第3章

图像平滑

本章主要讲述均值滤波、中值滤波、最小值滤波、最大值滤波、高斯滤波、二值图像滤波、数学形态学滤波、条件滤波等常用滤波方法，并讲述它们的思想和特点。

本章讲述滤波器的C++语言编程及其优化，包括“整数除法或者浮点数乘法和除法变为整数乘法和移位”“列积分”“积分图”“MMX及SSE”“直方图求中值”“直方图递推”“基于均值滤波的二值图像滤波”等程序优化技巧。

3.1 噪声与图像平滑

3.1.1 噪声

噪声(noise)这个词的原意是正常声音信号受到的干扰。凡是干扰人们休息、学习和工作以及对正常的声音产生干扰的声音，即不需要的声音，统称为噪声。后来，人们把这个概念推广到了更广泛的领域。

在图像处理领域，图像噪声是指存在于图像数据中不必要的干扰信息，表现为图像中引起较强视觉效果的孤立像素点或像素块。图像噪声的产生原因是多方面的：一是图像采集设备产生的噪声，比如图像传感器CCD和CMOS在采集图像过程中，由于受传感器材料属性、工作环境、电子元器件和电路结构等影响，会引入各种噪声，如电阻引起的热噪声、场效应管的沟道热噪声、暗电流噪声、像素的非均匀性噪声；二是在图像处理过程中，由于处理方法的缺陷，比如计算精度不够、模型考虑不全面、参数自适应能力不够、近似计算等产生的错误信息；三是图像数据在传输过程中受到外界的电磁波干扰，或者在用胶片存储图像时的胶片老化等。

对一个图像处理系统而言，噪声的来源分为外部噪声和内部噪声。外部噪声是指由于该系统外的因素引起的噪声，比如以电磁波辐射或经电源引入系统而引起的噪声，如电器设备、天体放电现象等引起的噪声，电源纹波引起的噪声，图3-1所示就是电源纹波引起的噪声。内部噪声是指来自系统内的因素引起的噪声，比如图像传感器材质特性本身产生的噪声、各种接头的抖动引起电流变化等产生的噪声、系统内部的电源芯片或者晶振温度漂移等产生的噪声，如图3-2所示。

图 3-1　电源纹波对某红外热像仪的条纹干扰

图 3-2　夜晚光照不足时某图像传感器的噪声

在对噪声的处理和分析上，常进行如下分类。

(1) 按统计特性划分：统计特性不随时间变化的噪声称为平稳噪声，反之称为非平稳噪声。

(2) 按噪声幅度分布形状划分：呈高斯分布的噪声称为高斯噪声，呈雷利分布的噪声称为雷利噪声。

(3) 按噪声频谱的形状划分：频谱均匀分布的噪声称为白噪声，频谱与频率成反比的噪声称为 $1/f$ 噪声，频谱与频率平方成正比的噪声称为三角噪声。

(4) 按噪声 $n(t)$ 和信号 $S(t)$ 之间的关系划分：输出信号为 $S(t)+n(t)$ 形式的噪声称为加性噪声，输出信号为 $S(t)(1+n(t))$ 形式的噪声称为乘性噪声。

为了便于分析和处理，往往将乘性噪声近似地认为是加性噪声，并总是假设信号和噪声互相统计独立。去除噪声的过程称作滤波，不同的滤波方法称作滤波器(filter)。

在图像处理和图像分析的算法中，为了保持一定的稳健性，一般需要先对原始图像做一定的滤波处理(称为图像预处理)。

3.1.2　图像平滑概述

在图像处理中，用作去掉图像噪声的各种滤波方法统称为图像平滑(image smoothing)。图像平滑的字面意思是使一个像素到其相邻像素的灰度变化是平滑的。

在图像处理中，从空间域的观点看，滤波就是去掉突然变大或变小的灰度值，用一个合适的灰度值替代该值。很显然，一个像素的灰度值是不是噪声，是由该像素在图像中的位置决定的，通过它与相邻像素的比较才能判定它的灰度值是异常(突然变大还是变小)还是正常。比如，一个人的收入为 1000 美元/月，这个收入在有些国家可能是属于贫困的，但在另一些国家可能是属于富裕的。那么，既然一个像素是否是噪声是由其邻域决定的，显然去掉噪声也得通过邻域计算，因此空间域的图像滤波属于邻域运算。

看下面一组测量数据 D_1：3　3　3　9　3　3　9　9　9　3　9　9　9，其数据曲线如图 3-3 所示。

图 3-3　原始数据曲线

该曲线上有 2 个明显的毛刺(噪声),如画圈处所示。在波形的前半截,数值从“3”突然变成“9”,在波形的后半截,数值从“9”突然变成“3”。

如前所言,空间域的图像滤波是邻域运算,而邻域有大小和形状之说,运算有何种运算之说,因此本章的主要内容就是围绕运算和邻域两个方面展开的。

3.2 均值滤波

3.2.1 均值滤波的定义

一个朴素的想法是,既然噪声表现为它的灰度值与周围相比有异常变化,比如突然变大或者突然变小,那么把它与周围的像素做灰度平均即可。比如,一个宿舍有 5 个同学,其中 1 个同学特别富有,那么把财富平均即可。也就是说,这种滤波算法的运算采用的是求均值,因此就称为均值滤波。

【定义 3-1】 均值滤波(average filtering)就是以当前像素为中心取一个邻域,用该区域的所有像素灰度值的均值作为该像素滤波后的灰度值。

那么如何设定邻域的大小和形状呢? 在一维数据上,可以取相邻 3 像素,也可以取相邻 5 像素,还可以取相邻 7 像素,总之邻域的大小是奇数(要以当前像素为中心左右对称)。

对前面的数据 D_1,假定取相邻 3 像素作为邻域,其均值滤波的计算过程和结果如下:

原值	求均值	均值
3	(邻域不完整,保持原值)	3
3	(3+3+3)/3	3
3	(3+3+9)/3	5
9	(3+9+3)/3	5
3	(9+3+3)/3	5
3	(3+3+9)/3	5
9	(3+9+9)/3	7
9	(9+9+9)/3	9
9	(9+9+3)/3	7
3	(9+3+9)/3	7
9	(3+9+9)/3	7
9	(9+9+9)/3	9
9	(邻域不完整,保持原值)	9

经过以上计算,数据变得平滑了,其结果是 3 3 5 5 5 5 7 9 7 7 7 9 9,如图 3-4 所示。

上述均值滤波过程可以描述为:

$$G(x)=\frac{g(x-1)+g(x)+g(x+1)}{3} \tag{3-1}$$

图 3-4　均值滤波后的结果

进一步形式化，一维均值滤波的形式化写法为：

$$G(x)=\frac{\sum_{j=-m}^{m} g(x+j)}{2m+1} \tag{3-2}$$

对于常规的二维图像而言，邻域常取矩形（习惯上称为滤波窗口，一般是正方形）。设此矩形的大小为 n 行 m 列（m，n 均为奇数），则二维均值滤波的形式化写法为：

$$G(x,y)=\frac{\sum_{i=-\frac{n}{2}}^{\frac{n}{2}}\sum_{j=-\frac{m}{2}}^{\frac{m}{2}} g(x+j,y+i)}{m\times n} \tag{3-3}$$

其含义就是以当前像素为中心取出矩形块内所有像素的灰度值累加，再除以矩形块内像素的个数，即均值。

3.2.2　邻域与卷积运算

式（3-3）可以描述矩形的、正方形的邻域，但除此之外，邻域还可以是其他任意形状，此时它就不能很好地表述了，该怎么处理呢？比如，采用当前像素周围 5 像素的均值时，如图 3-5 所示。

于是想到用一个小的图形来表示邻域，该图形称作模板（template）。该图形中参与运算的像素对应位置设为“1”，不参与运算的像素对应位置设为“0”，有时也用不填值代表 0，即图 3-6。可以看出图 3-6 是一个 3×3 的矩阵，记为 $\boldsymbol{T}$。则均值滤波的公式（3-3）用模板表示为：

$$G(x,y)=\frac{\sum_{i=-\frac{n}{2}}^{\frac{n}{2}}\sum_{j=-\frac{m}{2}}^{\frac{m}{2}} g(x+j,y+i)\boldsymbol{T}\left(\frac{m}{2}+j,\frac{n}{2}+i\right)}{\sum_{i=-\frac{n}{2}}^{\frac{n}{2}}\sum_{j=-\frac{m}{2}}^{\frac{m}{2}} \boldsymbol{T}\left(\frac{m}{2}+j,\frac{n}{2}+i\right)} \tag{3-4}$$

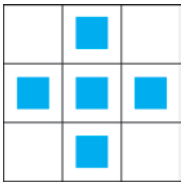


图 3-5　十字形邻域

0	1	0
1	1	1
0	1	0

图 3-6　模板表示

式(3-4)的含义是像素邻域的灰度值与模板的值做对应位置相乘,得到其总和,再用总和除以模板中不为 0 的元素的个数。显然,模板中取 0 的元素与对应像素的灰度值相乘后为 0,相当于对应像素的灰度值没有累加到总和中,即该像素没有起作用。

模板就是一个小的矩阵,像素邻域中的每个像素分别与该矩阵的每个元素对应相乘,这是一种卷积运算(convolution),所以模板又常称为卷积核,均值滤波又被看成是一种卷积运算。

因此,采用模板的形式可以表示任意的邻域。在图像处理中,常用的均值滤波模板的形状有方形和圆形两种,大小有 3×3 和 5×5 两种,如图 3-7～图 3-10 所示,式(3-4)中的分母值就是模板中数值不为 0 的像素个数。

$$\boldsymbol{T}^3=\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$$

图 3-7　3×3 的方形滤波器

$$\boldsymbol{T}^3=\frac{1}{5}\begin{bmatrix}0&1&0\\1&1&1\\0&1&0\end{bmatrix}$$

图 3-8　3×3 的十字形滤波器

$$\boldsymbol{T}^5=\frac{1}{25}\begin{bmatrix}1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\end{bmatrix}$$

图 3-9　5×5 的方形滤波器

$$\boldsymbol{T}^5=\frac{1}{21}\begin{bmatrix}0&1&1&1&0\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\0&1&1&1&0\end{bmatrix}$$

图 3-10　5×5 的圆形滤波器

3.2.3　均值滤波的特点

均值滤波的特点是:

(1) 滤波结果是灰度值大的像素在滤波后灰度值变小,灰度值小的像素在滤波后灰度值变大,图像总灰度值之和不变,即图像的均值(亮度)不变;

(2) 因为灰度值大的像素在滤波后灰度值变小、灰度值小的像素在滤波后灰度值变大,所以图像的标准差(对比度)变小,图像变模糊;

(3) 在目标和背景的边界上,滤波前后灰度值的变化尤其大,所以边界模糊得尤其突出;

(4) 均值滤波采用的邻域越大则模糊越突出。

图 3-11 是原始的灰度图像,其均值是 108,标准差是 42;图 3-12 是 5×5 的均值滤波的图像,其均值是 108,标准差是 39。图 3-13 是原始的灰度图像,其均值是 144,标准差是 101;图 3-14 是 5×5 的均值滤波的图像,其均值是 144,标准差是 101。

可以思考一下,图 3-12 和图 3-14 都是 5×5 的均值滤波图像,分别和原始图像相比,为什么图 3-12 标准差的变化较大而图 3-14 的标准差几乎没有变化。

图 3-11　原始图像(均值＝108，标准差＝42)

图 3-12　5×5 的均值滤波图像(均值＝108，标准差＝39)

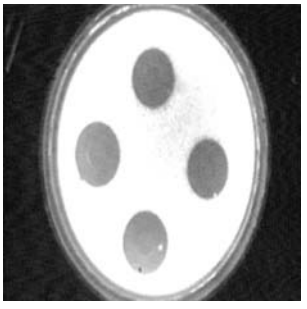


图 3-13　原始图像(均值＝144，标准差＝101)

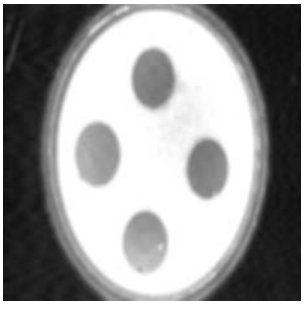


图 3-14　5×5 的均值滤波图像(均值＝144，标准差＝101)

3.2.4　基于列积分的快速均值滤波

求均值的过程就是一个多次累加和一次除法的过程，如果邻域大小为 101×101，难道均值滤波需要对每个像素求邻域均值且都进行 1 万多(10 201)次加法吗？那么一幅 1080p(1920×1080 像素)灰度图像的均值滤波岂不是要进行 200 多亿(21 152 793 600)次加法？显然均值滤波肯定不是这么计算的。本书作者在多年前提出了一种基于列积分的快速均值滤波方法。

在图 3-15 和图 3-16 中，两个框分表代表两个相邻像素的邻域。可以发现，相邻像素的邻域是重叠的，且重叠的部分相当大，这样就意味着重叠部分的灰度累加不用重复进行。对于 n 行×m 列的邻域，在某行中从一个像素处理到下一个像素时，邻域仅仅是去掉最左列积分并增加右列积分，如图 3-15 所示，仅访问 $2n$ 个数据；在某列中，从一个像素移动到下一个像素时，邻域仅仅是去掉最上行和增加下行，仅访问 $2m$ 个数据，如图 3-16 所示。显然，访问的数据量无论是 $2n$ 还是 $2m$ 都远远小于整个邻域的大小 $n \times m$。

实际远不止如此，计算还可以更加快速。在同一行上，如果先把 n 行数据进行按列求和(列积分)，并将列积分放到一个数组 sumCol[x]中，$x=0\cdots$width-1，则邻域从当前像素向右移动到其下一个像素时，只需要减去该邻域最左边的列积分，再加上其右边的列

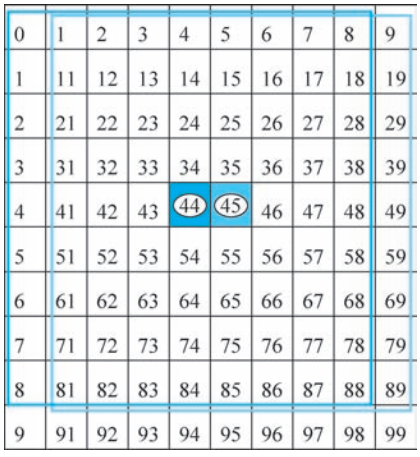


图 3-15 左右相邻的两个像素的邻域

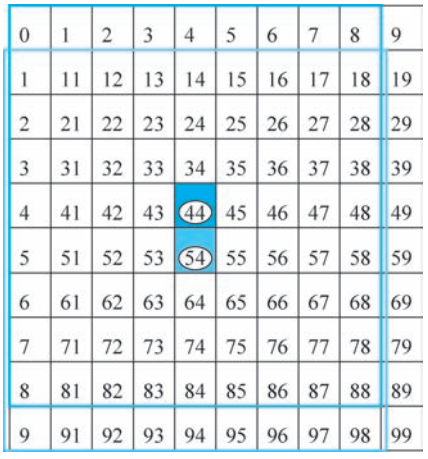


图 3-16 上下相邻的两个像素的邻域

积分,这样就只需要 1 次减法运算和 1 次加法运算。当从当前行移动到下一行时,则只需要把 sumCol[x]更新一下即可,这个更新也非常简单,只需要将 sumCol[x]减去该邻域的最上一行像素的灰度值,并加上其下一行像素的灰度值即可,也只需要 1 次减法运算和 1 次加法运算。算法描述如下。

【算法 3-1】 基于列积分的快速均值滤波

```
void RmwAvrFilterBySumCol( BYTE * pGryImg,              //原始灰度图像
                           int width, int height,       //图像的宽度和高度
                           int M, int N,                //滤波邻域: M 列 N 行
                           BYTE * pResImg               //结果图像
                         )
{   //没有对边界上邻域不完整的像素进行处理,可以采用变窗口的策略
    BYTE * pAdd, * pDel, * pRes;
    int halfx, halfy;
    int x, y;
    int sum,c;
    int sumCol[4096];                                   //约定图像宽度不大于 4096

    // step.1 ------------ 初始化 ---------------------------- //
    M = M/2 * 2 + 1;                                    //奇数化
    N = N/2 * 2 + 1;                                    //奇数化
    halfx = M/2;                                        //滤波器的 x 半径
    halfy = N/2;                                        //滤波器的 y 半径
    c = (1 << 23)/(M * N);                              //乘法因子
    memset(sumCol, 0, sizeof(int) * width);
    for (y = 0, pAdd = pGryImg; y < N; y++)
    {
        for (x = 0; x < width; x++) sumCol[x] += * (pAdd++);
    }
    // step.2 ------------ 滤波 ---------------------------- //
    for(y = halfy,pRes = pResImg + y * width,pDel = pGryImg; y < height - halfy; y++)
```

```
    {
        //初值
        for (sum = 0,x = 0; x < M; x++) sum += sumCol[x];
        //滤波
        pRes += halfx;                              //跳过左侧邻域不完整的像素
        for (x = halfx; x < width - halfx; x++)
        {
            //求灰度均值
            // * (pRes++) = sum/(N * M);
            * (pRes++) = (sum * c)>> 23;            //用整数乘法和移位代替除法
            //换列,更新灰度和
            sum -= sumCol[x - halfx];               //减去左边的列积分
            sum += sumCol[x + halfx + 1];           //加上右边的列积分
        }
        pRes += halfx;                              //跳过右侧邻域不完整的像素
        //换行,更新 sumCol
        for (x = 0; x < width; x++)
        {
            sumCol[x] -= * (pDel++);                //减去上一行像素的灰度值
            sumCol[x] += * (pAdd++);                //加上下一行像素的灰度值
        }
    }
    // step.3 ------------ 返回 ---------------------------- //
    return;
}
```

算法 3-1 求像素的灰度均值时仅需要 2 个加法、2 个减法、1 个乘法、1 个移位，共 6 个基本的整数运算，与邻域的大小 $n \times m$ 无关。此算法的巧妙之处在于采用了一个称为“列积分”的数组 sumCol。

该算法没有考虑图像边界上邻域不完整的像素的处理，可以采用变窗口的策略，对这些像素进行邻域不同的均值滤波，具体实现过程不再赘述。

另外，本算法并没有使用除法运算 sum/(N * M)，而是使用了(sum * c)≫23，这涉及一个编程技巧，即“整数除法或者浮点数乘法和除法变为整数乘法和移位”。在图像处理中，图像间的除法运算、均值滤波、直方图均衡化等算法中要用到除法运算，除法作为一个基本的运算也是常常用到的。但除法指令的执行速度是非常慢的(在很多 CPU 上，浮点数除法或乘法所用的时间大约是整数乘法的 10 倍，整数除法所用的时间大约是整数乘法的 3 倍)，因此，消除整数除法、浮点数乘法和除法就成了提高程序执行效率的关键要素。

设 $y=\frac{x}{b}$，x 和 y 都是整数，则在形式上有：

$$y=\frac{x}{b} \Rightarrow y=x \times \frac{1}{b} \Rightarrow y=x \times \frac{1 \times 2^k}{b} \div 2^k \tag{3-5}$$

令整数 $c=\frac{1 \times 2^k}{b}$，则有 $y=x \times c \div 2^k$，在计算机中一个整数除以 2 的 k 次方与该整

数右移 k 位是等价的,所以又有:

$$y=(x\times c)\gg k \tag{3-6}$$

至于 k 的取值,显然越大越好,k 越大则 c 越大,c 对 $\frac{1}{b}$ 替代的精度就越高,但一定得保证整数表达式 $x\times c$ 的值不能上溢(超过整数类型的数值表示范围),因此 k 的取值保证整数 y 有足够的精度即可。

例如,当 $x=255$,$b=9$ 时,则 $y=\lfloor x/9\rfloor=\lfloor 255/9\rfloor=28$。若令 $k=10$,则有 $c=\lfloor 1024/9\rfloor=113$,$y=(255\times 113)\gg 10=28$,与原来使用除法运算得到的值 28 相同。在图像处理中,如果原来是对每个像素都进行一次除法运算 $x/9$,那么现在就是只进行一次除法求出 c,然后对所有的像素进行一次整数乘法和移位运算即可。

同上,浮点的乘法表达式也可以转换为整数乘法和移位,设 $y=x\times a$,则在形式上有:

$$y=x\times a\Rightarrow y=x\times(2^k\times a)\div 2^k \tag{3-7}$$

此时令整数 $c=(2^k\times a)$,即可得到式(3-6)。

例如,当 $x=255$,$a=0.299$ 时,$y=\lfloor 255\times 0.299\rfloor=76$。若令 $k=10$,则有 $c=\lfloor 1024\times 0.299\rfloor=306$,$y=(255\times 306)\gg 10=76$,与原来使用浮点乘法运算得到的值 76 相同。

3.2.5 基于积分图的快速均值滤波

习惯上,常采用一种称作“积分图”的方法来实现均值滤波,该方法也能充分利用相邻像素的邻域交集,避免大量的重复运算。积分图的概念由 Paul Viola 等人(Detection using a boosted cascade of simple features. Computer Vision and Pattern Recognition, 2001,1: 8-14)提出,其定义如下。

【定义 3-2】 灰度图像积分图中任意一个像素 $s(x,y)$ 的值是从灰度图像的左上角 $(0,0)$ 与当前位置 (x,y) 所围成的矩形区域内的像素灰度值之和。

$$s(x,y)=\sum_{j=0}^{x}\sum_{i=0}^{y}g(j,i) \tag{3-8}$$

如在图 3-17 中:

$s(0,0)=1$

$s(3,0)=1+2+3+4$

$s(0,3)=1+9+17+25$

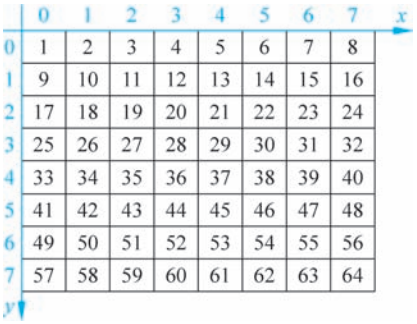
$s(3,3)=1+2+3+4+9+10+11+12+17+18+19+20+25+26+27+28$

在得到积分图后,均值滤波就变得非常简单。比如,图 3-18 中灰色区域的灰度值之和为 $s(3,3)-s(0,3)-s(3,0)+s(0,0)$。

显然,基于积分图求一块区域的灰度值之和只需要 2 个减法和 1 个加法,与区域的大小无关。有了灰度值之和,再除以邻域像素总个数,均值也就得到了。像素 (x,y) 的 n 行×m 列邻域内所有像素的灰度值之和 sumGry(x,y) 用积分图计算如下:

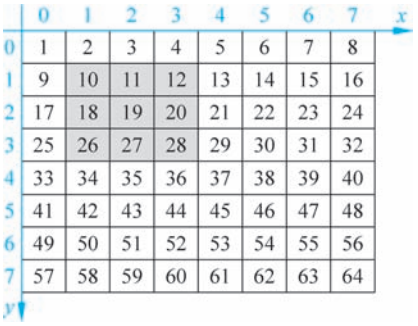
$$\text{sumGry}(x,y)=s(x_2,y_2)-s(x_1,y_2)-s(x_2,y_1)+s(x_1,y_1) \tag{3-9}$$

其中,$\mathrm{d}x=\left\lfloor\frac{m}{2}\right\rfloor$,$\mathrm{d}y=\left\lfloor\frac{n}{2}\right\rfloor$,$x_1=x-\mathrm{d}x-1$,$x_2=x+\mathrm{d}x$,$y_1=y-\mathrm{d}y-1$,$y_2=y+\mathrm{d}y$,$m$ 和 n 均为奇数。



	0	1	2	3	4	5	6	7	x
0	1	2	3	4	5	6	7	8	
1	9	10	11	12	13	14	15	16	
2	17	18	19	20	21	22	23	24	
3	25	26	27	28	29	30	31	32	
4	33	34	35	36	37	38	39	40	
5	41	42	43	44	45	46	47	48	
6	49	50	51	52	53	54	55	56	
7	57	58	59	60	61	62	63	64	
y									

图 3-17　积分图计算所用的原图



	0	1	2	3	4	5	6	7	x
0	1	2	3	4	5	6	7	8	
1	9	10	11	12	13	14	15	16	
2	17	18	19	20	21	22	23	24	
3	25	26	27	28	29	30	31	32	
4	33	34	35	36	37	38	39	40	
5	41	42	43	44	45	46	47	48	
6	49	50	51	52	53	54	55	56	
7	57	58	59	60	61	62	63	64	
y									

图 3-18　使用积分图计算灰色区域像素灰度值之和

【算法 3-2】 基于积分图的快速均值滤波

```
void RmwAvrFilterBySumImg( int * pSumImg,              //计算得到的积分图
                           int width, int height,      //图像的宽度和高度
                           int M, int N,               //滤波邻域：M 列 N 行
                           BYTE * pResImg              //结果图像
                         )
{   //没有对边界上邻域不完整的像素进行处理,可以采用变窗口的策略
    int * pY1, * pY2;
    BYTE * pRes;
    int halfx, halfy;
    int x, y, x1, x2;
    int sum, c;

    // step.1 ------------ 初始化 ---------------------------- //
    M = M/2 * 2 + 1;                                   //奇数化
    N = N/2 * 2 + 1;                                   //奇数化
    halfx = M/2;                                       //滤波器的 x 半径
    halfy = N/2;                                       //滤波器的 y 半径
    c = (1 << 23)/(M * N);                             //乘法因子
    // step.2 ------------ 滤波 ------------------------------ //
    for ( y = halfy + 1, pRes = pResImg + y * width, pY1 = pSumImg,
          pY2 = pSumImg + N * width;
          y < height - halfy;
          y++, pY1 += width, pY2 += width
        )
    {
        pRes += halfx + 1;                             //跳过左侧
        for (x = halfx + 1, x1 = 0, x2 = M; x < width - halfx; x++, x1++, x2++)
        //for (x1 = 0, x2 = M; x2 < width; x1++, x2++)   //上一行的简化形式,但不太容易读
        {
            sum = *(pY2 + x2) - *(pY2 + x1) - *(pY1 + x2) + *(pY1 + x1);
            *(pRes++) = (sum * c)>> 23;                //用整数乘法和移位代替除法
        }
```

```
            pRes += halfx;                                  //跳过右侧
        }
        // step.3 ------------ 返回 ---------------------------- //
        return;
    }
```

3.2.6 基于列积分的积分图实现

积分图的计算公式(3-8)非常简单,很容易实现。根据处理器架构的不同,比如CPU(比如X86的SIMD指令集)或者GPU(比如NV的SIMT架构),积分图应有多种不同的快速实现形式。根据3.2.4节中给出的"列积分"概念,下面给出一个基于列积分的积分图计算方法。

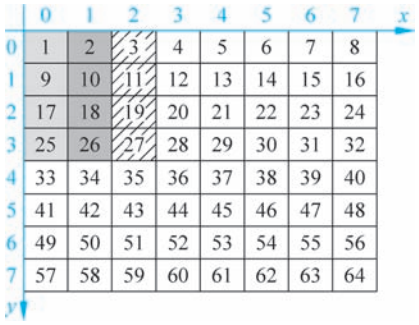


图 3-19 使用列积分的积分图计算

假设在 y_0 行上,已经知道了每列之和 $\mathrm{sumCol}(x,y_0)$,则积分图 $s(x,y_0)$ 的值为:

$$s(x,y_0)=\sum_{j=0}^{x}\mathrm{sumCol}(x,y_0) \qquad (3\text{-}10)$$

如图3-19中,已经知道了 $y=3$ 时的 $x=0$,$x=1$,$x=2$ 时列积分如下,分别如图3-19中不同底色所示。

$$\mathrm{sumCol}(0,3)=1+9+17+25$$
$$\mathrm{sumCol}(1,3)=2+10+18+26$$
$$\mathrm{sumCol}(2,3)=3+11+19+27$$

显然有:

$$s(0,3)=\mathrm{sumCol}(0,3)$$
$$s(1,3)=\mathrm{sumCol}(0,3)+\mathrm{sumCol}(1,3)$$
$$s(2,3)=\mathrm{sumCol}(0,3)+\mathrm{sumCol}(1,3)+\mathrm{sumCol}(2,3)$$

写成递推形式就是:

$$s(0,3)=\mathrm{sumCol}(0,3)$$
$$s(1,3)=s(0,3)+\mathrm{sumCol}(1,3)$$
$$s(2,3)=s(1,3)+\mathrm{sumCol}(2,3)$$

【算法 3-3】 基于列积分的积分图计算

```
void RmwDoSumGryImg( BYTE * pGryImg,                //原始灰度图像
                     int width,                     //图像的宽度
                     int height,                    //图像的高度
                     int * pSumImg                  //计算得到的积分图
                   )
{
    BYTE * pGry;
    int * pRes;
    int x, y;
```

```
    int sumCol[4096];                                    //约定图像宽度不大于 4096

    memset(sumCol, 0, sizeof(int) * width);
    for (y = 0, pGry = pGryImg, pRes = pSumImg; y < height; y++)
    {
        //最左侧像素的特别处理
        sumCol[0] += *(pGry++);
        *(pRes++) = sumCol[0];
        //正常处理
        for (x = 1; x < width; x++)
        {
            sumCol[x] += *(pGry++);                      //更新列积分
            *(pRes++) = *(pRes - 1) + sumCol[x];
        }
    }
    return;
}
```

3.2.7　基于 SSE 的积分图实现

图像数据是一种非常特别的数据，灰度值一般是 8 位(b)的，颜色分量也是 8 位，但是现在计算机的数据总线宽一般都是 64 位，这就是说在访问一个像素时，数据总线上传输的 64 位中只有 8 位是有用的，另外的都白白浪费了。为此，ARM 为处理图像等多媒体数据设计了 NEON 指令集；Intel 为处理图像等多媒体数据设计了 MMX 和 SSE 指令集，最早是 2000 年以前的 MMX(Multi Media Extensions，多媒体扩展指令集)，现在发展到了 SSE(Streaming SIMD Extensions，单指令多数据流扩展指令集)4. 2、AVX(Advanced Vector Extensions，高级矢量扩展)2. 0，它们都是采用 SIMD(Single Instruction Multiple Data，单指令多数据)机制，比如，MMX 的 64 位寄存器就可以同时存储 8 像素，意味着可以同时对这 8 像素进行处理，即用一条指令就实现了 8 个数据的运算，从而大大提高程序的运行效率。MMX、SSE、AVX 都是利用 CPU 内部的寄存器进行计算的，MMX 寄存器的宽度是 64 位，SSE 寄存器的宽度为 128 位，AVX 寄存器的宽度为 256 位。

采用 MMX 或者 SSE 实现 C/C++程序优化有两种方式：一种是嵌入式汇编的方式，需要将汇编代码嵌入 C/C++语句中，但这样的程序可读性很差；另外一种是内建函数的方式，可以像其他函数一样直接调用，这样程序可读性较好。这两种方式的执行效率是相等的，在 C/C++编程中，通常使用第二种方式。

【算法 3-4】　使用 SSE 实现积分图的快速计算

```
#include <nmmintrin.h>                                   //SIMD 指令集内建函数的头文件
void RmwDoSumGryImg_SSE( BYTE *pGryImg,                  //原始灰度图像
                         int width, int height,          //图像的宽度和高度
                         int *pSumImg                    //计算得到的积分图
                       )
{   //width 必须为 4 的倍数
    _declspec(align(16)) int sumCol[4096];               //约定图像宽度不大于 4096,16 字节对齐
```

```
    __m128i * pSumSSE,A;
    BYTE * pGry;
    int * pRes;
    int x, y;

    memset(sumCol, 0, sizeof(int) * width);
    for (y = 0, pGry = pGryImg, pRes = pSumImg; y < height; y++)
    {
        //0:需要特别处理
        sumCol[0] += * (pGry++);
        * (pRes++) = sumCol[0];
        //1
        sumCol[1] += * (pGry++);
        * (pRes++) = * (pRes - 1) + sumCol[1];
        //2
        sumCol[2] += * (pGry++);
        * (pRes++) = * (pRes - 1) + sumCol[2];
        //3
        sumCol[3] += * (pGry++);
        * (pRes++) = * (pRes - 1) + sumCol[3];
        //[4...width - 1]
        for (x = 4, pSumSSE = (__m128i * )(sumCol + 4); x < width; x += 4, pGry += 4)
        {
            //把变量的低 32 位(由 4 个 8 位整数组成)转换成 32 位的整数,即 4 个像素的灰度值
            A = _mm_cvtepu8_epi32(_mm_loadl_epi64((__m128i * )pGry));
            //4 个 32 位的整数相加,即同时更新 4 个列积分 sumCol
            * (pSumSSE++) = _mm_add_epi32( * pSumSSE, A);
            //递推
            * (pRes++) = * (pRes - 1) + sumCol[x + 0];
            * (pRes++) = * (pRes - 1) + sumCol[x + 1];
            * (pRes++) = * (pRes - 1) + sumCol[x + 2];
            * (pRes++) = * (pRes - 1) + sumCol[x + 3];
        }
    }
    return;
}
```

经实际测试,在某款 CPU 上,算法 3-4 比算法 3-3 的速度提高了约 2.8 倍,这是因为算法 3-4 使用了 SSE 内建函数,能够同时更新 4 个 sumCol 的值。内建函数是按照约定语法规则的函数,如果各家编译器支持该语法规则,则必须为使用者提供其函数,这些函数包含在编译器的运行库中,程序员不必单独编写代码,只需要调用这些函数即可,它们的实现由编译器厂商完成,比如在 Visual C++程序设计中,只需包含< nmmintrin.h >即可。

3.3 中值滤波

3.3.1 中值滤波的由来

仔细观察图 3-3 和图 3-4 就会发现,一个带有噪声的方波,经过均值滤波后却变成了一个斜波,滤波结果模糊了两类数值“3”和“9”之间的边界。从图 3-11 和图 3-12、图 3-13

和图 3-14 也看到均值滤波后的图像变模糊了。也就是说，均值滤波在去除噪声的同时，带来了很大的负面作用：它使得图像变模糊了。

如何解决这个问题呢？首先来分析一下变模糊的原因是什么。模糊的原因就是没有区分像素的类别，而将不同类别像素的灰度值进行了平均，比如没有区分是牛还是老鼠，而将牛的体重和老鼠的体重进行了累加，将其均值作为牛（老鼠）的均值滤波后的体重。那么，怎么区分是牛还是老鼠呢？难道在均值滤波以前还要先做分类？

体重在不同类别之间求平均是不对的（会生成不存在的物种），因此可以先在邻域内进行判断，如果牛的数量多就取牛的体重，如果老鼠的数量多就取老鼠的体重，是从里面挑出一个体重，而不是求均值去生成一个新的体重，这样就把分类的问题转换为谁多的问题。那么在邻域内是牛多还是老鼠多呢？显然，将它们的体重进行从小到大排序，谁多谁就出现在排序的最中间位置。比如，4 只老鼠 5 头牛，则排序后的第 5 号位置肯定是牛的体重；若 5 只老鼠 4 头牛，则排序后的第 5 号位置肯定是老鼠的体重。因此，取排序后位于最中间位置的体重作为滤波后的体重即可，这可能就是中值滤波的由来。

3.3.2　中值滤波的定义

【定义 3-3】 将 n（n 为奇数）个数据按其值 d_i 进行从大到小或者从小到大排列后得到一个有序序列 $d_0 d_1 \cdots d_{n-1}$，则 $d_{\lfloor \frac{n}{2} \rfloor}$ 称为中值。例如，有序序列 10，11，12，13，14，15，16，17，18 的 $n=9$，有 $\left\lfloor \frac{9}{2} \right\rfloor = 4$，则中值为 d_4，即 14。

【定义 3-4】 中值滤波（median filtering）就是以当前像素为中心取一个邻域，用该区域的所有像素灰度值的中值作为该像素滤波后的灰度值。

在前面的例子数据 D_1 中，取相邻 3 个像素作为邻域，其中值滤波的计算过程和结果如下：

原值	邻域值	中值
3	（邻域不完整，保持原值）	3
3	(3,3,3)	3
3	(3,3,9)	3
9	(3,9,3)	3
3	(9,3,3)	3
3	(3,3,9)	3
9	(3,9,9)	9
9	(9,9,9)	9
9	(9,9,3)	9
3	(9,3,9)	9
9	(3,9,9)	9
9	(9,9,9)	9
9	（邻域不完整，保持原值）	9

经过中值滤波以后，可以看出滤波后的结果完全去掉了噪声，这是因为像素受到噪声

污染后的灰度值在邻域内要么是最大值,要么是最小值,而不是中值。数据 D_1 的中值滤波结果如图 3-20 所示。

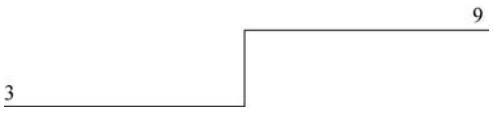


图 3-20 中值滤波后的结果

对图 3-13 取与图 3-14 相同大小的邻域进行中值滤波,得到的结果如图 3-21 所示。进一步增大滤波器的尺寸,得到图 3-22 和图 3-23。可以看出,随着滤波尺寸的变大,中值滤波得到的结果图像没有变得越来越模糊,黑色圆形区域的边界仍然清晰,且更加干净,说明噪声得到了滤除。这体现了中值滤波不会使图像变模糊,具有良好的边界清晰度保持能力。

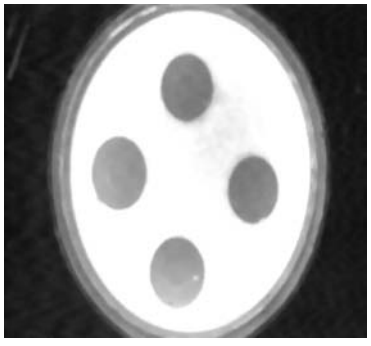


图 3-21 5×5 的中值滤波

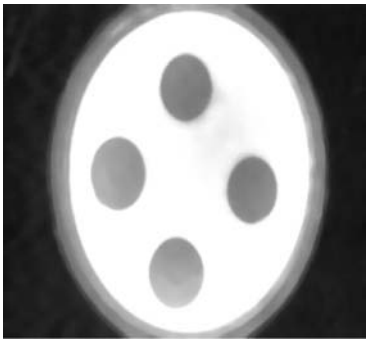


图 3-22 9×9 的中值滤波

但是当滤波器的大小增加到 21×21 时,黑色圆形区域却发生了严重的变形,如图 3-24 所示。这是因为随着邻域的增大,不能保证邻域内最多有 2 类目标,此时有了白色区域、黑色的圈、灰色的外环和黑色的背景,多达 4 类目标,不满足中值滤波的成立条件。

中值滤波在实际应用时,其邻域大小的选择要保证该邻域内最多有 2 类目标。至于受到噪声干扰的像素,其灰度值在邻域内要么最大,要么最小,一般不会是中值,所以会被滤除。当邻域不大时,可以认为邻域内最多有两类目标。

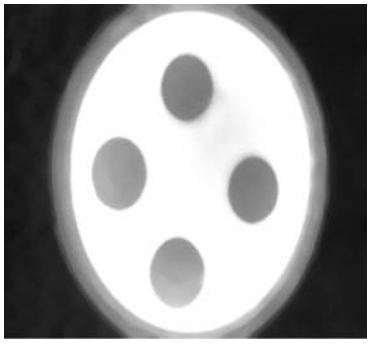


图 3-23 11×11 的中值滤波

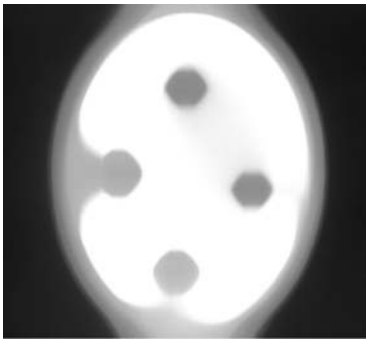


图 3-24 21×21 的中值滤波

3.3.3　中值滤波的特点

中值一定是邻域内某个像素的灰度值而不是某几个像素灰度的生成值(相比于它,均值就像是一个伪造的值),所以中值滤波不会使图像变模糊,这是它的优点;但是,中值滤波需要排序,而排序的复杂度远比相加求和大得多,所以中值滤波的速度要比均值滤波慢很多,这是它的缺点。其实,在图像处理中,排除编程技巧等因素,几乎可以说算法越复杂,则执行速度越慢,处理效果越好。

3.3.4　中值滤波的快速实现

如上所述,中值滤波需要排序,学过“数据结构”的人都知道在排序操作上花费时间非常多。假设邻域大小为 101×101,即 10 201 个像素,即使采用快速排序算法对它们的灰度值进行排序,按照快速排序时间复杂度 $O(n\mathrm{lb}n)$ 计算,也至少需要 135 840 次比较。求一个像素的邻域中值都需要十多万次的比较,这对于一幅动辄有几百万像素的图像而言,其速度之慢是无法忍受的。

回顾一下第 2 章的直方图,通过直方图很容易求出一个邻域的最大灰度值、最小灰度值和均值。如果给邻域建立一个灰度直方图,那么是否可以通过直方图求出这个邻域的灰度中值呢?比如一个班有 55 个同学考高等数学,有 55 个成绩,哪个成绩是中值呢?显然是在这个成绩之前有 27 个人。因此可以用一个计数器,从 0 分开始计数,若计数到某个成绩时计数值大于 27,则这个成绩就是中值。

【算法 3-5】　使用直方图求灰度中值

```
void GetMedianGry(int * histogram, int N, int * medGry)     //N 是像素总个数
{
    int g;
    int num;

    // step.1 -------------- 求灰度中值 -------------------------- //
    num = 0;
    for (g = 0; g < 256; g++)
    {
        num += histogram[g];
        if (2 * num > N) break; //num > N/2
    }
    * medGry = g;
    // step.2 -------------- 结束 -------------------------------- //
    return;
}
```

利用直方图中数据本身的有序特性,使得求图像的中值非常简单,大大减少了比较次数。从算法 3-5 可以看出,利用直方图求中值最多进行 256 次比较,因此可以利用直方图来快速得到中值。

此外,通过图 3-15 和图 3-16 可以发现,相邻像素的邻域有很大的交集区域,因此在求一个像素的邻域直方图时,若能利用其前一个像素的邻域直方图,就可以大大减少建立

直方图的时间。比如,中值滤波在同一行中是从左向右进行的,当从像素 A 向右移动一个像素到达像素 B 时,B 的邻域直方图就是 A 的邻域直方图丢掉其最左边列的像素灰度值,再增加一个新右边列的像素灰度值。

使用直方图进行中值滤波的算法如下。

【算法 3-6】 使用直方图的快速中值滤波

```
double RmwMedianFilter( BYTE * pGryImg, int width, int height,
                        int M, int N,          //滤波邻域: M 列 N 行
                        BYTE * pResImg
                       )
{  //每行建一个直方图,没有进行相邻行直方图递推和图像边界上像素变窗口等处理
    BYTE * pCur, * pRes;
    int halfx,halfy,x, y, i, j, y1, y2;
    int histogram[256];
    int wSize, j1, j2;
    int num, med, v;
    int dbgCmpTimes = 0;                   //搜索中值所需比较次数的调试

    M = M/2 * 2 + 1;                       //奇数化
    N = N/2 * 2 + 1;                       //奇数化
    halfx = M/2;                           //x 半径
    halfy = N/2;                           //y 半径
    wSize = (halfx * 2 + 1) * (halfy * 2 + 1); //邻域内像素总个数
    for (y = halfy, pRes = pResImg + y * width; y < height - halfy; y++)
    {
        //step.1 ---- 初始化直方图(每一行)
        y1 = y - halfy;
        y2 = y + halfy;
        memset(histogram, 0, sizeof(int) * 256);
        for (i = y1, pCur = pGryImg + i * width; i <= y2; i++, pCur += width)
        {
            for (j = 0; j < halfx * 2 + 1; j++)
            {
                histogram[ * (pCur + j)]++;
            }
        }
        //step.2 ----- 初始化中值(每一行)
        num = 0;                           //记录着灰度值从 0 到中值的个数
        for (i = 0; i < 256; i++)
        {
            num += histogram[i];
            if (num * 2 > wSize)
            {
                med = i;
                break;
            }
        }
        //滤波
```

```
            pRes += halfx;                          //没有处理图像左边界侧的像素
            for (x = halfx; x < width - halfx; x++)
            {
                //赋值
                *(pRes++) = med;
                //step.3 --- 直方图递推:减去当前邻域最左边的一列,添加邻域右侧的一个新列
                j1 = x - halfx;                     //最左边列
                j2 = x + halfx + 1;                 //右边的新列
                for (i = y1, pCur = pGryImg + i * width; i <= y2; i++, pCur += width)
                {
                                                    //减去最左边列
                    v = *(pCur + j1);
                    histogram[v]--;                 //更新直方图
                    if (v <= med) num--;            //更新 num
                    //添加右边的新列
                    v = *(pCur + j2);
                    histogram[v]++;                 //更新直方图
                    if (v <= med) num++;            //更新 num
                }
                //step.4 ----- 更新中值
                if (num * 2 < wSize)                //到上次中值 med 的个数少了,则 med 要变大
                {
                    for (med = med + 1; med < 256; med++)
                    {
                        dbgCmpTimes += 2;           //总的比较次数,调试用
                        num += histogram[med];
                        if (num * 2 > wSize) break;
                    }
                    dbgCmpTimes += 1;               //总的比较次数,调试用
                }
                else                                //到上次中值 med 的个数多了,则 med 要变小
                {
                    while ((num - histogram[med]) * 2 > wSize)          //若减去后,仍变小
                    {
                        dbgCmpTimes++;              //总的比较次数,调试用
                        num -= histogram[med];
                        med--;
                    }
                    dbgCmpTimes += 2;               //总的比较次数,调试用
                } //end of x
            }
            pRes += halfx;                          //没有处理图像右边界侧的像素
        } //end of y
        //返回搜索中值需要的平均比较次数
        return dbgCmpTimes * 1.0/((width - halfx * 2) * (height - halfy * 2));
    }
```

step.4 中有一个用来搜索中值的循环,其比较次数不固定。但是,从中值滤波的实践来看,平均搜索次数非常有限(一般就在 2～3,极个别图像会达到 9 左右),跟邻域大小

没有太直接的关系。比如邻域大小为 101×101,即有 10 201 个像素时,相邻两个像素的邻域有 10 201−101=10 100 个共同的像素,不会因为 101 个像素的变化就引起中值的剧烈变化,所以 step.4 中的循环执行次数是很少的(分析一下 dbgCmpTimes 的值就能知道),相比而言,step.3 的直方图递推浪费的时间更多。

3.4 极值滤波

分析均值滤波和中值滤波,可以知道它们分别是取一个数据集合(邻域内灰度值的集合)的均值和中值。对于一个数据集合而言,该数据集合除了有均值和中值外,还有 2 个最直观的特性,分别是最小值和最大值,那么最大值和最小值是否可以用于滤波呢?

如果事先能够知道噪声像素的灰度值是比周围像素的灰度值大,那么就可以使用邻域内的最小值来替代当前像素的灰度值,称为最小值滤波(minimum filtering)。反之,称为最大值滤波(maximum filtering)。

图 3-25 是带有反光的水面图像,若把反光当作噪声,那么采用最小值滤波应该可以去掉反光。图 3-26 是对图 3-25 使用 5×5 最小值滤波的结果,可以看到反光像素基本被去掉了。

图 3-25 原始图像

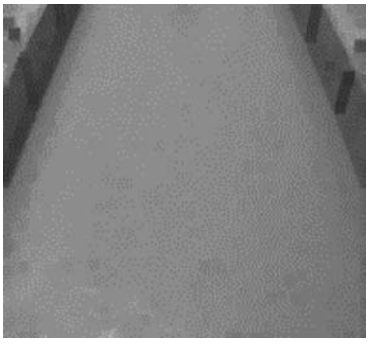


图 3-26 最小值滤波结果

3.5 高斯滤波

式(3-1)是对 3 个像素的灰度值求平均值,将其按照式(3-4)的模板形式进行表示,得到式(3-11),对应的模板如图 3-27 所示。图 3-28 虽然是 5 邻域均值滤波的模板,但是该模板中最左侧和最右侧的值为“0”,相当于没有起作用,它实际上就是 3 邻域的均值滤波。

$$G(x)=\frac{\sum_{j=-1}^{1} g(x+j)\times T(j)}{\sum_{j=-1}^{1} T(j)} \tag{3-11}$$

1	1	1

图 3-27　3 邻域均值滤波

0	1	1	1	0

图 3-28　5 邻域均值滤波

对于图 3-28 表示的模板，可以这样认为：因为该邻域内的最左侧像素、最右侧像素到中心像素的距离远了，所以把它们对 $G(x)$ 的贡献度设为 0，令它们对均值没有贡献。因此，模板中的“1”和“0”可以看成是权重，将图 3-27 转换为权重函数的形式即得到图 3-29。

简单地将权重设置成要么“1”要么“0”，没有柔和地体现“近处贡献度大、远处贡献度小”的特性。那么，用什么函数能够更好地体现贡献度的变换呢？均值为 0 的高斯函数是一个常用的权重函数，如图 3-30 所示。其表达式为：

$$T(j)=\frac{1}{\sqrt{2\pi\delta^2}}e^{\frac{-j^2}{2\delta^2}} \tag{3-12}$$

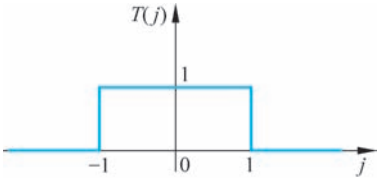


图 3-29　3 邻域均值滤波的权重函数

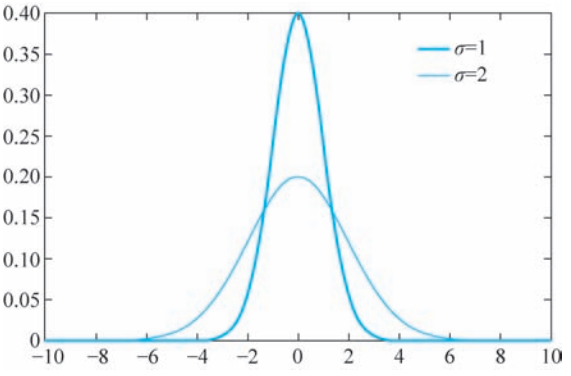


图 3-30　高斯函数

式(3-12)中，σ 为高斯函数的标准差。当 j 距离原点越远时，$T(j)$ 值越小。由于该函数有点复杂，不容易直观地看出它的取值，因此表 3-1 给出了 j 在不同取值下的 $T(j)$ 值。

表 3-1　高斯函数数值表

j	$e^{\frac{-j^2}{2\delta^2}}$ 的值	$\sigma=1$ 时 $T(j)$ 的值	$\sigma=2$ 时 $T(j)$ 的值
0	1.000 000	0.282 095	0.141 047
0.5σ	0.882 497	0.248 948	0.124 474
1.0σ	0.606 531	0.171 099	0.085 550
1.5σ	0.324 652	0.091 583	0.045 791
2.0σ	0.135 335	0.038 177	0.019 089
2.5σ	0.043 937	0.012 394	0.006 197
3.0σ	0.011 109	0.003 134	0.001 567

从表 3-1 可以看出，就 $e^{\frac{-j^2}{2\delta^2}}$ 的值而言，距离中心像素 3 倍 σ 距离的像素的权重 0.011 109 比中心像素的权重 1.0 降低了约 1/100，已经降到了可以忽略的程度。因此，使用高斯函

数加权均值滤波时,邻域的半径一般定为 3σ,半径超过 3σ 时,对滤波结果的影响不大。比如,当 $\sigma=1$,$j=3$ 时,$T(j)$ 的值为 0.003 134,即使像素 $g(x+j)$ 的值为最大值 255,$g(x+j)\times T(j)$ 的值也小于 1.0,可以忽略不计了。滤波邻域的半径和标准差 σ 具有一一对应的关系。

习惯上把使用高斯函数作为权重函数的均值滤波,该滤波称为高斯滤波(Gaussian average filtering)或者高斯平滑。同其他均值滤波一样,高斯滤波也会导致结果图像变模糊,所以有时也称为高斯模糊(Gaussian blur)。

图 3-30 和式(3-12)是一维的高斯函数,图像滤波的邻域一般是二维的,此时应采用二维高斯函数,如式(3-13)所示。

$$T(i,j)=\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-(j^2+i^2)}{2\sigma^2}} \tag{3-13}$$

高斯滤波的特点是:当 σ 越大时,高斯函数也越平缓,如图 3-30 所示;当 σ 越大时,邻域的大小也越大;邻域大小越大,就有更多的像素参加滤波,图像模糊得也就越突出,如图 3-31 和图 3-32 所示。

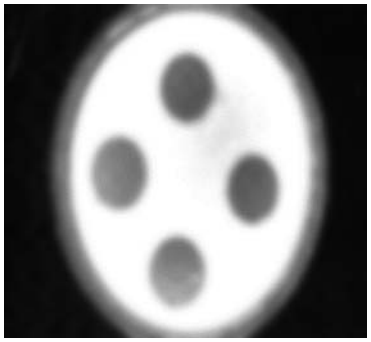


图 3-31 高斯滤波(半径为 4)

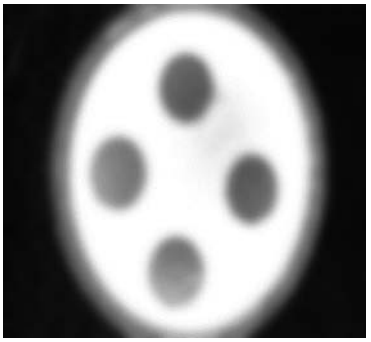


图 3-32 高斯滤波(半径为 5)

在实际应用中,为了快速计算,二维的高斯滤波可以被分解为两个一维的高斯滤波,比如先对图像按行进行一维的高斯滤波,再对得到的结果图像按列进行一维的高斯滤波。另外,为了提高高速缓存的命中率,从而提高速度,还可以把行滤波得到的结果图像进行 90°转置,把按列进行的一维高斯滤波变为按行的一维高斯滤波,最后再把得到的最终结果图像转置回来。

3.6 二值图像滤波与数学形态学滤波

3.6.1 基于均值滤波的二值图像滤波

前面以灰度图像为例,讲述了均值滤波、中值滤波、极值滤波以及高斯滤波,对于彩色图像的滤波完全可以按照灰度图像滤波的方法进行,只不过彩色图像中每个像素有 3 个分量而已。

那么对于二值图像有没有特别的滤波方法？首先来分析一下二值图像的特点。

(1) 二值图像只有两种灰度值，比如分别是 0 或者 255。

(2) 二值图像的灰度最小值和灰度最大值不用求取，最小值为 0，最大值为 255。

(3) 二值图像可以进行均值滤波，但是不能在结果图像中给像素赋值为 0 和 255 以外的其他灰度值。

(4) 二值图像可以进行中值滤波，但是肯定不用进行排序。在一个邻域内，若 0 的个数多，中值就是 0，若将其当作灰度图像，此时邻域内灰度均值小于 128；反之，中值就是 255，若将其当作灰度图像，此时邻域内灰度均值大于或等于 128。

基于以上特点，本书作者给出了一个基于均值滤波思想的二值图像最小值、最大值和中值滤波的方法(见算法 3-7)，其基本思想是二值图像当作灰度图像处理，具体如下。

(1) 当邻域内灰度均值大于或等于 255 时，赋值 255；否则，赋值 0。这就是最小值滤波。

(2) 当邻域内灰度均值大于 0 时，赋值 255；否则，赋值 0。这就是最大值滤波。

(3) 当邻域内灰度均值大于或等于 128 时，赋值 255；否则，赋值 0。这就是中值滤波。

【算法 3-7】 基于均值滤波的二值图像滤波

```
void RmwBinImgFilter( BYTE * pBinImg,                    //原始二值图像
                      int width, int height,             //图像的宽度和高度
                      int M, int N,                      //滤波邻域：M 列 N 行
                      double threshold,                 //灰度阈值,大于或等于该值时结果赋 255
                      BYTE * pResImg                     //结果图像
                     )
{   //没有对边界上邻域不完整的像素进行处理,可以采用变窗口的策略
    BYTE * pAdd, * pDel, * pRes;
    int halfx, halfy;
    int x, y, sum, sumThreshold;
    int sumCol[4096];                                    //约定图像宽度不大于 4096

    //step.1 ------------ 初始化 ---------------------------- //
    M = M/2 * 2 + 1;                                     //奇数化
    N = N/2 * 2 + 1;                                     //奇数化
    halfx = M/2;                                         //滤波器的 x 半径
    halfy = N/2;                                         //滤波器的 y 半径
    sumThreshold = max(1,(int)(threshold * M * N));      //转换为邻域内灰度值之和的阈值
    memset(sumCol, 0, sizeof(int) * width);
    for (y = 0, pAdd = pBinImg; y < N; y++)
    {
        for (x = 0; x < width; x++) sumCol[x] += * (pAdd++);
    }
    //step.2 ------------ 滤波 ---------------------------- //
    for(y = halfy, pRes = pResImg + y * width,pDel = pBinImg; y < height - halfy; y++)
    {
        //初值
        for (sum = 0,x = 0; x < M; x++) sum += sumCol[x];
        //滤波
        pRes += halfx;                                   //跳过左侧
        for (x = halfx; x < width - halfx; x++)
```

```
            {
                //求灰度均值
                *(pRes++) = (sum >= sumThreshold) * 255; //理解这个表达式的含义
                //换列,更新灰度值的和
                sum -= sumCol[x - halfx];                //减去左边列的灰度值
                sum += sumCol[x + halfx + 1];            //加上右边列的灰度值
            }
            pRes += halfx;                               //跳过右侧
            //换行,更新 sumCol
            for (x = 0; x < width; x++)
            {
                sumCol[x] -= *(pDel++);                  //减去上一行的灰度值
                sumCol[x] += *(pAdd++);                  //加上下一行的灰度值
            }
        }
        //step.3 ------------ 返回 ---------------------------- //
        return;
    }
```

这是一个非常简洁、快速的二值图像的最小值滤波、最大值滤波和中值滤波方法,仅需要改动 threshold 的值就能实现,且通过简单改变 threshold 的值就能实现比例滤波,比如取 64 时就是相当于邻域内有 25%像素为 255,滤波结果就是 255。所谓比例滤波,就是排序后的某个特定位置的值,比如最小值滤波相当于是 0%,中值滤波相当于是 50%,最大值滤波相当于是 100%。

3.6.2 二值图像的数学形态学滤波

对于二值图像的滤波,还有一种称为"数学形态学"(mathematical morphology)的滤波方法,简称形态学滤波。下面不具体讲述数学形态学的内涵,只讲一下形态学滤波的基本运算。

1. 膨胀与腐蚀

二值图像一般都是原始图像经过某种处理后得到的,它的像素具有明确的含义,比如黑色像素(以灰度值 0 表示)代表背景,白色像素(以灰度值 1 表示)代表目标,而目标往往具有某种几何形状,即具有某种形态。形态学滤波就是以目标形状为出发点来进行滤波的,并发展出了膨胀、腐蚀、开运算、闭运算等方法。比如,图 3-33 所示的白色目标内部有 1 个黑色孔洞(用带下画线的 0 代表),怎么把黑洞去掉呢?这时就可用膨胀(dilation)运算,如式(3-14)所示。

$$G(x,y)=(g\oplus T)(x,y)=\mathrm{OR}[g(x+j,y+i)\&T(j,i)] \tag{3-14}$$

其中,$\frac{-m}{2}\leqslant j\leqslant\frac{m}{2}$,$\frac{-n}{2}\leqslant i\leqslant\frac{n}{2}$,$T$ 类似图 3-6 所示,因为其具有某种结构,所以在形态学滤波中把这些模板称为结构元素(structure element)。式(3-14)与式(3-4)相比,就是把乘法换成了 & 运算,把求和换成了 OR 运算。因为 $G(x,y)$ 只能取 0 和 1,g 和 T 也都是只有 0 和 1 两种值,所以只要有一次 $g(x+j,y+i)\&T(j,i)$ 为 1,则 $G(x,y)$ 就取 1。比如,当 T 取图 3-6 中的值时,图 3-33 的结果就是图 3-34。对于图 3-33 中带下画线的 0 像素而言,其邻域与结构元素 T 做 & 运算时,能得到 4 个 1,位置如图 3-33 中带下画线的 1

所示，所以在结果图像图 3-34 中其值为 1。图 3-34 中多出了很多带下画线所示的 1，它们就是这些原本在原始图像图 3-33 中为“0”的像素，其邻域与结构元素 T 做 & 运算时有 1，所以滤波后变成了 1。

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

图 3-33　原始二值图像

0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

图 3-34　膨胀后的结果图像

相反地，如果白色目标外部有几处白色粘连，怎么把这些粘连去掉呢？这时就可用腐蚀运算(erosion)，如式(3-15)所示。

$$G(x,y)=(g\otimes T)(x,y)=\mathrm{AND}[g(x+j,y+i)\&T(j,i)] \tag{3-15}$$

腐蚀就是说，只有该像素邻域与结构元素 T 做 & 运算全部为 1 时，$G(x,y)$才为 1，只要一个为 0，$G(x,y)$就为 0。

其实，如果把 T 看成最大值滤波的邻域，所谓膨胀就是二值图像的最大值滤波。所谓膨胀运算就是因为二值图像中像素的值只有两种，所以对二值图像处理采用布尔代数逻辑运算，在布尔代数中，& 就相当于乘法，OR 就相当于加法。同理，如果把 T 看成最小值滤波的邻域，所谓腐蚀就是二值图像的最小值滤波。

2. 闭运算与开运算

图 3-35 和图 3-36 中都有正方形和圆形的目标。图 3-35 中目标内部有破损，图 3-36 中目标外部有粘连，如何进行处理呢？

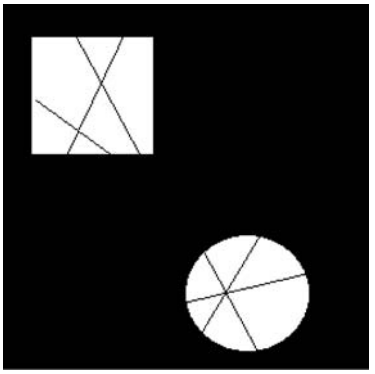


图 3-35　内部有破损的目标

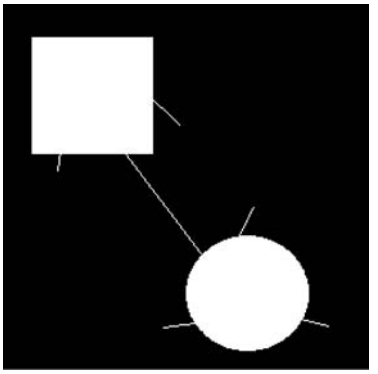


图 3-36　外部有粘连的目标

显然,对于图 3-35 可以采用 3×3 的方形结构元素进行膨胀运算,但膨胀后,目标被放大了一圈;因此,再对结果进行 3×3 的方形结构元素的腐蚀运算,再把目标缩小一圈;最终得到理想的结果。在形态学滤波中,把“先膨胀再腐蚀”的组合运算称为闭运算(close),闭运算的意思就是闭合目标内部的孔洞。

显然,对于图 3-36 可以采用 3×3 的方形结构元素进行腐蚀运算,但腐蚀后,目标被缩小了一圈;因此,再对结果进行 3×3 的方形结构元素的膨胀运算,再把目标放大一圈;最终得到理想的结果。在形态学滤波中,把“先腐蚀再膨胀”的组合运算称为开运算(open),开运算的意思就是分离开互相粘连的目标。

形态学滤波有一套较为晦涩的符号体系,不再细讲。闭运算和开运算的公式也不再细写。可以认为形态学滤波就是使用布尔代数和集合运算的二值图像滤波,形态学滤波能做到的,常规的滤波方法也能做到。在本质上,空间域滤波就是邻域的选定和邻域内计算函数的选定,形态学滤波和常规的滤波方法是一致的。

3.7 条件滤波

通过以上的滤波方法,像素(x,y)得到了新的灰度值$G(x,y)$。在图像实践中,是不是一定要用新灰度值替换原来的灰度值$g(x,y)$?如果不是,那么满足什么条件就替换成$G(x,y)$,否则保持原值$g(x,y)$呢?再者,假设邻域有A个像素,滤波运算是否一定要全部使用这A个像素?还有,像素能不能使用多个不同的邻域,从而得到多个滤波的结果,然后从中挑选一个最好的结果作为$G(x,y)$?本书把诸如此类带有一定条件的、带有选择运算的滤波,统一称为条件滤波。

常用的条件滤波方法主要有超限平滑、K个邻点平均法、多邻域枚举法均值滤波,下面讲述它们的基本原理。

3.7.1 超限平滑

所谓超限平滑,就是均值滤波得到的值$u(x,y)$和原值$g(x,y)$相比,超过了一定的程度C才使用新值,否则保持原值。超限平滑的表示如下:

$$G(x,y)=\begin{cases}u(x,y), & |u(x,y)-g(x,y)|\geqslant C\\ g(x,y), & \text{其他}\end{cases} \tag{3-16}$$

比如,在 3.2.1 节对D_1进行邻域为 3 个像素的均值滤波中,如果令$C=4$,则噪声“9”和“3”会被完美地移除。

此种情况使用超限平滑的合理性在于,在摄像系统拍摄得到的图像中,目标和背景之间至少存在着 1 个像素的过渡区,即目标和背景的边界像素灰度值是由目标的灰度逐渐变化到背景的灰度的。边界上的像素的灰度值$g(x,y)$与其滤波均值$u(x,y)$是接近的,而噪声的灰度值则与均值有较大的差异。

还有另外一种情况,比如进一步观察消除水面反光的图 3-26,会发现反光被消除了,但是左右两侧的墙壁区域明显变黑了。如何才能只消除水面的反光,而不改变其他像素的值呢?采用式(3-17)即可。

$$G(x,y)=\begin{cases}\min(x,y), & g(x,y)-\min(x,y)\geqslant C\\ g(x,y), & \text{其他}\end{cases}\tag{3-17}$$

其中，$\min(x,y)$是采用最小值滤波得到的值。使用超限平滑对图 3-25 进行滤波得到结果图像如图 3-37 所示，图 3-38 中的黑色像素代表这些像素满足条件 $g(x,y)-\min(x,y)\geqslant C$。

图 3-37　超限最小值滤波

图 3-38　被滤波的像素

3.7.2　K 个邻点平均法

所谓 K 个邻点平均法就是不使用邻域的全部像素，而是只使用其中的 K 个像素求均值；假设邻域中有 A 个像素，则 $K<A$。

在 A 个像素中，到底选取那 K 个呢？有一个基本原则就是选取与当前像素的灰度值最接近的那 K 个像素。比如，在 3.2.1 节对 D_1 进行邻域为 3 个像素的均值滤波中，如果令 $K=2$，则噪声“9”和“3”会得到减弱，且非噪声像素不受影响。

此种情况使用 K 个邻点平均法的合理性在于，因为噪声的灰度值跟正常像素的灰度值不接近，所以噪声的灰度值能被其周围（即邻域）的灰度值修改掉；而目标像素或者背景像素的邻域内，肯定有多个同类像素，设为 K 个，用这 K 个同类像素求均值，肯定不会产生模糊。

在邻域内选择与当前像素的灰度值最接近的 K 个灰度值，肯定需要比较操作，因此这个滤波器可以看成既有中值滤波的排序操作，又有均值滤波的求和操作。

3.7.3　多邻域枚举法均值滤波

在邻域内选择 K 个像素是不太容易的。不妨认为，在一个邻域内，属于同一类别 K 个像素的分布形式一共有 N 种，如果把每一种都作为一个模板，就是 N 个模板。这样每个模板都得到一个均值，总共得到了 N 个均值，从而从这 N 个均值中选择一个最佳模板的均值作为滤波结果。

如何选择最佳模板呢？显然，若某个模板内包含中心像素的 K 个像素都来自相同类，则该模板肯定是最佳的。如何评价模板内的像素来自相同类呢？显然模板内像素灰度值的均方差越小，则说明该模板越具有灰度一致性，即越有可能不包含边缘和不包含异类目标，因此该模板就越佳。

上述做法是同时采用 N 种邻域,因此本书中将其称为多邻域枚举法均值滤波。一种常用的多邻域枚举法的模板如图 3-39 所示。

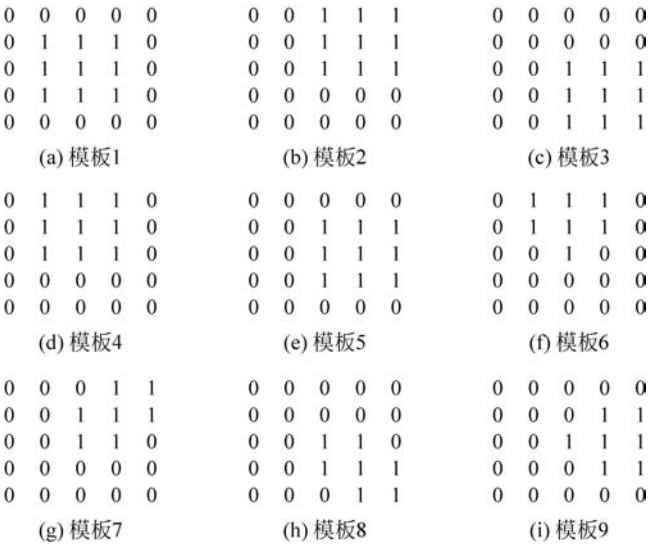


图 3-39 多邻域枚举法均值滤波模板

该系列共有 9 个模板,它们从 5×5 邻域的 25 个像素分别选取 9 个像素和 7 个像素。每个模板分别代表的含义是比较清楚的,在此不再赘述。

3.8 本章小结

本章讲述了均值滤波、中值滤波、最小值滤波、最大值滤波、高斯滤波、二值图像滤波、形态学滤波等常用滤波方法,讲述了它们的思想、特点以及 C/C++ 编程优化,包括“整数除法或者浮点法乘法和除法变为整数乘法和移位”“列积分”“积分图”“MMX 及 SSE”“直方图求中值”“直方图递推”“基于均值滤波的二值图像滤波”等技巧。

通俗地说,这些滤波器就是把像素特定邻域内的灰度值构成一个数据集合,使用该集合的某种取值来替换该像素原来的灰度值。邻域一般又称为模板,模板有大小、形状和取值等特性,取值可按位置加权,比如使用高斯函数加权。

另外,可以把图像中每个像素邻域内的灰度值构成集合,来进行滤波,称为像素级滤波,本章给出的算法程序都是像素级滤波;也可以把图像序列中具有相同坐标位置的不同帧上像素的灰度值构成集合来进行滤波,称为帧级滤波,在运动目标检测中,常用帧级滤波求取背景图像;也可以在同一幅图像中,给每个像素取个较大的邻域当作图像块,在本图中搜索与之相似的多个图像块,用这些块的中心像素灰度值构成集合来进行滤波,滤波的结果替换该像素的灰度值,称为块级滤波(non-local filter,简称非邻域滤波)。

在实际应用中可根据具体情况,设计出各种复杂形状的滤波器;可以采用各种各样的邻域,甚至可以是从多个候选的邻域中选取一个最合理的邻域,可从邻域中选取全部或

者部分的像素，计算它们的均值、中值、极值等，甚至可以是排序后的某个特定位置的值（简称比例滤波）；还可多个滤波器组合使用，以及条件滤波等。

深刻理解图像空间域平滑的实质和各种滤波方法的特点，掌握编程优化方法，并灵活运用，尤其是多种滤波方法的组合使用，使得滤波器之间能够取长补短，在应用中往往能起到事半功倍的效果。

作业与思考

3.1　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？

3.2　对一个图像 A 作 3×3 的均值滤波得到图像 B，对图像 B 再做 3×3 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近似得到图像 C？

3.3　在图像处理中，彩色到灰度转换公式为：gry＝0.299×red＋0.587×green＋0.114×blue，用 C/C++ 语言编程把图像 H0301Rgb.bmp（见图 3-40）转换为灰度图像 H0301Gry.bmp，分别使用“整数除法或者浮点数乘法和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()）。

3.4　参照算法 3-1，采用 C/C++ 语言编程，实现对于灰度图像 H0302Gry.bmp（见图 3-41）每行上的一维均值滤波（邻域大小为 1 行 M 列，M＝21）。

3.5　参照算法 3-2，采用 C/C++ 语言编程，实现对于灰度图像 H0302Gry.bmp（见图 3-41）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。

图 3-40　H0301Rgb.bmp

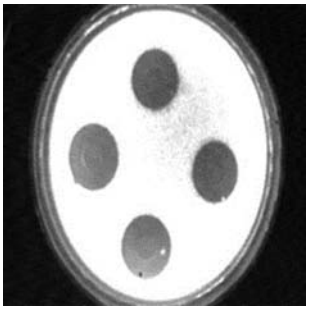


图 3-41　H0302Gry.bmp

3.6　使用 MMX 或者 SSE 指令集，用 C/C++ 语言编程实现灰度图像 H0302Gry.bmp（见图 3-41）的反相。

3.7　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。

3.8 采用高速摄像机(2000f/s)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,如何才能得到一幅没有炮弹的场景图像(一般称为背景图像)?

3.9 $\boldsymbol{T}^3=\begin{bmatrix}0&1&0\\1&1&1\\0&1&0\end{bmatrix}$ 和 $\boldsymbol{T}^5=\begin{bmatrix}0&0&0&0&0\\0&0&1&0&0\\0&1&1&1&0\\0&0&1&0&0\\0&0&0&0&0\end{bmatrix}$ 等价吗?

3.10 $\boldsymbol{T}^3=\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$ 有什么特点?使用它进行图像平滑需要乘法和除法运算吗?

3.11 使用标准差等于 1.0 的高斯函数计算一维卷积模板 $T(j)(-3\leqslant j\leqslant 3)$,得到 $T=\{0.003\,134,0.038\,177,0.171\,099,0.282\,095,0.171\,099,0.038\,177,0.003\,134\}$,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?

3.12 使用 C/C++语言编程,对灰度图像 H0303Gry.bmp(见图 3-42)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。

3.13 算法 3-7 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?

3.14 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 D_1 进行邻域为 3 像素的均值滤波,写出滤波结果。

3.15 使用 Photoshop 中的 Median Filter 或者 Gaussian Blur,估计拍摄灰度图像 H0304Gry.bmp(见图 3-43)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。

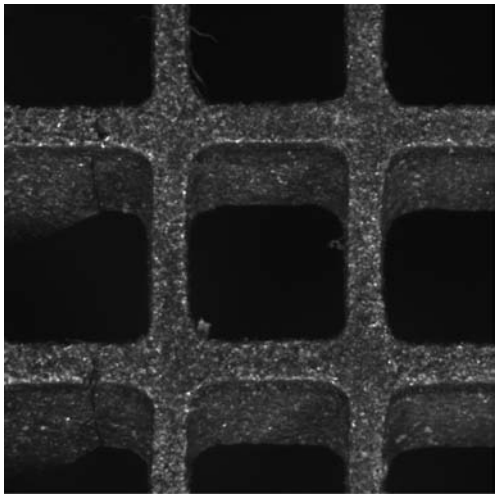


图 3-42 H0303Gry.bmp

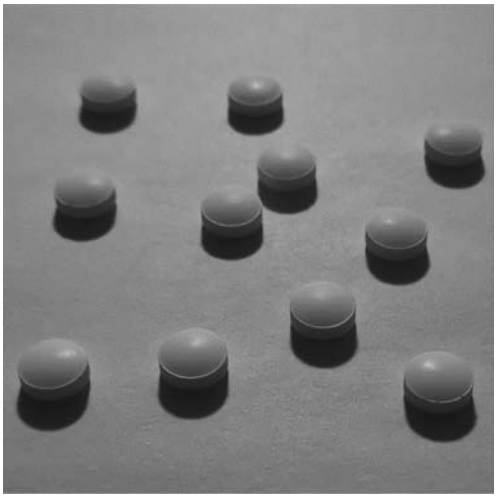


图 3-43 H0304Gry.bmp