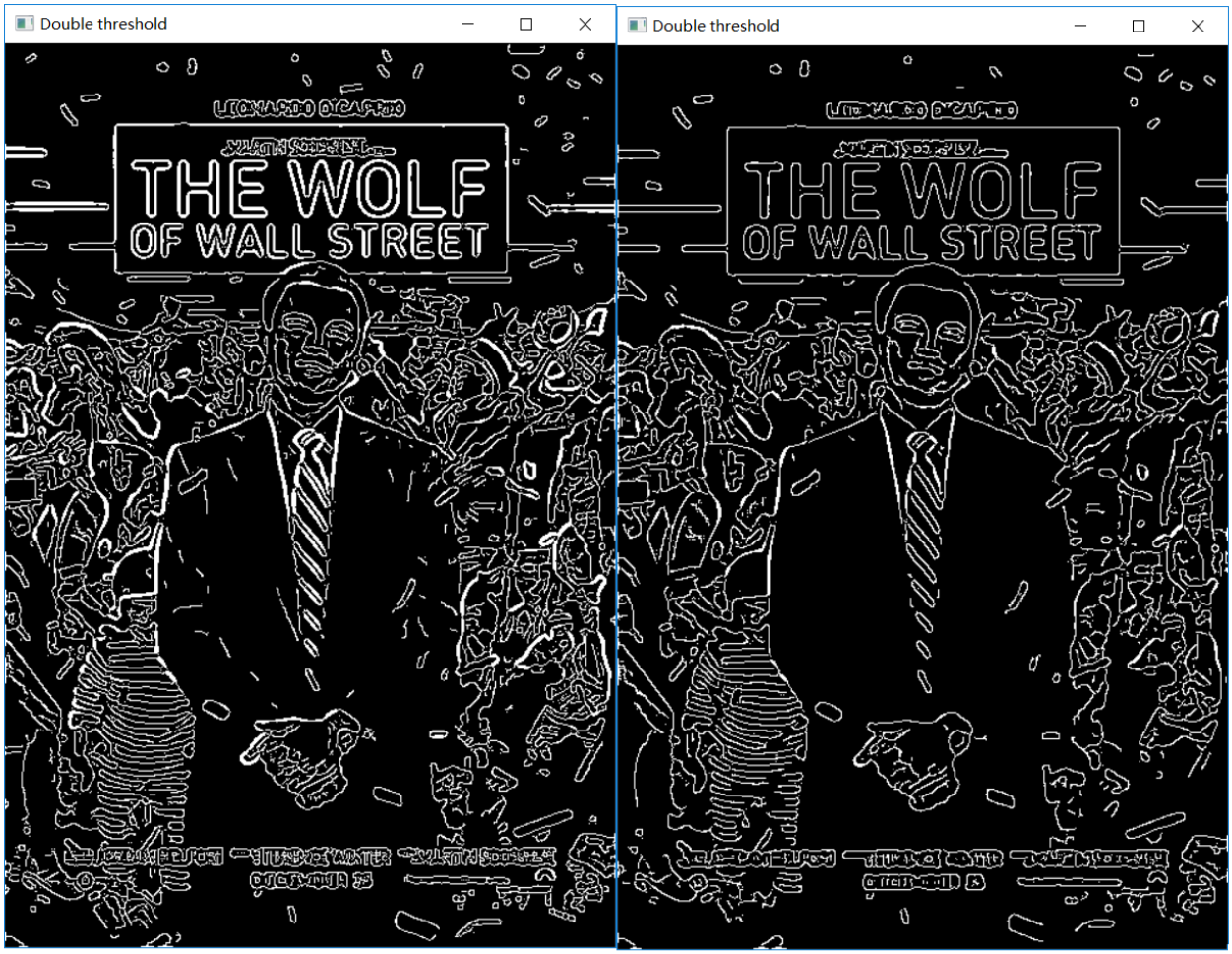
可见删去了一些短边。

测试参数：

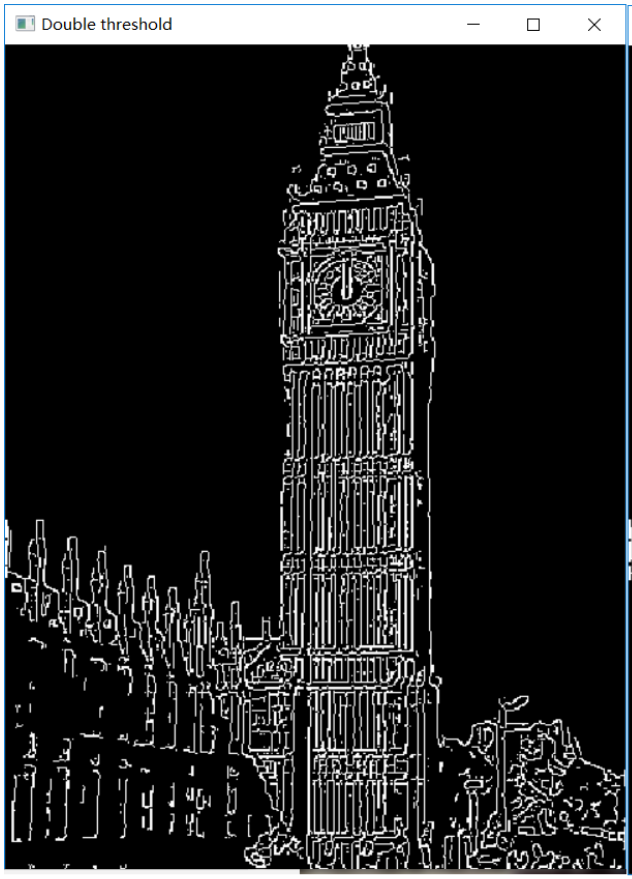
1. 参数：gaussian filter矩阵的半径r（实际矩阵size为1 + 2 * r），其他参数为`σ = 2, low threshold = 20, high threshold = 40`。结果（双阈值处理后）：



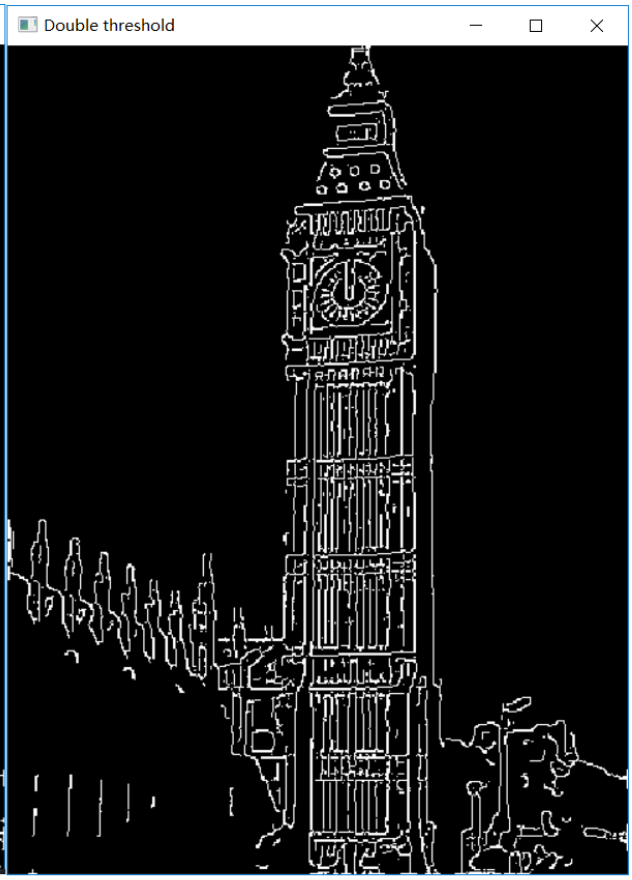
r = 2　　r = 5


Double threshold
THE WOLF
OF WALL STREET
Double threshold
THE WOLF
OF WALL STREET

r = 2 r = 5

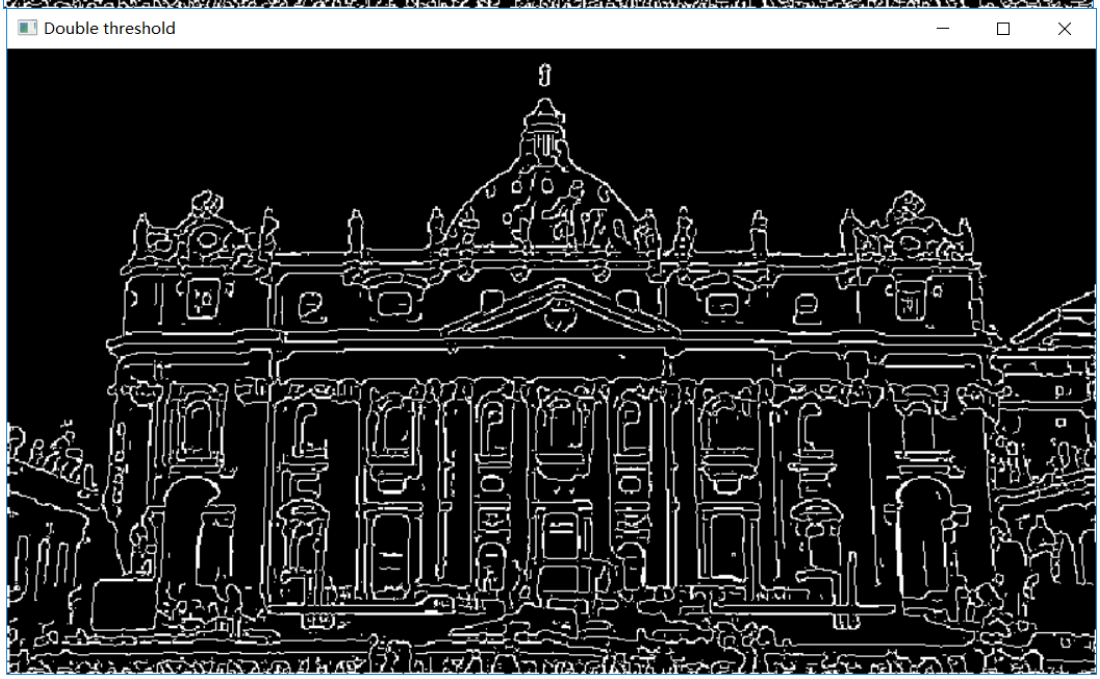


r = 2



r = 5

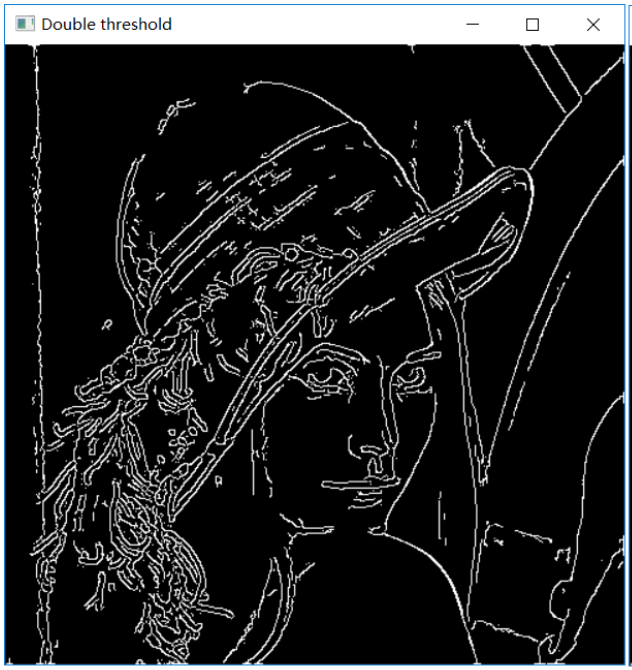
r = 2



r = 5

可以看到，gaussian filter矩阵的size越大，每个像素点相对周围做正态平均的范围就会越大，也就是会最模糊，导致最终结果捕捉的细节就会越少。

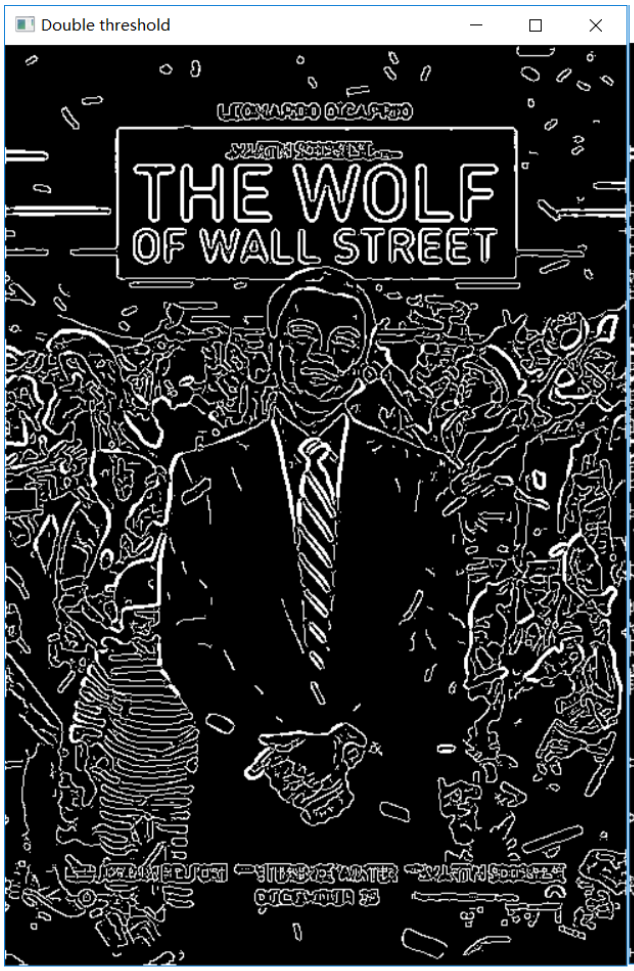
2. 参数：σ，其他参数为`r = 2, low threshold = 20, high threshold = 40`。结果（双阈值处理后）：


Double threshold

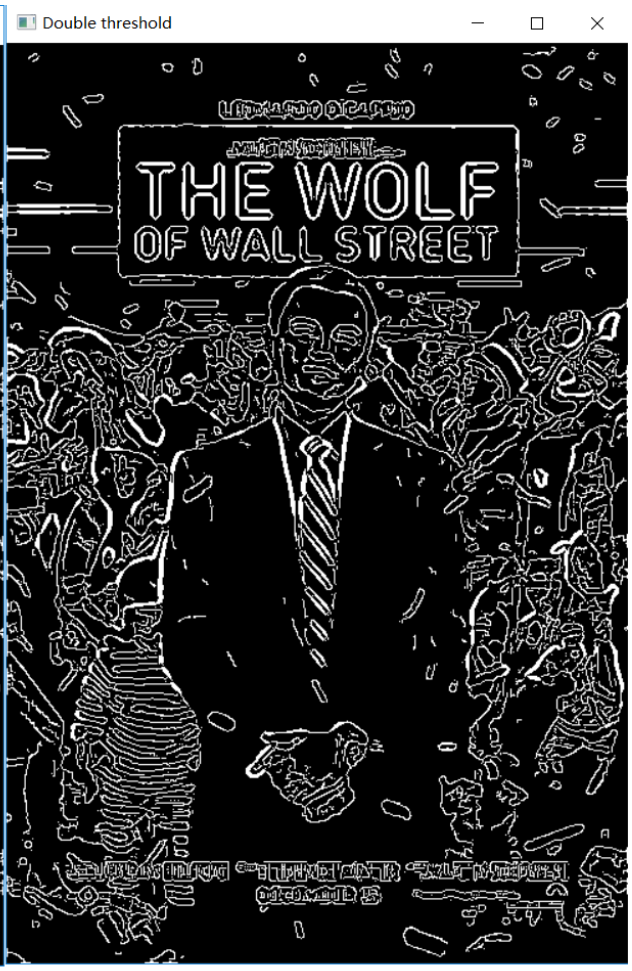
σ = 2


Double threshold

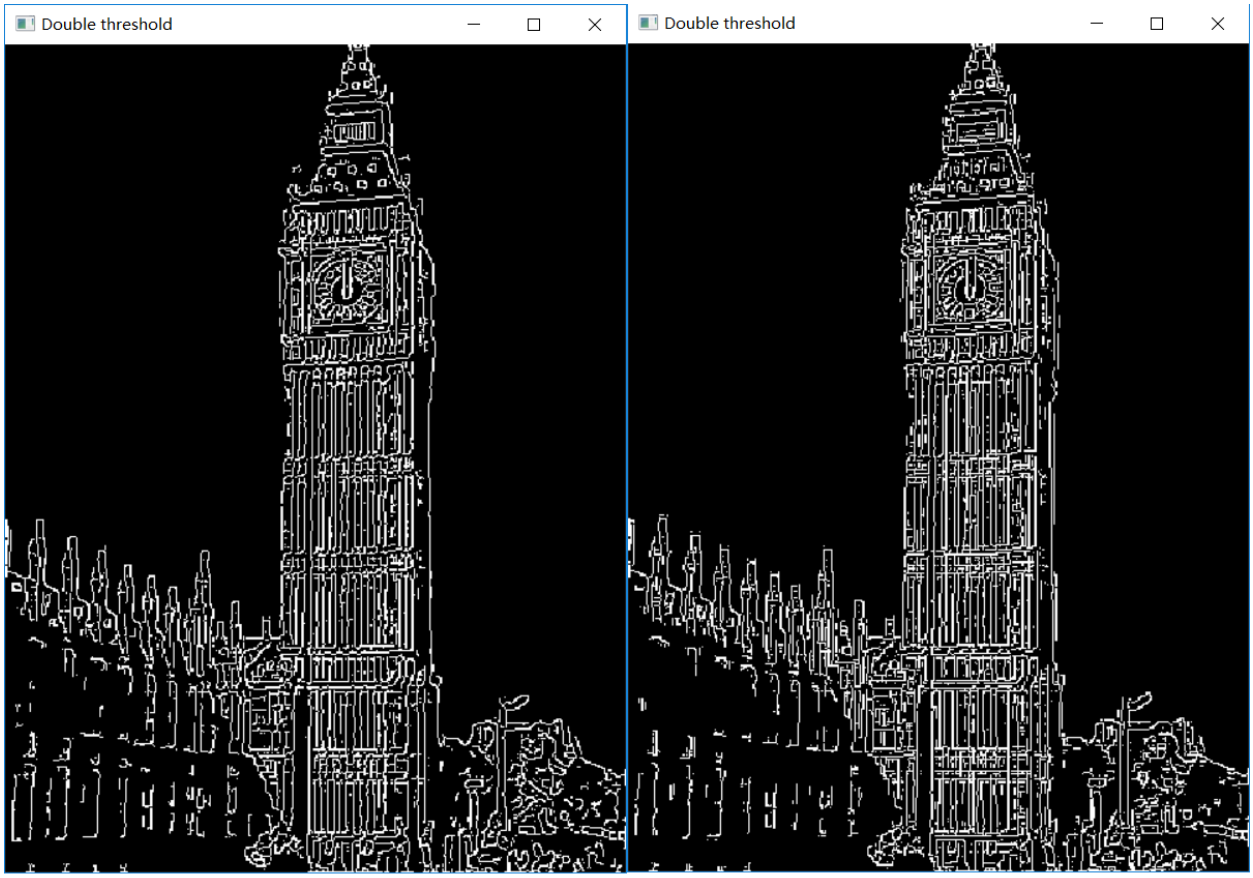
σ = 10


Double threshold
THE WOLF
OF WALL STREET

σ = 2


Double threshold
THE WOLF
OF WALL STREET

σ = 10

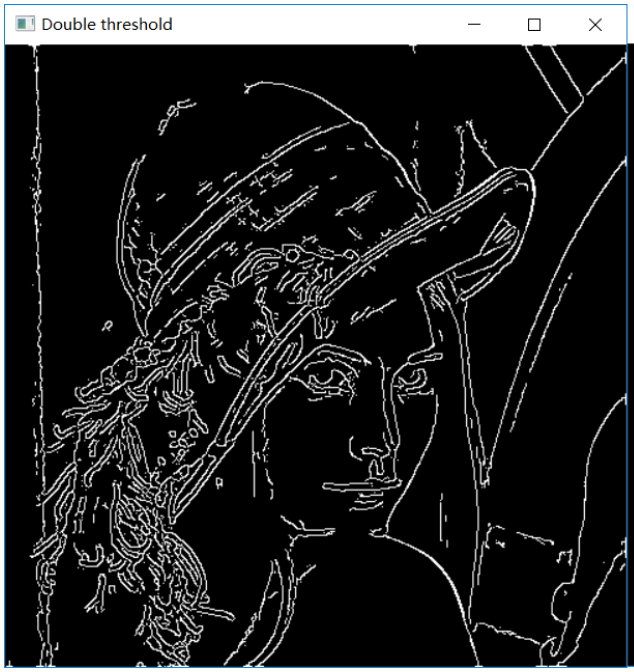


σ = 2

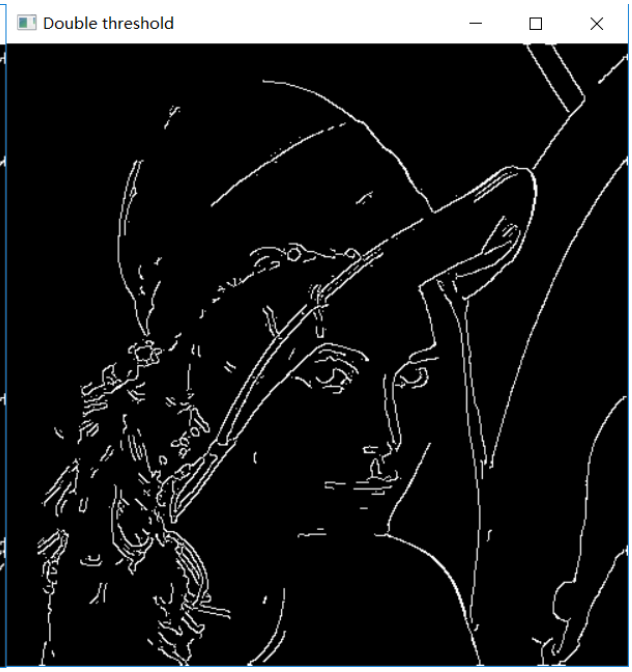
σ = 10

可以看到，σ相比于gaussian filter矩阵的size来说虽然影响没有那么大，但也会导致最终结果捕捉的细节的多少。σ越大，细节（小的、杂的边或像素点）会多些。

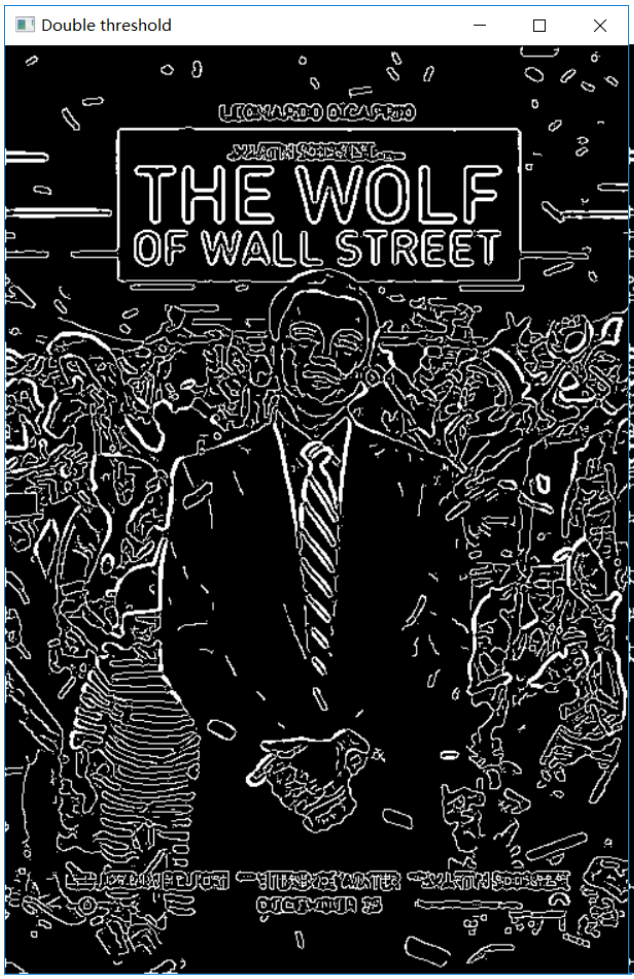
3. 参数：高低阈值，其他参数为`r = 2, σ = 2`。结果（双阈值处理后）：



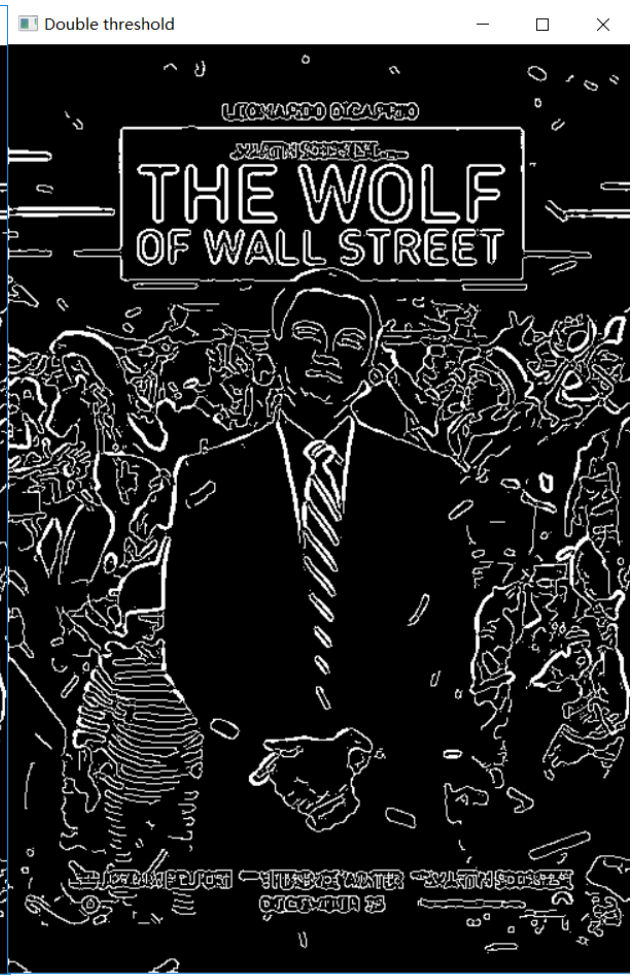
l = 20, h = 40



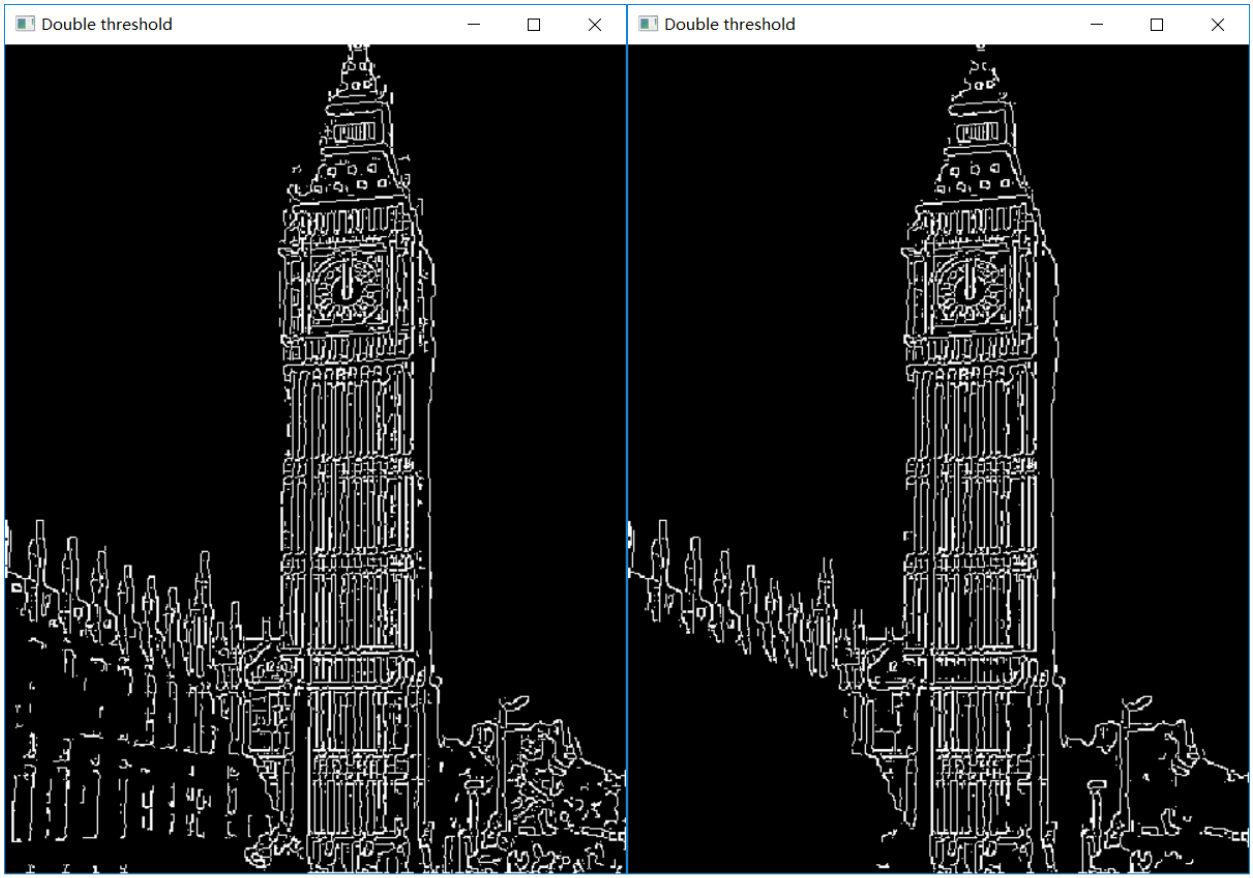
l = 40, h = 80



l = 20, h = 40



l = 40, h = 80



l = 20, h = 40

l = 40, h = 80

l = 20,
h = 40

l = 40,
h = 80

可以明显看出双阈值的筛选效果，设置高阈值可以提供更简洁的信息，而设置低阈值可以捕获到更多的信息。

以上就是本次实验的内容，非常期待下次的实验。