浙江大学实验报告

专业：计算机科学与技术
姓名：NoughtQ
学号：1145141919810
日期：2024 年 11 月 29 日

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

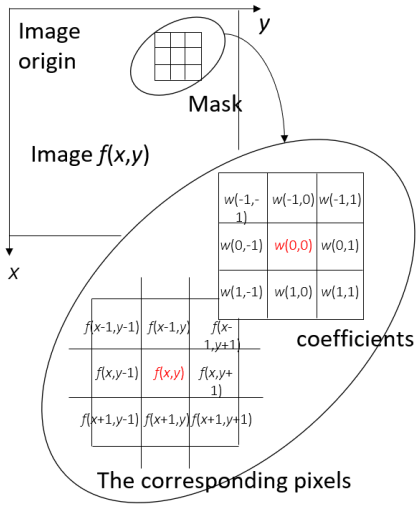
1. 实现图像均值滤波
2. 通过拉普拉斯变换，实现图像增强

二、 实验内容和原理

2.1 图像滤波

滤波器(filter)是一个大小为$M \times N$的窗口，窗口中的元素与原图像的处于窗口内的像素进行某种运算，结果作为新图像的一个像素。当窗口滑过原图像并完成上述运算之后，就能够得到一幅新图像。

- 滤波器的别名：掩模(mask)、核(kernel)、模板(template)，窗口(window)
- 滤波器子图像中的值是系数值，而不是像素值，它代表了影响新像素产生的权重
- 滤波器的执行流程：在待处理图像中逐点移动**掩模**(mask)，在每一点(x, y)处，滤波器在该点的响应通过实现定义的关系来计算。
  - 对于线性空间滤波，其响应由滤波器系数与滤波掩模扫过区域的对应像素值的乘积之和给出



响应值$R = w(-1,-1)f(x-1,y-1) + w(-1,0)f(x-1,y) + ... + w(0,0)f(x,y) + ... + w(1,0)f(x+1,y) + w(1,1)f(x+1,y+1)$

- 滤波器的响应值：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：

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

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

- 简化形式为：$R = \sum_{i=1}^{mn} w_i z_i$

图像在传输过程中，由于传输信道、采样系统质量较差，或受各种干扰的影响，而造成图像毛糙，此时，就需对图像进行平滑处理。平滑可以抑制高频成分，但也使图像变得模糊。

平滑空间滤波器(spatial smoothing filter)用于模糊处理和减少噪声。模糊处理经常用于预处理，例如，在提取大的目标之前去除图像中一些琐碎的细节，桥接直线或曲线的缝隙。

2.2 图像均值滤波

平滑线性空间滤波器的输出是包含在滤波掩模邻域内像素的简单平均值。因此，这些滤波器也称为**均值滤波器**。均值滤波器的主要应用是去除图像中的不相干细节，即那些与滤波掩模尺寸相比更小的像素区域。

- 简单平均，表示窗口中每一个像素对响应的贡献是一样的
- 加权平均，表示窗口中的像素对相应的贡献有大小之分
- 两个3×3平滑（均值）滤波器掩模，每个掩模前边的乘数等于它的系数值的和，以计算平均值

$$\frac{1}{9}\times\begin{array}{|c|c|c|}\hline 1&1&1\\\hline 1&1&1\\\hline 1&1&1\\\hline\end{array}\qquad \frac{1}{16}\times\begin{array}{|c|c|c|}\hline 1&2&1\\\hline 2&4&2\\\hline 1&2&1\\\hline\end{array}$$

它的方程为：

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

其中，滤波器大小为$(2a+1)\times(2b+1)$，w为滤波器，f为输入图像，g为输出图像。

滤波掩模的大小与图像的平滑效果有直接的关系。当掩模比较小时，可以观察到在整幅图像中有轻微的模糊，当掩模大小增加，模糊程度也随之增加。

2.3 图像锐化滤波

- **锐化滤波器**(sharpening filter)的作用：突出图像中的细节或者增强被模糊了的细节。
- **微分算子**(differential operator)是实现锐化的工具，其响应程度与图像在该点处的突变程度有关。微分算子增强了边缘和其他突变（如噪声）并削弱了灰度变化缓慢的区域。
  - 基于二阶微分的图像增强——**拉普拉斯算子**(Laplacian operator)
  - 基于一阶微分的图像增强——**梯度法**(gradient-based method)
- 对于函数$f(x)$，我们用差分(difference)来表示微分算子：$\frac{\partial f}{\partial x} = f(x+1) - f(x)$
  - 类似地，二阶差分为：$\frac{\partial^2 f}{\partial x^2} = f(x+1) + f(x-1) - 2f(x)$

2.3.1 梯度法增强

对于二元函数$f(x,y)$，定义一个二维向量：

$$\nabla f = \begin{pmatrix} G_x \\ G_y \end{pmatrix} = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

它的模值(magnitude)为：

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

当对整幅图像计算梯度时，运算量会很大，因此，在实际操作中，常用绝对值代替平方与平方根运算近似求梯度的模值：$\nabla f \approx |G_x| + |G_y|$

2.3.2 拉普拉斯变换增强

对于二元函数$f(x,y)$，拉普拉斯算子为：

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

- 沿 x 方向的二阶偏微分为：$\frac{\partial^2 f}{\partial x^2} = f(x+1,y) + f(x-1,y) - 2f(x,y)$
- 沿 y 方向的二阶偏微分为：$\frac{\partial^2 f}{\partial y^2} = f(x,y+1) + f(x,y-1) - 2f(x,y)$
- 因此，离散形式的拉普拉斯算子为：

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

0	1	0
1	-4	1
0	1	0

可以看到，这类拉普拉斯算子是旋转不变的，具有各向同性。

· 对角线方向上的元素也可以考虑进来，这样就扩展了掩膜的设计：

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

1	1	1
1	-8	1
1	1	1

此类拉普拉斯算子也是旋转不变的，具有各向同性。

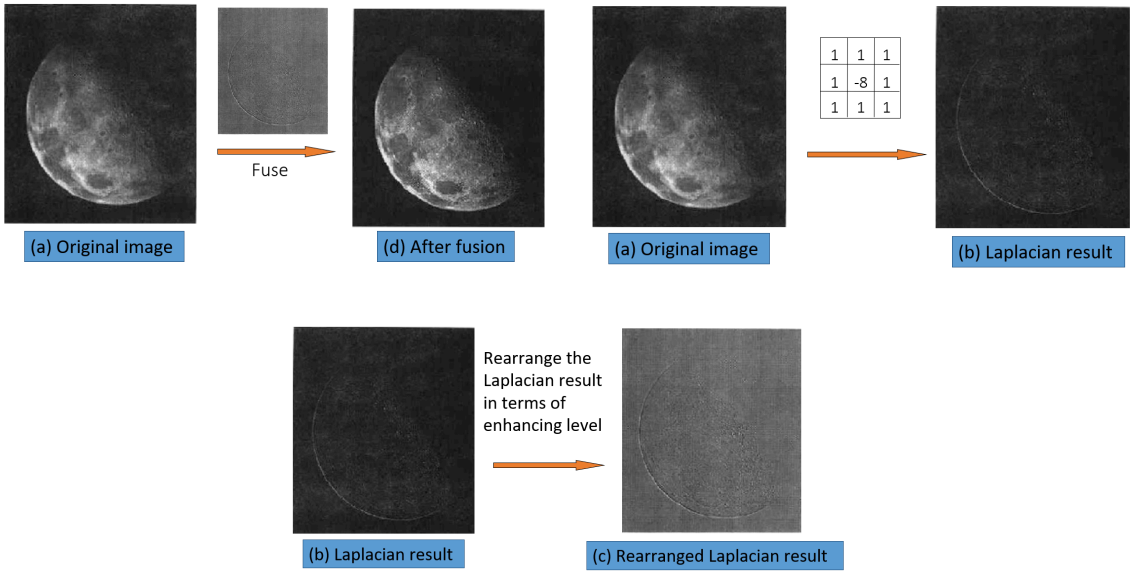
· 当拉普拉斯滤波后的图像与其它图像合并时（相加或相减），则必须考虑符号上的差别

应用

基于拉普拉斯算子的图像增强：

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

将原始图像和拉普拉斯图像叠加在一起的简单方法可以保护拉普拉斯锐化处理的效果，同时又能复原背景信息。



三、 实验步骤与分析

3.1 图像均值滤波

下面给出与图像均值滤波相关的代码：

```
// 均值滤波操作
BMPFILE MeanFilter(BMPFILE bf) {
    BMPFILE newImg;
    BYTE val;
    LONG width, height;
    int x, y, i, j, tmpX, tmpY;
    int pos, halfLen;
    double sumR, sumG, sumB;

    // 分配空间
    newImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));

    // 将旧图的数据拷贝到新图上
```

```
    memcpy(&(newImg->bmfh), &(bf->bmfh), sizeof(BITMAPFILEHEADER));
    memcpy(&(newImg->bmih), &(bf->bmih), sizeof(BITMAPINFOHEADER));
    newImg->aBitmapBits =
            (BYTE *)malloc(sizeof(BYTE) * newImg->bmih.biSizeImage);
    width = newImg->bmih.biWidth;
    height = newImg->bmih.biHeight;
    halfLen = FILTERWINDOWLEN / 2;

    // 均值滤波
    for (y = 0; y < bf->bmih.biHeight; y++)
        for (x = 0; x < bf->bmih.biWidth; x++) {
            sumR = sumG = sumB = 0;
            for (i = -halfLen; i <= halfLen; i++)
                for (j = -halfLen; j <= halfLen; j++) {
                    tmpY = y + i < 0 ?
                        0 : (y + i >= height ? height - 1 : y + i);
                    tmpX = x + j < 0 ?
                          0 : (x + j >= width ? width - 1 : x + j);
                    pos = tmpY * ColorBitWidth + tmpX * 3;
                    sumB += bf->aBitmapBits[pos];
                    sumG += bf->aBitmapBits[pos + 1];
                    sumR += bf->aBitmapBits[pos + 2];
                }

            pos = y * ColorBitWidth + x * 3;
                  newImg->aBitmapBits[pos] = rearrangeComp(sumB /
(FILTERWINDOWLEN * FILTERWINDOWLEN));
                newImg->aBitmapBits[pos + 1] = rearrangeComp(sumG /
(FILTERWINDOWLEN * FILTERWINDOWLEN));
                newImg->aBitmapBits[pos + 2] = rearrangeComp(sumR /
(FILTERWINDOWLEN * FILTERWINDOWLEN));
        }

    return newImg;
}
```

这里展开介绍一下均值滤波的处理细节：

- 对图像的每个像素点进行滤波处理时，由于是根据以该像素点为中心的窗口来进行像素值的调整，因此对于边上的像素，会出现窗口越界问题。我采取的操作是：对于越界坐标，将其“拉回”至边界处，用边界上的像素值替代。
  - 还有一种可行的方法是尝试给图像周围填充一圈像素，这样就无需对边界做特殊处理，但由于时间原因，我并没有进行试验。

- 为了确保计算均值的精度，我将 sumB、sumG、sumR 设为 double 类型。
- 窗口的边长为 FILTERWINDOWLEN，是一个常数，可以在 bmp.h 中修改它的值。

3.2 图像拉普拉斯变换增强

下面给出与图像拉普拉斯变换增强相关的代码：

```
// 询问拉普拉斯算子类型
void askLapFilterType(void) {
     printf("The program provides two types of available Laplacian
operators:\n");
    printf("1)\n0  1  0\n1 -4  1\n0  1  0\n");
    printf("2)\n1  1  1\n1 -8  1\n1  1  1\n");
    printf("Please choose one of the operators(input 1 or 2): ");
    scanf("%d", &LapFilterChoice);

    if (LapFilterChoice < 1 || LapFilterChoice > 2) {
        printf("Invalid choice, try again!\n");
        exit(1);
    } else {
        printf("Valid choice!\n\n");
    }
}

// 拉普拉斯变换增强（锐化滤波）操作
BMPFILE LaplacianFilter(BMPFILE bf) {
    BMPFILE newImg;
    BYTE val;
    LONG width, height;
    int x, y, i, j, tmpX, tmpY;
    int pos, halfLen, choice, factor;
    double sumR, sumG, sumB;

    // 分配空间
    newImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));

    // 将旧图的数据拷贝到新图上
    memcpy(&(newImg->bmfh), &(bf->bmfh), sizeof(BITMAPFILEHEADER));
    memcpy(&(newImg->bmih), &(bf->bmih), sizeof(BITMAPINFOHEADER));
    newImg->aBitmapBits =
            (BYTE *)malloc(sizeof(BYTE) * newImg->bmih.biSizeImage);
    width = newImg->bmih.biWidth;
    height = newImg->bmih.biHeight;
    halfLen = FILTERWINDOWLEN / 2;
```

```
    askLapFilterType();

    // 拉普拉斯变换（锐化滤波）
    // 拉普拉斯算子:
    // 0  1  0
    // 1 -4  1
    // 0  1  0
    if (LapFilterChoice == 1) {
        for (y = 0; y < bf->bmih.biHeight; y++)
            for (x = 0; x < bf->bmih.biWidth; x++) {
                sumR = sumG = sumB = 0;
                for (i = -halfLen; i <= halfLen; i++)
                    for (j = -halfLen; j <= halfLen; j++) {
                        if (abs(i) + abs(j) == 2)
                            continue;
                        tmpY = y + i < 0 ?
                          0 : (y + i >= height ?
                                              height - 1 : y + i);
                        tmpX = x + j < 0 ?
                          0 : (x + j >= width ?
                                              width - 1 : x + j);
                        pos = tmpY * ColorBitWidth + tmpX * 3;
                        factor = tmpX | tmpY ?  1 : -4;
                        sumB += bf->aBitmapBits[pos] * factor;
                        sumG += bf->aBitmapBits[pos + 1] * factor;
                        sumR += bf->aBitmapBits[pos + 2] * factor;
                    }

                pos = y * ColorBitWidth + x * 3;
                newImg->aBitmapBits[pos] =
                    rearrangeComp(bf->aBitmapBits[pos] *
                                  (LAPWINDOWSIZE1 + 1) - sumB);
                newImg->aBitmapBits[pos + 1] =
                    rearrangeComp(bf->aBitmapBits[pos + 1] *
                                  (LAPWINDOWSIZE1 + 1) - sumG);
                newImg->aBitmapBits[pos + 2] =
                    rearrangeComp(bf->aBitmapBits[pos + 2] *
                                  (LAPWINDOWSIZE1 + 1) - sumR);
            }
    // 拉普拉斯算子:
    // 1  1  1
```

```
    // 1 -8  1
    // 1  1  1
    } else if (LapFilterChoice == 2) {
        for (y = 0; y < bf->bmih.biHeight; y++)
            for (x = 0; x < bf->bmih.biWidth; x++) {
                sumR = sumG = sumB = 0;
                for (i = -halfLen; i <= halfLen; i++)
                    for (j = -halfLen; j <= halfLen; j++) {
                        tmpY = y + i < 0 ?
                          0 : (y + i >= height ?
                                              height - 1 : y + i);
                        tmpX = x + j < 0 ?
                          0 : (x + j >= width ?
                                              width - 1 : x + j);
                        pos = tmpY * ColorBitWidth + tmpX * 3;
                        factor = tmpX | tmpY ?  1 : -8;
                        sumB += bf->aBitmapBits[pos] * factor;
                        sumG += bf->aBitmapBits[pos + 1] * factor;
                        sumR += bf->aBitmapBits[pos + 2] * factor;
                    }

                pos = y * ColorBitWidth + x * 3;
                newImg->aBitmapBits[pos] =
                  rearrangeComp(bf->aBitmapBits[pos] *
                  ((FILTERWINDOWLEN * FILTERWINDOWLEN) + 1) - sumB);
                newImg->aBitmapBits[pos + 1] =
                  rearrangeComp(bf->aBitmapBits[pos + 1] *
                  ((FILTERWINDOWLEN * FILTERWINDOWLEN) + 1) - sumG);
                newImg->aBitmapBits[pos + 2] =
                  rearrangeComp(bf->aBitmapBits[pos + 2] *
                  ((FILTERWINDOWLEN * FILTERWINDOWLEN) + 1) - sumR);
            }
    } else {
        printf("Invalid choice, try again!\n");
        exit(1);
    }

    return newImg;
}
```

这里展开介绍一下均值滤波的处理细节：

- 这里采用与均值滤波相同的方法来处理窗口越界问题。

- 程序提供了两种课上讲到过的拉普拉斯算子，分别是：

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

  左边的拉普拉斯算子只考虑了 x 方向和 y 方向的像素点，而右边的拉普拉斯算子则在左边的基础上将对角线上的像素点也考虑进来了。

- 以左边的拉普拉斯算子为例，根据**“实验内容和原理”**一节中给出的公式：

$$\nabla^2 f = \sum_{i=-1}^{1} \sum_{j=-1}^{1} f(x+i, y+j) - 5f(x,y) \tag{1}$$

$$g(x,y) = f(x,y) - \nabla^2 f(x,y) \tag{2}$$

  (1)式中的系数 5 指的是拉普拉斯算子中非零元素的个数。

  将(1)(2)式联立，可以得到：

$$g(x,y) = 6f(x,y) - \sum_{i=-1}^{1} \sum_{j=-1}^{1} f(x+i, y+j) \tag{3}$$

  所以程序在计算拉普拉斯变换的时候，无论采用何种算子，都会直接使用形如(3)式的这一化简形式。

- 程序里还有一个直接生成拉普拉斯变换中间图像的函数，但由于实现方式与上述函数类似，因此这里不再给出，唯一的不同在于这里直接将拉普拉斯算子作为像素值。

3.3 主程序

主程序结构较为简单，故不再解释。但需要注意的是：最后被我注释掉的那几行语句仅供测试使用，正常情况下用户不应执行这些语句。如果想要执行这些语句的话，需要先注释掉从 `choice = ...` 到 `WriteBMPFile(...);` 之间的语句，然后去掉原先被注释掉的那几行的注释符号，最后重新编译和运行，可以得到拉普拉斯变换的中间图像。

```
#include "bmp.h"
#include <stdio.h>
#include <stdlib.h>

int main() {
    int choice;
    BMPFILE oldImg, newImg;

    // 分配存储空间
    oldImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));
    newImg = (BMPFILE)malloc(sizeof(struct tagBMPFILESTRUCT));
```

```
    // 读取原图
    oldImg = ReadBMPFile();

    // 询问选择
    choice = AskforChoicev5();

    // 根据选择执行不同的简单几何变换
    switch (choice) {
        case 15: newImg = MeanFilter(oldImg); break;
        case 16: newImg = LaplacianFilter(oldImg); break;
        default: printf("Invalid choice!\n"); exit(1);
    }

    // 得到新图
    WriteBMPFile(newImg, choice);

    // 直接生成拉普拉斯变换的中间图像
    // newImg = LaplacianFilterMid(oldImg);
    // WriteBMPFile(newImg, 161);

    return 0;
}
```

四、 实验环境及运行方法

4.1 实验环境

我的开发环境如下:

- Windows 11 24H2
- gcc version 14.2.0
- GNU Make 4.4.1

4.2 运行方法

若要运行主程序，需要遵循以下步骤:

> **注意**
>
> 在执行以下命令前，请检查一下路径下 `./代码/test` 是否存在`.bmp` 图片，并且检查一下`./代码/scripts/bmp.h` 文件的宏定义 `BMPFILEPATH` 是否指的是该图片。若有问题请自行修改，否则程序无法正常运行。

1. 将目录切换至 ./代码/build
2. 执行以下命令:

```
# 编译代码
$ make

# 运行可执行文件
$ ./lab5
Successfully open the file!
Size: 1548022(bit)
ColorBitWidth: 2144
Width: 714
Height: 722
Image Size: 1547968
Here are choices of spatial filtering.
15) Mean filter
16) Laplacian sharpening filter
Other numbers can cancel the filtering operation.
Please input your choice(15 or 16):
```

现在程序询问执行何种滤波操作，可以选择的方法有:

- 均值滤波（序号 15）
- 拉普拉斯变换的锐化滤波（序号 16）

下面以输入 16 为例:

```
Please input your choice(15 or 16): 16
Valid choice!
Please wait a minute for the generation of the new image...
The program provides two types of available Laplacian operators:
1)
0  1  0
1 -4  1
0  1  0
2)
1  1  1
1 -8  1
1  1  1
Please choose one of the operators(input 1 or 2):
```

现在程序询问用户使用何种拉普拉斯算子，一共有两种选择，这里以输入 2 为例:

```
Please choose one of the operators(input 1 or 2): 2
```

```
Valid choice!

Finish the conversion successfully!
```

此时程序完成了对图像的拉普拉斯变换的锐化滤波操作。

3. 来到 `./代码/tests` 目录，此时可以看到滤波后的图像。

> **注意**
>
> 在进行拉普拉斯变换的锐化滤波操作时，一定要确保程序宏定义 `FILTERWINDOWLE` 的值为 3，即窗口大小为 `3x3`；而在进行均值滤波的时候，可以通过改变该字段来设置窗口大小，得到不同滤波效果的图像。

五、 实验结果展示

> **注意**
>
> 本报告采用 typst 书写，但是 typst 不支持插入 BMP 图像文件，因此这里展示的是它们的 PNG 形式，对应的 BMP 形式见目录 `./代码/tests`。

5.1 图像均值滤波

Figure 8: 原图

Figure 9: 均值滤波，窗口大小为 3×3

Figure 10: 均值滤波，窗口大小为 9×9

Figure 11: 均值滤波，窗口大小为 15×15

Figure 12: 均值滤波，窗口大小为 21×21

可以看到，均衡滤波能够减少原图像中的噪点，使图像更加平滑。随着滤波窗口的增大，图像会变得越来越模糊，所以窗口并不是越大越好。下面再给出另一个例子：

Figure 13: 原图

Figure 14: 均值滤波，窗口大小为 3×3

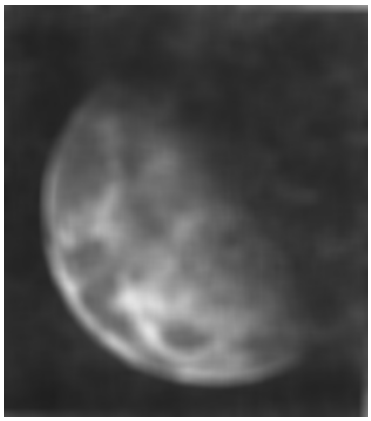


Figure 15: 均值滤波，窗口大小为 12×12

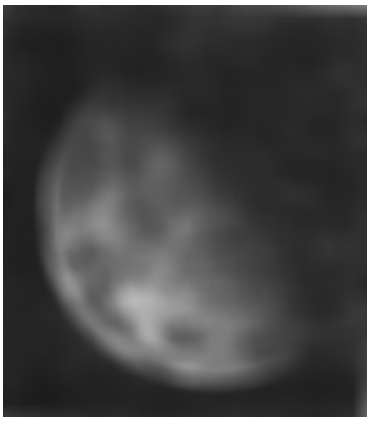


Figure 16: 均值滤波，窗口大小为 21×21

5.2 图像拉普拉斯变换增强

Figure 17: 原图

Figure 18: 拉普拉斯变换的锐化滤波得到的增强图像，使用第一种拉普拉斯算子

Figure 19: 拉普拉斯变换的中间图像

Figure 20: 拉普拉斯变换的锐化滤波得到的增强图像，使用第二种拉普拉斯算子

Figure 21: 拉普拉斯变换的中间图像

可以看到，拉普拉斯变换的锐化滤波能够较为明显地增强原图像，且第二种拉普拉斯算子（将对角线的像素算进去的那个）的锐化效果更明显。但是过强的锐化会导致某些点的像素值过大，因而第二种拉普拉斯变换会带来更多的白点。下面再给出另一个例子：

Figure 22: 原图

Figure 23: 拉普拉斯变换的锐化滤波得到的增强图像，使用第一种拉普拉斯算子

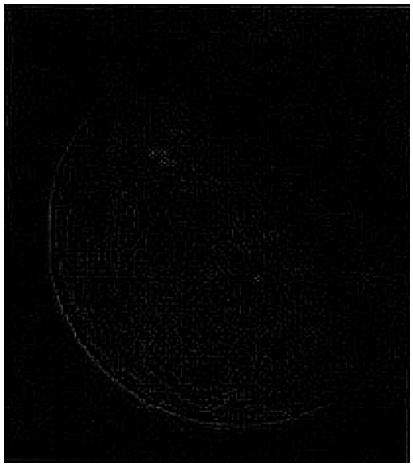


Figure 24: 拉普拉斯变换的中间图像

Figure 25: 拉普拉斯变换的锐化滤波得到的增强图像，使用第二种拉普拉斯算子

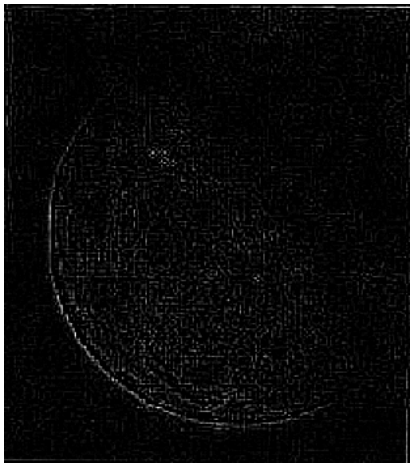


Figure 26: 拉普拉斯变换的中间图像

六、 心得体会

由于本次实验主要是根据滤波公式来调整图像的像素值大小来实现图像的平滑和锐化，因此代码量不是很大，完成得还算比较轻松顺利。简单的代码便能产生不同的滤波效果，让我感觉非常神奇有趣。

可能实验中最需要注意的是窗口的越界问题，但是比较容易处理，只需要稍微留点心就行。总的来说，本次实验加深了我对均值滤波、以拉普拉斯变换为代表的锐化滤波等空间滤波方法的认识和理解。