实验三：空域滤波

学号：SA22225286 姓名：孟寅磊 日期：20221002

实验内容

> 1. 利用均值模板平滑灰度图像
>
>   具体内容：利用OpenCV对图像像素进行操作，分别利用$3*3$、$5*5$、$9*9$的均值模板平滑灰度图像
>
> 2. 利用高斯模板平滑灰度图像
>
>   具体内容：利用OpenCV对图像像素进行操作，分别利用$3*3$、$5*5$、$9*9$的高斯模板平滑灰度图像
>
> 3. 利用Laplacian、Robert、Sobel模板锐化灰度图像
>
>   具体内容：利用OpenCV对图像像素进行操作，分别利用Laplacian、Robert、Sobel模板锐化灰度图像
>
> 4. 利用高提升滤波算法增强灰度图像
>
>   具体内容：利用OpenCV对图像像素进行操作，设计高提升滤波算法增强图像
>
> 5. 利用均值模板平滑彩色图像
>
>   具体内容：利用OpenCV对图像像素的RGB三个通道进行操作，分别利用$3*3$、$5*5$、$9*9$的均值模板平滑彩色图像
>
> 6. 利用高斯模板平滑彩色图像
>
>   具体内容：利用OpenCV对图像像素的RGB三个通道进行操作，分别利用$3*3$、$5*5$、$9*9$的高斯模板平滑彩色图像
>
> 7. 利用Laplacian、Robert、Sobel模板锐化彩色图像
>
>   具体内容：利用OpenCV对图像像素的RGB三个通道进行操作，分别利用Laplacian、Robert、Sobel模板锐化彩色图像

实验完成情况

1. 基本原理

线性空间滤波器在图像f和滤波器核w之间执行乘积之和运算，我们在图像中移动核，使其中心和各个像素重合，然后将核的系数与对应像素相乘再相加赋于原像素。一般来说，大小为$m \times n$的核对大小为$M \times N$的图像的线性空间滤波可以表示为

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{3.1}$$

式中，x和y发生变化，使得核的中心能够访问f中的每个像素。(x,y)的值不变时，该式实现乘积之和。这是实现线性滤波的核心工具，我们首先编写两个函数，使得该运算能分别在灰度图像和彩色图像上执行。

> 用于对灰度图像执行卷积操作的函数

```
/// @brief convolution operation to a gray-scale image using an opertor
/// @param src input image
```

```
/// @param dst the result of convolution, it will be used by the caller
/// @param kernel the operator
/// @param border_type the borderType, using BORDER_CONSTANT or
///                    BORDER_REPLICATE or BORDER_REFLECT
void convolution(Mat &src, vector<vector<double>> &dst,
                 vector<vector<double>> kernel, int border_type) {
    CV_Assert(src.type() == CV_8UC1  &&
              src.rows == dst.size() &&
              src.cols == dst[0].size());
    int border = (kernel.size() - 1) / 2;
    int i_dst  = 0;
    int j_dst  = 0;
    double con = 0;
    Mat filled;
    // form the border around the image
    copyMakeBorder(src, filled, border, border, border, border,
border_type);
    int rows = src.rows + 2 * border;
    int cols = src.cols + 2 * border;
    // process each pixel
    for (int i = border; i < rows - border; ++i) {
        for (int j = border; j < cols - border; ++j) {
            i_dst = i - border;
            j_dst = j - border;
            // convolution operation
            con = 0;
            for (int ki = i - border; ki <= i + border; ++ki)
                for (int kj = j - border; kj <= j + border; ++kj)
                    con += (kernel[ki-i_dst][kj-j_dst] *
                            filled.at<uchar>(ki, kj));
            dst[i_dst][j_dst] = con;
        }
    }
}
```

用于对彩色图像执行卷积操作的函数

```
/// @brief convolution operation to a rgb image using an opertor
/// @param src input image
/// @param dst the result of convolution, it will be used by the caller
/// @param kernel the operator
/// @param border_type the borderType, using BORDER_CONSTANT or
///                    BORDER_REPLICATE or BORDER_REFLECT
void convolution(Mat &src, vector<vector<Vec3d>> &dst,
                 vector<vector<double>> kernel, int border_type) {
    CV_Assert(src.type() == CV_8UC3  &&
              src.rows == dst.size() &&
              src.cols == dst[0].size());
    int border = (kernel.size() - 1) / 2;
    int i_dst  = 0;
    int j_dst  = 0;
    Mat filled;
    vector<double> cons(3, 0);
    // form the border around the image
```

```
    copyMakeBorder(src, filled, border, border, border, border,
border_type);
    int rows = src.rows + 2 * border;
    int cols = src.cols + 2 * border;
    // process each pixel
    for (int i = border; i < rows - border; ++i) {
        for (int j = border; j < cols - border; ++j) {
            i_dst = i - border;
            j_dst = j - border;
            // convolution operation
            cons[0] = cons[1] = cons[2] = 0;
            for (int ki = i - border; ki <= i + border; ++ki)
                for (int kj = j - border; kj <= j + border; ++kj)
                    for (int ch = 0; ch < 3; ++ch)
                        cons[ch] += (kernel[ki-i_dst][kj-j_dst] *
                                     filled.at<Vec3b>(ki, kj)[ch]);
            dst[i_dst][j_dst] = Vec<double, 3>(cons[0], cons[1], cons[2]);
        }
    }
}
```

2. 利用均值模板（高斯模板）平滑灰度（彩色）图像

大小为$m \times m$的均值模板是如下的阵列

$$\frac{1}{m^2}\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}_{m\times m} \tag{3.2}$$

高斯核由如下的公式计算

$$w(s,t) = G(s,t) = Ke^{-\frac{s^2+t^2}{2\sigma^2}} \tag{3.3}$$

在具体操作中高斯核是对式(3.3)取样得到的，规定s和t的值，然后计算函数在这些坐标处的值，这些值是核的系数。通过将核的系数除以各系数之和实现核的归一化。σ控制高斯函数关于其均值的"展开度"，其值越大，对图像的平滑效果越明显。一个大小为3×3的高斯模板如下

$$\frac{1}{4.8976}\begin{bmatrix} 0.3679 & 0.6065 & 0.3679 \\ 0.6065 & 1.0000 & 0.6065 \\ 0.3679 & 0.6065 & 0.3679 \end{bmatrix} \tag{3.4}$$

我们首先编写用于计算大小为$m \times m$，标准差为σ的高斯核的函数如下

```
/// @brief get the gaussian kernel used to blur an image.
/// @param m the size of gaussian kernel is m*m,and m is an odd number
/// @param sigma the standard deviation of gaussian function
/// @return a reference to a normalized gaussian kernel used to blur an
image
vector<vector<double>> gaussian_kernel(int m, double sigma) {
    vector<vector<double>> kernel(m, vector<double>(m, 0));
    double sum = 0;
    double r_square = 0;
    double center = (m - 1) / 2;
    for (int i = 0; i < m; ++i) {
```

```
        for (int j = 0; j < m; ++j) {
            r_square = pow((i - center), 2) + pow((j - center), 2);
            kernel[i][j] = exp(-(r_square / (2 * sigma * sigma)));
            sum += kernel[i][j];
        }
    }
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < m; ++j)
            kernel[i][j] /= sum;
    return kernel;
}
```

基于上述的卷积操作和模板，用于平滑灰度（彩色）图像的函数实现如下

```
/// @brief blur an gray-scale or rgb image using a box or gaussian kernel,
///        the in-place processing is supported.
/// @param src input image
/// @param dst output image
/// @param m the size of kernel is m*m,and m is an odd number
/// @param sigma the standard deviation of gaussian function, if sigma == 0,
///              we use a box kernel
void blur(Mat &src, Mat &dst, int m, double sigma) {
    CV_Assert(src.type() == CV_8UC1 || src.type() == CV_8UC3);
    // create a box kernel
    const double elem = 1.0 / m / m;
    vector<vector<double>> box_kernel(m, vector<double>(m, elem));
    if (src.type() == CV_8UC1) {
        // gray-scale image
        vector<vector<double>> tmp_dst(src.rows, vector<double>(src.cols,
0));
        if (sigma == 0)
            convolution(src, tmp_dst, box_kernel, BORDER_REFLECT);
        else
            convolution(src, tmp_dst, gaussian_kernel(m, sigma),
BORDER_REFLECT);
        dst.create(src.size(), CV_8UC1);
        for (int i = 0; i < dst.rows; ++i)
            for (int j = 0; j < dst.cols; ++j)
                dst.at<uchar>(i, j) = saturate_cast<uchar>(tmp_dst[i][j]);
    } else {
        // rgb image
        vector<vector<Vec3d>> tmp_dst(src.rows, vector<Vec3d>(src.cols,
Vec3d(0, 0, 0)));
        if (sigma == 0)
            convolution(src, tmp_dst, box_kernel, BORDER_REFLECT);
        else
            convolution(src, tmp_dst, gaussian_kernel(m, sigma),
BORDER_REFLECT);
        dst.create(src.size(), CV_8UC3);
        for (int i = 0; i < dst.rows; ++i)
            for (int j = 0; j < dst.cols; ++j)
                for (int ch = 0; ch < 3; ++ch)
                    dst.at<Vec3b>(i, j)[ch] = saturate_cast<uchar>
(tmp_dst[i][j][ch]);
    }
```

```
}
```

3. 利用高提升滤波算法增强灰度图像

从图像中减去一幅平滑后的图像称为钝化掩蔽，它由如下步骤组成：

> 1. 模糊原图像。
> 2. 从原图像减去模糊后的图像（产生的差称为模板）。
> 3. 将模板与原图像相加。

令$\bar{f}(x,y)$表示模糊后的图像，公式形式的模板为

$$g_{mask}(x,y) = f(x,y) - \bar{f}(x,y) \tag{3.5}$$

将加权后的模板与原图像相加：

$$g(x,y) = f(x,y) + kg_{mask}(x,y) \tag{3.6}$$

当$k > 1$时，这个过程称为高提升滤波。代码实现如下

```
void highboost_filter(Mat &src, Mat &dst, int m, double sigma, double k) {
    CV_Assert(src.type() == CV_8UC1);
    const int rows = src.rows;
    const int cols = src.cols;
    Mat blurred;
    blur(src, blurred, m, sigma);
    Mat mask (rows, cols, CV_8UC1);
    for (int i = 0; i < rows; ++i)
        for (int j = 0; j < cols; ++j)
            mask.at<uchar>(i, j) =
            saturate_cast<uchar>(src.at<uchar>(i, j) - blurred.at<uchar>(i,
j));
    imshow("mask", mask);
    Mat temp_src = src.clone();
    dst.create(src.size(), CV_8UC1);
    for (int i = 0; i < rows; ++i)
        for (int j = 0; j < cols; ++j)
            dst.at<uchar>(i, j) =
            saturate_cast<uchar>(k * mask.at<uchar>(i, j) +
temp_src.at<uchar>(i, j));
}
```

4. 利用Laplacian模板锐化灰度（彩色）图像

最简单的各向同性导数算子是拉普拉斯，对于两个变量的函数$f(x,y)$，它定义为

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{3.7}$$

再将对角方向整合到数字拉普拉斯核的定义中，我们可以构造出下面四个拉普拉斯核

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{3.8}$$

拉普拉斯是导数算子，因此会突出图像中的急剧过渡，并且不强调缓慢变化的灰度区域。这往往会产生具有灰色边缘和其他不连续性的图像，它们都叠加在暗色无特征背景上。将拉普拉斯图像与原图像相加，就可以恢复背景特征，同时保留拉普拉斯的锐化效果。因此我们使用拉普拉斯锐化图像的基本方法是

$$g(x,y) = f(x,y) + c[\nabla^2 f(x,y)] \tag{3.9}$$

$f(x,y)$和$g(x,y)$分别是输入图像和锐化后的图像。若使用(3.8)中的前两个核，$c=-1$；若使用(3.8)中的后两个核，$c=1$。使用Laplacian算子锐化图像的算法如下

```
/// @brief sharpen an gray-scale or rgb image using a laplacian operator,
///        the in-place processing is supported.
/// @param src input image
/// @param dst output image
void laplacian_sharpen(Mat &src, Mat &dst) {
    CV_Assert(src.type() == CV_8UC1 || src.type() == CV_8UC3);
    vector<vector<double>> laplacian1 = {{ 0, 1, 0},{ 1,-3, 1},{ 0, 1, 0}};
    vector<vector<double>> laplacian2 = {{ 1, 1, 1},{ 1,-7, 1},{ 1, 1, 1}};
    vector<vector<double>> laplacian3 = {{ 0,-1, 0},{-1, 5,-1},{ 0,-1, 0}};
    vector<vector<double>> laplacian4 = {{-1,-1,-1},{-1, 9,-1},{-1,-1,-1}};
    if (src.type() == CV_8UC1) {
        // gray-scale image
        vector<vector<double>> tmp_dst(src.rows, vector<double>(src.cols,
0));
        convolution(src, tmp_dst, laplacian1, BORDER_REFLECT);
        dst.create(src.size(), CV_8UC1);
        for (int i = 0; i < dst.rows; ++i)
            for (int j = 0; j < dst.cols; ++j)
                dst.at<uchar>(i, j) = saturate_cast<uchar>(tmp_dst[i][j]);
    } else {
        // rgb image
        vector<vector<Vec3d>> tmp_dst(src.rows, vector<Vec3d>(src.cols,
Vec3d(0, 0, 0)));
        convolution(src, tmp_dst, laplacian1, BORDER_REFLECT);
        dst.create(src.size(), CV_8UC3);
        for (int i = 0; i < dst.rows; ++i)
            for (int j = 0; j < dst.cols; ++j)
                for (int ch = 0; ch < 3; ++ch)
                    dst.at<Vec3b>(i, j)[ch] = saturate_cast<uchar>
(tmp_dst[i][j][ch]);
    }
}
```

5. 使用Robert(Sobel)算子锐化灰度（彩色）图像

在图像处理中，一阶导数是用梯度幅度实现的。图像f在坐标(x,y)处的梯度定义为二维列向量

$$\nabla f \equiv grad(f) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{3.10}$$

向量∇f的幅度表示为$M(x,y)$，其中

$$M(x,y) = mag(\nabla f) = \sqrt{g_x^2 + g_y^2} \tag{3.11}$$

是梯度向量方向的变化率在(x,y)处的值。$M(x,y)$是与原图像大小相同的图像，它是x和y在f的所有像素位置上变化时创建的。实践中称这幅图像为梯度图像。根据上面几个公式的离散近似构造的$Robert$交叉梯度算子如下

$$\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix} \tag{3.12}$$

构造的$Sobel$算子如下

$$\begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix}\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix} \tag{3.13}$$

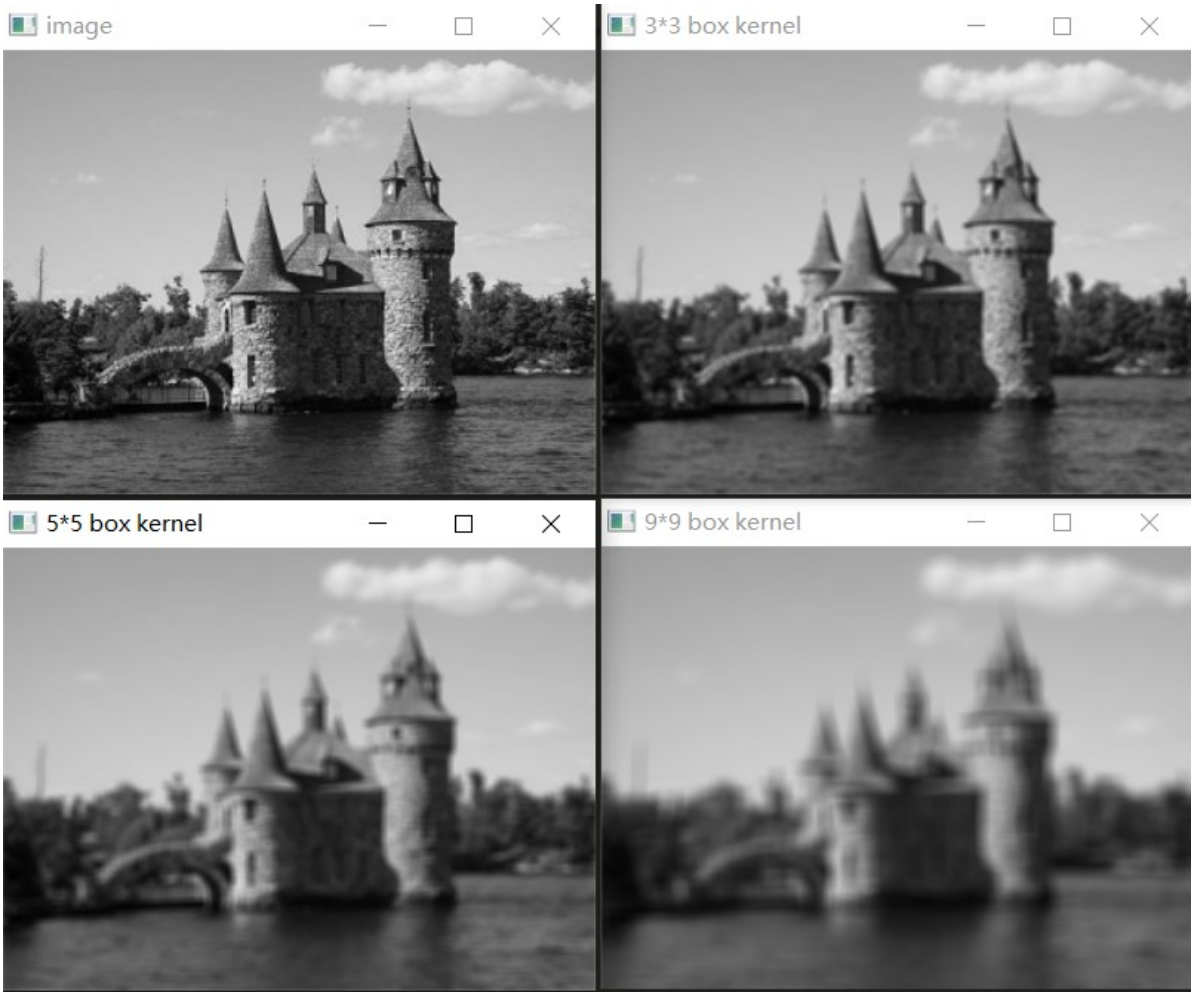
使用$Robert(Sobel)$算子锐化灰度（彩色）图像的算法如下

```
/// @brief sharpen an gray-scale or rgb image using a robert or sobel operator,
///        the in-place processing is supported.
/// @param src input image
/// @param dst output image
/// @param kx derivative x operator
/// @param ky derivative y operator
void gradient_sharpen(Mat &src, Mat &dst,
                      vector<vector<double>> kx,
                      vector<vector<double>> ky) {
    CV_Assert(src.type() == CV_8UC1 || src.type() == CV_8UC3);
    const int rows = src.rows;
    const int cols = src.cols;
    if (src.type() == CV_8UC1) {
        // gray-scale image
        vector<vector<double>> gx (rows, vector<double>(cols, 0));
        vector<vector<double>> gy (rows, vector<double>(cols, 0));
        vector<vector<double>> Mxy(rows, vector<double>(cols, 0));
        convolution(src, gx, kx, BORDER_REFLECT);
        convolution(src, gy, ky, BORDER_REFLECT);
        for (int i = 0; i < rows; ++i)
            for (int j = 0; j < cols; ++j)
                Mxy[i][j] = abs(gx[i][j]) + abs(gy[i][j]);
        dst.create(src.size(), CV_8UC1);
        for (int i = 0; i < dst.rows; ++i)
            for (int j = 0; j < dst.cols; ++j)
                dst.at<uchar>(i, j) = saturate_cast<uchar>(Mxy[i][j]);
    } else {
        // rgb image
        vector<vector<Vec3d>> gx (rows, vector<Vec3d>(cols, Vec3d(0, 0, 0)));
        vector<vector<Vec3d>> gy (rows, vector<Vec3d>(cols, Vec3d(0, 0, 0)));
        vector<vector<Vec3d>> Mxy(rows, vector<Vec3d>(cols, Vec3d(0, 0, 0)));
        convolution(src, gx, kx, BORDER_REFLECT);
        convolution(src, gy, ky, BORDER_REFLECT);
        for (int i = 0; i < rows; ++i)
            for (int j = 0; j < cols; ++j)
                for (int ch = 0; ch < 3; ++ch)
                    Mxy[i][j][ch] = abs(gx[i][j][ch]) + abs(gy[i][j][ch]);
```

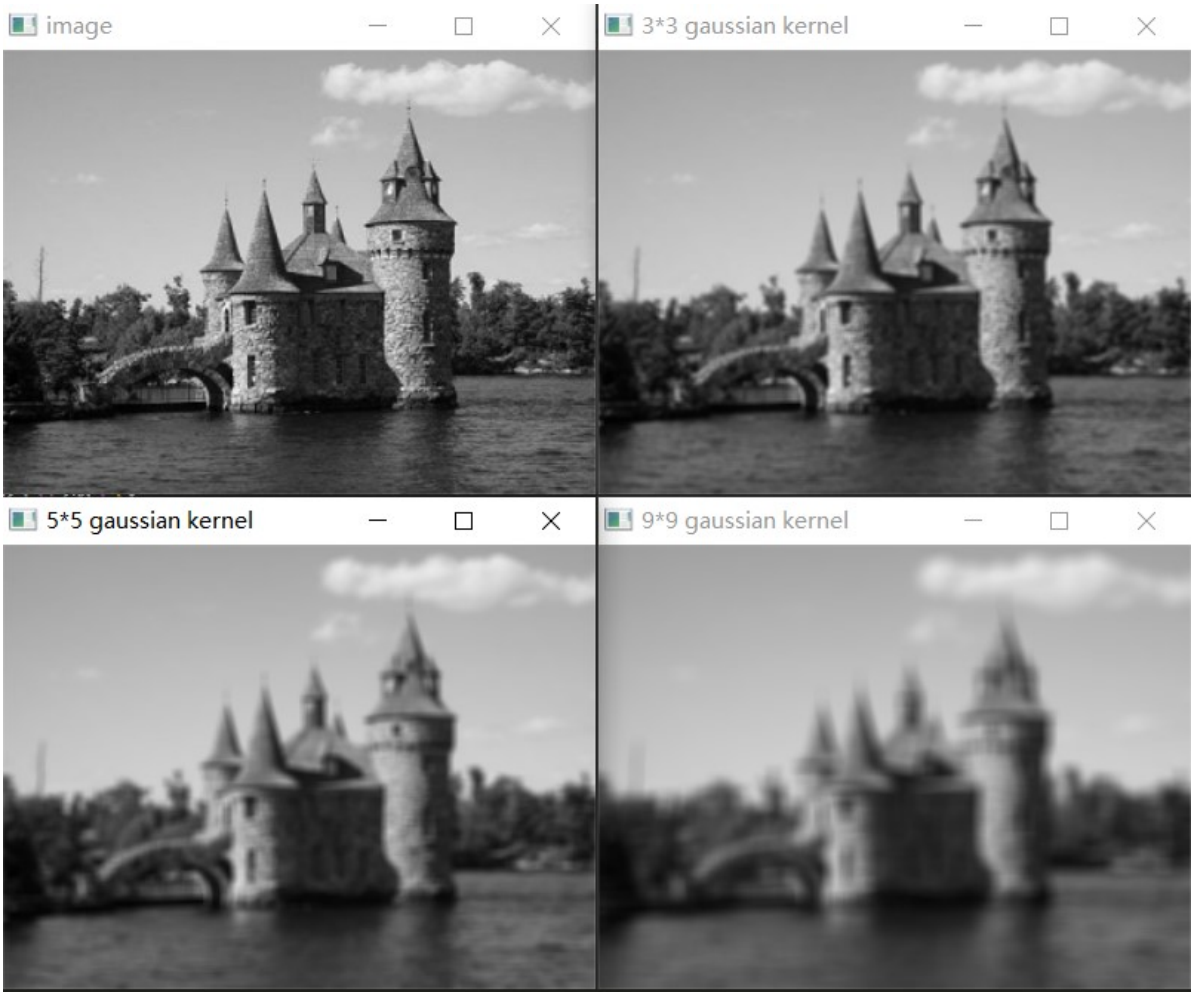
```
        dst.create(src.size(), CV_8UC3);
        for (int i = 0; i < dst.rows; ++i)
            for (int j = 0; j < dst.cols; ++j)
                for (int ch = 0; ch < 3; ++ch)
                    dst.at<Vec3b>(i, j)[ch] = saturate_cast<uchar>(Mxy[i][j][ch]);
    }
}
```

实验结果

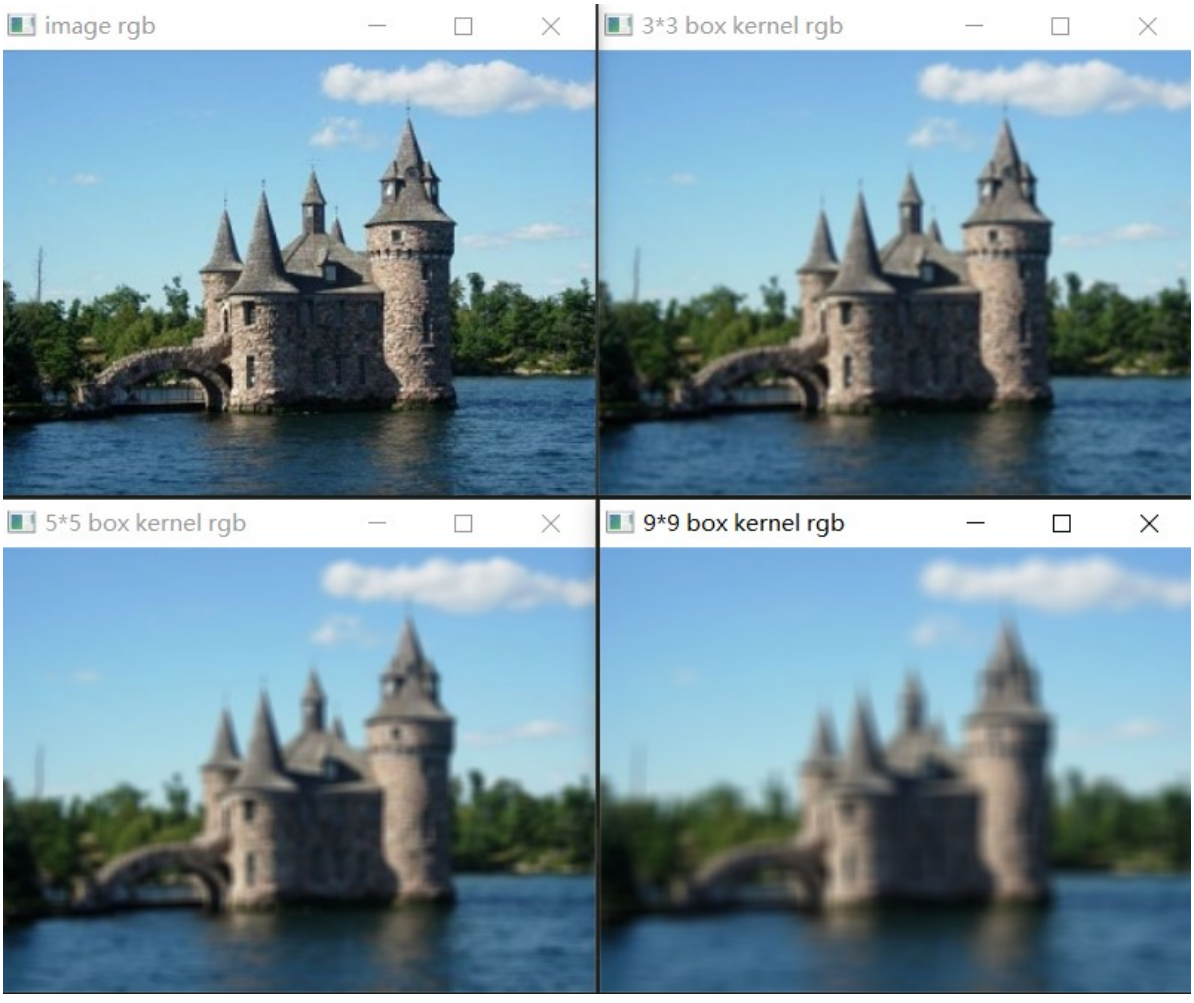
- [x] 使用均值模板平滑灰度图像



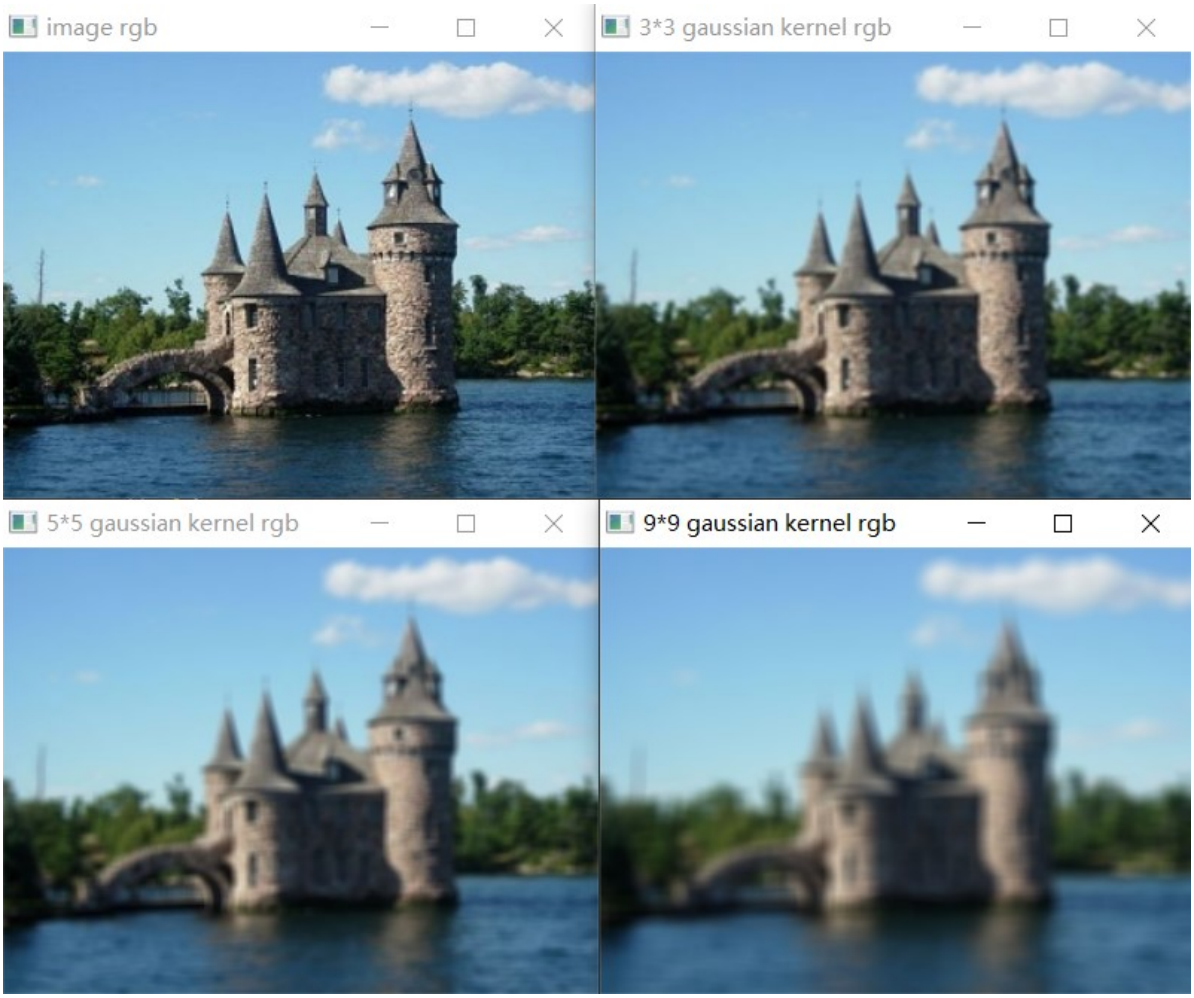
- [x] 使用高斯模板平滑灰度图像



- [x] 使用均值模板平滑彩色图像

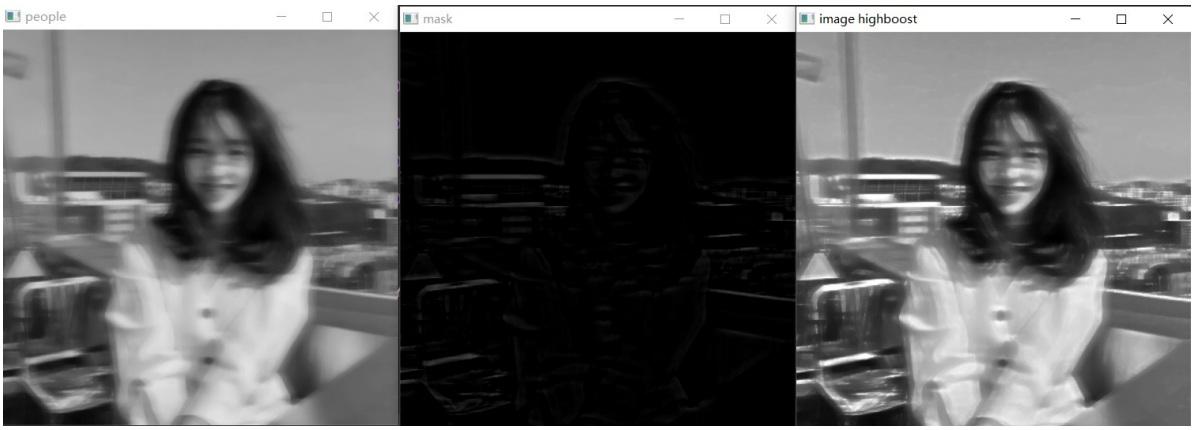


- [x] 使用高斯模板平滑彩色图像



- [x] 使用高提升算法增强灰度图像

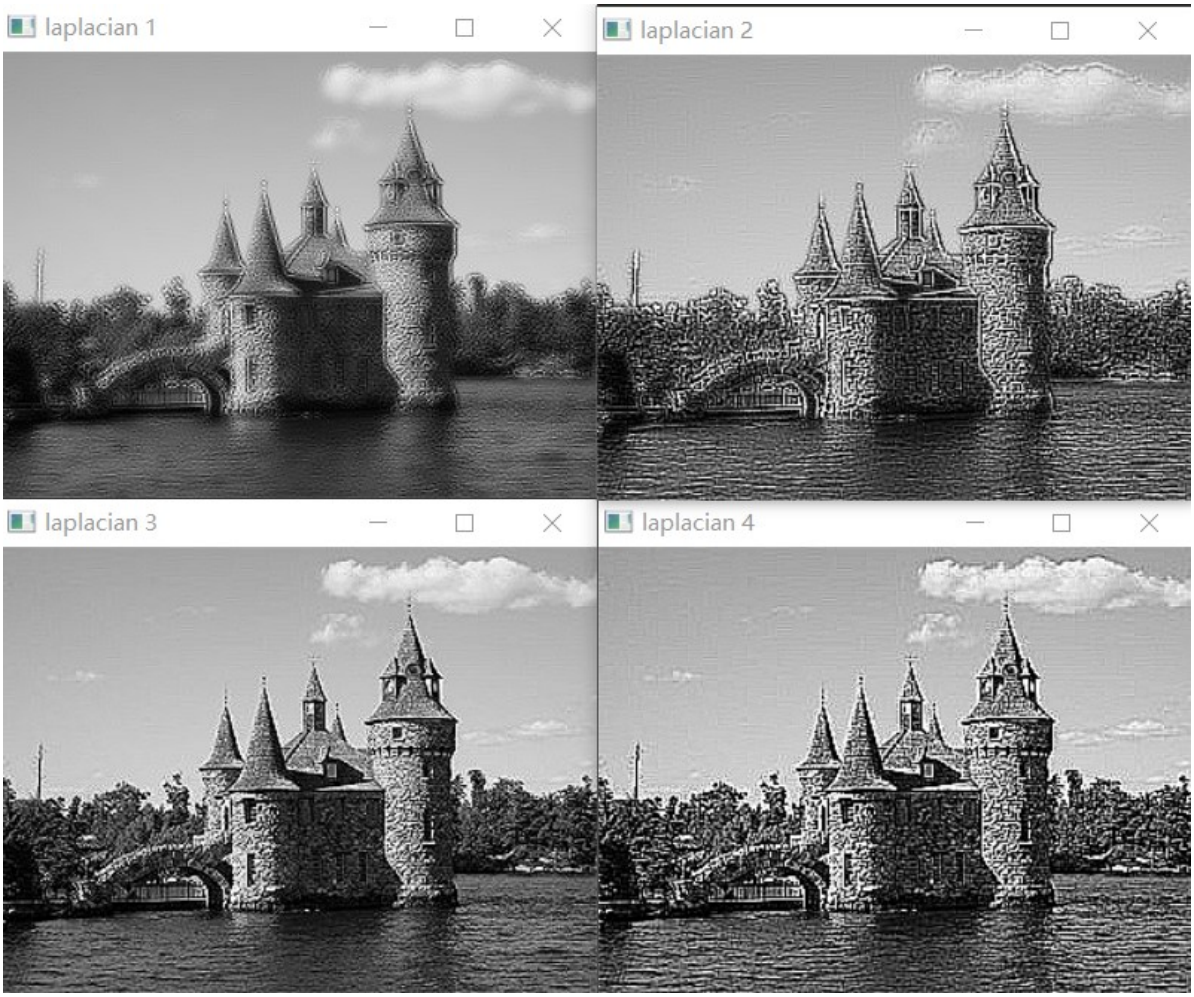
> 左边的灰度图像首先使用了大小为15×15，$\sigma = 4$的高斯模板进行平滑处理，然后使用了$k = 1.8$的高提升算法形成了右边的图像。中间的图像是所用的模板



- [x] 使用Laplacian模板锐化灰度（彩色）图像

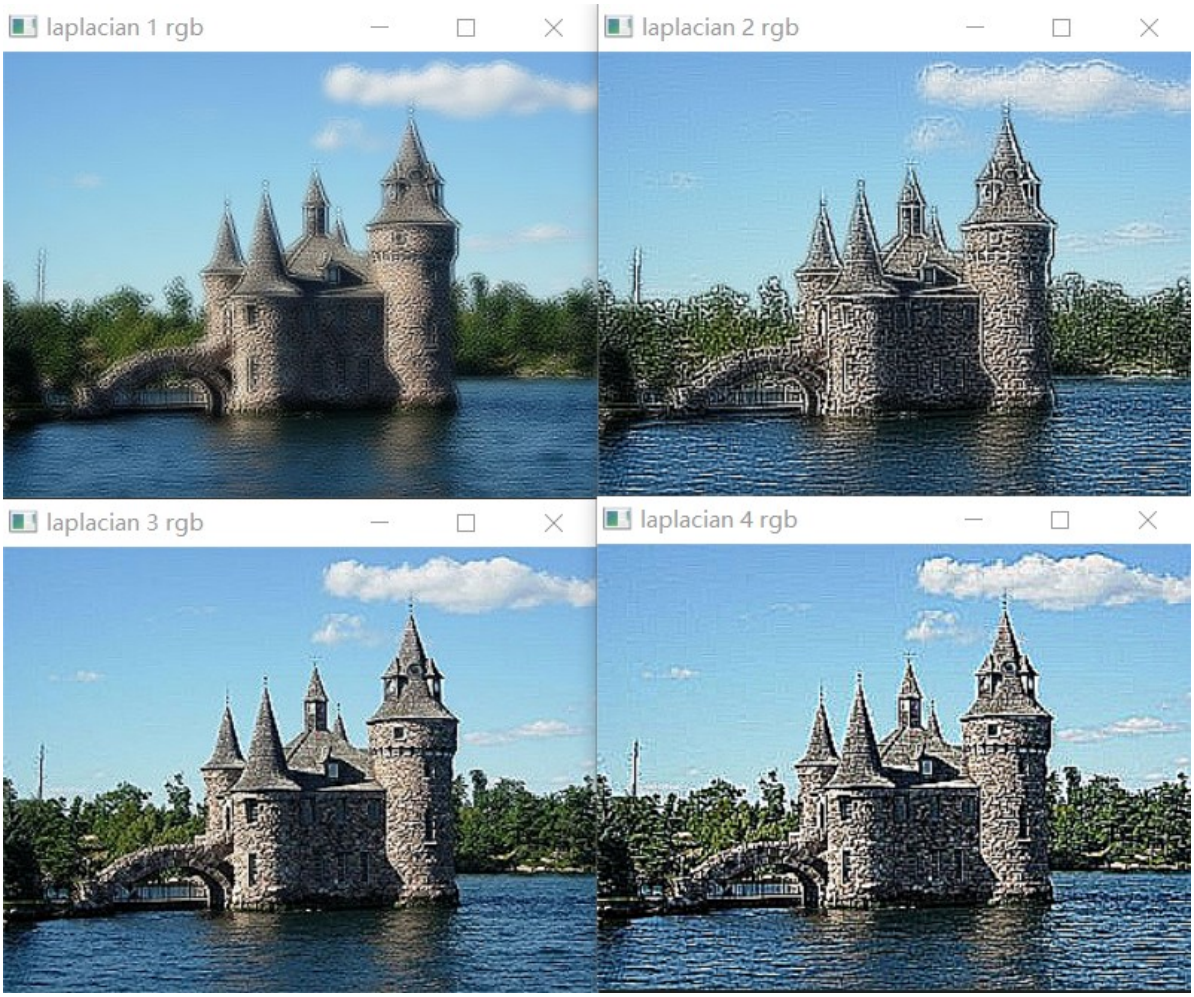
> 灰度图像原图

> 使用四种Laplacian模板锐化后

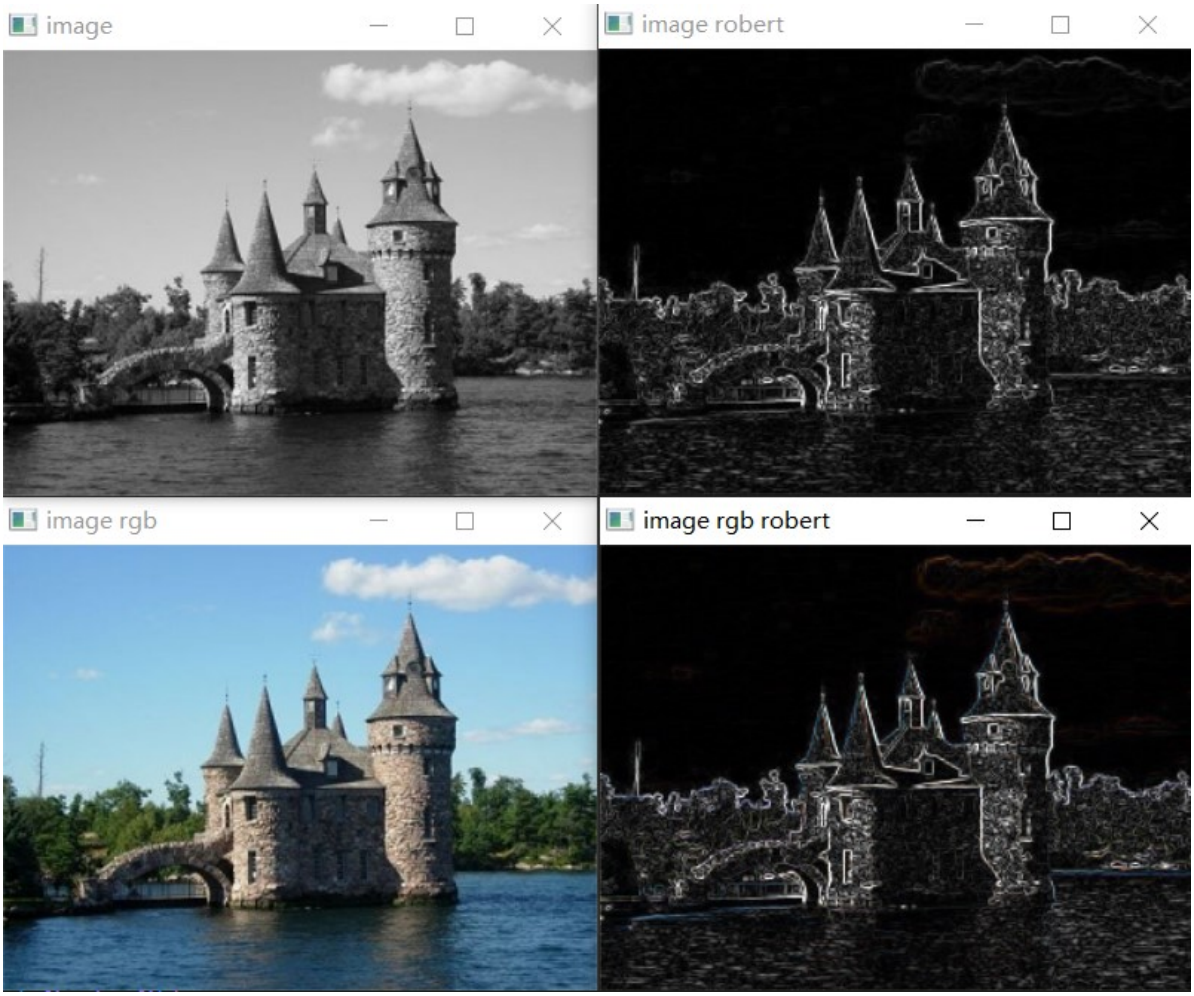


> 彩色图像原图

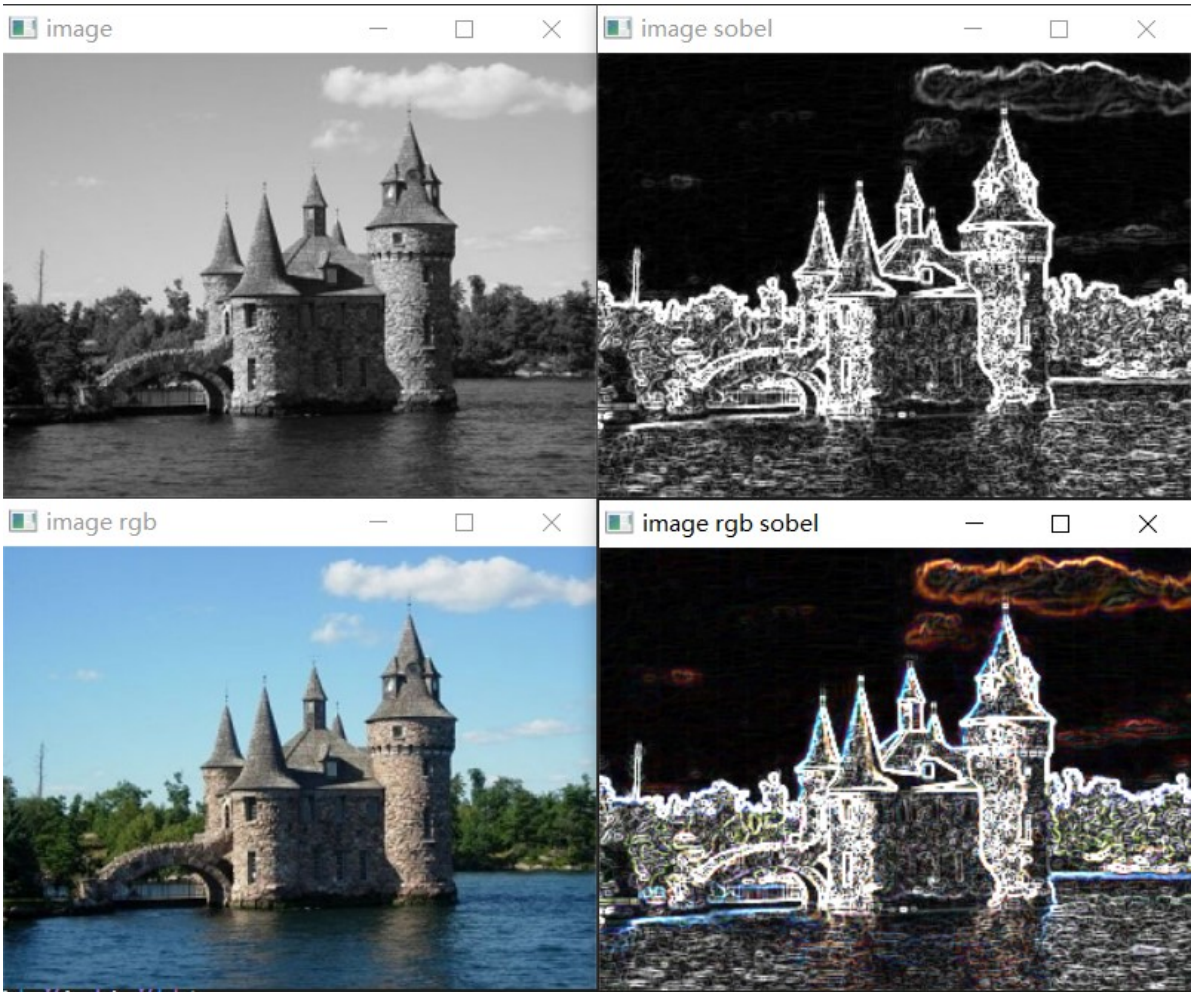
> 使用四种Laplacian模板锐化后



- [x] 使用Robert模板锐化图像



- [x] 使用Sobel模板锐化图像



完整的源代码见附件`lab3.cpp`。