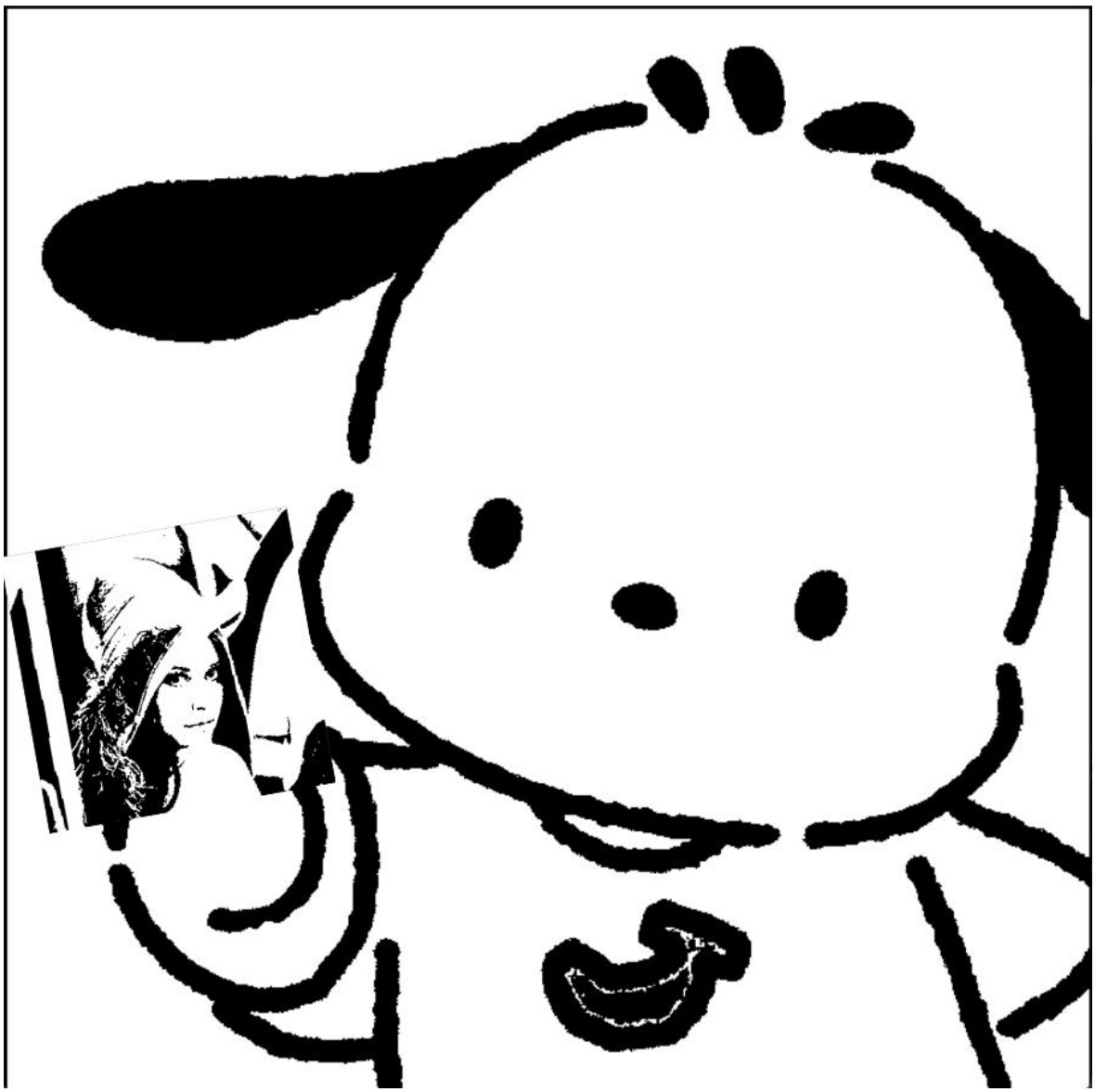
高级程序语言设计–实验报告

矩阵运算



报告名称：实验报告–矩阵运算

班级：国 03

学号：2253156

姓名：闫浩扬

完成日期：2023 年 5 月 20 日

一．功能描述

编写了一个矩阵操作小程序，将多项矩阵基础功能整合与菜单中，使其可以实现以下功能：

（1）矩阵加法：将两个列数，行数都相同的矩阵相加，并输出和矩阵
（2）矩阵数乘：输出将整数与矩阵中的每一个元素分别相乘所得的矩阵。
（3）矩阵转置：将 M×N 的矩阵 A 的行换成同序数的列得到 N×M 的矩阵 A^T。
（4）矩阵乘法：矩阵 A 的列数和矩阵 B 的行数相等，进行矩阵乘法并输出结果。
（5）Hadamard 乘积：约束与加法相同，对应元素运算变成乘法。
（6）矩阵卷积：此处要求：kernel size = 3，padding = 1，stride = 1，dilation = 1；
（7）卷积应用：实现一个利用卷积操作应用与图像处理的简单示例：假设矩阵 A 为原灰度图 demolena.jpg（大小为 256*256），灰度值为矩阵值（int），分别采用不同卷积核（矩阵 B）进行卷积运算，并将得到的矩阵除以卷积核的总和（如 B1 的卷积核总和为 9，总和为 0 时不做处理），保存为灰度图观察结果
（8）OTSU 算法：使用 OTSU 算法对 lena 图像进行二值化。并在 OTSU 算法的基础上实现图片主体的提取。

二．设计思路

2.1 基础功能的实现

（1）矩阵的构造

受前期作业启发，决定封装一个矩阵类，其中包括矩阵的数据以及一些矩阵运算需要使用的必要函数。

```
class Matrix //矩阵类
{
private:
    int row, col; //行和列
    int** matx;
public:
    Matrix(int r, int c);
    Matrix(int r, int c, int** init);
    Matrix(int r, int c, uchar **init);
    // 使用二维数组初始化矩阵
    void initialize(int data[][3]) { ... }
    // 析构函数
    ~Matrix() { ... }
    int getrow() { ... }
    int getcol() { ... }
    uchar** toUcharPtrArray() const { ... }
    Matrix operator+(const Matrix& x2);
    Matrix operator*(int k);
    Matrix operator/(int k);
    Matrix operator*(Matrix& x2);
    Matrix& operator=(const Matrix& x2);
    int get_sum();
    void check(Mat& img,int sum);
    Matrix Hadamard(const Matrix& x2) const;
    Matrix convolution(const Matrix& b) const;
    friend Matrix trans(const Matrix& a);
    friend istream& operator>>(istream& in, Matrix& x);
    friend ostream& operator<<(ostream& out, Matrix& x);
};
```

这个矩阵类（Matrix）实现了一个简单的矩阵数据结构，具有一些基本的矩阵操作和功能。

类成员变量：

row 和 col：分别表示矩阵的行数和列数。

matx：是一个二维整数数组指针，用于存储矩阵的元素。采用动态内存申请的方式，有利于节省内存空间，并且可以满足各种大小的矩阵构造。

构造函数：

Matrix(int r, int c)：创建一个空的 r 行 c 列的矩阵。

Matrix(int r, int c, int** init)：创建一个 r 行 c 列的矩阵，并使用二维整数数组 init 来初始化矩阵。

Matrix(int r, int c, uchar **init)：创建一个 r 行 c 列的矩阵，并使用二维无符号字符数组 init 来初始化矩阵。

析构函数：

~Matrix()：析构函数，由于矩阵采用动态内存申请，所以必须要在析构函数中将动态内存 delete，释放掉矩阵占用的内存。

成员函数：

initialize(int data[][3])：使用二维整数数组 data 初始化矩阵的元素（主要是为了传入卷积核，但是这样又有些复杂）。

getrow() 和 getcol()：创建一个接口，分别返回矩阵的行数和列数。因为矩阵中 row，col 为私有成员，为了安全访问，保护私有成员，使用接口访问。

toUcharPtrArray()：将矩阵转换为二维无符号字符指针数组，并返回该数组的指针。

get_sum()：计算矩阵中所有元素的和，用于卷积核的总和计算，这样方便以后添加不同的卷积核。

check(Mat& img, int sum)：灰度图像范围为 0-255，而矩阵在卷积操作以后有可能出现超范围的数值，在将卷积以后的矩阵赋值给 Mat 对象时，需要检查一下数值范围。

（2）矩阵的运算

基础运算：

受前期作业启发，使用运算符重载可以极大地提高效率，并且简单易读。

operator+：矩阵加法运算符重载

形式：Matrix operator+(const Matrix& x2);

功能：实现两个矩阵相加的操作。将当前矩阵与参数 x2 的对应元素相加，生成一个新的矩阵作为结果返回。

operator*：矩阵乘法运算符重载（数乘）

形式：Matrix operator*(int k);

功能：实现矩阵与标量之间的乘法操作。将当前矩阵的每个元素乘以参数 k，生成一个新的矩阵作为结果返回。

operator/：矩阵除法运算符重载

形式：Matrix operator/(int k);

功能：实现矩阵与标量之间的除法操作。将当前矩阵的每个元素除以参数 k，生成一个新的矩阵作为结果返回。

operator*：矩阵乘法运算符重载（矩阵之间的乘法）

形式：Matrix operator*(Matrix& x2);

功能：实现矩阵与矩阵之间的乘法操作。当前矩阵的列数必须与参数 x2 的行数相等。运算结果为当前矩阵与参数 x2 的乘积矩阵，生成一个新的矩阵作为结果返回。

operator=：赋值运算符重载

形式：Matrix& operator=(const Matrix& x2);

功能：实现矩阵的赋值操作。将参数 x2 的矩阵内容赋值给当前矩阵，并返回当前矩阵的引用。

进阶运算：

Hadamard(const Matrix& x2)：返回当前矩阵和另一个矩阵 x2 的 Hadamard 积（对应元素相乘）结果。

convolution(const Matrix& b)：返回当前矩阵和另一个矩阵 b 的卷积结果。

friend Matrix trans(const Matrix& a)：友元函数，用于计算矩阵 a 的转置矩阵。

friend istream& operator>>(istream& in, Matrix& x) 和 **friend ostream& operator<<(ostream& out, Matrix& x)**：友元函数，重载输入输出运算符，允许直接通过流进行矩阵的输入和输出。

（3）矩阵的卷积操作

要求：kernel size = 3，padding = 1，stride = 1，dilation = 1；根据网上查询，得知，卷积后的输出矩阵大小应为(Input-Kernel+2*Padding)/Stride+1;由于本题中 Kernel 大小为 3*3，代入可知，结果矩阵的大小恰为其本身大小。

实现逻辑：

1.定义变量和结果矩阵：首先，根据卷积核 b 的大小和当前矩阵的大小，定义一些变量， bRows、bCols（卷积核的大小）、outRows 和 outCols（卷积结果的大小）。然后，创建了一个结果矩阵 result。

2.创建填充矩阵：由于 padding=1，创建了一个填充矩阵 paddedMatrix，其大小比当前矩阵的行

和列分别增加了 2。这样做是为了在当前矩阵的外围填充零值，以处理边界像素的卷积操作。

3.填充矩阵：使用嵌套的循环遍历 paddedMatrix 的每个位置，并根据位置的坐标值进行填充。对于边界位置，将其值设置为零；对于内部位置，将其值设置为当前矩阵相应位置的值。

4.执行卷积操作：使用嵌套的循环遍历结果矩阵 result 的每个位置，并对应地进行卷积操作。

5.加权求和：对于 result 中的每个位置，使用嵌套的循环遍历卷积核 b 的每个位置。将 paddedMatrix 中对应位置的像素值与 b 中对应位置的权重相乘，并将乘积累加到 result 中的对应位置。

6.返回结果矩阵：将生成的结果矩阵 result 作为方法的返回值。

```
Matrix Matrix::convolution(const Matrix& b) const {
    int bRows = 3;
    int bCols = 3;
    int outRows = row;
    int outCols = col;
    Matrix result(outRows, outCols);

    int paddedRows = row + 2;
    int paddedCols = col + 2;
    Matrix paddedMatrix(paddedRows, paddedCols);
    // 在矩阵A的外围填充0
    for (int i = 0; i < paddedRows; ++i) {
        for (int j = 0; j < paddedCols; ++j) {
            if (i == 0 || i == paddedRows - 1 || j == 0 || j == paddedCols - 1) {
                paddedMatrix.matx[i][j] = 0;
            }
            else {
                paddedMatrix.matx[i][j] = matx[i - 1][j - 1];
            }
        }
    }
    //卷积操作
    for (int i = 0; i < outRows; ++i) {
        for (int j = 0; j < outCols; ++j) {
            int sum = 0;
            for (int k = 0; k < bRows; ++k) {
                for (int m = 0; m < bCols; ++m) {
                    sum += paddedMatrix.matx[i + k][j + m] * b.matx[k][m];
                }
            }
            result.matx[i][j] = sum;
        }
    }
    return result;
}
```

（4）卷积应用

1.定义卷积核矩阵：根据给定的初始化数据，创建了六个卷积核矩阵 B1、B2、B3、B4、B5 和 B6。这些卷积核矩阵用于进行不同的卷积操作。

2.读取图像：使用 OpenCV 的 imread 函数读取名为 "demolena.jpg" 的图像，并将图像的灰度值存储在 Mat 类型的变量 image 中。

3.创建矩阵对象：根据读取的图像，创建了矩阵对象 a，其行数和列数与图像的行数和列数相同。使用 Mat2Vec 函数将图像的灰度值转换为二维数组，并通过构造函数将其传递给矩阵 a 进行初始化。

4.执行卷积操作并可视化结果：对于每个卷积核，创建了相应的结果矩阵对象（c1、c2、c3、c4、c5 和 c6）。然后，通过调用 convolution 方法将矩阵 a 与相应的卷积核进行卷积操作，并将结果存储在相应的结果矩阵对象中。

5.计算卷积核的和并进行归一化：对于每个卷积核，通过调用 get_sum 方法获取卷积核元素的和。如果和为零，将其设置为 1，以避免除以零的错误。

6.创建结果图像：为每个卷积操作创建一个 CV_8UC1 类型的图像对象（img1、img2、img3、img4、img5 和 img6）。

7.检查并处理结果：通过调用 check 方法，将相应的结果矩阵对象转换为 CV_8UC1 类型的图像，并存储在相应的结果图像对象中。

8.显示结果图像：使用 OpenCV 的 imshow 函数将每个结果图像显示在相应的窗口中。

（5）OTSU 算法二值化

没看见可以使用 opencv 中的库函数，自己写了一个。

OTSU 算法的基本思路：

1.计算灰度直方图：遍历图像的所有像素，统计每个灰度级别的像素数量，形成灰度直方图。

2.计算总像素数：统计图像的总像素数以及权重。

3.初始化类内方差最大值和最佳阈值：初始化类内方差的最大值为 0，并将最佳阈值初始化为 0。遍历灰度级别：从灰度级别 0 开始，遍历所有可能的阈值。

4.计算类内方差：在每个阈值处，将图像分为两个类别：背景（低于阈值）和前景（高于阈值）。计算背景和前景的像素数、像素平均值和像素方差。然后，使用以下公式计算类内方差：

类内方差 = 背景像素数 * 前景像素数 *（背景平均值 - 前景平均值）^ 2

5.更新最大类内方差和最佳阈值：如果计算得到的类内方差大于最大类内方差，则更新最大类内方差和最佳阈值。

6.阈值分割图像：根据最佳阈值将图像进行二值化分割，将低于阈值的像素标记为背景，高于阈值的像素标记为前景。

7.输出分割结果：得到二值化的分割结果图像。

用到如下重要函数：

下面是各个函数的逻辑说明：

1. binImg(const vector<vector<int>>& image, vector<vector<int>>& binImage, int threshold)：

首先获取原始图像的行数和列数。调整二值化图像的大小，初始化为与原始图像相同的尺寸。 遍历原始图像的每个像素点，若像素值大于等于阈值，则在二值化图像中对应位置像素值设为 255（白色），否则设为 0（黑色）。

2. OSTUCUT(uchar** originImage, const vector<vector<int>>& image, vector<vector<int>>& binImage, int threshold)：

获取原始图像的行数和列数。调整截取图像的大小，初始化为与原始图像相同的尺寸。遍历原始图像的每个像素点，若像素值大于等于阈值，则在截取图像中对应位置保留原始像素值，否则设为0。

3. int BestThreshold(const std::vector<int>& histogram)：

统计灰度直方图中像素的总数量 `totalPixels`。

计算灰度直方图的加权和 `sumWeight`，即每个灰度级别的像素值乘以对应的像素数量的累加和。初始化变量 `sumB`、`weightBK`、`weightFG`、`bestVariance` 和 `threshold`。遍历灰度直方图的每个灰度级别：累加当前灰度级别的像素数量到 `weightBK`。若 `weightBK` 为0，则跳过当前灰度级别。计算 `weightFG`，即剩余像素数量。 累加当前灰度级别的加权像素值到 `sumB`。计算背景和前景的平均像素值 `averageBK` 和 `averageFG`。计算类间方差 `varBetween`。若 `varBetween` 大于 `bestVariance`，更新 `bestVariance` 和 `threshold`。返回最佳阈值 `threshold`。

4. void OTSU(Mat imageMat, int choice)：

将原始图像转换为像素矩阵 `imageVec`。根据像素矩阵构建灰度图像的二维向量 `image`。构建灰度直方图 `histogram`，统计每个灰度级别的像素数量。调用 `BestThreshold` 函数计算最佳阈值 `bestThreshold`。根据选择参数 `choice` 进行不同操作： 若 `choice` 为1，调用 `binImg` 函数进行二值化，并显示结果图像。若 `choice` 为2，调用 `OSTUCUT` 函数进行截取，并显示截取的图像。

5. `void show_otsu()`：

读取图像文件，并将其分别传入 `OTSU` 函数进行二值化或截取操作。显示不同图像的结果。以上是给定代码中各个函数的逻辑说明，它们结合在一起使用了OTSU算法来进行图像的阈值分割和截取操作。

(6)OTSU算法分割主体

观察，发现主体均为偏白的（灰度偏向255），背景均要求黑色（灰度0），OTSU算法已经将最佳阈值计算出来，改变每个像素点的灰度即实现了二值化，但是此处的分割并不是要实现二值化，而是分离主体，保留原有灰度，所以在check函数判断赋值时，应当加入下图这个判断条件，如果大于阈值，应当将原有灰度赋值给结果图像，而不是同二值化一样赋值255即可。

```
if (image[i][j] >= threshold) {
    binImage[i][j] = originImage[i][j];
}
else
    binImage[i][j] = 0;
```

,因此考虑单独写一个和 binImg 函数类似的 OSTUCUT 函数。

三．在实验过程中遇到的问题及解决方法

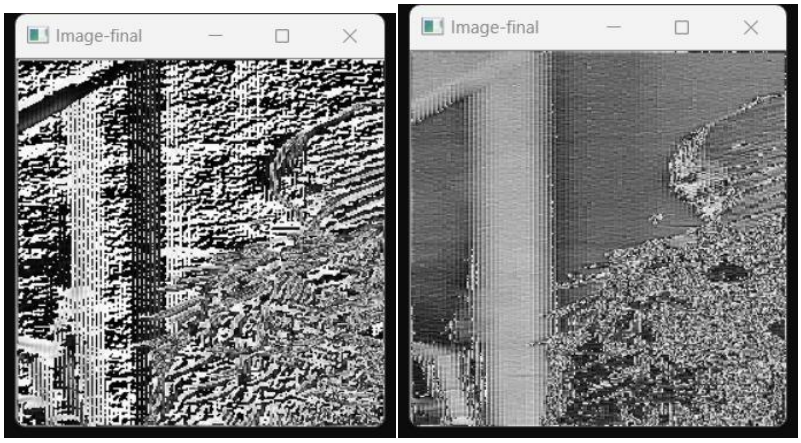
1. 矩阵大小错误

做基础项时使用静态变量定义矩阵，matx[10][10]二维数组，到图像处理时发现矩阵会超限，修改变大以后，发现会报错，搜索发现由于这个数组在类内，不能定义太大，并且如果静态定义，不论矩阵大小，耗费内存会很大。所以采取动态内存申请的方式，动态定义矩阵。

2. 卷积操作有误

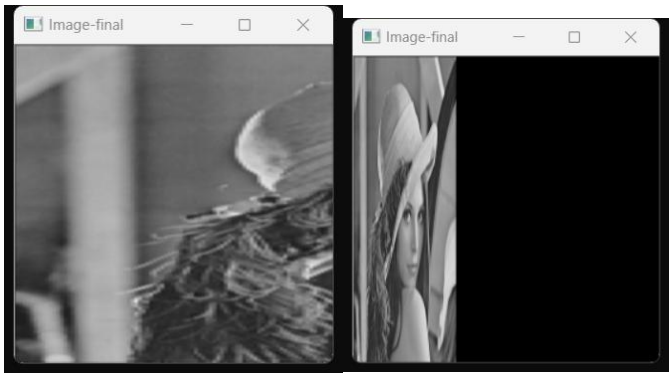
卷积时刚开始未考虑填充，计算出错，后来修改，在卷积时，新建一个填充矩阵，在边缘赋值 0，中间为原矩阵进行卷积，这样可以考虑到边缘像素的卷积。

3. 图像像素点颜色不对，太锐利



图片大体是对的，所以排除卷积问题，猜想是灰度值判断出现错误，导致每个像素的灰度出现问题。在灰度赋值时添加判断大于 255 值置 255，小于 0 置 0；

4. 图像显示不全，比例不对



Lena 图像显示不全或比例不对。仔细观察可得，这个图片只有原图的前 1/3 部分那么可猜测原因有二，可能是临近三个元素的值是相同的，要么是数组太小，而第二种猜测通过打印图像行列的大小被否，故而解决方法是使用 mat.at<>()类成员函数赋值的时候把 i 变成 3*j;

```
for (int i = 0; i < mat.rows; ++i)
{
    for (int j = 0; j < mat.cols; ++j)
    {

        array[i][j] = mat.at<uchar>(i, 3*j);
        if (mat.at<uchar>(i, 3 * j) < 0) {
            array[i][j] = 0;
        }
        if (mat.at<uchar>(i, 3 * j) > 255) {
            array[i][j] = 255;
        }
    }
}
```

5. OTSU 算法阈值计算错误

阈值计算大体思路了解，但是实际编写时出现问题，类内方差和类间方差没有全计算，导致阈值计算出现问题，耗费较长时间。

四. 心得体会

1. 初步了解了面向对象编程的优点，可以极大地提高效率，方便编写与查错等。
2. 理解了运算符重载的优点，方便直观。
3. 体会到了成员函数与友元函数的格子用法与特点。以及函数的返回类型应该由他的功能所决定。
4. 在出现实际与预期不同时，应当有条理地逐个猜测，逐个分析，逐个查错，不能瞎改乱改，或者遇到问题先用搜索引擎搜索。
5. 有意识地将一些代码封装为单独的函数，可以提高可读性
6. 注释太少，一些关键代码处应当增加一些注释。
7. 体会到动态内存申请的优点，节省空间，但是也要注意及时释放，如矩阵类中动态内存的释放可以放在析构函数中，这样可以自动释放。
8. 遇到问题无头绪时，可以多搜索，借助一些工具帮助修改。

五．源代码

```
#include <conio.h>
#include <iostream>
#include <opencv2/opencv.hpp>
#include<vector>
using namespace cv;
using namespace std;

class Matrix //矩阵类
{
private:
    int row, col; //行和列
    int** matx;
public:
    Matrix(int r, int c);
    Matrix(int r, int c, int** init);
    Matrix(int r, int c, uchar **init);
    // 使用二维数组初始化矩阵
    void initialize(int data[][3]) {
        for (int i = 0; i < row; ++i) {
            for (int j = 0; j < col; ++j) {
                matx[i][j] = data[i][j];
            }
        }
    }
    // 析构函数
    ~Matrix() {
        // 释放矩阵内存
        for (int i = 0; i < row; ++i) {
            delete[] matx[i];
        }
        delete[] matx;
    }
    int getrow() {
        return row;
    }
```

```
    int getcol() {
        return col;
    }
    uchar** toUcharPtrArray() const {
        uchar** array = new uchar * [row];
        for (int i = 0; i < row; ++i) {
            array[i] = new uchar[col];
            for (int j = 0; j < col; ++j) {
                array[i][j] = static_cast<uchar>(matx[i][j]);
            }
        }
        return array;
    }
    Matrix operator+(const Matrix& x2);
    Matrix operator*(int k);
    Matrix operator/(int k);
    Matrix operator*(Matrix& x2);
    Matrix& operator=(const Matrix& x2);
    int get_sum();
    void check(Mat& img,int sum);
    Matrix Hadamard(const Matrix& x2) const;
    Matrix convolution(const Matrix& b) const;
    friend Matrix trans(const Matrix& a);
    friend istream& operator>>(istream& in, Matrix& x);
    friend ostream& operator<<(ostream& out, Matrix& x);
};
Matrix::Matrix(int r, int c) :row(r), col(c)
{
    matx = new int* [row];
    for (int i = 0; i < row; ++i)
        matx[i] = new int[col];
    for (int i = 0; i < row; ++i)
        for (int j = 0; j < col; ++j)
            matx[i][j] = 0;
}
Matrix::Matrix(int r, int c, uchar** init)
{
    row = r;
```

```
    col = c;
    matx = new int* [row];
    for (int i = 0; i < row; ++i)
        matx[i] = new int[col];
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            matx[i][j] = (int)init[i][j]; // 使用初始矩阵的对应元素进行初始化
        }
    }
}
Matrix::Matrix(int r, int c, int** init)
{
    row = r;
    col = c;
    matx = new int* [row];
    for (int i = 0; i < row; ++i)
        matx[i] = new int[col];
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            matx[i][j] = init[i][j]; // 使用初始矩阵的对应元素进行初始化
        }
    }
}
Matrix& Matrix::operator=(const Matrix& x2)
{
    row = x2.row;
    col = x2.col;
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            matx[i][j] = x2.matx[i][j];
        }
    }
    return *this;
}

Matrix Matrix::operator+(const Matrix& x2)
{
    Matrix tmp(row, col);
```

```
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            tmp.matx[i][j] = matx[i][j] + x2.matx[i][j];
        }
    }
    return tmp;
}
Matrix Matrix::operator*(int k)
{
    Matrix tmp(row, col);
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            tmp.matx[i][j] = matx[i][j] * k;
        }
    }
    return tmp;
}
Matrix Matrix::operator/(int k)
{
    Matrix tmp(row, col);
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            tmp.matx[i][j] = matx[i][j] / k;
        }
    }
    return tmp;
}
Matrix Matrix::operator*(Matrix& x2)
{
    Matrix result(row, x2.col);
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < x2.col; ++j) {
            result.matx[i][j] = 0;
            for (int k = 0; k < col; ++k) {
                result.matx[i][j] += matx[i][k] * x2.matx[k][j];
            }
        }
    }
```

```
    return result;
}

istream& operator>>(istream& in, Matrix& x)
{
    for (int i = 0; i < x.row; i++) {
        for (int j = 0; j < x.col; j++) {
            in >> x.matx[i][j];
        }
    }
    return in;
}
ostream& operator<<(ostream& out, Matrix& x)
{
    for (int i = 0; i < x.row; i++) {
        for (int j = 0; j < x.col; j++) {
            out << x.matx[i][j] << " ";
        }
        out << endl;
    }
    return out;
}
Matrix Matrix::Hadamard(const Matrix& x2) const
{
    Matrix tmp(row, col);
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            tmp.matx[i][j] = matx[i][j] * x2.matx[i][j];
        }
    }
    return tmp;
}
Matrix Matrix::convolution(const Matrix& b) const {
    int bRows = 3;
    int bCols = 3;
    int outRows = row;
    int outCols = col;
    Matrix result(outRows, outCols);
```

```
    int paddedRows = row + 2;
    int paddedCols = col + 2;
    Matrix paddedMatrix(paddedRows, paddedCols);
    // 在矩阵 A 的外围填充 0
    for (int i = 0; i < paddedRows; ++i) {
        for (int j = 0; j < paddedCols; ++j) {
            if (i == 0 || i == paddedRows - 1 || j == 0 || j == paddedCols - 1) {
                paddedMatrix.matx[i][j] = 0;
            }
            else {
                paddedMatrix.matx[i][j] = matx[i - 1][j - 1];
            }
        }
    }
    //卷积操作
    for (int i = 0; i < outRows; ++i) {
        for (int j = 0; j < outCols; ++j) {
            int sum = 0;
            for (int k = 0; k < bRows; ++k) {
                for (int m = 0; m < bCols; ++m) {
                    sum += paddedMatrix.matx[i + k][j + m] * b.matx[k][m];
                }
            }
            result.matx[i][j] = sum;
        }
    }
    return result;
}
int Matrix::get_sum()
{
    int sum = 0;
    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < col; ++j) {
            sum += matx[i][j];
        }
    }
```

```
    return sum;
}
void Matrix::check(Mat &img,int sum)
{
    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < col; ++j) {
            matx[i][j] /= sum;
            if (matx[i][j] < 0)
                matx[i][j] = 0;
            if (matx[i][j] > 255)
                matx[i][j] = 255;
            img.at<uchar>(i, j) = matx[i][j];
        }
    }
}

void wait_for_enter()
{
    cout << endl
        << "按回车键继续";
    while (_getch() != '\r')
        ;
    cout << endl
        << endl;
    cin.clear();
    cin.ignore(1024, '\n');
}

uchar** Mat2Vec(Mat mat)
{

    uchar** array = new uchar * [mat.rows];
    for (int i = 0; i < mat.rows; ++i)
        array[i] = new uchar[mat.cols];
    for (int i = 0; i < mat.rows; ++i)
    {
        for (int j = 0; j < mat.cols; ++j)
```

```
        {

            array[i][j] = mat.at<uchar>(i, 3*j);
            if (mat.at<uchar>(i, 3 * j) < 0) {
                array[i][j] = 0;
            }
            if (mat.at<uchar>(i, 3 * j) > 255) {
                array[i][j] = 255;
            }
        }
    }

    return array;
}
Mat Vector2Mat(vector<vector<int>>& imageVec)
{
    int numRows = (int)imageVec.size();
    int numCols = (int)imageVec[0].size();

    Mat imageMat(numRows, numCols, CV_8UC1);

    for (int i = 0; i < numRows; i++) {
        for (int j = 0; j < numCols; j++) {
            imageMat.at<uchar>(i, j) = static_cast<uchar>(imageVec[i][j]);
        }
    }

    return imageMat;
}
cv::Mat Vec2Mat(uchar** imageData, int numRows, int numCols)
{
    cv::Mat imageMat(numRows, numCols, CV_8UC1);

    for (int i = 0; i < numRows; i++) {
        uchar* rowPtr = imageMat.ptr<uchar>(i);
        for (int j = 0; j < numCols; j++) {
            rowPtr[j] = imageData[i][j];
        }
```

```
    }

    return imageMat;
}

char menu()
{
    cout << "*********************************************************" << endl;
    cout << " *      1 矩阵加法      2 矩阵数乘      3 矩阵转置     *" << endl;
    cout << " *      4 矩阵乘法      5 Hadamard 乘积  6 矩阵卷积     *" << endl;
    cout << " *      7 卷积应用      8 OTSU 算法     0 退出系统     *" << endl;
    cout << "*********************************************************" << endl;
    char menu;
    while (1) {
        menu = _getch();
        if (menu >= '0' && menu <= '8') {
            break;
        }
        else {
            cout << "输入错误" << endl;
            getchar();
            continue;
        }
    }
    system("cls");
    return menu;
}
void matriplus()
{
    int row, col;
    cout << "请输入矩阵 A 的行和列数:" << endl;
    cin >> row >> col;
    Matrix a(row, col);
    cout << "请输入矩阵 A 的数据:" << endl;
    cin >> a;
    cout << "矩阵 A 为:" << endl;
    cout << a;
    int row2, col2;
```

```
    cout << "请输入矩阵 B 的行和列数:" << endl;
    cin >> row2 >> col2;
    if (row != row2 || col != col2)
    {
        cout << "矩阵行和列不匹配，无法加和" << endl;
        return;
    }
    else {
        Matrix b(row, col);
        cout << "请输入矩阵 B 的数据:" << endl;
        cin >> b;
        cout << "矩阵 B 为:" << endl;
        cout << b;
        Matrix c(row, col);
        c = a + b;
        cout << "矩阵 A+B 的和为:" << endl;
        cout << c;
    }
}
void nummulti()
{
    int row, col;
    cout << "请输入矩阵的行和列数:" << endl;
    cin >> row >> col;
    Matrix a(row, col);
    cout << "请输入矩阵的数据:" << endl;
    cin >> a;
    cout << "矩阵为:" << endl;
    cout << a;
    int k;
    cout << "请输入要数乘的整数 k" << endl;
    cin >> k;
    a = a * k;
    cout << "数乘 " << k << " 后,矩阵为:" << endl;
    cout << a;
}
Matrix trans(const Matrix& a)
{
```

```
    Matrix a2(a.col, a.row);
    for (int i = 0; i < a.row; ++i) {
        for (int j = 0; j < a.col; j++) {
            a2.matx[j][i] = a.matx[i][j];
        }
    }
    return a2;
}
void matritrans()
{
    int row, col;
    cout << "请输入矩阵的行和列数:" << endl;
    cin >> row >> col;
    Matrix a(row, col);
    cout << "请输入矩阵的数据:" << endl;
    cin >> a;
    cout << "矩阵为:" << endl;
    cout << a;
    Matrix a2(col, row);
    cout << "矩阵转置后为:" << endl;
    a2 = trans(a);
    cout << a2;
}
void matrimulti()
{
    int row, col;
    cout << "请输入矩阵 A 的行和列数:" << endl;
    cin >> row >> col;
    Matrix a(row, col);
    cout << "请输入矩阵 A 的数据:" << endl;
    cin >> a;
    cout << "矩阵 A 为:" << endl;
    cout << a;
    int row2, col2;
    cout << "请输入矩阵 B 的行和列数:" << endl;
    cin >> row2 >> col2;
```

```
    //判断是否可以做乘法
    if (col!=row2) {
        cout << "行和列不匹配，无法进行矩阵乘法" << endl;
        return;
    }
    else {
        Matrix b(row2, col2);
        cout << "请输入矩阵 B 的数据:" << endl;
        cin >> b;
        cout << "矩阵 B 为:" << endl;
        cout << b;
        Matrix c(row, col2);
        c = a * b;
        cout << "矩阵 A 和 B 的乘积为:" << endl;
        cout << c;
    }
}
void hadamulti()
{
    int row, col;
    cout << "请输入矩阵 A 的行和列数:" << endl;
    cin >> row >> col;
    Matrix a(row, col);
    cout << "请输入矩阵 A 的数据:" << endl;
    cin >> a;
    cout << "矩阵 A 为:" << endl;
    cout << a;
    int row2, col2;
    cout << "请输入矩阵 B 的行和列数:" << endl;
    cin >> row2 >> col2;
    if(row!=row2||col!=col2)
    {
        cout << "矩阵行和列不匹配，无法做 Hadamard 乘积" << endl;
        return;
    }
    else {
        Matrix b(row2, col2);
        cout << "请输入矩阵 B 的数据:" << endl;
```

```
        cin >> b;
        cout << "矩阵 B 为:" << endl;
        cout << b;
        Matrix c(row, col);
        c = a.Hadamard(b);
        cout << "矩阵 A+B 的 Hadamard 乘积为:" << endl;
        cout << c;
    }

}
void conv(Matrix& b) //传入卷积核
{
    int row, col;
    cout << "请输入矩阵 A 的行和列数:" << endl;
    cin >> row >> col;
    Matrix a(row, col);
    cout << "请输入矩阵 A 的数据:" << endl;
    cin >> a;
    cout << "矩阵 A 为:" << endl;
    cout << a;
    Matrix result(row,col);
    result = a.convolution(b);
    cout << "卷积结果为:" << endl;
    cout << result << endl;
    cout << result.getrow() << "行 " << result.getcol()<<"列 " << endl;
}
void demo()
{

    int init1[3][3] = { {1,1,1},
                        { 1,1,1},
                        { 1,1,1} };
    Matrix B1(3, 3);
    B1.initialize(init1);
    int init2[3][3] = { {-1,-2,-1},
                  { 0,0,0},
                  { 1,2,1} };
    Matrix B2(3, 3);
```

```
B2.initialize(init2);
int init3[3][3] = { {-1,0,1},
                    { -2,0,2},
                    { -1,0,1} };
Matrix B3(3, 3);
B3.initialize(init3);
int init4[3][3] = { {-1,-1,-1},
                    { -1,9,-1},
                    { -1,-1,-1} };
Matrix B4(3, 3);
B4.initialize(init4);
int init5[3][3] = { {-1,-1,0},
                    { -1,0,1},
                    { 0,1,1} };
Matrix B5(3, 3);
B5.initialize(init5);
int init6[3][3] = { {1,2,1},
                    { 2,4,2},
                    { 1,2,1} };
Matrix B6(3, 3);
B6.initialize(init6);
 Mat image =
     imread("demolena.jpg"); // 图像的灰度值存放在格式为Mat 的变量 image 中
 int sum = 0;
 Matrix a(image.rows, image.cols, Mat2Vec(image));
 Matrix c1(image.rows, image.cols);
 c1 = a.convolution(B1);
 sum = B1.get_sum();
 if (sum == 0)
     sum = 1;
 Mat img1(image.rows, image.cols, CV_8UC1);
 c1.check(img1,sum);
 imshow("B1", img1);

 Matrix c2(image.rows, image.cols);
 c2 = a.convolution(B2);
 sum = B2.get_sum();
 if (sum == 0)
```

```
    sum = 1;
Mat img2(image.rows, image.cols, CV_8UC1);
c2.check(img2,sum);
imshow("B2", img2);

Matrix c3(image.rows, image.cols);
c3 = a.convolution(B3);
sum = B3.get_sum();
if (sum == 0)
    sum = 1;
Mat img3(image.rows, image.cols, CV_8UC1);
c3.check(img3,sum);
imshow("B3", img3);

Matrix c4(image.rows, image.cols);
c4 = a.convolution(B4);
sum = B4.get_sum();
if (sum == 0)
    sum = 1;
Mat img4(image.rows, image.cols, CV_8UC1);
c4.check(img4,sum);
imshow("B4", img4);

Matrix c5(image.rows, image.cols);
c5 = a.convolution(B5);
sum = B5.get_sum();
if (sum == 0)
    sum = 1;
Mat img5(image.rows, image.cols, CV_8UC1);
c5.check(img5,sum);
imshow("B5", img5);

Matrix c6(image.rows, image.cols);
c6 = a.convolution(B6);
sum = B6.get_sum();
if (sum == 0)
    sum = 1;
Mat img6(image.rows, image.cols, CV_8UC1);
```

```
    c6.check(img6,sum);
    imshow("B6", img6);

    waitKey(0);
    return;
}

/* 有问题，阈值计算不对
double calculateVariance(const std::vector<int>& histogram, int threshold)
{
    int totalPixels = 0;
    int sum = 0;
    for (int i = 0; i <= threshold; i++)
    {
        totalPixels += histogram[i];
        sum += i * histogram[i];
    }
    if (totalPixels == 0)
        return 0;
    double mean1 = sum / static_cast<double>(totalPixels);

    int totalPixels2 = 0;
    int sum2 = 0;
    for (int i = threshold + 1; i < histogram.size(); i++)
    {
        totalPixels2 += histogram[i];
        sum2 += i * histogram[i];
    }
    if (totalPixels2 == 0)
        return 0;
    double mean2 = sum2 / static_cast<double>(totalPixels2);

    double variance1 = 0.0;
    double variance2 = 0.0;

    for (int i = 0; i <= threshold; i++)
    {
        double diff = i - mean1;
```

```
        variance1 += diff * diff * histogram[i];
    }
    variance1 /= (totalPixels + totalPixels2);

    for (int i = threshold + 1; i < histogram.size(); i++)
    {
        double diff = i - mean2;
        variance2 += diff * diff * histogram[i];
    }
    variance2 /= (totalPixels + totalPixels2);

    return variance1 + variance2;
}

void OSTUbin(const vector<int>& histogram, int& bestThreshold, double& bestVariance)// OTSU二值化
{
    int totalPixels = 0;
    for (int i = 0; i < 256; ++i)
    {
        totalPixels += histogram[i];
    }
    bestThreshold = 0; //最佳阈值
    bestVariance = 0.0;//最佳方差
    for (int threshold = 0; threshold < 256; ++threshold)
    {
        double variance = calculateVariance(histogram, threshold);
        //cout << variance <<"@@" << endl;
        if (variance > bestVariance) {
            bestVariance = variance;
            bestThreshold = threshold;
            cout <<"最好阈值"<< bestThreshold << endl;
        }
    }
}
*/

void binImg(const vector<vector<int>>& image, vector<vector<int>>& binImage, int threshold)
{
```

```
    int row_img = (int)image.size();
    int col_img = (int)image[0].size();
    //调整二值化图像大小
    binImage.resize(row_img, std::vector<int>(col_img, 0));
    //根据阈值将图像像素点二值化
    for (int i = 0; i < row_img; ++i) {
        for (int j = 0; j < col_img; ++j) {
            if (image[i][j] >= threshold) {
                binImage[i][j] = 255;
            }
            else
                binImage[i][j] = 0;
        }
    }
}
void OSTUCUT(uchar** originImage,const vector<vector<int>>& image, vector<vector<int>>& binImage, int
threshold)
{
    int row_img = (int)image.size();
    int col_img = (int)image[0].size();
    binImage.resize(row_img, std::vector<int>(col_img, 0));
    for (int i = 0; i < row_img; ++i) {
        for (int j = 0; j < col_img; ++j) {
            if (image[i][j] >= threshold) {
                binImage[i][j] = originImage[i][j];
            }
            else
                binImage[i][j] = 0;
        }
    }
}
int BestThreshold(const std::vector<int>& histogram)//计算最佳阈值
{
    int totalPixels = 0;
    for (int i = 0; i < histogram.size(); ++i)
    {
        totalPixels += histogram[i];
    }
```

```
    float sumWeight = 0.0f;
    for (int i = 0; i < histogram.size(); ++i)
    {
        sumWeight += i * histogram[i];
    }

    int sumB = 0;
    int weightBK = 0;
    int weightFG = 0;

    float bestVariance = 0.0f;
    int threshold = 0;

    for (int i = 0; i < histogram.size(); ++i)
    {
        weightBK += histogram[i];
        if (weightBK == 0) continue;

        weightFG = totalPixels - weightBK;
        if (weightFG == 0) break;

        sumB += i * histogram[i];

        float averageBK = static_cast<float>(sumB) / weightBK;
        float averageFG = static_cast<float>(sumWeight - sumB) / weightFG;

        float varBetween = static_cast<float>(weightBK) * static_cast<float>(weightFG) * (averageBK
- averageFG) * (averageBK - averageFG);

        if (varBetween > bestVariance)
        {
            bestVariance = varBetween;
            threshold = i;
        }
    }

    return threshold;
```

```
}

void OTSU(Mat imageMat,int choice) //choice==1 二值化  //choice==2 截取图像
{
    vector<vector<int>> image;
    int numRows = imageMat.rows;
    int numCols = imageMat.cols;
    imshow("Ori", imageMat);
    uchar** imageVec=Mat2Vec(imageMat);
    for (int i = 0; i < numRows; i++) {
        std::vector<int> rowVector;
        for (int j = 0; j < numCols; j++) {
            rowVector.push_back(imageVec[i][j]); // 将每个像素值添加到行向量中
        }
        image.push_back(rowVector); // 将每一行的向量添加到图像向量中
    }
    vector<int> histogram(256, 0);
    for (int i = 0; i < image.size(); i++) {
        const std::vector<int>& row = image[i];
        for (int j = 0; j < row.size(); j++) {
            int pixel = row[j];
            histogram[pixel]++;
        }
    }
    //OTSU 二值化
    int bestThreshold = 0;
    double bestVariance = 0.0;
    bestThreshold = BestThreshold(histogram);
    //bestThreshold = 100;
    //图像二值化
    vector<vector<int>> binImage;
    if (choice == 1) {
        binImg(image, binImage, bestThreshold);
        imshow("OTSU 二值化", Vector2Mat(binImage));
    }
    if (choice == 2) {
        OSTUCUT(imageVec, image, binImage, bestThreshold);
        imshow("OTSU 截取主体", Vector2Mat(binImage));
```

```
    }
    waitKey(0);
}
void show_otsu()
{
    Mat image1 =
        imread("demolena.jpg");
    cout << "Choice=1,下面演示利用 OTSU 算法二值化 lena 图像" << endl;
    OTSU(image1,1); //二值化 lena
    Mat image2 =
        imread("brain.jpg",1);
    cout << "Choice=2,下面演示利用 OTSU 算法截取 brain 图像" << endl;
    OTSU(image2,2); //截取
    Mat image3 =
        imread("ship.jpg",1);
    cout << "Choice=2,下面演示利用 OTSU 算法截取 ship 图像" << endl;
    OTSU(image3,2); //截取
    Mat image4 =
        imread("polyhedrosis.jpg",1);
    cout << "Choice=2,下面演示利用 OTSU 算法截取 polyhedrosis 图像" << endl;
    OTSU(image4,2); //截取
    Mat image5 =
        imread("snowball.jpg",1);
    cout << "Choice=2,下面演示利用 OTSU 算法截取 snowball 图像" << endl;
    OTSU(image5,2); //截取
    waitKey(0);
}

int main()
{
    // 定义相关变量
    char choice = 0;
    char ch = 0;
    int init[3][3] = {{-1,0,1},
                      { -1,0,1},
                      { -1,0,1} };
    Matrix B0(3, 3);
    B0.initialize(init);
```

```
while (true)
{
    system("cls");
    choice = menu();
    if (choice == '0') // 选择退出
    {
        cout << "\n 确定退出吗?" << endl;
        cin >> ch;
        if (ch == 'y' || ch == 'Y')
            break;
        else
            continue;
    }

switch (choice)
    {
        case '1':
            matriplus();
            wait_for_enter();
            break;
        case '2':
            nummulti();
            wait_for_enter();
            break;
        case '3':
            matritrans();
            wait_for_enter();
            break;
        case '4':
            matrimulti();
            wait_for_enter();
            break;
        case '5':
            hadamulti();
            wait_for_enter();
            break;
        case '6':
            conv(B0);
```

```
                wait_for_enter();
                break;
            case '7':
                demo();
                wait_for_enter();
                break;
            case'8':
                show_otsu();
                break;
            default:
                cout << "\n 输入错误，请从新输入" << endl;
                wait_for_enter();
        }
    }
    return 0;
}
```