专业：计算机科学与技术
姓名：秦嘉俊
学号：3210106182
日期：2022 年 11 月 12 日

# 浙江大学 实验报告

课程名称：图像信息处理 指导老师：宋明黎 成绩_____

实验名称：图像均值滤波和图像拉普拉斯变换增强

## 一、 实验目的和要求

1. 实现图像的均值滤波。
2. 利用拉普拉斯变换，实现图像的增强 (锐化)

## 二、 实验内容和原理

### 2.1 滤波

图像滤波 (Image Filter)，即在尽量保留图像细节特征的条件下对目标图像的噪声进行抑制，是图像预处理中不可缺少的操作，其处理效果的好坏将直接影响到后续图像处理和分析的有效性和可靠性。

图像的滤波就是放置一个大小为 $M \times N$ 的窗口，其中窗口中的元素对窗口中原始图像的相应像素进行操作，并将结果保存为新图像中的像素。滤波也被称作：遮罩 (mask)、内核 (kernel)、模板 (template) 等。滤波器中的元素是系数而不是像素值，它们表示应用于原始图像中像素的权重。

对于图像中的每个像素，根据滤波窗口中元素之间的定义关系计算滤波后对应的像素值。空间线性滤波将系数与对应像素之间相乘再求和来计算滤波后的像素值。

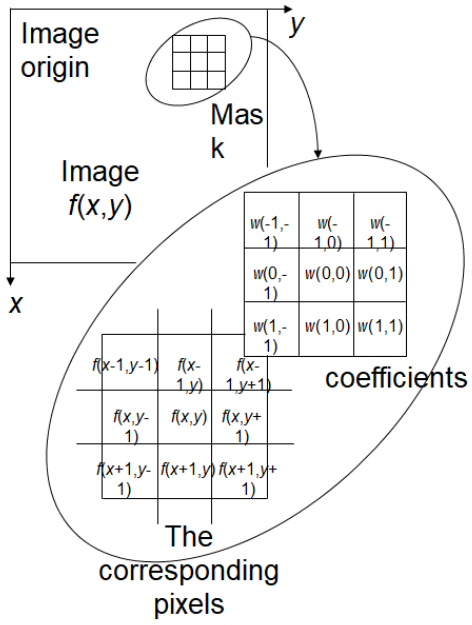



图 1: 滤波的过程

在上图中，The corresponding 代表滤波窗口对应的原图像像素值，coefficients 代表滤波的窗口中的系数。于是我们有这样的关系: $R = w(-1,-1)f(x-1,y-1) + w(-1,0)f(x-1,y) + \ldots + w(0,0)f(x,y) + \ldots + w(1,0)f(x+1,y) + w(1,1)f(x+1,y)$

这实质上是一种卷积操作，卷积表示为 $h(x) = f(x) * h(x) = \frac{1}{M} = \sum_{t=0}^{M-1} f(t)h(x-t)$

通常，掩模的长宽都为奇数。假设分别为 $2a+1$ 和 $2b+1$. 当窗口中心处于像素 $(x,y)$ 处时，新的像素值为：对图像 $f$ 中所有像素都与掩模进行运算之后，最终产生一幅新图像 $g$.

$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t)f(x+s,y+t)$ 即 $R = w_1z_1 + w_2z_2 + \cdots + w_{mn}z_{mn} = \sum_{i=1}^{m} w_iz_i$

当发现图像中的伪影 (Artifacts) 或噪点 (Noise) 过多时，滤波可以对图像进行平滑操作，以抑制噪点，减少伪影。但是，平滑操作会使图像变得模糊。

平滑可以减少噪声和模糊，这可用于预处理。例如，当我们只想提取大目标时，可以删除细微的细节。

空间滤波可以分为线性平滑滤波和统计排序滤波。

## 2.2 线性滤波

线性平滑滤波的输出是遮罩中像素的平均值。它也被称为均值滤波。均值滤波主要用于细微的细节去除，即去除比遮罩窗口还小的不需要的区域。

$$\frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

图 2: 均值滤波的遮罩窗口

简单的均值计算中，遮罩窗口中的像素对最终结果的贡献相同。加权平均，遮罩窗口中的像素对最终结果的贡献不均。不过均值滤波中，每个遮罩的系数都应该等于所有遮罩窗口内系数之和的倒数，以获得平均值。

因此，我们可以将均值滤波的公式写成：

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

其中，遮罩窗口的大小为 $(2a+1) \times (2b+1)$, $w$ 是小窗，$f$ 是输入的图像, $g$ 是输出的图像。

遮罩窗口的大小对于最终结果非常重要。当遮罩很小时，模糊效果非常细微，反之亦然。我们为了获得对目标的简要描述，使用均值滤波对图像进行模糊处理，以去除较小的对象，同时保留较大的对象。因此，遮罩的大小取决于要合并到背景中的对象。因此，遮罩窗口的大小取决于要合并到背景中的对象。

与均值滤波 (空间线性滤波) 相比，统计滤波是一种非线性空间滤波，其效果基于掩码窗口中像素值

的排序。中心像素的值取决于窗口中的排序结果。最流行的统计滤波是中值滤波。在本次试验中不再讲述。

## 2.3 应用拉普拉斯变换进行图像增强

锐化处理的主要目的是突出灰度的过渡部分。补偿轮廓，增强图像的边缘及灰度跳变的部分，使图像变得清晰，增强图像中的细节或锐化模糊部分。

图像锐化的用途多种多样，应用范围从电子印刷和医学成像到工业检测和军事系统的制导等。图像的模糊可以通过积分来实现（均值处理与积分类似）。而图像锐化，则需要微分来实现。

微分算子是一种锐化工具，其响应取决于相邻像素值之间的变化。微分算子增强了图像中的边缘和其他明显变化（包括噪声），并减弱了细微变化。基本上。微分算子的响应程度与图像在用算子操作的这一点的突变程度成正比，这样，图像微分增强边缘和其他突变（如噪声），而削弱灰度变化缓慢的区域。

使用一阶微分的图像增强被称为基于梯度的图像增强 (gradient based method) ，而使用二阶微分的图像增强被称为拉普拉斯算子 (Laplacian operator)。

对于一个整数值函数 $f(x)$ 来说，我们使用差分来表示微分算子：$\dfrac{\partial f}{\partial x} = f(x+1) - f(x)$

类似地可以定义二阶微分算子：$\dfrac{\partial^2 f}{\partial x^2} = f(x+1) + f(x-1) - 2f(x)$

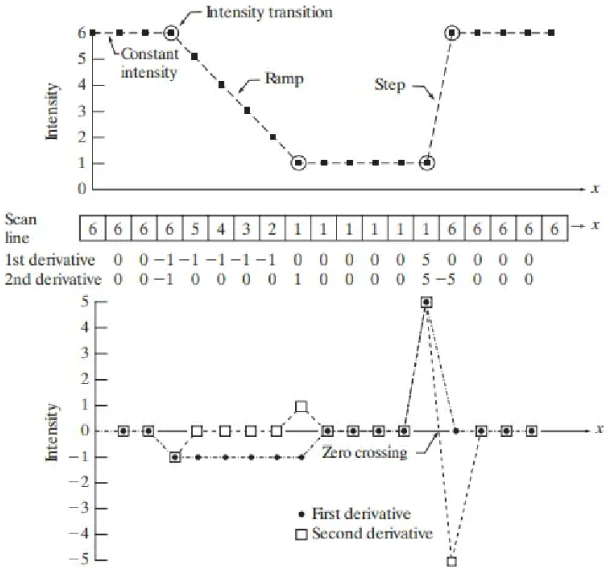



图 3: 一阶差分和二阶差分

对于一个二元函数 $f(x,y)$ 来说，我们首先定义一个二维的向量：

$$\nabla f = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix}$$

它的幅值 (Magnitude) 被表示为：

$$\nabla f = (G_x^2 + G_y^2)^{\frac{1}{2}} = \left[ (\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2 \right]^{\frac{1}{2}}$$

计算图像中所有像素的梯度非常耗时。因此，通常使用绝对值替换原始梯度幅值。

$$\nabla f \approx |G_x| + |G_y|$$

对函数 $f(x, y)$，拉普拉斯算子定义如下

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

可以证明，最简单的各向同性导数算子是拉普拉斯算子 (Rosenfeld and Kak [1982])。因为任何阶导数都是线性运算，所以拉普拉斯算子是线性算子。为了以离散形式表示该方程，我们使用之前的定义。

- 在 $x$ 方向，我们有 $\frac{\partial^2 f}{\partial x^2} = f(x+1, y) + f(x-1, y) - 2f(x, y)$
- 在 $y$ 方向，我们有 $\frac{\partial^2 f}{\partial y^2} = f(x, y+1) + f(x, y-1) - 2f(x, y)$

因此，从前面三个方程可以看出，两个变量的离散拉普拉斯函数是:

$$\nabla^2 f(x, y) = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

这个方程可以使用之前的滤波器掩码来实现，它给出了以 90° 为增量旋转的各向同性结果。实现机制与线性平滑滤波相同，只是在这里使用了不同的系数。

也可考虑对角线方向上的元素。得到的公式如下: $\nabla^2 f(x, y) = f(x-1, y-1) + f(x, y-1) + f(x+1, y-1) + f(x-1, y) f(x+1, y) + f(x-1, y+1) + f(x, y+1) + f(x+1, y+1) - 8f(x, y)$

或者也写作: $\nabla^2 f = \sum_{i=-1}^{1} \sum_{j=-1}^{1} f(x+i, y+j) - 9f(x, y)$

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | −4 | 1 | 1 | −8 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | −1 | 0 | −1 | −1 | −1 |
| −1 | 4 | −1 | −1 | 8 | −1 |
| 0 | −1 | 0 | −1 | −1 | −1 |

图 4: 拉普拉斯算子的遮罩窗口

上图显示了用于实现拉普拉斯算子的滤波掩码。此外，在 2.5（c）和（d）是从二阶导数的定义中得到的，二阶导数是前述公式的负数。这两个遮罩窗口会产生相同的结果，但在将拉普拉斯滤波图像与另一图像组合（通过加法或减法）时，必须记住符号的差异。

由于拉普拉斯算子是一种导数算子，它的使用会突出显示图像中的强度不连续性，而不强调具有缓慢变化的强度级别的区域。这将倾向于生成具有灰色边缘线和其他不连续性的图像，所有这些图像都叠加在

黑暗、无特征的背景上。只需将拉普拉斯图像添加到原始图像中，即可“恢复”背景特征，同时仍保持拉普拉斯图像的锐化效果。

也就是说，如果使用的定义具有负中心系数，则我们减去而不是添加拉普拉斯图像以获得锐化结果。因此，我们使用拉普拉斯函数进行图像锐化的基本方法是：

$$g(x,y) = f(x,y) + c\left[\nabla^2 f(x,y)\right]$$

其中 $f(x,y)$ 和 $g(x,y)$ 分别为输入的图像和锐化后的图像，也就是说：

$$g(x,y) = \begin{cases} f(x,y) - \nabla^2 f(x,y), If\ the\ center\ of\ the\ mask\ is\ negative \\ f(x,y) + \nabla^2 f(x,y), If\ the\ center\ of\ the\ mask\ is\ positive \end{cases}$$

我们将拉普拉斯变换后的图像与原始图像进行融合，就可以保持锐化效果，恢复原始视觉信息。

缩放拉普拉斯图像的典型方法是将其最小值相加，使新的最小值为零，然后将结果缩放到全局范围。公式如下:

$$f_m = f - min(f)$$
$$f_s = K(\frac{f_m}{\max(f_m)})$$

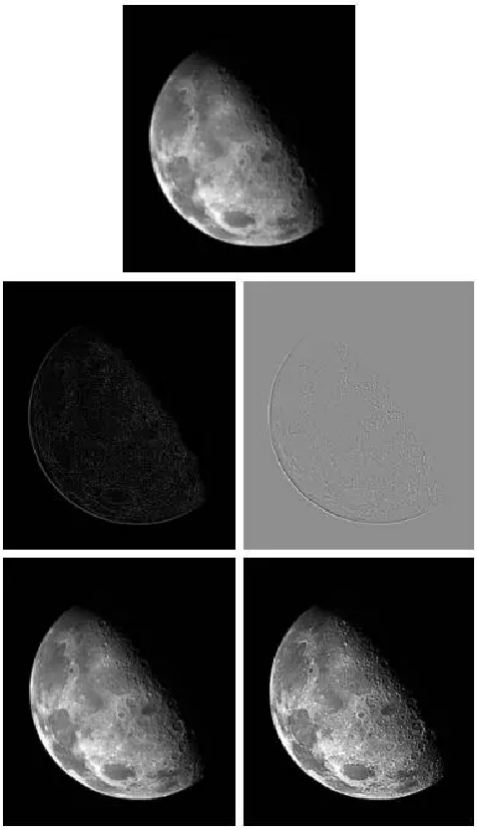


图 5: 利用拉普拉斯算子的图片锐化结果

如上图所示，进行这种变化后的图像的主要特征是边缘和尖锐的强度不连续。背景以前是黑色的，现在由于缩放而变成灰色。这种灰色外观是典型的拉普拉斯现象。显示了使用拉普拉斯算子最终获得的结果，该图像中的细节明显比原始图像中的更清晰。将原始图像添加到拉普拉斯算子恢复了图像中的整体强

度变化，拉普拉斯算子增加了强度不连续位置的对比度。最终结果是图像中的小细节得到了增强，背景色调得到了合理保留。

# 三、 实验步骤与分析

其中，图像信息头、图像文件头等结构体的定义，以及图像的输出基本同前，这里不再重复。本次实验我们的程序可以输入一张 24 位彩色 BMP 图像或者一张 8 位 BMP 灰度图像。

## 3.1 BMP 文件的读入

```
int main()
{
    BMPFILE a, b;
    FILE *fp;
    /* 读入 */
    fp = fopen("LN.bmp", "rb"); // rb 打开一个二进制文件
    if (!fp) {
        printf("BMP Image Not Found!\n");
        exit(0);
    }
    printf("Successfully open the image\n");
    fread(&(a.bmfh), sizeof(BITMAPFILEHEADER), 1, fp);
    fread(&(a.bmih), sizeof(BITMAPINFOHEADER), 1, fp);
    ImageHeight = a.bmih.biHeight;
    ImageWidth = a.bmih.biWidth;

    if(! a.bmih.biSizeImage) // 注意 biSizeImage 可能为 0 !
        a.bmih.biSizeImage = a.bmfh.bfSize - a.bmfh.OffBits;
    ImageSize = a.bmih.biSizeImage; // 所有像素所占的字节数.
    row_byte = (a.bmih.biBitCount / 8 * ImageWidth + 3) / 4 * 4; // 注意字节必须是 4 的整数倍
    if (a.bmih.biBitCount == 8) // 如果是 8 位图片的话，我们需要加上调色板
        for (int i = 0; i < 256; i++)
            a.aColors[i].rgbBlue = a.aColors[i].rgbGreen = a.aColors[i].rgbRed = i,
            b.aColors[i].rgbBlue = b.aColors[i].rgbGreen = b.aColors[i].rgbRed = i;
    a.aBitmapBits = (BYTE *)malloc(sizeof(BYTE) * ImageSize);
    fread(a.aBitmapBits, ImageSize * sizeof(BYTE), 1, fp);
    fclose(fp);
    /* 开始操作 */
    memcpy(&(b.bmfh), &(a.bmfh), sizeof(BITMAPFILEHEADER));
    memcpy(&(b.bmih), &(a.bmih), sizeof(BITMAPINFOHEADER));
    b.aBitmapBits = (BYTE *)malloc(sizeof(BYTE) * ImageSize);
    MeanFiltering(&a, &b, 21);
    Print(&b, "MeanFiltering.bmp");
    Laplace_Transform(&a, &b);
    Print(&b, "Laplace8.bmp");
    Laplace_Transform2(&a, &b);
    Print(&b, "Laplace4.bmp");
}
```

在 main 中我们的结构体 a 是读入的 BMP 文件，b 用来存储图像变换后的 BMP 文件。和以往略有不同的点在于，我们如果读入是灰度图，那么在图像读入时就为其加上调色板，以便后面操作。

读入结束后，我们依次对这张图象进行均值滤波，和两种拉普拉斯变换。

### 3.2 均值滤波函数 Mean_Filter

```
/*
函数返回(x,y) 对应第几个 bytes, 这里的形参 bytes 表示一行有多少字节, pixel 表示一个像素是 1/3 bytes
*/
int Get_Position(int x, int y, int bytes, int pixels)
{
    if (x < 0) x = 0; /* padding */
    if (y < 0) y = 0; /* padding */
    if (x >= ImageHeight) x = ImageHeight - 1; /* padding */
    if (y >= ImageWidth) y = ImageWidth - 1; /* padding */
    return x * bytes + y * pixels;
}
/* 均值滤波 对 a 操作, 结果存在 ans 中
我们的窗口是 scale * scale 大小的, 要求 scale 为奇数 */
void MeanFiltering(BMPFILE *a, BMPFILE *ans, int scale)
{
   int i, j, k1, k2;
   int length = scale / 2; /* 对于中心点为 (i,j) 的窗口,
   窗口范围是 i:[i-scale/2,i+scale/2] j:[j-scale/2,j+scale/2 */
   for (i = 0; i < ImageHeight; i++)
      for (j = 0; j < ImageWidth; j++)
         if (a->bmih.biBitCount == 24) { /* 判断是否为彩色图 */
         double sumR = 0, sumG = 0, sumB = 0;
         for (k1 = i - length; k1 <= i + length; k1++)
            for (k2 = j - length; k2 <= j + length; k2++) {
               int now = Get_Position(k1, k2, row_byte, 3); /* 彩色图就 同时对三个通道均值滤波 */
               sumB += a->aBitmapBits[now];
               sumG += a->aBitmapBits[now+1];
               sumR += a->aBitmapBits[now+2];
            }
         int now = Get_Position(i, j, row_byte, 3);
         ans->aBitmapBits[now] = sumB / (scale * scale);
         ans->aBitmapBits[now+1] = sumG / (scale * scale);
         ans->aBitmapBits[now+2] = sumR / (scale * scale);
      }
      else {
         int sum = 0;
         for (k1 = i - length; k1 <= i + length; k1++)
            for (k2 = j - length; k2 <= j + length; k2++) {
               int now = Get_Position(k1, k2, row_byte, 1); /* 灰度图只需要对一个通道均值滤波 */
               sum += a->aBitmapBits[now];
            }
         int now = Get_Position(i, j, row_byte, 1);
         ans->aBitmapBits[now] = sum / (scale * scale);
      }
}
```

在图片边界进行均值滤波时，我们应该做出相应的调整。调整的方法有以下几种：将图片四周扩大一圈，也就是 padding，padding 是增加各个边的像素的数量，目的是保持我们生成的图片不要边小，与原图保持一致，增加的像素数值与边界像素保持一致；另一种思路就是舍去边界像素点，保证遮罩窗口的边

界不超过图片边界即可；我这里采用的思路就是向内取值，也就是说，如果窗口的边界超过了图片边界，我们就用最近的不发生越界的窗口代替。例如，假设 scale(遮罩窗口从中心向四周拓展的长度) 取 1，那么遮罩窗口长度就是 3，在 (0,0) 这个点的均值滤波结果，我们实际上取的是 $[0,3]\times[0,3]$ 这个范围的均值。

为了实现 padding 的思想，我们修改了 `Get_Position` 的代码，如果目前要计算的坐标 $(x,y)$ 超过了原有图像的边界，我们就对坐标进行自动调整，以便计算。

具体说明详见代码及代码注释。

### 3.3 图像锐化函数 Laplace_Transform

```
int Update(int x)
{
    if (x < 0) return 0;
    if (x > 255) return 255;
    return x;
}
/* 第一种拉普拉斯变换
1 1 1
1 -8 1
1 1 1 */
void Laplace_Transform(BMPFILE *a, BMPFILE *ans)
{
    int i, j, k1, k2;
    for (i = 0; i < ImageHeight; i++)
        for (j = 0; j < ImageWidth; j++)
            if (a->bmih.biBitCount == 24) { /* 判断是否为彩色图 */
            int sumR = 0, sumG = 0, sumB = 0;
            for (k1 = i - 1; k1 <= i + 1; k1++)
                for (k2 = j - 1; k2 <= j + 1; k2++) {
                    int now = Get_Position(k1, k2, row_byte, 3);
                    sumB += a->aBitmapBits[now];
                    sumG += a->aBitmapBits[now+1];
                    sumR += a->aBitmapBits[now+2];
                }
            int now = Get_Position(i, j, row_byte, 3);
            ans->aBitmapBits[now] = Update(10 * a->aBitmapBits[now] - sumB); /* 这里直接把原图加上，需要update */
            ans->aBitmapBits[now+1] = Update(10 * a->aBitmapBits[now+1] - sumG);
            ans->aBitmapBits[now+2] = Update(10 * a->aBitmapBits[now+2] - sumR);
        }
        else {
            int sum = 0;
            for (k1 = i - 1; k1 <= i + 1; k1++)
                for (k2 = j - 1; k2 <= j + 1; k2++) {
                    int now = Get_Position(k1, k2, row_byte, 1);
                    sum += a->aBitmapBits[now];
                }
            int now = Get_Position(i, j, row_byte, 1);
            ans->aBitmapBits[now] = Update(10 * a->aBitmapBits[now] - sum);
        }
}
/* 第二种拉普拉斯变换
0 1 0
1 -4 1
```

```
   0 1 0 */
   void Laplace_Transform2(BMPFILE *a, BMPFILE *ans)
   {
      int i, j, k;
      int dx[5] = {-1, 0, 0, 0, 1};
      int dy[5] = {0, -1, 0, 1, 0}; /* (i+dx[k], j+dx[k]) 可以遍历五个位置*/
      for (i = 0; i < ImageHeight; i++)
         for (j = 0; j < ImageWidth; j++)
            if (a->bmih.biBitCount == 24) { /* 判断是否为彩色图 */
            int sumR = 0, sumG = 0, sumB = 0;
            for (k = 0; k <= 4; k++) {
               int now = Get_Position(i + dx[k], j + dy[k], row_byte, 3);
               sumB += a->aBitmapBits[now];
               sumG += a->aBitmapBits[now+1];
               sumR += a->aBitmapBits[now+2];
            }
            int now = Get_Position(i, j, row_byte, 3);
            ans->aBitmapBits[now] = Update(6 * a->aBitmapBits[now] - sumB); /* 这里直接把原图加上，需要update */
            ans->aBitmapBits[now+1] = Update(6 * a->aBitmapBits[now+1] - sumG);
            ans->aBitmapBits[now+2] = Update(6 * a->aBitmapBits[now+2] - sumR);
         }
         else {
            int sum = 0;
            for (k = 0; k <= 4; k++) {
               int now = Get_Position(i + dx[k], j + dy[k], row_byte, 1);
               sum += a->aBitmapBits[now];
            }
            int now = Get_Position(i, j, row_byte, 1);
            ans->aBitmapBits[now] = Update(6 * a->aBitmapBits[now] - sum);
         }
   }
```

我们这里实现了两种遮罩窗口，`Laplace_Transform` 实现的是 $\begin{bmatrix}1 & 1 & 1\\ 1 & -8 & 1\\ 1 & 1 & 1\end{bmatrix}$，其拉普拉斯算子为

$$\nabla^2 f(x,y) = \sum_{i=-1}^{1}\sum_{j=-1}^{1} f(x+i,y+j) - 9f(x,y)$$

加上原图后得到

$$g(x,y) = f(x,y) - \nabla^2 f(x,y) = 10f(x,y) - \sum_{i=-1}^{1}\sum_{j=-1}^{1} f(x+i,y+j)$$

. 因此我们这里直接利用化简后的形式。需要注意的是这里运算中可能超过 BYTE 的范围，因此我们用 `int` 计算，最后通过 `Update` 函数将像素 rearrange 到 $[0,255]$ 这个区间上。

`Laplace_Transform2` 类似，实现的是 $\begin{bmatrix}0 & 1 & 0\\ 1 & -4 & 1\\ 0 & 1 & 0\end{bmatrix}$.

具体说明详见代码及代码注释。

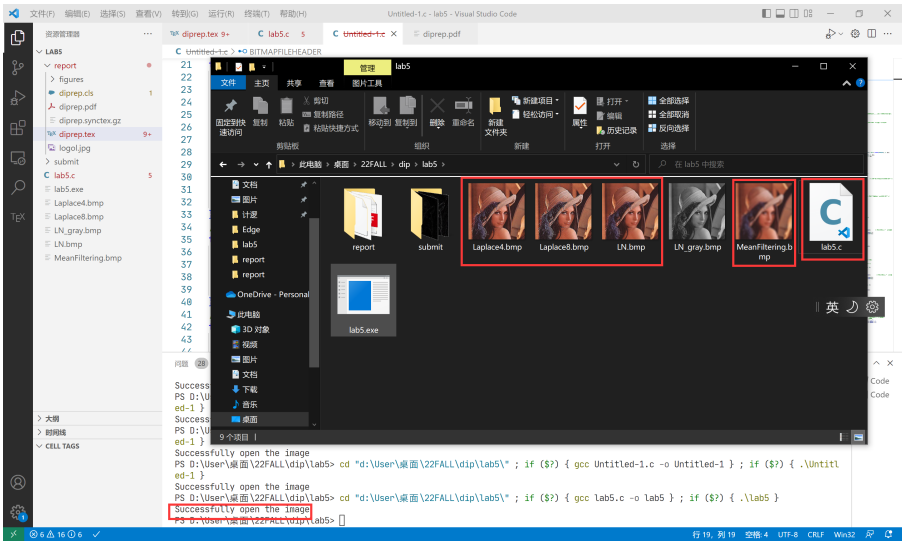



图 6: 运行程序

# 四、 实验环境及运行方法

## 4.1 实验环境

Windows 10 系统

gcc 10.3.0 (tdm64-1) x86_64-w64-mingw32

## 4.2 运行方法

源文件为 lab5.c, 在源文件里有我们的样例输入, 一张 24 位彩色 BMP 图像文件 (LN.bmp)(或者一张 8 位灰度图像 LN_gray.bmp), 使用 VSCode 打开这个文件夹，并选中 lab5.c 点击 Run Code 即可开始运行。当终端出现“Successfully open the image”时说明我们成功打开了图像，否则会输出“BMP Image Not Found!”. 成功运行后会得到 3 个新图像，MeanFiltering.bmp，是均值滤波的结果。Laplace8，是拉普拉斯变换进行图像锐化的结果，其中，选取的窗口为 $\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

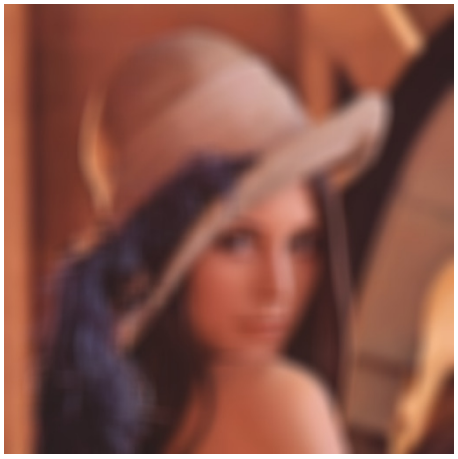
Laplace4，是拉普拉斯变换进行图像锐化的结果，选取的窗口为 $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$.

# 五、 实验结果展示

我们依然用经典的莱纳图作为测试用例

(a) 输入图像 (LN.bmp)

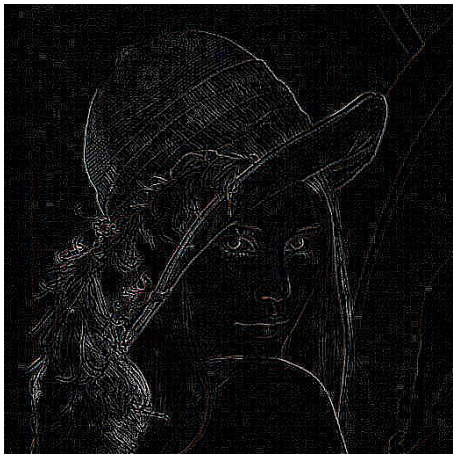
(b) 窗口大小为 3 的均值滤波 (MeanFiltering_LN3.bmp)



(c) 窗口大小为 21 的均值滤波 (MeanFiltering_LN21.bmp)

(d) 拉普拉斯变换 1 (Laplace_LN8.bmp)

(e) 拉普拉斯变换 2 (Laplace_LN4.bmp)



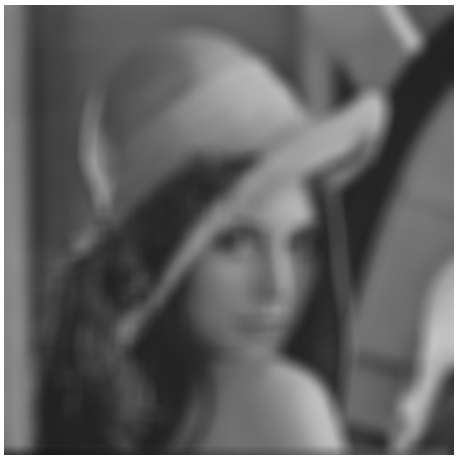
(f) 拉普拉斯变换中间图像 1 (Laplaceop8.bmp)

(g) 拉普拉斯变换中间图像 2 (Laplaceop4.bmp)

输入图像样例 (8 位灰度 Lena 图):

(h) 输入图像 (LN_gray.bmp)

(i) 窗口大小为 3 的均值滤波 (MeanFiltering_LNgray3.bmp)

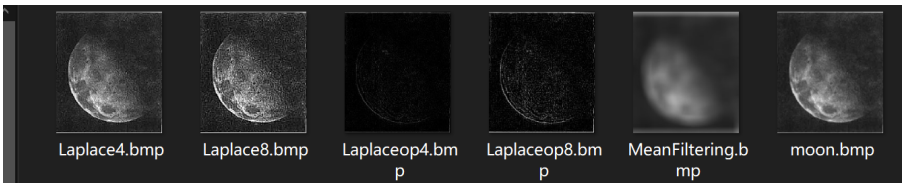


(j) 窗口大小为 21 的均值滤波 (MeanFiltering_LNgray21.bmp)

(k) 拉普拉斯变换 1
(Laplace_LNgray8.bmp)

(l) 拉普拉斯变换 2
(Laplace_LNgray4.bmp)

此外我们还测试了一个样例




# 六、 心得体会

如果不进行放缩，进行拉普拉斯变换的直接输出结果会有十分密集的像素点，所以才需要进行放缩的操作，放缩指 Update，将结果映射到 [0,255]，这样才能得到合适的拉普拉斯变换后的结果。

这是因为拉普拉斯变换的结果有正有负，在 unsigned char 中负值其实是当做加上 255 后的正值。对于灰度图来说直接进行输出的话负值接近白色，正值接近黑色，因此才有了上图这样的效果。进行放缩处理后才有了第 5 部分中输出图像的结果。

如果进行拉普拉斯变换的图像增强的话，则完全不需要放缩处理，直接将变换结果与原图的图像数据进行 fuse 即可。要考虑溢出时的情形。fuse 的结果可能超过 unsigned char 的范围，即 [0,255]。溢出会造成图像出现斑点：

不过总体来说，本次作业并不难，只是进行像素数值的运算。