专业：计算机科学与技术
姓名：薛柔
学号：3220104854
日期：2023/10/29

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：均值滤波和拉普拉斯增强

一、实验目的和要求

1. 学会对图像进行均值滤波
2. 学会对图像进行拉普拉斯增强操作

二、实验内容和原理

实验内容：

1. 图像的均值滤波 (Mean filtering)
2. 图像的拉普拉斯增强 (Laplacian enhancement)

实验原理：

1. 卷积 (Convolution)

定义：g(x) 是两个一维函数 f(x) 和 h(x) 的卷积，当

$$g(x) = f(x) * h(x) = \int_{-\infty}^{\infty} f(t)h(x-t)dt$$

这意味着两个一维函数的卷积可以表述为它们乘积的积分。通常我们称 f(x) 为输入函数， h(x) 为卷积函数。

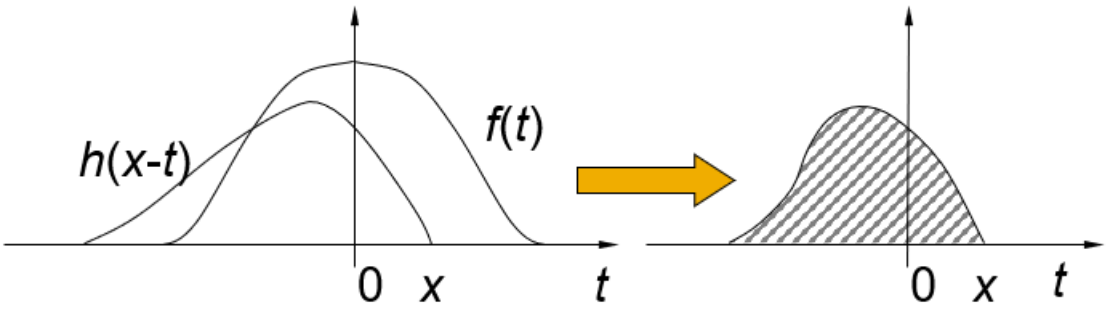


图 2.1 卷积的图示

对图像进行卷积即可达到滤波效果，其本质上等同于计算图像像素的加权和。

2. 空间滤波 (Spatial filtering)

1) 滤波器 (Filter)：滤波器是一个以 M×N 为大小的窗口，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。结果将保存为新图像中的像素。滤波器中的元素是系数，而不是像素值，该值表示施加在原始图像中像素上的权重。

对于图像中的每个像素 (x, y) ，其在滤波器中的响应值是根据滤波器中元素之间的预定义关系计算的。对于空间线性滤波，响应值是通过将系数与其相应像素之间的乘法相加（即加权平均）来计算的。

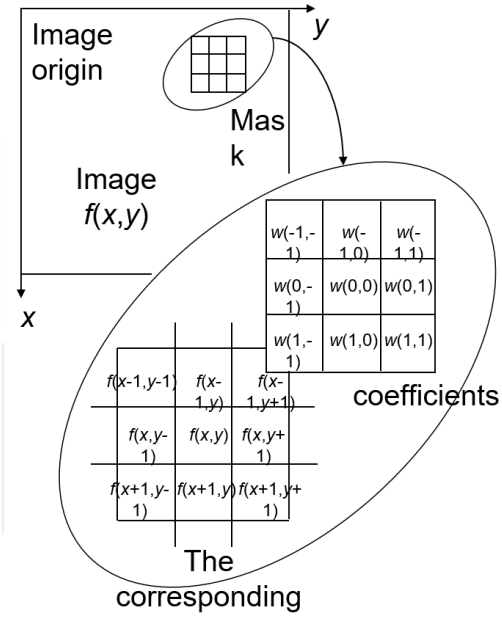


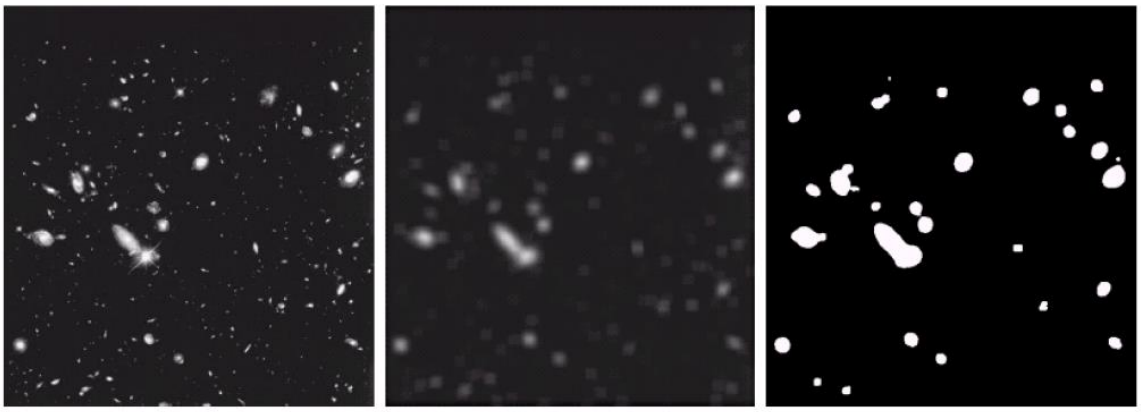
图 2.2 滤波器图示

2) 平滑滤波：当发现图像中的噪声过多时，可以对图像进行平滑处理，以减少噪声。但是，平滑操作会使图像模糊，丢失信息。可用于预处理图像，只保留较大的目标。

线性平滑滤波器的输出是滤波器中像素的平均值。它也被称为均值滤波器。均值滤波主要用于去除细微的细节，即消除小于滤波器大小的不需要的区域。蒙版的大小取决于想保留部分的大小，前面的系数使所有格子的权值和为 1。下面以 3*3 的滤波器为例。

$$\frac{1}{9}\times\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\qquad \frac{1}{16}\times\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$$

图 2.3 简单均值滤波（左）和加权均值滤波的滤波器（右）



a b c

FIGURE 3.36 (a) Image from the Hubble Space Telescope. (b) Image processed by a 15 × 15 averaging mask. (c) Result of thresholding (b). (Original image courtesy of NASA.)

图 2.4 均值滤波的效果图

除线性滤波之外，还有统计滤波。统计滤波是一种非线性的空间滤波，中心像素的值取决于窗口中的排序结果。最流行的统计过滤器是中位数过滤器。将中心像素替换为邻域中的中值。其降噪能力出色，与平均滤波器相比，引入的模糊更少。

3) 锐化滤波：为了增强图像中的细节或锐化模糊的部分，我们会使用锐化滤波。其中使用的工具是差分算子，其响应取决于相邻像素值之间的变化。差分算子能加强图像中的边缘和其他明显的变化（包括噪点），并削弱细微的变化。

对于图像数据，差分算子就是相邻两像素的像素值之差，即

$$\frac{\partial f}{\partial x} = f(x+1) - f(x)$$

同理，二阶差分为相邻两像素的差分之差，即

$$\frac{\partial^2 f}{\partial x^2} = f(x+1) + f(x-1) - 2f(x)$$

对于一般的二维图像，我们将 x, y 方向上的差分写成一个向量，即

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

它的模即为两个分量的平方和再开方，然而这种运算十分耗时，因此将其近似为分量的绝对值之和，即

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

图像处理中，常使用二阶差分对图像进行增强。定义拉普拉斯算子为：

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

由前面的定义和近似得：

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

将对角线上的像素也考虑进来得：

$$\begin{aligned} \nabla^2 f = [&f(x-1,y-1) + f(x,y-1) + f(x+1,y-1) \\ &+ f(x-1,y) + f(x+1,y) \\ &+ f(x-1,y+1) + f(x,y+1) + f(x+1,y+1)] \\ &- 8f(x,y) \end{aligned}$$

将它们写成滤波器的形式以及右边遮罩的大致效果图如下（它们均是旋转不变的）：

0	1	0
1	-4	1
0	1	0

1	1	1
1	-8	1
1	1	1

图 2.5 拉普拉斯算子的滤波器

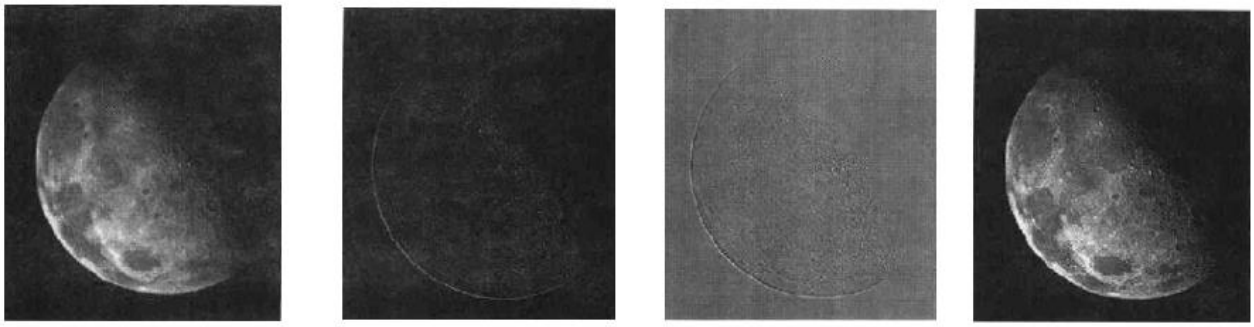


图 2.5 拉普拉斯增强的效果图

三、实验步骤与分析

1. 图像的均值滤波

实验 1 中读取到图像数据和输出图像不多做赘述。下面是进行均值滤波操作的函数。a 是存储原图信息的数组指针，函数实现将图像用 3*3 的蒙版等权重地进行滤波。对于边界的像素，我的处理方式是对周围有意义的像素值求平均，例如左上角的第一个像素，将其本身以及右、下、右下的周围三个像素求平均作为结果。

```
1.  void simple_mean(PIXEL **a)
2.  {
3.      FILE *fp;
4.      PIXEL *new_pixel[bmih.biHeight];
5.      for (int i=0;i<bmih.biHeight;i++){
6.          new_pixel[i] = (PIXEL*)malloc(sizeof(PIXEL)*(bmih.biWidth));
7.      }
8.      for(int i=0;i<bmih.biHeight;i++){
9.          for(int j=0;j<bmih.biWidth;j++){
10.             double sum_r=0;
11.             double sum_g=0;
12.             double sum_b=0;
13.
14.             int cnt=9;
15.             //滤波窗口相对于当前像素的位置
16.             for(int n=-1;n<2;n++){
17.                 for(int m=-1;m<2;m++){
18.                     if(i+n<0||j+m<0||i+n>=bmih.biHeight||j+m>=bmih.biWidth){
19.                         cnt--;//周围像素不在图像内，求和的像素数减少
20.                     }else{//周围像素在图像中，求和
21.                         sum_r+=a[i+n][j+m].red;
22.                         sum_g+=a[i+n][j+m].green;
23.                         sum_b+=a[i+n][j+m].blue;
24.                     }
25.                 }
26.             }//求周围像素值和的平均数
27.             new_pixel[i][j].red=sum_r/cnt;
28.             new_pixel[i][j].green=sum_g/cnt;
29.             new_pixel[i][j].blue=sum_b/cnt;
30.         }
31.     }
32.     //输出图像，省略
33.     //释放内存
34.     for (int i=0;i<bmih.biHeight;i++){
35.         free(new_pixel[i]);
```

```
    }
}
```

除此之外，与简单均值滤波类似，我还实现了加权均值滤波。权重在代码中给出，大体为中间权重高，周围权重较低。

```
void weight_mean(PIXEL **a)
{
    FILE *fp;
    int weight[3][3]={{1,2,1},{2,4,2},{1,2,1}};//权重

    PIXEL *new_pixel[bmih.biHeight];
    for (int i=0;i<bmih.biHeight;i++){
        new_pixel[i] = (PIXEL*)malloc(sizeof(PIXEL)*(bmih.biWidth));
    }
    for(int i=0;i<bmih.biHeight;i++){
        for(int j=0;j<bmih.biWidth;j++){
            double sum_r=0;
            double sum_g=0;
            double sum_b=0;

            int cnt=16;
            //滤波窗口相对于当前像素的位置
            for(int n=-1;n<2;n++){
                for(int m=-1;m<2;m++){
                    if(i+n<0||j+m<0||i+n>=bmih.biHeight||j+m>=bmih.biWidth){
                        cnt-=weight[1+n][1+m];//周围像素不在图像内，求和的像素数（代权重）减少
                    }else{//周围像素在图像中，乘以权重再求和
                        sum_r+=a[i+n][j+m].red*weight[1+n][1+m];
                        sum_g+=a[i+n][j+m].green*weight[1+n][1+m];
                        sum_b+=a[i+n][j+m].blue*weight[1+n][1+m];
                    }
                }
            }
            new_pixel[i][j].red=sum_r/cnt;
            new_pixel[i][j].green=sum_g/cnt;
            new_pixel[i][j].blue=sum_b/cnt;
        }
    }
    //输出图像，略
    //释放内存
    for (int i=0;i<bmih.biHeight;i++){
        free(new_pixel[i]);
    }
```

2. 拉普拉斯增强

运用原理中提到的公式，对图像进行拉普拉斯增强。第一部分是，用拉普拉斯算子算出二阶差分，因为有正有负，所以新申请了 int 类型的数组 change 来存储这一数据。

```
void laplacian_enhance(PIXEL **a)
{
    FILE *fp;
    //尝试了多种遮罩
    //int weight[3][3]={{0,1,0},{1,-4,1},{0,1,0}};
    //int weight[3][3]={{0,-1,0},{-1,4,-1},{0,-1,0}};
    //int weight[3][3]={{1,1,1},{1,-8,1},{1,1,1}};
    int weight[3][3]={{-1,-1,-1},{-1,8,-1},{-1,-1,-1}};

    PIXEL *new_pixel[bmih.biHeight];
    for (int i=0;i<bmih.biHeight;i++){
        new_pixel[i] = (PIXEL*)malloc(sizeof(PIXEL)*(bmih.biWidth));
    }
    //存储拉普拉斯算子的结果
    //0,1,2 分别存储 r,g,b
    int *change[3];
    for (int i=0;i<3;i++){
        change[i] = (int*)malloc(sizeof(int)*(bmih.biHeight*bmih.biWidth));
    }
    for(int i=0;i<bmih.biHeight;i++){
        for(int j=0;j<bmih.biWidth;j++){
            double sum_r=0;
            double sum_g=0;
            double sum_b=0;
            //滤波窗口相对于当前像素的位置
            for(int n=-1;n<2;n++){
                for(int m=-1;m<2;m++){
                    if(i+n<0||j+m<0||i+n>=bmih.biHeight||j+m>=bmih.biWidth){
                        continue;//像素位置不合法
                    }else{//计算加权和
                        sum_r+=a[i+n][j+m].red*weight[1+n][1+m];
                        sum_g+=a[i+n][j+m].green*weight[1+n][1+m];
                        sum_b+=a[i+n][j+m].blue*weight[1+n][1+m];
                    }
                }
            }
            int new_r=sum_r;
            int new_g=sum_g;
            int new_b=sum_b;
            change[0][i*bmih.biHeight+j]=new_r;
```

```
            change[1][i*bmih.biHeight+j]=new_g;
            change[2][i*bmih.biHeight+j]=new_b;
            //用于输出中间图像
            if(new_r>255)   new_r=255;
            if(new_g>255)   new_g=255;
            if(new_b>255)   new_b=255;

            if(new_r<0)   new_r=0;
            if(new_g<0)   new_g=0;
            if(new_b<0)   new_b=0;
            new_pixel[i][j].red=new_r;
            new_pixel[i][j].green=new_g;
            new_pixel[i][j].blue=new_b;
        }
    }
```

接下来是将这一结果 Rearrange，并加到原来的像素值上。这里我用的方法是，乘以一个小数，我们也将这一过程输出图片，用于比较。

```
    //Rearrange 后的中间图像
    for(int i=0;i<bmih.biHeight;i++){
        for(int j=0;j<bmih.biWidth;j++){
            new_pixel[i][j].red=new_pixel[i][j].red*0.2;
            new_pixel[i][j].green=new_pixel[i][j].green*0.2;
            new_pixel[i][j].blue=new_pixel[i][j].blue*0.2;
        }
    }

    //叠加增强
    for(int i=0;i<bmih.biHeight;i++){
        for(int j=0;j<bmih.biWidth;j++){
            int new_r=a[i][j].red+change[0][i*bmih.biHeight+j]*0.2;
            int new_g=a[i][j].green+change[1][i*bmih.biHeight+j]*0.2;
            int new_b=a[i][j].blue+change[2][i*bmih.biHeight+j]*0.2;
            //防止溢出
            if(new_r>255)   new_r=255;
            if(new_g>255)   new_g=255;
            if(new_b>255)   new_b=255;

            if(new_r<0)   new_r=0;
            if(new_g<0)   new_g=0;
            if(new_b<0)   new_b=0;

            new_pixel[i][j].red=new_r;
            new_pixel[i][j].green=new_g;
```

```
27.             new_pixel[i][j].blue=new_b;
28.         }
29.     }
30. //输出图像
```

这样我们就得到了锐化后的图像。

四、实验环境及运行方法

- 实验环境：Windows 11 系统

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

- 运行方法：先将 dip_hw5.exe 文件和 42.bmp 以及 gray.bmp 放在同一个目录下，运行 dip_hw5.exe 文件，输入 0 则对彩色图像进行操作，选 1 则对灰色图像进行操作。注意，再次运行后原来的结果图像会被覆盖。等待 1s 左右会得到 5 张新的图像。它们的命名分别为对原始图像进行的操作名（例如简单均值滤波操作后的图像被命名为 simple_mean，具体名称在结果展示中给出）。当终端出现“Successfully open the image”时说明我们成功打开了原始图像，否则会输出“BMP Image Not Found!”。

  如果运行不成功，可以将文件夹中的.c 源文件重新编译运行。

五、实验结果展示

下面先展示均值滤波的结果：

大遮罩，可以获得更明显（即更模糊）的效果，下面是 5*5 遮罩的效果。

图 5.1 输入图像 42.bmp（左）和简单均值滤波后的图像 simple_mean.bmp（右）

下面的左图为 3*3 遮罩的简单均值滤波，相较于 5*5 模糊效果较弱。右图为 3*3 遮罩的加权均值滤波，能保留更多的重点。

图 5.2 简单均值滤波后的图像 simple_mean.bmp（左）加权均值滤波后的图像 weight_mean.bmp（右）

下图为拉普拉斯增强的过程图，对于彩色图，我在三个色彩通道中分别增强。

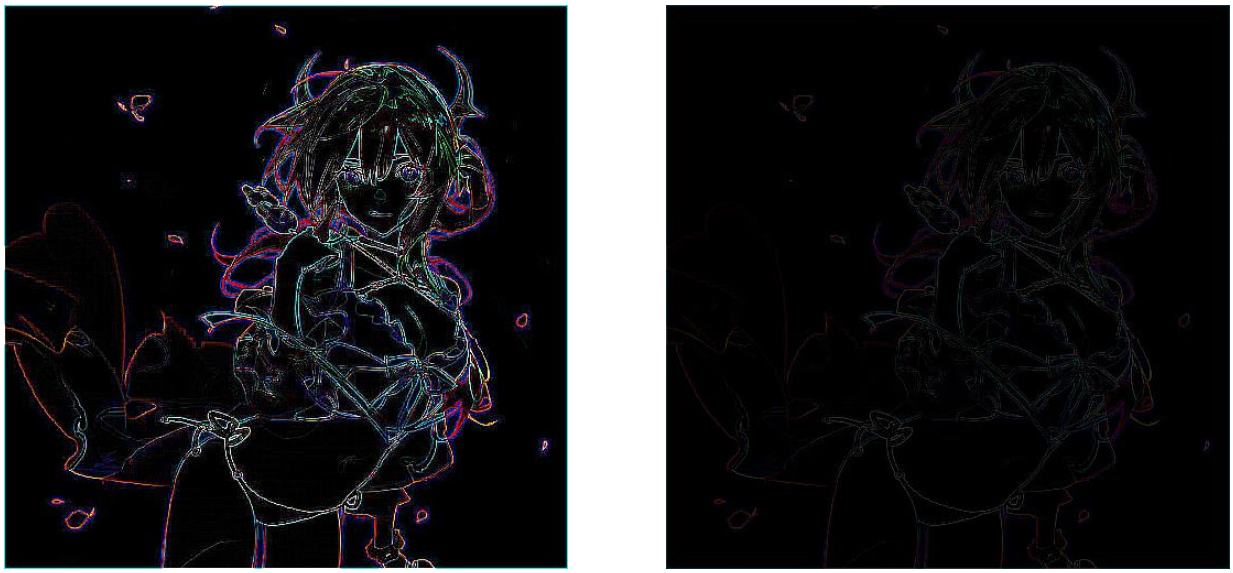


图 5.3 计算拉普拉斯算子后的中间图 Laplacian_result.bmp（左）

和 Rearrange 后的 rearranged_Laplacian_result.bmp（右）

将输入图像与增强结果放在一起作对比。可以看到，结果图像明显锐化，线条更加明显，达到了我们想要的效果。


图 5.4 输入图像 42.bmp（左）和拉普拉斯增强后的图像 Laplacian_enhance.bmp（右）


图 5.5 灰度图像计算拉普拉斯算子后的中间图 Laplacian_result.bmp（左）
和 Rearrange 后的 rearranged_Laplacian_result.bmp（右）


图 5.6 灰度输入图像 gray.bmp（左）和拉普拉斯增强后的图像 Laplacian_enhance.bmp（右）

六、心得体会

本次还有很多可以额外探索的地方：例如如何找到合适的遮罩大小、改变系数得到更好滤波效果。可以看到由于我的原始图像相较而言不算很小，适当增大滤波器的大小可以获得更好的模糊和锐化效果。值得一提的是，尽管展示图片在报告 pdf 中的对比不强烈，但用图片查看器看我的锐化结果图像，效果是较为明显的。

只要掌握基本原理，无论是均值滤波，还是非线性的中值滤波，实现都是较为容易的。因此它们在图像处理中得到了广泛的应用。