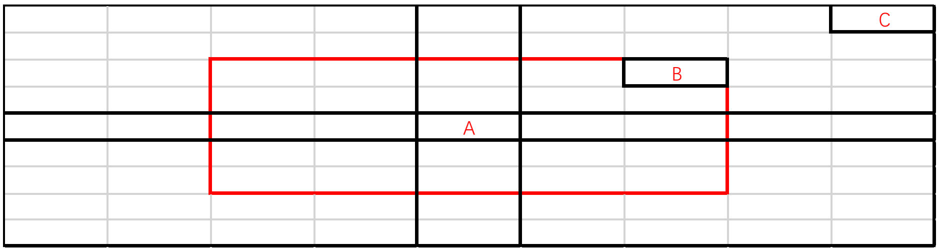
第三章作业 黄海浪 9181040G0818

1. 滤波前图 3-11 和滤波后图 3-12 的标准差变化较明显，为什么滤波前图 3-13 和滤波后图 3-14 的标准差几乎没有变化？
   答：原图 3-11 的色彩鲜明，边缘信息多，边缘亮度的突变较多。而图 3-13 边缘信息少，在边缘处亮度大多是渐变的，突变较少。将滤波窗口变大，也可以使得 3-13 变换后与原图相比有较大的差异。

2. 对一个图像A作 5×5 的均值滤波得到图像 B，对图像B再做 5×5 的均值滤波得到图像C。那么，对图像A做多大的均值滤波可以直接得到图像C？
   答：两次 5*5，做第二次 5*5 时的方格里面的数据最多来原与某个方向的 4 块，那么直接对 A 做 9*9 的均值滤波可以达到两次 5*5 的效果，如图：



3. 在图像处理中，彩色到灰度转换公式为：$gry = 0.299 \times red + 0.587 \times green + 0.114 \times blue$，请用 C/C++编程把彩色图像 H0301Rgb.bmp 转成为灰度图像 H0301Gry.bmp，请分别使用“整数除法或者浮点乘法除法变为整数乘法和移位”的编程技巧和直接计算，分别使用 Debug 和 Release 编译，并各执行 1000 次，比较它们的时间花费（C/C++中的时间函数为 clock_t t=clock()）。
   答：
   两个公式分别为：$gry = 0.299 \times red + 0.587 \times green + 0.114 \times blue$
   $gry = (306*red + 601*green + 116*blue)>>10$
   因为 1000 次有点比较不出，所以运行了 3000 次，结果如下：

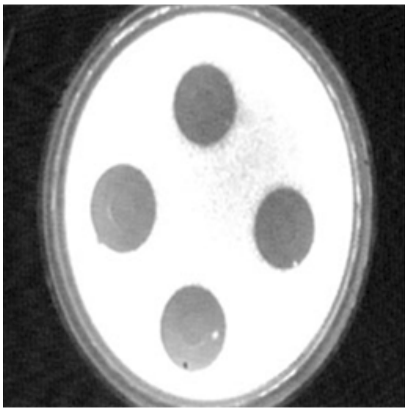
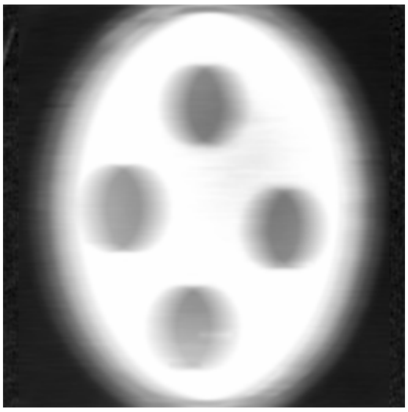
	debug	release
浮点乘	3343.95	1240.14
整数乘+移位	2853.62	850.589

主要代码：

```
    clock_t t_start = clock();
    for (unsigned times = 0; times < 3000; ++times) {
        auto *pCurImg = data;
        //非查表法 1
        for (auto *pGrayImg = grayData, *pEndGrayImg = pGrayImg + newImgSize; pGrayImg < pEndGrayImg;) {
            *(pGrayImg++) = 0.114 * (*(pCurImg++)) + 0.587 * (*(pCurImg++)) + 0.299 * (*(pCurImg++));
        }
        //非查表法 2
//          for (auto *pGrayImg = grayData, *pEndGrayImg = pGrayImg + newImgSize; pGrayImg < pEndGrayImg;) {
//              *(pGrayImg++) = (116 * (*(pCurImg++)) + 601 * (*(pCurImg++)) + 306 * (*(pCurImg++))) >> 10;
//          }
    }
    clock_t t_end = clock();
    std::cout << (t_end - t_start) / 1000.0;
```

4. 参照算法 3-1，采用 C/C++编程，实现对于灰度图像 H0302Gry.bmp 每行上的一维均值滤波（邻域大小为 1 行 M 列，M=21）。

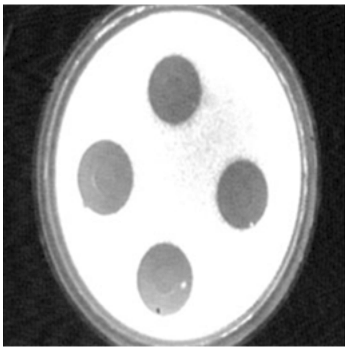
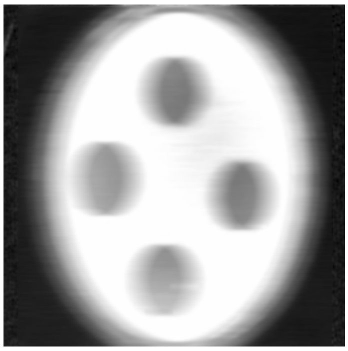
答：下面分别为原图和均值滤波后图（边缘没有处理）

主要代码：

```cpp
/// 均值滤波列积分
bool BmpFile::avrFilterBySumCol(unsigned M, unsigned N) {
    // 窗口 M列 N行
    unsigned half_x = M >> 1;
    unsigned half_y = N >> 1;
    M = (half_x << 1) | 0x0001;
    N = (half_y << 1) | 0x0001;
    unsigned C = (1 << 23) / (M * N);
    unsigned width = infoHeader.bitmap_width;
    unsigned height = infoHeader.bitmap_height;
    auto *sumCol = new unsigned[width];
    memset(sumCol, 0, width << 2);

    // 初始化sumCol
    auto *pAdd = data, *pDel = data;
    for (auto *pEnd = data + N * width; pAdd < pEnd;) {
        for (unsigned x = 0; x < width; ++x) {
            sumCol[x] += *(pAdd++);
        }
    }
    auto *resData = new unsigned char[infoHeader.image_size];
    memcpy(resData, data, infoHeader.image_size);
    // 进行滤波 跳过边缘的处理
    for (auto *pCur = resData + half_y * width, *pEnd = resData + (height - half_y) * width; pCur < pEnd;) {
        // 初始化sum
        unsigned sum = 0;
        for (unsigned x = 0; x < M; ++x) {
            sum += sumCol[x];
        }
        pCur += half_x;
        for (unsigned x = half_x, end_x = width - half_x; x < end_x; ++x) {
            // 乘法换除法
            *(pCur++) = (sum * C) >> 23;
            // 更新sum
            sum -= sumCol[x - half_x];
            sum += sumCol[x + half_x + 1];
        }
        pCur += half_x;
        for (unsigned x = 0; x < width; ++x) {
            sumCol[x] -= *(pDel++);
            sumCol[x] += *(pAdd++);
        }
    }
    delete[] data;
    data = resData;

    delete[] sumCol;

    return true;
}
```

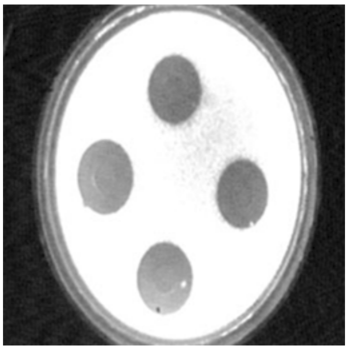
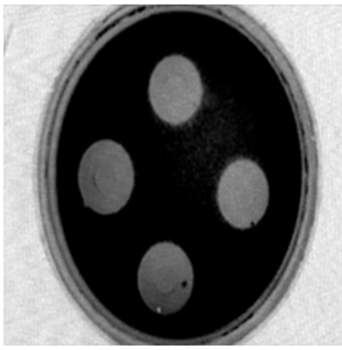
5. 参照算法 3-2，采用 C/C++编程，实现对于灰度图像 H0302Gry.bmp 每行上的一维均值滤波（邻域大小为 1 行 M 列，M=21），要使用一维积分图。
答：使用了基于列积分的计算方法得到了一维积分图，在计算时，左边缘和上边缘会少处理一行/一列像素，为了能够直接使用公式 graph(x,y)+graph(m,n)-graph(x,n)-graph(m,y)
最终效果如下：

主要代码：

```cpp
///均值滤波积分图
bool BmpFile::avrFilterByGraph(unsigned int M, unsigned int N) {
    if (sumGraph == nullptr) {
        this->getSumGryGraphBySumCol();
    }
    // 窗口 M列 N行
    unsigned half_x = M >> 1;
    unsigned half_y = N >> 1;
    M = (half_x << 1) | 0x0001;
    N = (half_y << 1) | 0x0001;
    unsigned C = (1 << 23) / (M * N);
    unsigned width = infoHeader.bitmap_width;
    unsigned height = infoHeader.bitmap_height;

    auto *resData = new unsigned char[infoHeader.image_size];
    memcpy(resData, data, infoHeader.image_size);
    // 进行滤波 跳过边缘的处理
    auto *pRes = resData + half_y * width;
    auto *pSmallSumGraph = sumGraph;
    auto *pBigSumGraph = sumGraph + N * width;

    // +1 是确保 算sum的公式为正确的（4个值），否则应该为一个值或者两个值 但这样会少处理一行一列像素
    for (unsigned y = half_y + 1, end_y = height - half_y;
         y < end_y; ++y, pSmallSumGraph += width, pBigSumGraph += width) {
        pRes += half_x + 1;
        unsigned sum = 0;
        for (unsigned x = half_x + 1, end_x = width - half_x, x1 = 0, x2 = M; x < end_x; ++x, ++x1, ++x2) {
            sum = *(pBigSumGraph + x2) + *(pSmallSumGraph + x1) - *(pBigSumGraph + x1) - *(pSmallSumGraph + x2);
        }
        pRes += half_x;
    }

    delete[] data;
    data = resData;

    return true;
}
```

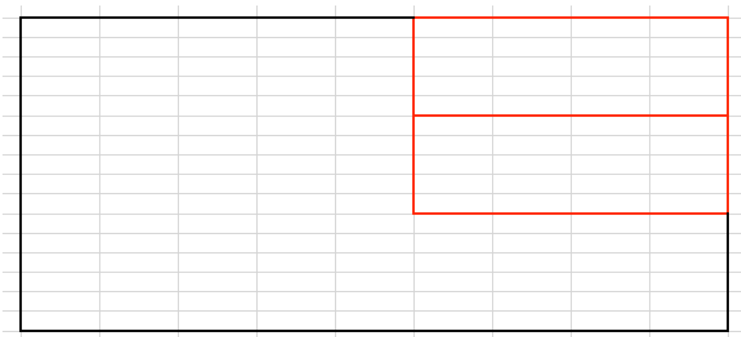
6. 使用 MMX 或者 SSE 指令集，用 C/C++编程实现灰度图像 H0302Gry.bmp 的反相。
答：利用_mm_set1_epi8(0xff)加载了一个全 1 的值给类型__m128i 后会自动高位补 0，再利用_mm_xor_si128(*pSSE, F1)进行异或处理，达到取反的效果。
最终效果如下：

主要代码：

```
///图片反向 SSE
void BmpFile::InvertBySSE() const {
    // F1 高位会自动补1
    __m128i F1 = _mm_set1_epi8( b: 0xff);
    __m128i *pSSE = (__m128i *) data;
    for (auto *pCur = data, *pEnd = data + infoHeader.image_size; pCur < pEnd; pCur += 16) {
        *(pSSE++) = _mm_xor_si128(*pSSE, F1);
    }
}
```

7. 使用算法 3-6 对灰度图像 H0302Gry.bmp 进行中值滤波，分别使用 5 × 5，13 × 13，21 × 21 大小的邻域进行中值滤波，比较这三种邻域时 dbgCmpTimes 的值，并进行分析。
答：5X5: 6.43626；13X13: 5.851；21X21: 5.91256；
三者值差距不大，和窗口大小基本无关，对于同一张图片，从代码中看，可以理解为更新直方图后，小窗口的值和大窗口部分重叠，比较大窗口时比较次数可以简单理解为比较小窗口之和，窗口越大移动窗口次数越少，会在一定程度上降低比较次数。(比如对于 H0304Gry.bmp，5X5: 3.01665；13X13: 2.81271；21X21: 2.75748；)



8. 采用高速摄像机（2000fps）对准一个区域进行拍照，该区域有一发炮弹高速飞过。摄像机得到了M幅场景图像，但M幅图像的每一幅中都有炮弹，请问如何才得到一幅没有炮弹的场景图像（一般称为背景图像）？
   答：新建一副和拍摄图像一样大的背景图，将 M 副图像的图对应的像素点用“中值滤波”，利用 M 副图像即可得到没有炮弹的背景图像。

9.

$T^3 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$和$T^5 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$等价吗？

答：等价，大部分区域处理的效果一样，但是如果忽略边缘的处理，那么在边缘处不一样。

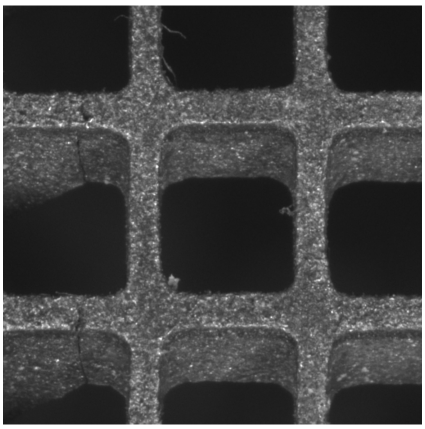
10.

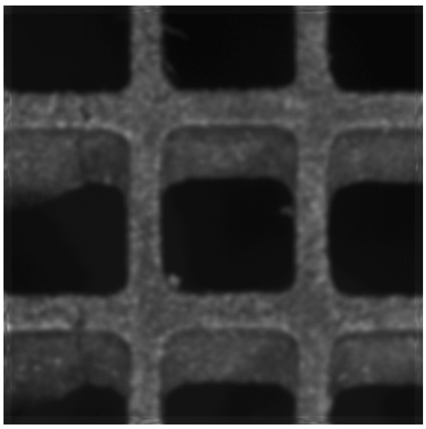
$T^3 = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$有什么特点？使用它进行图像平滑需要乘法和除法运算吗？

答：1，2，4，16 分别为 2^0，2^1，2^2，2^4，可以不用乘法除法运算，直接用移位。

11. 使用标准差等于 1.0 的高斯函数计算一维卷积模板 T(j)(-3<=j<=3)，得到 T={0.003134，0.038177，0.171099，0.282095，0.171099，0.038177，0.003134}，按式(3-11)实现高斯滤波时，会有大量的浮点计算，如何消除这些浮点运算？
   答：取 T 中最小的那个数 N，乘以 X 后得到的值为整数，再次计算$M = \lfloor \log(X) \rfloor + 1$，最后的这个 M 即为我们需要的移位数，然后将 N 变为$N' = \lfloor N \ll M \rfloor$，计算公式则变为$ans = N' * Y \gg M$，如 T 中最小为 0.003134，$M = \lfloor \log(10^6) \rfloor + 1 = 20$，则 0.003134 变为 3286，Y*0.003134 变为 3286*Y>>M。

12. 使用 C/C++编程，对灰度图像 H0303Gry.bmp 实现标准差$\delta = 3$的高斯滤波，使用两个一维高斯平滑的串联实现，并且在像素处理循环中不使用浮点运算。
   答：用了移位，选取最小的一个，按照题 11 方式计算时 M 可能过大，造成计算溢出，这时取 M 最大为 24 位，因为像素亮度最大为 8 位值即 255。
   图片结果如下：





12 题主要代码：

```
bool BmpFile::GaussianFilter(int const &sigma) {
    int wSize = 3 * sigma + 1;
    auto *T = new double[wSize];
    auto *T_1 = new unsigned[wSize];
    auto *resData = new unsigned char[infoHeader.image_size];
    memcpy(resData, data, infoHeader.image_size);
    // 计算值
    for (int j = 0; j < wSize; ++j) {
        T[j] = calTj(j, sigma);
    }
    // 将值转为整数类型的
    unsigned M = 0;
    double minNum = T[wSize - 1];
    // 即余下的值 与 原来的值相比基本无变化
    while (1e6 * (minNum - int(minNum)) > minNum) {
        minNum *= 2;
        M++;
    }
    // 避免计算时溢出
    M = fmin( lcpp_x: 24, M);
    for (int j = 0; j < wSize; ++j) {
        T_1[j] = unsigned(T[j] * (1 << M));
    }
    // 开始计算
    for (auto *pCur = data + wSize, *pEnd = data + infoHeader.image_size - wSize, *pRes = resData + wSize;
         pCur < pEnd; pCur += wSize << 1, pRes += wSize << 1) {
        auto *rowEnd = pCur + infoHeader.bitmap_width - (wSize << 1);
        while (pCur < rowEnd) {
            unsigned sum = (*pCur * T_1[0]) >> M;
            for (unsigned x = 1; x < wSize; ++x) {
                sum += (*(pCur + x) * T_1[x]) >> M;
                sum += (*(pCur - x) * T_1[x]) >> M;
            }
            *(pRes++) = sum;
            pCur++;
        }
    }
    // 转一下
    auto *resData2 = new unsigned char[infoHeader.image_size];
    for (unsigned y = 0; y < infoHeader.bitmap_height; ++y) {
        for (unsigned x = 0; x < infoHeader.bitmap_width; ++x) {
            *((resData2 + x * infoHeader.bitmap_height) + y) = *((resData + y * infoHeader.bitmap_width) + x);
        }
    }
    // 重新计算resData
    memcpy(resData, resData2, infoHeader.image_size);
    // 开始计算
    for (auto *pCur = resData + wSize, *pEnd = resData + infoHeader.image_size - wSize, *pRes = resData2 + wSize;
         pCur < pEnd; pCur += wSize << 1, pRes += wSize << 1) {
        auto *rowEnd = pCur + infoHeader.bitmap_height - (wSize << 1);
        while (pCur < rowEnd) {
            unsigned sum = (*pCur * T_1[0]) >> M;
            for (unsigned x = 1; x < wSize; ++x) {
                sum += (*(pCur + x) * T_1[x]) >> M;
                sum += (*(pCur - x) * T_1[x]) >> M;
            }
            *(pRes++) = sum;
            pCur++;
        }
    }
    // 再转回去
    for (unsigned y = 0; y < infoHeader.bitmap_width; ++y) {
        for (unsigned x = 0; x < infoHeader.bitmap_height; ++x) {
            *((resData + x * infoHeader.bitmap_width) + y) = *((resData2 + y * infoHeader.bitmap_height) + x);
        }
    }
    delete[] resData2;

    delete[] data;
    data = resData;

    delete[] T;
    delete[] T_1;
    return true;
}
```

13. 算法 3-6 中 threshold 取何值时能够实现二值图像的最小值滤波、最大值滤波和中值滤波？

答：最大值滤波：threshold=255，即领域内所有值为 255 才取 255。

最小值滤波：threshold=0，即领域内所有值为 0 才取 0。

中值滤波：threshold=127，即领域内所有值平均下来大于等于 127 取 255，否则取 0。

14. 请分别使用超限平滑和K个邻点平均法，对 3.2.1 节中的数据 D1 进行邻域为 3 个像素的均值滤波，写出滤波结果。

超限平滑：令阈值 C=4

原值	求均值	均值
3	领域不完整，保持原值	3
3	（3+3+3）/3=3<4	3
3	（3+3+9）/3=5>4	5
9	（3+9+3）/3=5>4	5
3	（3+9+3）/3=5>4	5
3	（3+9+3）/3=5>4	5
9	（9+9+3）/3=7>4	7
9	（9+9+9）/3=9>4	9
9	（9+9+3）/3=7>4	7
3	（9+9+3）/3=7>4	7
9	（9+9+3）/3=7>4	7
9	（9+9+9）/3=9>4	9
9	领域不完整，保持原值	9

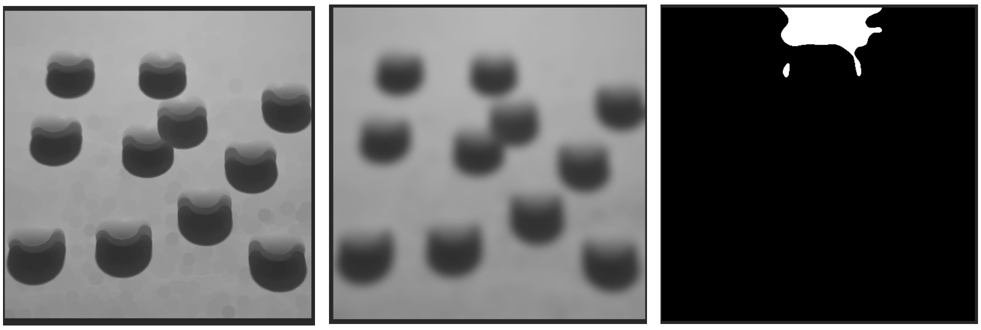
K 个邻点平均法：取 K=2

原值	求均值	均值
3	领域不完整，保持原值	3
3	（3+3）/2=3	3
3	（3+3）/2=3	3
9	（3+3）/2=3	3
3	（3+3）/2=3	3
3	（3+3）/2=3	3
9	（9+9）/2=9	9
9	（9+9）/2=9	9
9	（9+9）/2=9	9
3	（9+9）/2=9	9
9	（9+9）/2=9	9
9	（9+9）/2=9	9
9	领域不完整，保持原值	9

15. 使用 Photoshop 中的 median Filter 或者 Gaussian Blur，估计拍摄灰度图像 H0304Gry.bmp 时光源的位置，给出不含有药片的光照图像，并通过观察药片的阴影确认得到的光照图像是正确的。

答：首先进行最小值滤波，再进行高斯滤波，最后调整阈值。

图片结果如下，可以看到光是从图片上方的方向打过来的。



第一次实验报告

第二章作业更正：

题目 4（修改了第 290 行，以及去掉变量 A（代码中的 sum））

```
280 /// 直方图均衡化
281 bool BmpFile::histogramEqualization() {
282     // 获取直方图
283     if (histogram == nullptr) {
284         this->getHistogram();
285     }
286     auto *LUT = new unsigned char[char_express];
287 
288     for (unsigned i = 0, sum = 0; i < char_express; ++i) {
289         sum += histogram[i];
290         LUT[i] = ((sum << 8) - sum) / infoHeader.image_size;
291     }
292 
293     // 修改后
294 //    LUT[0] = ((histogram[0] << 8) - histogram[0]) / infoHeader.image_size;
295 //    for (unsigned i = 1; i < char_express; ++i) {
296 //        histogram[i] += histogram[i - 1];
297 //        LUT[i] = ((histogram[i] << 8) - histogram[i]) / infoHeader.image_size;
298 //    }
299 
300     for (auto *pCur = data, *pEnd = pCur + infoHeader.image_size; pCur < pEnd; pCur++) {
301         *pCur = LUT[*pCur];
302     }
303 
304     delete[] LUT;
305     return true;
306 }
```

题目 5：

直接用灰度图像的直方图均衡化，然后得到

题目 7：

bmp 文件储存，行顺序反的。彩色图像行储存为 BGR，结果如下：

第二章题目 7 修改后主要代码：

```
/// 加载一个 14位raw 图像
BmpFile *BmpFile::load14Raw(const std::string &path, unsigned int width, unsigned int height) {
    std::ifstream infile(path, mode: std::ios::in | std::ios::binary);
    if (!infile) {
        std::cerr << "Open file failed." << std::endl;
        throw;
    }
    auto size = width * height;
    auto *data = new short int[size];
    // 读取文件
    infile.seekg( off: 0, dir: std::ios::beg).read(reinterpret_cast<char *>(data), n: size * sizeof(short int));
    infile.close();
    // 新建一个bmp文件
    auto *bmp = createGrayBmp(width, height);
    // 初始化bmp的数据
    bmp->data = new unsigned char[bmp->infoHeader.image_size];
    memset(bmp->data, c: 0, bmp->infoHeader.image_size);

    // 获取 14位图 直方图
    unsigned short_int_express = 1 << 14;
    auto *histogram = new unsigned[short_int_express];
    memset(histogram, c: 0, len: sizeof(unsigned) << 14);
    for (auto *pCur = data, *pEnd = pCur + size; pCur < pEnd;) {
        histogram[*(pCur++)]++;
    }
    // 计算得到查找表 （直方图均衡化）
    auto *LUT = new unsigned char[short_int_express];
    for (unsigned i = 0, sum = 0; i < short_int_express; ++i) {
        sum += histogram[i];
        LUT[i] = ((sum << 8) - sum) / size;
    }

    // 将数据按照响应格式储存
    auto *resImg = new unsigned char[bmp->infoHeader.image_size];
    auto *pDesCur = resImg;
    for (auto *pCur = data, *pEnd = pCur + size; pCur < pEnd; pCur++) {
        *(pDesCur++) = LUT[*pCur];
    }
    bmp->writeBitData(width, height, resImg);

    delete[] resImg;
    delete[] data;
    delete[] histogram;
    delete[] LUT;

    return bmp;
}
```

不足之处：

1. 类里面函数用了很多 new，但是每次写完 new 我就会加一行 delete，然后再在他们之间写其他代码，也不知道如何去改进。

2. 然后 MMX 指令求 sumCol 哪一块怎么调试也调不对，4 个数可以分开成一对 两个数，但是分出来的两个数分不开。

成长：

这门课带来了太多太多以前没有接触的编程思维，显得有点措手不及，但是仔细做了之后感觉想的会变多了点。图像处理的知识也越发丰富起来，要能处理，还要处理得快。总的来说，这门课很有意思，学到了很多其他课学不到的知识，非常感谢老师。

实验最终代码：

utils/bmpFile.h （主要部分）

```
class BmpFile {
public:
    const unsigned char_express = 1 << 8;
    // 是否是 RGB 图
    bool isRgb = true;
    bitmap_file_header fileHeader;
    bitmap_info_header infoHeader;
    bitmap_palette *palette = nullptr;
    unsigned fileHeaderSize = sizeof(fileHeader);
    unsigned infoHeaderSize = sizeof(infoHeader);
    unsigned paletteSize = sizeof(bitmap_palette);
    unsigned paletteNum = 0;

    // 存文件时用的 header
    unsigned char *header = nullptr;
    unsigned headerSize = 0;
    // 文件数据
    unsigned char *data = nullptr;

    // 直方图
    unsigned *histogram = nullptr;
    // 亮度
    double bright = 0;
    // 对比度
    double contrast = 0;

    //积分图
    unsigned *sumGraph = nullptr;

    ~BmpFile() {
        delete[] this->palette;
        delete[] this->header;
        delete[] this->data;
        delete[] this->histogram;
        delete[] this->sumGraph;
    }

    //新建 bmp 图像
    static BmpFile *createGrayBmp(unsigned width, unsigned height);

    //将数据转为 8 位的数据
```

```
    void writeBitData(unsigned width, unsigned height, unsigned char *_data)
const;

    // 将已有的信息写入头
    void copyHeader2Men();

    // 加载图片
    void load(const std::string &);

    // 保存图片
    void save(const std::string &);

    // 反向
    void Invert() const;

    // 如果是灰色 则弄伪彩色
    bool gryPseudoColor() const;

    // rgb 到 gray 图片
    bool rgb2Gray();

    //均值方差规定化
    bool MVR(const double &bright, const double &contrast);

    //得到直方图
    void getHistogram();

    // 得到亮度和对比度
    void getBrightContrast();

    //直方图均衡化
    bool histogramEqualization();

    // 拿到彩色的 Histogram
    void getHistogramRgb();

    //raw 加载
    static BmpFile *load14Raw(const std::string &path, unsigned width,
unsigned height);

    //彩色灰度图像 浮点乘 到 整数乘 测试
    bool rgb2Gray_test1();

    //均值滤波列积分
```

```
    bool avrFilterBySumCol(unsigned M, unsigned N);

    //积分图实现
    void getSumGryGraphBySumCol();

    //SSE 实现
    void getSumGryGraphBySSE();

    //均值滤波 积分图
    bool avrFilterByGraph(unsigned M, unsigned N);

    //反向 SSE
    void InvertBySSE() const;

    //中值滤波
    double medianFilter(unsigned M, unsigned N);

    //高斯滤波
    bool GaussianFilter(int const &sigma);
};
```

utils/bmpFile.cpp（主要部分）

```cpp
#include "bmpFile.h"

////////////其他操作///////////
///图片反向
void BmpFile::Invert() const {
    // 换成 int 一次 4 个字节
    for (int *pCurImg = reinterpret_cast<int *>(data), *pEndImg =
            pCurImg + infoHeader.image_size / sizeof(int) * sizeof(unsigned
char);
         pCurImg != pEndImg; ++pCurImg) {
        *pCurImg = ~*pCurImg;
    }
}

///灰度图像假彩色 这里用蓝色
bool BmpFile::gryPseudoColor() const {
    if (isRgb) {
        return false;
    }
    memset(palette, 0, paletteNum * paletteSize);
    for (unsigned i = 0; i < paletteNum; ++i) {
        palette[i].blue = i;
    }
    return true;
}

///rgb 图像转灰度图像
bool BmpFile::rgb2Gray() {
    if (!isRgb) {
        return false;
    }
    // char 表达的最大值 每一个存查找表
    const unsigned &size = char_express;
    auto *red = new double[size];
    auto *blue = new double[size];
    auto *green = new double[size];
    memset(red, 0, size << 3);
    memset(blue, 0, size << 3);
    memset(green, 0, size << 3);

    // 删除已经有的调色板 重新建立 256 色调色板
    delete[] this->palette;
```

```
    palette = new bitmap_palette[size];
    // 调色板数量 默认 256
    paletteNum = size;
    // 初始化 0 默认黑色
    memset(palette, 0, sizeof(bitmap_palette) << 8);
    for (unsigned i = 0; i < size; ++i) {
        // 查找表
        red[i] = 0.299 * i;
        green[i] = 0.587 * i;
        blue[i] = 0.114 * i;
        // 调色板 正常色彩 亮度
        palette[i].red = i;
        palette[i].blue = i;
        palette[i].green = i;
    }

    // 真实图片大小
    unsigned newImgSize = infoHeader.bitmap_width * infoHeader.bitmap_height;
    // 修正对齐
    unsigned img_size = newImgSize;
    unsigned tmpHeaderSize = fileHeaderSize + infoHeaderSize + paletteSize *
paletteNum;
    img_size += 4 - ((tmpHeaderSize + img_size) % 4);

    // 开始赋值
    auto *grayData = new unsigned char[img_size];
    memset(grayData, 0, img_size);

//    clock_t t_start = clock();
//    for (unsigned times = 0; times < 1000; ++times) {
    auto *pCurImg = data;
    //查表法 BGR 顺序
    for (auto *pGrayImg = grayData, *pEndGrayImg = pGrayImg + newImgSize;
pGrayImg < pEndGrayImg;) {
        *(pGrayImg++) = int(blue[*(pCurImg++)] + green[*(pCurImg++)] +
red[*(pCurImg++)]);
    }
    //非查表法
//        for (auto *pGrayImg = grayData, *pEndGrayImg = pGrayImg +
newImgSize; pGrayImg < pEndGrayImg;) {
//            *(pGrayImg++) = 0.114 * (*(pCurImg++)) + 0.587 * (*(pCurImg++))
+ 0.299 * (*(pCurImg++));
//        }
//    }
```

```
//    clock_t t_end = clock();
//    std::cout << (t_end - t_start);


    // 修正 data
    delete[] this->data;
    this->data = grayData;

    // 修正图片信息
    fileHeader.offset_bits = tmpHeaderSize;
    fileHeader.file_size = tmpHeaderSize + img_size;
    infoHeader.image_size = img_size;
    infoHeader.planes = 1;
    infoHeader.image_depth = 8;
    this->isRgb = false;

    // 释放临时内存
    delete[] red;
    delete[] blue;
    delete[] green;

    return true;
}

///均值方差规定化（针对于灰度图像）
bool BmpFile::MVR(const double &_bright, const double &_contrast) {
    if (this->isRgb) {
        return false;
    }

    // 获取亮度
    this->getBrightContrast();

    auto *LUT = new unsigned char[char_express];
    for (unsigned i = 0; i < char_express; ++i) {
        LUT[i] = fmax(0, fmin(255, (i - this->bright) * _contrast /
this->contrast + _bright));
    }

    for (auto *pCur = data, *pEnd = pCur + infoHeader.image_size; pCur <
pEnd; pCur++) {
        *pCur = LUT[*pCur];
    }
    delete[] LUT;
    return true;
```

```
}

/// 获取直方图
void BmpFile::getHistogram() {
    delete[] histogram;
    histogram = new unsigned[char_express];
    memset(histogram, 0, sizeof(unsigned) << 8);
    for (auto *pCur = data, *pEnd = pCur + infoHeader.image_size; pCur <
pEnd;) {
        histogram[*(pCur++)]++;
    }
}

/// 获取亮度对比度
void BmpFile::getBrightContrast() {
    if (histogram == nullptr) {
        this->getHistogram();
    }
    unsigned sum = 0;
    for (unsigned i = 0; i < char_express; ++i) {
        sum += histogram[i] * i;
    }
    bright = double(sum) / infoHeader.image_size;

    contrast = 0.0;
    for (unsigned i = 0; i < char_express; ++i) {
        contrast += histogram[i] * (i - bright) * (i - bright);
    }
    contrast = sqrt(contrast / (infoHeader.image_size - 1));
}

/// 直方图均衡化
bool BmpFile::histogramEqualization() {
    // 获取直方图
    if (histogram == nullptr) {
        this->getHistogram();
    }
    auto *LUT = new unsigned char[char_express];

    for (unsigned i = 0, sum = 0; i < char_express; ++i) {
        sum += histogram[i];
        LUT[i] = ((sum << 8) - sum) / infoHeader.image_size;
    }
```

```
    // 修改后
//    LUT[0] = ((histogram[0] << 8) - histogram[0]) / infoHeader.image_size;
//    for (unsigned i = 1; i < char_express; ++i) {
//        histogram[i] += histogram[i - 1];
//        LUT[i] = ((histogram[i] << 8) - histogram[i]) /
infoHeader.image_size;
//    }

    for (auto *pCur = data, *pEnd = pCur + infoHeader.image_size; pCur <
pEnd; pCur++) {
        *pCur = LUT[*pCur];
    }

    delete[] LUT;
    return true;
}

/// 加载一个 14 位 raw 图像
BmpFile *BmpFile::load14Raw(const std::string &path, unsigned int width,
unsigned int height) {
    std::ifstream infile(path, std::ios::in | std::ios::binary);
    if (!infile) {
        std::cerr << "Open file failed." << std::endl;
        throw;
    }
    auto size = width * height;
    auto *data = new short int[size];
    // 读取文件
    infile.seekg(0, std::ios::beg).read(reinterpret_cast<char *>(data), size
* sizeof(short int));
    infile.close();
    // 新建一个 bmp 文件
    auto *bmp = createGrayBmp(width, height);
    // 初始化 bmp 的数据
    bmp->data = new unsigned char[bmp->infoHeader.image_size];
    memset(bmp->data, 0, bmp->infoHeader.image_size);

    // 获取 14 位图 直方图
    unsigned short_int_express = 1 << 14;
    auto *histogram = new unsigned[short_int_express];
    memset(histogram, 0, sizeof(unsigned) << 14);
    for (auto *pCur = data, *pEnd = pCur + size; pCur < pEnd;) {
        histogram[*(pCur++)]++;
    }
```

```
    // 计算得到查找表 （直方图均衡化）
    auto *LUT = new unsigned char[short_int_express];
    for (unsigned i = 0, sum = 0; i < short_int_express; ++i) {
        sum += histogram[i];
        LUT[i] = ((sum << 8) - sum) / size;
    }

    // 将数据按照响应格式储存
    auto *resImg = new unsigned char[bmp->infoHeader.image_size];
    auto *pDesCur = resImg;
    for (auto *pCur = data, *pEnd = pCur + size; pCur < pEnd; pCur++) {
        *(pDesCur++) = LUT[*pCur];
    }
    bmp->writeBitData(width, height, resImg);

    delete[] resImg;
    delete[] data;
    delete[] histogram;
    delete[] LUT;

    return bmp;
}

///彩色灰度图像 浮点乘 到 整数乘 测试
bool BmpFile::rgb2Gray_test1() {
    if (!isRgb) {
        return false;
    }
    // char 表达的最大值 每一个存查找表
    const unsigned &size = char_express;

    // 删除已经有的调色板 重新建立 256 色调色板
    delete[] this->palette;
    palette = new bitmap_palette[size];
    // 调色板数量 默认 256
    paletteNum = size;
    // 初始化 0 默认黑色
    memset(palette, 0, sizeof(bitmap_palette) << 8);
    for (unsigned i = 0; i < size; ++i) {
        // 调色板 正常色彩 亮度
        palette[i].red = i;
        palette[i].blue = i;
        palette[i].green = i;
    }
```

```
    // 真实图片大小
    unsigned newImgSize = infoHeader.bitmap_width * infoHeader.bitmap_height;
    // 修正对齐
    unsigned img_size = newImgSize;
    unsigned tmpHeaderSize = fileHeaderSize + infoHeaderSize + paletteSize *
paletteNum;
    img_size += 4 - ((tmpHeaderSize + img_size) % 4);

    // 开始赋值
    auto *grayData = new unsigned char[img_size];
    memset(grayData, 0, img_size);

    clock_t t_start = clock();
    for (unsigned times = 0; times < 3000; ++times) {
        auto *pCurImg = data;
        //非查表法 1
        for (auto *pGrayImg = grayData, *pEndGrayImg = pGrayImg + newImgSize;
pGrayImg < pEndGrayImg;) {
            *(pGrayImg++) = 0.114 * (*(pCurImg++)) + 0.587 * (*(pCurImg++)) +
0.299 * (*(pCurImg++));
        }
        //非查表法 2
//        for (auto *pGrayImg = grayData, *pEndGrayImg = pGrayImg +
newImgSize; pGrayImg < pEndGrayImg;) {
//            *(pGrayImg++) = (116 * (*(pCurImg++)) + 601 * (*(pCurImg++)) +
306 * (*(pCurImg++))) >> 10;
//        }
    }
    clock_t t_end = clock();
    std::cout << (t_end - t_start) / 1000.0;


    // 修正 data
    delete[] this->data;
    this->data = grayData;

    // 修正图片信息
    fileHeader.offset_bits = tmpHeaderSize;
    fileHeader.file_size = tmpHeaderSize + img_size;
    infoHeader.image_size = img_size;
    infoHeader.planes = 1;
    infoHeader.image_depth = 8;
    this->isRgb = false;
```

```
    return true;
}

/// 均值滤波列积分
bool BmpFile::avrFilterBySumCol(unsigned M, unsigned N) {
    // 窗口 M 列 N 行
    unsigned half_x = M >> 1;
    unsigned half_y = N >> 1;
    M = (half_x << 1) | 0x0001;
    N = (half_y << 1) | 0x0001;
    unsigned C = (1 << 23) / (M * N);
    unsigned width = infoHeader.bitmap_width;
    unsigned height = infoHeader.bitmap_height;
    auto *sumCol = new unsigned[width];
    memset(sumCol, 0, width << 2);

    // 初始化 sumCol
    auto *pAdd = data, *pDel = data;
    for (auto *pEnd = data + N * width; pAdd < pEnd;) {
        for (unsigned x = 0; x < width; ++x) {
            sumCol[x] += *(pAdd++);
        }
    }
    auto *resData = new unsigned char[infoHeader.image_size];
    memcpy(resData, data, infoHeader.image_size);
    // 进行滤波 跳过边缘的处理
    for (auto *pCur = resData + half_y * width, *pEnd = resData + (height -
half_y) * width; pCur < pEnd;) {
        // 初始化 sum
        unsigned sum = 0;
        for (unsigned x = 0; x < M; ++x) {
            sum += sumCol[x];
        }
        pCur += half_x;
        for (unsigned x = half_x, end_x = width - half_x; x < end_x; ++x) {
            // 乘法换除法
            *(pCur++) = (sum * C) >> 23;
            // 更新 sum
            sum -= sumCol[x - half_x];
            sum += sumCol[x + half_x + 1];
        }
        pCur += half_x;
        for (unsigned x = 0; x < width; ++x) {
            sumCol[x] -= *(pDel++);
```

```
            sumCol[x] += *(pAdd++);
        }
    }
    delete[] data;
    data = resData;

    delete[] sumCol;

    return true;
}

///基于列积分的积分图实现
void BmpFile::getSumGryGraphBySumCol() {
    delete[] sumGraph;
    unsigned width = infoHeader.bitmap_width;
    auto *sumCol = new unsigned[width];
    memset(sumCol, 0, width << 2);
    sumGraph = new unsigned[infoHeader.image_size];
    auto *pRes = sumGraph;
    for (auto *pCur = data, *pEnd = data + infoHeader.image_size; pCur <
pEnd;) {
        sumCol[0] += *(pCur++);
        *(pRes++) = sumCol[0];
        for (unsigned x = 1; x < width; ++x, ++pRes) {
            sumCol[x] += *(pCur++);
            *pRes = *(pRes - 1) + sumCol[x];
        }
    }
    delete[] sumCol;
}

/// 用指令集获取积分图  有错误 还需要调试...
void BmpFile::getSumGryGraphBySSE() {
    delete[] sumGraph;
    sumGraph = new unsigned[infoHeader.image_size];
    unsigned width = infoHeader.bitmap_width;
    auto *sumCol = new unsigned[width];
    auto *pRes = sumGraph;
    memset(sumCol, 0, width << 2);
//   auto *p = std::align(1 << 8, width, reinterpret_cast<void *&>(sumCol),
reinterpret_cast<size_t &>(width));

    __m128i *pSumSSE, A;
```

```
    for (auto *pCur = data, *pEnd = data + infoHeader.image_size; pCur <
pEnd;) {
        sumCol[0] += *(pCur++);
        *(pRes++) = sumCol[0];
        for (unsigned x = 1; x <= 3; ++x, ++pRes) {
            sumCol[x] += *(pCur++);
            *pRes = *(pRes - 1) + sumCol[x];
        }
        pSumSSE = (__m128i *) (sumCol + 4);
        for (unsigned x = 4; x < width; pCur += 4) {
            A = _mm_cvtepi16_epi64(_mm_loadl_epi64((__m128i *) pCur));
            *(pSumSSE++) = _mm_add_epi32(*pSumSSE, A);
            for (unsigned i = 0; i < 4; ++i, ++pRes) {
                *(pRes) = *(pRes - 1) + sumCol[x++];
            }
        }
    }
    delete[] sumCol;
}

///均值滤波积分图
bool BmpFile::avrFilterByGraph(unsigned int M, unsigned int N) {
    if (sumGraph == nullptr) {
        this->getSumGryGraphBySumCol();
    }
    // 窗口 M 列 N 行
    unsigned half_x = M >> 1;
    unsigned half_y = N >> 1;
    M = (half_x << 1) | 0x0001;
    N = (half_y << 1) | 0x0001;
    unsigned C = (1 << 23) / (M * N);
    unsigned width = infoHeader.bitmap_width;
    unsigned height = infoHeader.bitmap_height;

    auto *resData = new unsigned char[infoHeader.image_size];
    memcpy(resData, data, infoHeader.image_size);
    // 进行滤波 跳过边缘的处理
    auto *pRes = resData + half_y * width;
    auto *pSmallSumGraph = sumGraph;
    auto *pBigSumGraph = sumGraph + N * width;

    // +1 是确保 算 sum 的公式为正确的（4 个值），否则应该为一个值或者两个值 但这样会少处理
一行一列像素
    for (unsigned y = half_y + 1, end_y = height - half_y;
```

```
         y < end_y; ++y, pSmallSumGraph += width, pBigSumGraph += width) {
        pRes += half_x + 1;
        unsigned sum = 0;
        for (unsigned x = half_x + 1, end_x = width - half_x, x1 = 0, x2 = M;
x < end_x; ++x, ++x1, ++x2) {
            sum = *(pBigSumGraph + x2) + *(pSmallSumGraph + x1) -
*(pBigSumGraph + x1) - *(pSmallSumGraph + x2);
            *(pRes++) = (sum * C) >> 23;
        }
        pRes += half_x;
    }

    delete[] data;
    data = resData;

    return true;
}

///图片反向 SSE
void BmpFile::InvertBySSE() const {
    // F1 高位会自动补 1
    __m128i F1 = _mm_set1_epi8(0xff);
    __m128i *pSSE = (__m128i *) data;
    for (auto *pCur = data, *pEnd = data + infoHeader.image_size; pCur <
pEnd; pCur += 16) {
        *(pSSE++) = _mm_xor_si128(*pSSE, F1);
    }
}

///中值滤波
double BmpFile::medianFilter(unsigned int M, unsigned int N) {
    // 窗口 M 列 N 行
    unsigned half_x = M >> 1;
    unsigned half_y = N >> 1;
    M = (half_x << 1) | 0x0001;
    N = (half_y << 1) | 0x0001;
    unsigned wSize = M * N;
    unsigned width = infoHeader.bitmap_width;
    unsigned height = infoHeader.bitmap_height;
    unsigned tmp_histogram[char_express];
    // 领域内像素总个数
    unsigned dbgCmpTimes = 0;
    auto *resData = new unsigned char[infoHeader.image_size];
    memcpy(resData, data, infoHeader.image_size);
```

```cpp
auto *pRes = resData + half_y * width;
for (unsigned y = half_y, end_y = height - half_y; y < end_y; ++y) {
    // 初始化直方图 每一行
    unsigned y1 = y - half_y;
    unsigned y2 = y + half_y;
    memset(tmp_histogram, 0, char_express << 2);
    auto *pCur = data + y1 * width;
    // 领域内的计算
    for (unsigned i = y1; i <= y2; ++i, pCur += width) {
        for (unsigned j = 0, end_j = (half_x << 1) + 1; j < end_j; ++j) {
            ++tmp_histogram[*(pCur + j)];
        }
    }
    // 计算中值
    unsigned num = 0;
    unsigned med = -1;
    for (unsigned i = 0; i < char_express; ++i) {
        num += tmp_histogram[i];
        if ((num << 1) > wSize) {
            med = i;
            break;
        }
    }
    // 进行滤波
    pRes += half_x;
    for (unsigned x = half_x, end_x = width - half_x; x < end_x; ++x) {
        *(pRes++) = med;
        // 更新直方图
        unsigned j1 = x - half_x;
        unsigned j2 = x + half_x + 1;
        pCur = data + y1 * width;
        // 对每一行
        for (unsigned i = y1; i <= y2; ++i, pCur += width) {
            unsigned v = *(pCur + j1);
            // 减左边
            --tmp_histogram[v];
            if (v <= med) {
                --num;
            }
            // 加右边
            v = *(pCur + j2);
            ++tmp_histogram[v];
            if (v <= med) {
                ++num;
```

```
                }
            }
            if ((num << 1) < wSize) {
                for (med++; med < char_express; ++med) {
                    dbgCmpTimes += 2;
                    num += tmp_histogram[med];
                    if ((num << 1) > wSize) { break; }
                }
                ++dbgCmpTimes;
            } else {
                while (((num - tmp_histogram[med]) << 1) > wSize) {
                    ++dbgCmpTimes;
                    num -= tmp_histogram[med];
                    --med;
                }
                dbgCmpTimes += 2;
            }
        }
        pRes += half_x;
    }
    delete[] data;
    data = resData;

    return double(dbgCmpTimes) / (width - (half_x << 1)) / (height - (half_y
<< 1));
}


double calTj(double const &j, double const &sigma) {
    double sigma_2 = sigma * sigma;
    return exp(-(j * j) * 0.5 / sigma_2) / sqrt(2 * M_PI * sigma_2);
}

bool BmpFile::GaussianFilter(int const &sigma) {
    int wSize = 3 * sigma + 1;
    auto *T = new double[wSize];
    auto *T_1 = new unsigned[wSize];
    auto *resData = new unsigned char[infoHeader.image_size];
    memcpy(resData, data, infoHeader.image_size);
    // 计算值
    for (int j = 0; j < wSize; ++j) {
        T[j] = calTj(j, sigma);
    }
    // 将值转为整数类型的
    unsigned M = 0;
```

```
    double minNum = T[wSize - 1];
    // 即余下的值 与 原来的值相比基本无变化
    while (1e6 * (minNum - int(minNum)) > minNum) {
        minNum *= 2;
        M++;
    }
    // 避免计算时溢出
    M = fmin(24, M);
    for (int j = 0; j < wSize; ++j) {
        T_1[j] = unsigned(T[j] * (1 << M));
    }
    // 开始计算
    for (auto *pCur = data + wSize, *pEnd = data + infoHeader.image_size -
wSize, *pRes = resData + wSize;
         pCur < pEnd; pCur += wSize << 1, pRes += wSize << 1) {
        auto *rowEnd = pCur + infoHeader.bitmap_width - (wSize << 1);
        while (pCur < rowEnd) {
            unsigned sum = (*pCur * T_1[0]) >> M;
            for (unsigned x = 1; x < wSize; ++x) {
                sum += (*(pCur + x) * T_1[x]) >> M;
                sum += (*(pCur - x) * T_1[x]) >> M;
            }
            *(pRes++) = sum;
            pCur++;
        }
    }
    // 转一下
    auto *resData2 = new unsigned char[infoHeader.image_size];
    for (unsigned y = 0; y < infoHeader.bitmap_height; ++y) {
        for (unsigned x = 0; x < infoHeader.bitmap_width; ++x) {
            *((resData2 + x * infoHeader.bitmap_height) + y) = *((resData + y
* infoHeader.bitmap_width) + x);
        }
    }
    // 重新计算 resData
    memcpy(resData, resData2, infoHeader.image_size);
    // 开始计算
    for (auto *pCur = resData + wSize, *pEnd = resData +
infoHeader.image_size - wSize, *pRes = resData2 + wSize;
         pCur < pEnd; pCur += wSize << 1, pRes += wSize << 1) {
        auto *rowEnd = pCur + infoHeader.bitmap_height - (wSize << 1);
        while (pCur < rowEnd) {
            unsigned sum = (*pCur * T_1[0]) >> M;
            for (unsigned x = 1; x < wSize; ++x) {
```

```
                sum += (*(pCur + x) * T_1[x]) >> M;
                sum += (*(pCur - x) * T_1[x]) >> M;
            }
            *(pRes++) = sum;
            pCur++;
        }
    }
    // 再转回去
    for (unsigned y = 0; y < infoHeader.bitmap_width; ++y) {
        for (unsigned x = 0; x < infoHeader.bitmap_height; ++x) {
            *((resData + x * infoHeader.bitmap_width) + y) = *((resData2 + y *
infoHeader.bitmap_height) + x);
        }
    }
    delete[] resData2;

    delete[] data;
    data = resData;

    delete[] T;
    delete[] T_1;
    return true;
}
```