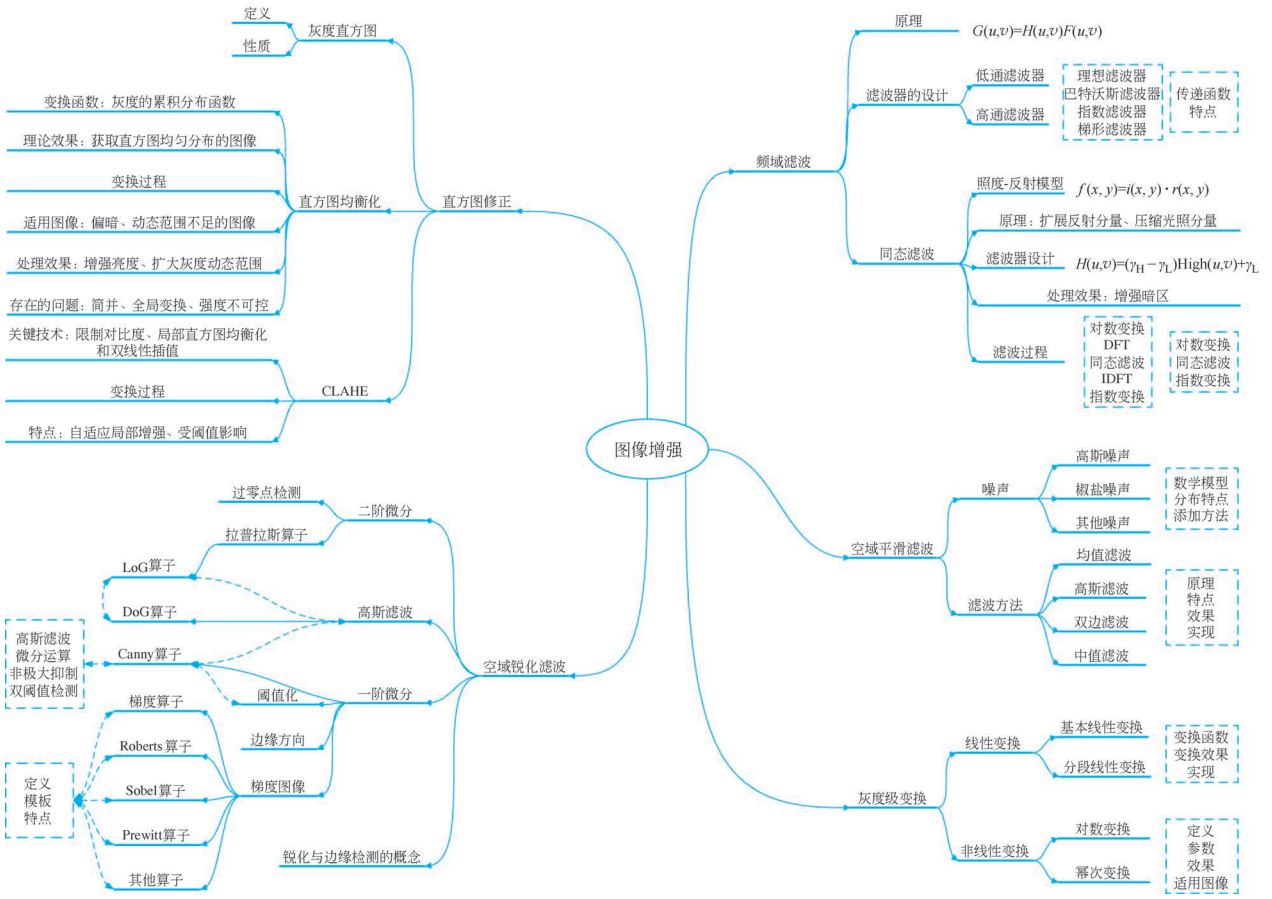
第5章 图像增强

CHAPTER 5

本章思维导图



图像增强(Image Enhancement)是将一幅图像中的有用信息(即感兴趣信息)进行增强，同时抑制无用信息(即干扰信息或噪声)，改善图像质量以增强图像的视觉效果，或者增强图像的感兴趣部分以利于计算机处理。典型的图像增强技术有基于灰度级变换、直方图修正的对比度增强，有抑制图像中噪声的图像平滑，有增强图像中细节或边缘的图像锐化等。由于图像平滑和锐化常采用滤波的方式进行，也称为滤波处理，可以在空间域进行，也可以在频域进行。随着技术的发展，一些新型技术被用于图像增强处理，如模糊增强、基于人类视觉的增强等；图像增强处理也被用于特定情形下的图像，并衍生出一系列的新方法，如去雾增强、低照度图像增强等。本章主要讲解典型的图像增强算法及其仿真实现。

5.1　灰度级变换

灰度级变换就是借助变换函数将输入的像素灰度值映射成一个新的输出值，通过改变像素的亮度值来增强图像，如式(5-1)所示。

$$g(x,y)=T[f(x,y)] \tag{5-1}$$

其中，$f(x,y)$是输入图像，$g(x,y)$是变换后的输出图像，T 是灰度级变换函数。由于灰度级变换一般是将过暗的图像灰度值进行重新映射，扩展灰度级范围，使其分布在整个灰度值区间，又称为扩展。

由式(5-1)可看出，变换函数 T 的不同将导致不同的输出，其实现的变换效果也不一样。因此，在实际应用中，可以通过灵活地设计变换函数 T 来实现各种处理。

5.1.1　线性灰度级变换

线性灰度级变换指变换前后灰度级呈现线性关系，方法简便，易于理解，主要用于调整亮度、对比度。

1. 基本线性灰度级变换

最基本的线性灰度级变换如式(5-2)所示。

$$g(x,y)=f(x,y)\cdot\tan\theta \tag{5-2}$$

变换效果由变换函数的倾角 θ 所决定：当 $\theta=45°$，图像灰度无变化，如图 5-1(a)所示；当 $\theta<45°$，变换后灰度取值范围压缩，灰度值降低，图像均匀变暗，如图 5-1(b)所示；当 $\theta>45°$，变换后灰度取值范围拉伸，灰度值增大，图像均匀变亮，如图 5-1(c)所示。因此，可以根据图像的亮度，选择不同的倾角实现不同的处理效果。图 5-1 中 L 表示灰度级数目。

如果变换函数不经过原点，线性灰度级变换表示为

$$g(x,y)=\alpha f(x,y)+\beta \tag{5-3}$$

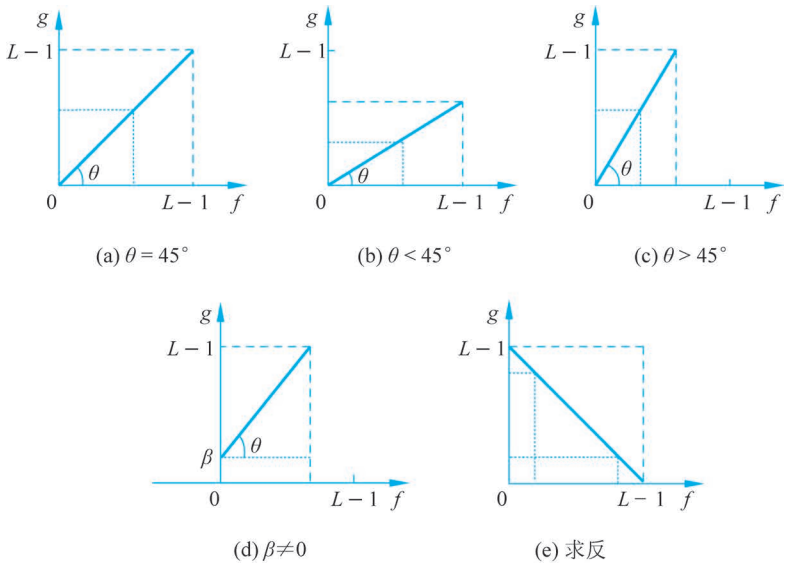
即 $\alpha=\tan\theta$，β 为偏移，变换函数如图 5-1(d)所示。式(5-2)其实是式(5-3)中 $\beta=0$ 的情况。

当灰度级变换函数如图 5-1(e)所示时，图像高低灰度值反转，即暗变亮、亮变暗。

【例 5.1】 编写程序，设置线性变换函数倾角 θ 和偏移 β，对灰度图像进行线性灰度级变换。

解： 图像采用矩阵表示，设置 θ 和 β 后，直接按照式(5-2)和式(5-3)对矩阵进行运算即可。程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('couple.bmp')
Image = Image / 255
```



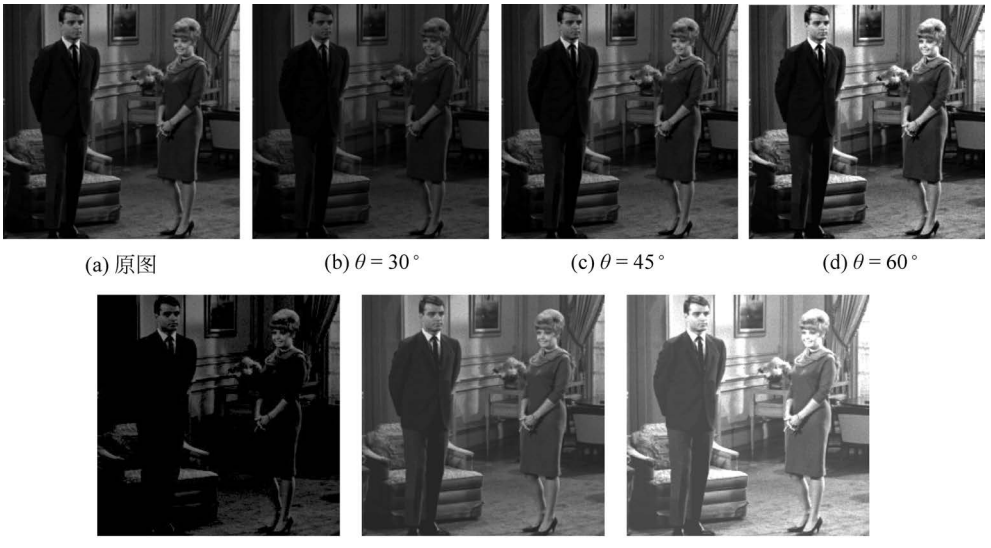
(a) $\theta = 45°$　(b) $\theta < 45°$　(c) $\theta > 45°$

(d) $\beta \neq 0$　(e) 求反

图 5-1　基本线性灰度级变换

```
result1, result2 = Image.copy(), Image.copy()
theta = [np.pi / 6, np.pi / 4, np.pi / 3]                    #设置倾角θ分别为π/6、π/4、π/3
for i in range(3):                                           #按式(5-2)进行变换
    result = Image * np.tan(theta[i])
    result1 = cv.hconcat((result1, result))                  #变换后图像水平拼接,方便显示
cv.imshow("Image and brightness adjustment: beta = 0", result1)
beta = [-30 / 255, 30 / 255, 60 / 255]                       #设置偏移β分别为-30、30、60
for i in range(3):                                           #按式(5-3)进行变换
    result = Image * np.tan(theta[i]) + beta[i]
    result2 = cv.hconcat((result2, result))
cv.imshow("Image and brightness adjustment: beta is not equal to 0", result2)
cv.waitKey()
```

程序运行结果如图 5-2 所示。



(a) 原图　(b) $\theta = 30°$　(c) $\theta = 45°$　(d) $\theta = 60°$

(e) $\theta = 30°, \beta = -30$　(f) $\theta = 45°, \beta = 30$　(g) $\theta = 60°, \beta = 60$

图 5-2　线性灰度级变换效果

OpenCV 中 intensity_transform 模块的 autoscaling 函数可以将图像的灰度级线性扩展到[0,255]，其调用格式为

```
cv.intensity_transform.autoscaling(input, output) -> None
```

参数 input 可以是 BGR 或灰度图像数据。

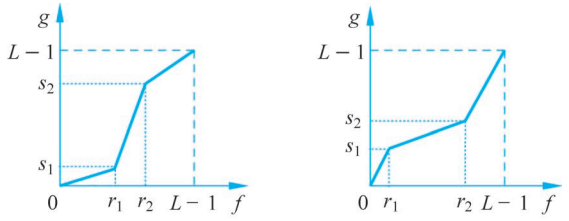
另有 convertScaleAbs 函数，按式(5-3)变换后，取绝对值并转化为无符号 8 位数据，其调用格式为

```
cv.convertScaleAbs(src[, dst[, alpha[, beta]]]) -> dst
```

参数 src 是输入的原图像，可以是单通道或多通道。可以尝试用这两个函数改写例 5.1。

2. 分段线性灰度级变换

将输入图像的灰度级区间分段，各段分别作线性灰度级变换，称为分段线性灰度级变换，是一种常用的灰度级变换方法。图 5-3 所示为分段线性灰度级变换的示意图，将输入灰度分为了三段，灰度区间$[0,r_1)$变换为$[0,s_1)$，灰度区间$[r_1,r_2)$变换为$[s_1,s_2)$，灰度区间$[r_2,L-1]$变换为$[s_2,L-1]$。



(a) 分段线性变换1　　(b) 分段线性变换2

图 5-3　分段线性变换函数示意图

可以看出，随着参数r_1、s_1、r_2、s_2的不同，每段灰度的变化也不一样，所以，可以根据实际需要，灵活设置参数取值，实现不同的变换效果。在图 5-3(a)中，由于$r_1>s_1$，实现了低灰度的范围压缩，灰度值降低；由于$r_2<s_2$，第三段线性函数的倾角小于 45°，实现了高灰度的范围压缩，但灰度值增大；整幅图像低灰度更低，高灰度更高，实现了对比度增强。在图 5-3(b)中，由于$r_1<s_1$，实现了低灰度的范围拉伸，灰度值增大；由于$r_2>s_2$，第三段线性函数的倾角大于 45°，实现了高灰度的范围拉伸，但灰度值降低；整幅图像低灰度提升，高灰度降低，实现了对比度降低。

图 5-3 所示的三段式线性灰度级变换函数如式(5-4)所示。

$$g(x,y)=\begin{cases}\dfrac{s_1}{r_1}f(x,y), & 0\leqslant f(x,y)<r_1\\[2ex] \dfrac{s_2-s_1}{r_2-r_1}[f(x,y)-r_1]+s_1, & r_1\leqslant f(x,y)<r_2\\[2ex] \dfrac{L-1-s_2}{L-1-r_2}[f(x,y)-r_2]+s_2, & r_2\leqslant f(x,y)<L\end{cases}\tag{5-4}$$

OpenCV 中的 intensity_transform 模块的 contrastStretching 函数可以实现三段式的分段线性灰度变换，将灰度级扩展到[0,255]，其调用格式为

```
cv.intensity_transform.contrastStretching(input, output, r1, s1, r2, s2) -> None
```

参数 input 可以是 BGR 或灰度图像数据，r1、s1、r2、s2 即式(5-4)中的r_1、s_1、r_2、s_2。

【例 5.2】 编写程序，采用图 5-3 所示的三段式线性变换函数对图像进行灰度级变换。

解：可以设定参数 r_1、s_1、r_2、s_2，然后根据式(5-4)对图像中的灰度级进行变换，本例直接使用 contrastStretching 函数，程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('panda.bmp', cv.IMREAD_GRAYSCALE)
r1, r2, s1, s2 = 80, 200, 30, 220                          #设置分段线性变换函数参数
result1, result2 = Image.copy(), Image.copy()
cv.intensity_transform.contrastStretching(Image, result1, r1, s1, r2, s2)
r1, r2, s1, s2 = 30, 220, 80, 200
cv.intensity_transform.contrastStretching(Image, result2, r1, s1, r2, s2)
cv.imshow("Original Image", Image)
cv.imshow("ContrastStretching 1", result1)
cv.imshow("ContrastStretching 2", result2)
cv.waitKey()
```

程序运行结果如图 5-4 所示。图 5-4(a)为原图；图 5-4(b)为 $r_1=80$、$s_1=30$、$r_2=200$、$s_2=220$ 时的处理效果，低灰度更低，高灰度更高，图像对比度得到增强；图 5-4(c)为 $r_1=30$、$s_1=80$、$r_2=220$、$s_2=200$ 时的处理效果，低灰度提升，高灰度降低，图像对比度降低。

(a) 原图　(b) r_1=80、r_2=200、s_1=30、s_2=220　(c) r_1=30、r_2=220、s_1=80、s_2=200

图 5-4　分段线性灰度级变换效果图

5.1.2　非线性灰度级变换

采用非线性变换函数实现灰度级的变换，可以实现比线性变换更加灵活的变换效果，常用的有对数变换和幂变换等。

灰度级的对数变换如式(5-5)所示。

$$g(x,y)=\alpha\log[f(x,y)+1] \tag{5-5}$$

其中，α 是尺度比例系数，$[f(x,y)+1]$是为了避免对 0 求对数，确保 $\log[f(x,y)+1]\geqslant 0$。式(5-5)实际是先对图像进行对数变换，再进行线性拉伸，以保证灰度值分布合理。

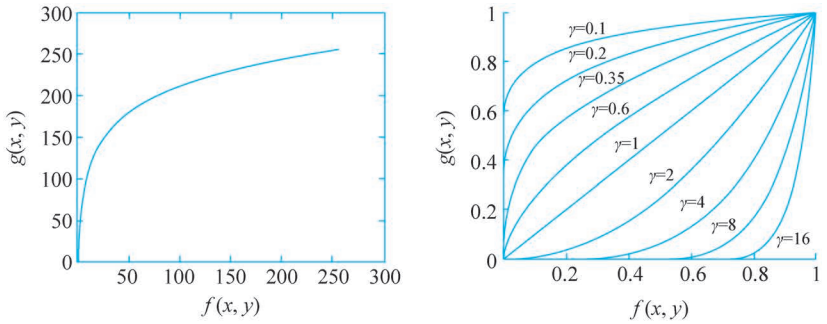
对数变换函数图形如图 5-5(a)所示，图像的低灰度区扩展，高灰度区压缩，一般适用于处理过暗图像。

灰度级的幂变换如式(5-6)所示。

$$g(x,y)=\alpha[f(x,y)]^{\gamma} \tag{5-6}$$

其中，γ 为正常数，决定了幂变换函数的图形以及灰度级变换效果，α 为尺度比例系数。

当 γ 取不同值时，可以得到一簇变换曲线。如图 5-5(b)所示。当 $\gamma=1$ 时，幂变换为线性变换；当 $0<\gamma<1$ 时，幂变换扩展中低灰度区，压缩高灰度区，使得图像变亮，增强图像中暗区的细节；当 $\gamma>1$ 时，幂变换扩展中高灰度区，压缩低灰度区，使得图像变暗，增强图像中亮区的细节。因此，幂变换也称为 gamma 校正，幂变换的指数值就是 gamma 值。



(a) 对数变换函数图　　(b) 幂变换函数图

图 5-5　非线性灰度级变换函数

OpenCV 中 intensity_transform 模块的 logTransform 和 gammaCorrection 函数可以实现灰度级的对数变换和幂变换，输出图像取值均在[0,255]之间，其调用格式为

```
cv.intensity_transform.logTransform(input, output) -> None
cv.intensity_transform.gammaCorrection(input, output, gamma) -> None
```

参数 input 可以是 BGR 或灰度图像数据。

【例 5.3】　编写程序，对图像进行对数变换和幂变换。

解：可以设定参数 α 和 γ，然后根据式(5-5)和式(5-6)对图像进行对数变换和幂变换，本例直接使用 logTransform 和 gammaCorrection 函数，程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread('couple.bmp', cv.IMREAD_GRAYSCALE)
result1, result2, result3 = Image.copy(), Image.copy(), Image.copy()
cv.intensity_transform.logTransform(Image, result1)                 #对数变换
cv.intensity_transform.gammaCorrection(Image, result2, 0.35)        #幂变换,0<γ<1
cv.intensity_transform.gammaCorrection(Image, result3, 2)           #幂变换,γ>1
cv.imshow("Original Image", Image)
cv.imshow("Log transformation", result1)
cv.imshow("Gamma correction:gamma = 0.35", result2)
cv.imshow("Gamma correction:gamma = 2", result3)
cv.waitKey()
```

程序运行结果如图 5-6 所示。图 5-6(a)为原图，图像较暗；图 5-6(b)为对数变换处理效果，低灰度得到大幅度提升，图像变亮很多；图 5-6(c)为 $\gamma=0.35$ 的幂变换，拉伸了低灰度区，图像变亮，但变亮程度弱于图 5-6(b)的对数变换；图 5-6(d)为 $\gamma=2$ 的幂变换，图像变暗。

(a) 原图　(b) 对数变换　(c) 幂变换γ=0.35　(d) 幂变换γ=2

图 5-6　非线性灰度级变换

5.2 直方图修正法

直方图是数字图像处理中一个常用的工具，是多种处理方法的基础，本节学习直方图的概念以及利用直方图进行灰度级变换的方法。

5.2.1 灰度直方图

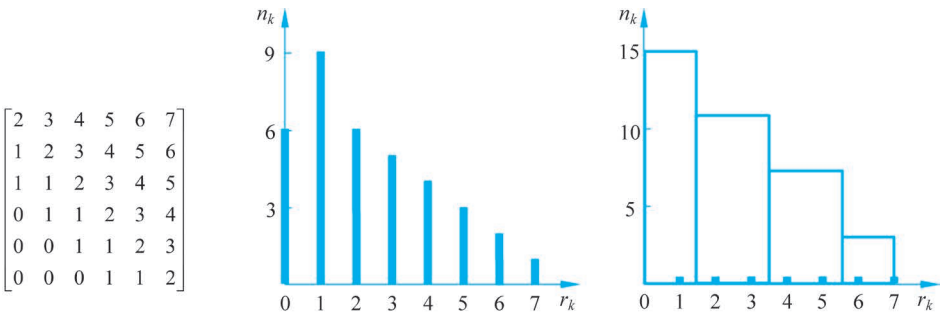
1. 灰度直方图的定义

以灰度级为横坐标，以图像中灰度出现的次数（频数、概率）为纵坐标，绘制的图形称为灰度直方图，它反映了图像中灰度的分布状况。灰度直方图的定义如式(5-7)所示。

$$p(r_k)=\frac{n_k}{M\cdot N} \tag{5-7}$$

其中，$M\cdot N$ 为一幅数字图像的分辨率，也就是总像素数，n_k 是呈现第 k 级灰度 r_k 的像素数，$p(r_k)$ 为灰度级 r_k 出现的相对频数。

可以通过扫描图像，统计各个灰度出现的次数，计算频数并绘制灰度直方图。如一幅 6×6 分辨率的图像可以用如图 5-7(a)所示的矩阵表示，共有 0～7 八个灰度级，灰度级分布统计如表 5-1 所示，则可以绘制并显示图像的灰度直方图，如图 5-7(b)所示。



(a) 原图的矩阵表示　(b) 灰度直方图1　(c) 灰度直方图2

图 5-7　数字图像及其直方图

表 5-1　灰度级分布统计

r_k	0	1	2	3	4	5	6	7
n_k	6	9	6	5	4	3	2	1
$p(r_k)$	6/36	9/36	6/36	5/36	4/36	3/36	2/36	1/36

也可以将灰度范围分为几个区间，统计各灰度在各区间内的像素数目，如表 5-2 所示，绘制的灰度直方图如图 5-7(c)所示。

表 5-2　灰度区间分布统计

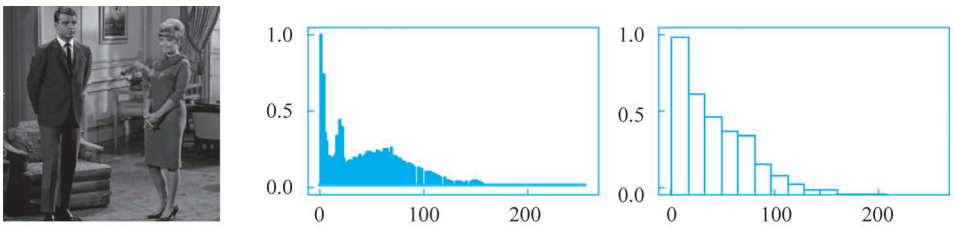
r_k	[0,1]	[2,3]	[4,5]	[6,7]
n_k	15	11	7	3
$p(r_k)$	15/36	11/36	7/36	3/36

【例 5.4】　编写程序，统计并显示图像的灰度直方图。

解：采用扫描图像的方法统计各个灰度出现的次数，进而计算其频数并绘制灰度直方图，程序如下。

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
Image = cv.imread('couple.bmp', cv.IMREAD_GRAYSCALE)
height, width = np.shape(Image)
hist1, hist2 = np.zeros([256, 1]), np.zeros([1, 16])
for y in range(height):
    for x in range(width):
        hist1[Image[y, x]] += 1                          #统计各灰度级出现的次数
        hist2[0, Image[y, x] // 16] += 1                 #统计各灰度区间内像素数
hist1 /= hist1.max()
hist2 /= hist2.max()
cv.imshow("Source image", Image)
fig, ax = plt.subplots(1, 2)
ax[0].stem(range(256), hist1, linefmt='black', markerfmt='')     #绘制256个级别的直方图
ax[0].set_title('Histogram with 256 bins')
ax[1].bar(list(range(0, 241, 16)), hist2[0], width=16, align='edge',
          ec=[0, 0, 0], fc=[1, 1, 1])                    #灰度级分为16个区间,绘制直方图
ax[1].set_title('Histogram with 16 bins')
plt.rcParams['font.sans-serif'] = ['Times New Roman']
plt.tight_layout()
plt.show()
```

程序运行结果如图 5-8 所示。图 5-8(a)为原图，图 5-8(b)为 256 个级别的灰度直方图，图 5-8(c)为 16 个区间的灰度直方图，两幅灰度直方图都进行了归一化。



(a) 原图　(b) 256个级别的灰度直方图　(c) 16个区间的灰度直方图

图 5-8　统计并绘制灰度直方图

彩色图像有 3 个色彩通道，可以分别对每个通道统计灰度直方图或亮度直方图。

OpenCV 中的 calcHist 函数可以用于统计灰度直方图，其调用格式如下：

```
cv.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]]) -> hist
```

参数 images 是输入图像的列表；channels 指明要进行统计的色彩通道；mask 是和图像同等大小的模板矩阵，选择参与统计的像素；histSize 指明直方图分区数；ranges 指明各区间的边界，如果均匀分区，只需要指明第一个区间的下边界和最后一个区间的上边界；accumulate 指明是否累积统计。

例 5.4 采用 calcHist 函数统计灰度直方图的程序如下：

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
Image = cv.imread('couple.bmp', cv.IMREAD_GRAYSCALE)
h_size, h_range = [256], (0, 256)
hist = cv.calcHist([Image], [0], None, h_size, h_range)
plt.stem(range(256), hist/hist.max(), linefmt='black', markerfmt='')
plt.rcParams['font.sans-serif'] = ['Times New Roman']
plt.title('Histogram')
```

```
plt.show()
```

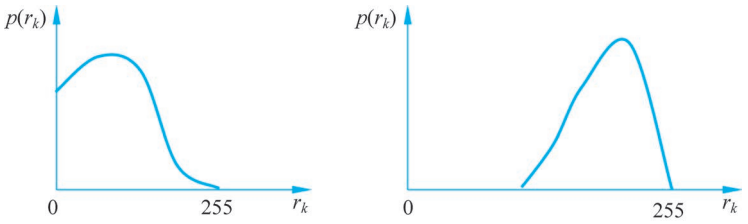
程序运行结果如图 5-8(b)所示。

2. 灰度直方图的性质

一幅图像的灰度直方图通常具有如下性质：

(1) 灰度直方图不具有空间特性。灰度直方图描述了灰度在各区间范围内的像素个数，但不能反映图像像素空间位置信息，即不能通过灰度直方图了解到各灰度在图像中出现的位置。

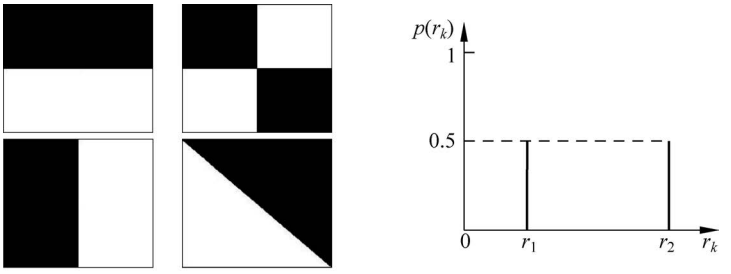
(2) 灰度直方图反映图像大致描述，如图像灰度范围、灰度级分布、整幅图像平均亮度等。图 5-9 所示为两幅图像的灰度直方图，可以从中判断出图像的相关特性。在图 5-9(a)中，大部分像素值集中在低灰度级区域，图像偏暗；图 5-9(b)中的图像则相反，大部分像素的灰度集中在高灰度区域，图像偏亮；两幅图像都存在动态范围不足的现象。



(a) 图像偏暗　　(b) 图像偏亮

图 5-9　灰度动态范围不足的图像灰度直方图

(3) 一幅图像唯一对应相应的灰度直方图，而不同的图像可以具有相同的灰度直方图。因灰度直方图只是统计图像中灰度出现的次数，与各个灰度出现的位置无关，因此，不同的图像可能具有相同的灰度直方图。图 5-10(a)为 4 幅大小相同、空间灰度分布不同的二值图像，但具有相同的灰度直方图，如图 5-10(b)所示。



(a) 4幅不同的图像　　(b) 灰度直方图

图 5-10　不同图像具有相同直方图分布特性

5.2.2　直方图均衡化

直方图修正法也是通过构造灰度级变换函数对图像进行变换的，使变换后的图像的直方图达到一定的要求。设变量 r 表示原图像中像素的灰度级，变量 s 表示增强后新图像中的灰度级，均已进行了归一化，即 $0 \leqslant r \leqslant 1$，$0 \leqslant s \leqslant 1$。根据灰度级变换的原理，有

$$s = T(r) \tag{5-8}$$

变换函数 T 需要满足两个条件：

(1) $T(r)$在 $0 \leqslant r \leqslant 1$ 区域内单值单调增加，以保证灰度级从黑到白的次序不变。

(2) $T(r)$在 $0\leqslant r\leqslant 1$ 区域内满足 $0\leqslant s\leqslant 1$，以保证变换后的像素灰度级仍在允许的灰度级范围内。

基于直方图的灰度级变换的核心就是寻找满足这两个条件的变换函数 $T(r)$，不同的变换函数对应不同的方法，直方图均衡化采用灰度级 r 的累积分布函数作为变换函数，即

$$s=T(r)=\int_0^r p_r(\omega)\mathrm{d}\omega \tag{5-9}$$

其中，$p_r(r)$表示灰度级 r 的概率密度函数，$T(r)$随着 r 增大，单值单调增加，最大为 1，满足两个条件。

根据概率论知识，用 $p_r(r)$和 $p_s(s)$分别表示 r 和 s 的概率密度函数，有

$$p_s(s)=p_r(r)\cdot\frac{\mathrm{d}r}{\mathrm{d}s}=p_r(r)\cdot\frac{1}{p_r(r)}=1 \tag{5-10}$$

即利用 r 的累积分布函数作为变换函数，产生一幅灰度级分布具有均匀概率密度的图像。

一幅数字图像，共有 L 个灰度等级，总像素个数为 $M\cdot N$，第 j 级灰度 r_j 对应的像素数为 n_j，直方图均衡化的变换函数 $T(r)$为

$$s_k=T(r_k)=\sum_{j=0}^{k}p_r(r_j)=\sum_{j=0}^{k}\frac{n_j}{M\cdot N} \tag{5-11}$$

对一幅数字图像进行直方图均衡化处理的算法步骤如下。

(1) 统计原始图像直方图，即计算 $p_r(r)$。

(2) 由式(5-11)计算新的灰度级 s_k。

(3) 修正 s_k 为合理的灰度级。数字图像灰度级有限，$0\sim k$ 的灰度级的概率之和未必是合理的灰度级，所以，需要修正，也就是四舍五入到最近的灰度级。

(4) 计算新的直方图，即计算 $p_s(s)$。

(5) 用处理后的新灰度代替处理前的灰度，生成新图像。

【例 5.5】 假定一幅分辨率为 64×64，灰度级为 8 级的图像，其灰度分布如表 5-3 所示，对其进行直方图均衡化处理。

表 5-3　例 5.5 中图像的灰度级分布

灰度级 r_k	0	1/7	2/7	3/7	4/7	5/7	6/7	1
像素数 n_k	790	1023	850	656	329	245	122	81
$p_r(r_k)$	0.19	0.25	0.21	0.16	0.08	0.06	0.03	0.02

解： 由原图的灰度分布统计可看出，图像中绝大部分像素集中在低灰度区，图像整体偏暗。

(1) 计算新的灰度级。

$$s_0=T(r_0)=\sum_{j=0}^{0}P_r(r_j)=P_r(r_0)=0.19$$

$$s_1=T(r_1)=\sum_{j=0}^{1}P_r(r_j)=P_r(r_0)+P_r(r_1)=0.19+0.25=0.44$$

依此类推，可得到

$$s_2=0.19+0.25+0.21=0.65\quad s_3=0.19+0.25+0.21+0.16=0.81$$

$$s_4=0.89\quad s_5=0.95\quad s_6=0.98\quad s_7=1$$

(2) 修正 s_k 为合理的灰度级 s_k'。

$$s_0=0.19\approx\frac{1}{7}\quad s_1=0.44\approx\frac{3}{7}\quad s_2=0.65\approx\frac{5}{7}\quad s_3=0.81\approx\frac{6}{7}$$

$$s_4=0.89\approx\frac{6}{7}\quad s_5=0.95\approx1\quad s_6=0.98\approx1\quad s_7=1$$

则新图像对应只有 5 个不同灰度级别，为 1/7,3/7,5/7,6/7,1。即

$$s'_0=\frac{1}{7}\quad s'_1=\frac{3}{7}\quad s'_2=\frac{5}{7}\quad s'_3=\frac{6}{7}\quad s'_4=1$$

(3) 计算新的直方图。

$$p_s(s'_0)=p_r(r_0)=0.19$$

$$p_s(s'_1)=p_r(r_1)=0.25$$

$$p_s(s'_2)=p_r(r_2)=0.21$$

$$p_s(s'_3)=p_r(r_3)+p_r(r_4)=0.16+0.08=0.24$$

$$p_s(s'_4)=p_r(r_5)+p_r(r_6)+p_r(r_7)=0.06+0.03+0.02=0.11$$

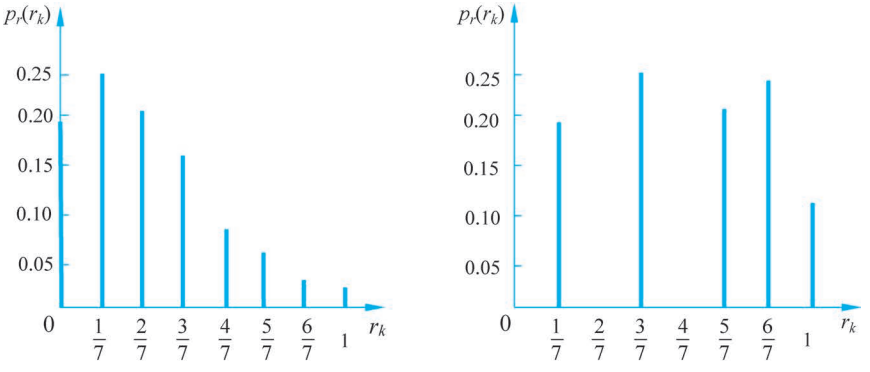
(4) 生成新图像。

按照表 5-4 中变换前后的灰度对应关系改变像素的灰度，即可生成新的图像。

表 5-4 直方图均衡化变换前后灰度级对应关系

变换前灰度级	0	1/7	2/7	3/7	4/7	5/7	6/7	1
变换后灰度级	1/7	3/7	5/7	6/7	6/7	1	1	1

原始图像的直方图和直方图均衡化处理后的图像直方图显示结果如图 5-11 所示。可看出，图 5-11(b)中对应的变换后的新直方图比图 5-11(a)中的原图像的直方图要平坦很多。理想情况下，经过直方图均衡化处理的图像直方图应是十分均匀平坦的，但实际情况并非如此，和理论分析有差异，这是由于图像在直方图均衡化处理过程中，灰度级作“近似简并”引起的结果。



(a) 原始图像直方图　　(b) 均衡化后图像的直方图

图 5-11 直方图均衡化处理前后的直方图分布对比

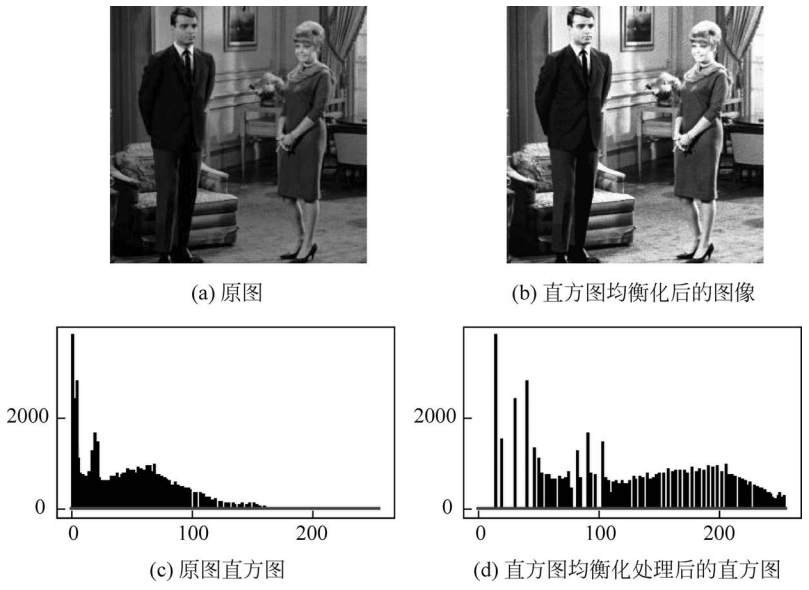
【例 5.6】 编写程序，对图像进行直方图均衡化。

解：按照例 5.5 所示过程实现直方图均衡化，程序如下。

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
```

```
Image = cv.imread('couple.bmp', cv.IMREAD_GRAYSCALE)
height, width = np.shape(Image)
h_size, h_range = [256], (0, 256)
hist = cv.calcHist([Image], [0], None, h_size, h_range)          #统计原图的直方图
result = np.zeros([height, width])
s = np.zeros([256, 1])                                           #定义 s 数组
s[0, 0] = hist[0, 0]
for i in range(1, 256):
    s[i, 0] = s[i-1, 0] + hist[i, 0]                             #累加生成新的灰度级
for y in range(height):
    for x in range(width):
        result[y, x] = s[Image[y, x], 0] / (height * width)      #生成新图像
result = (result * 255).astype(np.uint8)
hist_s = cv.calcHist([result], [0], None, h_size, h_range)       #统计新图像的直方图
fig, ax = plt.subplots(2, 2)
ax[0, 0].imshow(Image, plt.cm.gray), ax[0, 0].set_axis_off()     #显示原图
ax[0, 0].set_title('Original image')
ax[1, 0].stem(range(256), hist, linefmt='black', markerfmt='')   #显示原图直方图
ax[1, 0].set_title('Original histogram')
ax[0, 1].imshow(result, plt.cm.gray), ax[0, 1].set_axis_off()    #显示均衡化后的图像
ax[0, 1].set_title('Result image')
ax[1, 1].stem(range(256), hist_s, linefmt='black', markerfmt='') #显示新图像的直方图
ax[1, 1].set_title('Balanced histogram')
plt.rcParams['font.sans-serif'] = ['Times New Roman']
plt.tight_layout()
plt.show()
```

程序运行结果如图 5-12 所示。图 5-12(a)为原图，图像较暗；图 5-12(b)为均衡化后的图像，整体变亮，图像的视觉效果变好；图 5-12(d)为均衡化后图像的直方图，与原图直方图(图 5-12(c))相比，动态范围扩大，高灰度像素数增加；但和理论分析中的均匀分布有差异。



(a) 原图　(b) 直方图均衡化后的图像

(c) 原图直方图　(d) 直方图均衡化处理后的直方图

图 5-12　直方图均衡化处理前后的图像以及直方图

OpenCV 中的 equalizeHist 函数可以实现对 8 位灰度图像的直方图均衡化，其调用格式如下：

```
cv.equalizeHist(src[, dst]) -> dst
```

采用 equalizeHist 函数改写例 5.6 程序如下：

```
import cv2 as cv
import numpy as np
Image = cv.imread('couple.bmp', cv.IMREAD_GRAYSCALE)
result = cv.equalizeHist(Image)
cv.imshow("Original image", Image)
cv.imshow("Result image", result)
cv.waitKey()
```

程序运行结果如图 5-12(b)所示。

关于直方图均衡化分析的内容，请扫描二维码，查看讲解。

5.2.3 限制对比度自适应直方图均衡化

全局直方图均衡化方法简单高效，但是，图像中不同区域分布不同，使用同一种变换效果未必理想，而且实际应用中常常需要增强某些局部细节，因此，可以采用自适应直方图均衡化(Adaptive Histogram Equalization，AHE)方法，将图像划分为不重叠的子块，对每块分别进行均衡化。很明显，这样处理后，各子块之间会出现不连续现象，即有明显块效应，可以通过双线性插值方法改善。AHE 方法在相对均匀区域容易过度放大噪声，可以通过限制对比度增强的方法克服，即限制对比度自适应直方图均衡化(Contrast Limited Adaptive Histogram Equalization，CLAHE)方法。CLAHE 方法与全局直方图均衡化相比，主要区别在于限制对比度、局部直方图均衡化和双线性插值三方面，下面分别进行介绍。

第 8 集
微课视频

设子块的分辨率为 $w \cdot w$，统计块内的灰度分布 $p(r)$，在块内进行直方图均衡化的变换函数可以表示为

$$T(r)=\frac{255F(r)}{w \cdot w} \tag{5-12}$$

其中，$F(r)$是灰度级的累积分布函数。为防止对比度被过度拉伸，可以限制 $T(r)$的斜率。由于

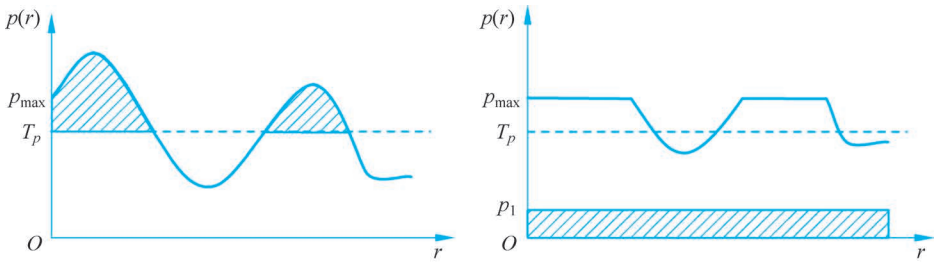
$$\frac{\mathrm{d}T(r)}{\mathrm{d}r}=\frac{255p(r)}{w \cdot w} \tag{5-13}$$

可知，限制变换函数的斜率可以通过限制直方图的高度实现，即限制 $p(r)$的最大值为 $p_{\max}$。设阈值为 T_p，调整直方图为

$$p(r)=\begin{cases} p(r)+p_1, & p(r)<T_p \\ p_{\max}, & p(r)\geqslant T_p \end{cases} \tag{5-14}$$

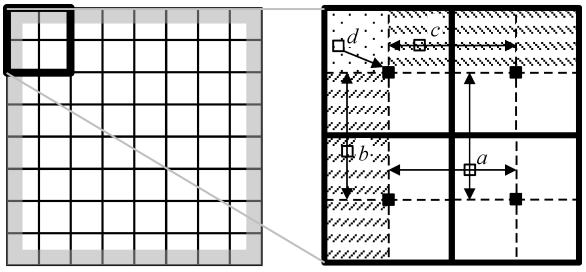
直方图的调整如图 5-13 所示，对直方图进行裁剪使其低于上限 $p_{\max}$，裁剪掉的部分均匀分布在整个灰度区间上，以保证直方图总面积不变。按照调整后的直方图进行均衡化，对比度不会过度增强，可以避免过度放大噪声。

利用插值运算消除块效应如图 5-14 所示。图 5-14(a)所示为图像的分块情况，将图像分为 8×8 个子块，阴影部分是边界像素，即落在图像四角的四个子块中心点围成的四边形之外的像素。左上角 2×2 个子块如图 5-14(b)所示，其中，黑色小方块是子块中心，按照各块的变换函数进行处理：a 点的值由其周围 4 个子块中心变换后的值进行双线性插值计算；b、c 点的值各由相邻两个子块线性插值计算；d 点的值根据所在子块的变换函数计算。



(a) 调整前　　(b) 调整后

图 5-13　调整直方图限制对比度



(a) 分块情况　　(b) 插值示意

图 5-14　利用插值运算消除块效应

综上所述，CLAHE 方法的运算过程整理如下。

(1) 将图像分为不重叠的子块。常见是 8×8 个子块，如果图像长宽不是 8 的倍数，可以通过边界填塞补充。

(2) 统计各子块的直方图，设置参数，调整直方图。

(3) 局部直方图均衡化。

(4) 块间双线性插值。

OpenCV 中的 CLAHE 类封装了该方法的相关函数，apply 函数可以对灰度图像实现 CLAHE 增强，getClipLimit、setClipLimit 函数可以分别获取和设置对比度限制阈值，getTilesGridSize、setTilesGridSize 函数可以分别获取和设置行列方向上的子块数目；另外，CLAHE 类通过 createCLAHE 实例化。这些函数的调用格式如下：

```
cv.CLAHE.apply(src[, dst]) -> dst
cv.CLAHE.getClipLimit( ) -> retval
cv.CLAHE.getTilesGridSize() -> retval
cv.CLAHE.setClipLimit(clipLimit) -> None
cv.CLAHE.setTilesGridSize(tileGridSize) -> None
cv.createCLAHE([, clipLimit[, tileGridSize]]) -> retval
```

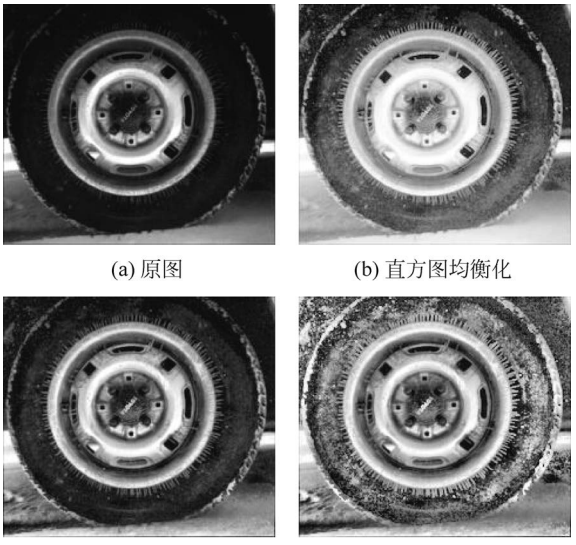
【例 5.7】 编写程序，对图像进行限制对比度自适应直方图均衡化。

解： 程序如下。

```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
Image = cv.imread('tire.tif', cv.IMREAD_GRAYSCALE)
result1 = cv.equalizeHist(Image)                                    # 全局直方图均衡化
clahe = cv.createCLAHE(clipLimit = 4, tileGridSize = (8, 8))       # 创建 CLAHE 实例
result2 = clahe.apply(Image)
clahe = cv.createCLAHE(clipLimit = 40, tileGridSize = (8, 8))      # CLAHE 实例 2,不同的限制程度
```

```
result3 = clahe.apply(Image)
fig, ax = plt.subplots(2, 2)
ax[0, 0].imshow(Image, plt.cm.gray), ax[0, 0].set_axis_off()
ax[0, 0].set_title('Original image')
ax[0, 1].imshow(result1, plt.cm.gray), ax[0, 1].set_axis_off()
ax[0, 1].set_title('Result of HE')
ax[1, 0].imshow(result2, plt.cm.gray), ax[1, 0].set_axis_off()
ax[1, 0].set_title('Result of CLAHE, clipLimit = 4')
ax[1, 1].imshow(result3, plt.cm.gray), ax[1, 1].set_axis_off()
ax[1, 1].set_title('Result of CLAHE, clipLimit = 40')
plt.rcParams['font.sans - serif'] = ['Times New Roman']
plt.tight_layout()
plt.show()
```

程序运行结果如图 5-15 所示，图 5-15(a)为原图，图 5-15(b)为直方图均衡化的结果，整幅图像采用同一个变换函数，图像上下两部分细节增强程度不一样；图 5-15(c)和图 5-15(d)是 CLAHE 处理结果，整幅图像均得到增强，而且限制对比度的阈值不一样，处理结果也不一样，阈值越大，对比度增强越大，暗区的噪声也越大。



(a) 原图　(b) 直方图均衡化

(c) 低阈值的CLAHE　(d) 高阈值的CLAHE

图 5-15　限制对比度自适应直方图均衡化

5.3　空间域平滑滤波

在图像的获取、传输和存储过程中，常常会受到各种噪声的干扰，影响图像质量。抑制或消除图像中存在的噪声称为图像平滑(Image Smoothing)，通过在像素邻域内进行模板运算以抑制噪声的方法称为空间域平滑滤波。

5.3.1　图像中的噪声

根据噪声和图像信号的关系，将噪声分为两种形式：加性噪声和乘性噪声。

加性噪声与图像信号不相关，含噪声图像 $g(x,y)$ 可表示为理想无噪声图像 $f(x,y)$ 与噪声 $n(x,y)$ 之和，即

$$g(x,y)=f(x,y)+n(x,y) \tag{5-15}$$

乘性噪声与图像信号相关，往往随图像信号的变化而变化，如果噪声和信号成正比，则含噪声图像 $g(x,y)$ 表示为

$$g(x,y)=f(x,y)+f(x,y)\cdot n(x,y) \tag{5-16}$$

为了分析处理方便，在信号变化很小时，往往将乘性噪声近似看作加性噪声，而且总是假定信号和噪声是互相独立的。

一般噪声是随机信号，通常用概率分布函数描述。常见的噪声有高斯噪声、椒盐噪声、泊松噪声等。

高斯噪声分布在每个像素上，幅度值是随机的，分布近似符合高斯正态特性。高斯噪声的概率密度函数为

$$p(x)=\frac{1}{\sqrt{2\pi}\sigma}e^{-(x-\mu)^2/2\sigma^2} \tag{5-17}$$

其中，μ 为随机变量 x 的均值，σ^2 为 x 的方差。描述的噪声值有 95%落在$(\mu-2\sigma,\mu+2\sigma)$范围内。

椒盐噪声的概率密度函数为

$$p(x)=\begin{cases}P_a, & x=a\\ P_b, & x=b\\ 0, & \text{其他}\end{cases} \tag{5-18}$$

其中，$b>a$。当 $P_a\neq 0$，$P_b\neq 0$ 时，尤其是它们近似相等时，描述的噪声值将类似于随机撒在图像上的胡椒和盐粉颗粒，因此称为椒盐噪声，具有幅度值近似相等但出现位置随机分布的特性。

泊松噪声的概率密度函数为

$$p(x=k)=\frac{\lambda^k e^{-\lambda}}{k!},\quad \lambda>0 \tag{5-19}$$

另外还有服从瑞利分布、均匀分布、指数分布等的噪声，其概率密度函数不再一一介绍。

在图像处理中，常根据数学模型生成噪声，与图像相加生成含噪声图像用于仿真实验。利用 Python 编程，可以采用 Numpy 库 random 模块的随机函数生成随机数矩阵模拟噪声。

【例 5.8】 编写程序，给图像添加高斯噪声和椒盐噪声。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
height, width = np.shape(Image)
density = 0.05                                          #设置椒盐噪声密度
num_noise = int(height * width * density)               #受椒盐噪声干扰的像素数
noisedI_sp = Image.copy()
for i in range(num_noise):
    x = np.random.randint(1, width)
    y = np.random.randint(1, height)                    #随机选择受椒盐噪声干扰的点
    if np.random.randint(0,2) == 0:
        noisedI_sp[y, x] = 0                            #添加椒噪声
    else:
        noisedI_sp[y, x] = 255                          #添加盐噪声
gauss = np.random.normal(0, 0.05, (height, width))
        #生成服从高斯分布的随机矩阵，均值为 0，标准差为 0.05，矩阵和图像大小相同
noisedI_g = Image / 255 + gauss
```

```
cv.imshow("Original image", Image)
cv.imshow("Noisy image: Salt and pepper noise", noisedI_sp)
cv.imshow("Noisy image: Gaussian noise", noisedI_g)
cv.waitKey()
```

程序运行结果如图 5-16 所示。

(a) 原图　　(b) 椒盐噪声　　(c) 高斯噪声

图 5-16　给图像添加噪声

5.3.2　均值滤波

均值滤波，又称邻域平均法，是图像空间域平滑滤波中最基本的方法之一，其基本思想是以某一像素为中心，在它的周围选择一个邻域，用邻域内所有像素值的均值代替原来像素值，通过降低噪声点与周围像素的差值抑制噪声。

输入图像 $f(x,y)$，经均值滤波处理后，得到输出图像 $g(x,y)$，即

$$g(x,y)=\frac{1}{M_S N_S}\sum_{(i,j)\in S} f(i,j) \tag{5-20}$$

其中，S 是像素(x,y)周围的邻域，一般选方形区域，M_S 和 N_S 是邻域 S 的宽和高。

均值滤波可以采用模板运算表示。典型的均值模板中所有系数都取相同值，如 3×3 和 5×5 的简单均值模板如式(5-21)所示。

$$\boldsymbol{H}_1=\frac{1}{3\times3}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\quad \boldsymbol{H}_2=\frac{1}{5\times5}\begin{bmatrix}1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\end{bmatrix} \tag{5-21}$$

也称为归一化的盒式滤波器。

若邻域内有噪声存在，经过均值滤波，噪声的幅度会大幅降低，但点与点之间的灰度差值会变小，将导致边缘模糊。邻域越大，模糊越严重。

OpenCV 中 blur 函数可以实现典型的均值滤波，boxFilter 函数可以实现盒式滤波，其调用格式如下：

```
cv.blur(src, ksize[, dst[, anchor[, borderType]]]) -> dst
cv.boxFilter(src, ddepth, ksize[, dst[, anchor[, normalize[, borderType]]]]) -> dst
```

blur 函数参数中的 src 是输入的原图，可以包含多个色彩通道；ksize 指模板宽和高；anchor 指明模板原点，默认值(−1,−1)表示模板中心为原点；borderType 指明边界像素的填塞方式。boxFilter 函数参数中的 normalize 指明是否使用邻域内像素数 $M_S N_S$ 归一化，默认为 True；ddepth 指明输出图像深度，设为−1 时和原图深度一致。

【例 5.9】　编写程序，给图像添加高斯噪声并进行均值滤波。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
gauss = np.random.normal(0, 0.05, np.shape(Image))
noisedI = Image / 255 + gauss                                    # 高斯噪声图像
averI1 = cv.blur(noisedI, ksize = (3, 3), borderType = cv.BORDER_REPLICATE)
                                                                 # 3 × 3 均值滤波
averI2 = cv.boxFilter(noisedI, ddepth = -1, ksize = (7, 7), normalize = True,
                      borderType = cv.BORDER_REPLICATE)          # 7 × 7 均值滤波
cv.imshow("Original image", Image)
cv.imshow("Noisy image", noisedI)
cv.imshow("Image smoothed by Averaging filter (3, 3)", averI1)
cv.imshow("Image smoothed by Averaging filter (7, 7)", averI2)
cv.waitKey()
```

程序运行结果如图 5-17 所示。可以看出，均值滤波抑制高斯噪声效果明显，但会使图像变得模糊，均值滤波邻域半径越大，图像的模糊程度越大。

(a) 高斯噪声图像　(b) 3×3模板滤波　(c) 7×7模板滤波

图 5-17　均值滤波效果

5.3.3　高斯滤波

在均值滤波中，周围邻域内像素参与运算的程度一致，而在实际中，距离中心像素近的像素相对较远的像素与中心像素的相关性更强，对最终结果的影响应该更大，因此，可以采用加权滤波的方法。高斯滤波是邻域内的点按照高斯函数加权进行滤波，也就是图像与高斯函数的卷积运算。

零均值、标准差为 σ 的二维高斯函数，如式(5-22)所示。

$$h(x,y)=\frac{1}{2\pi\sigma^2}\mathrm{e}^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5-22}$$

零均值、标准差为 1 的二维高斯函数曲线如图 5-18 所示，可以看出，高斯函数曲线为钟形，离中心原点越近，函数值越大；离中心原点越远，函数值越小。这一特性使得高斯函数常被用来进行权值分配。

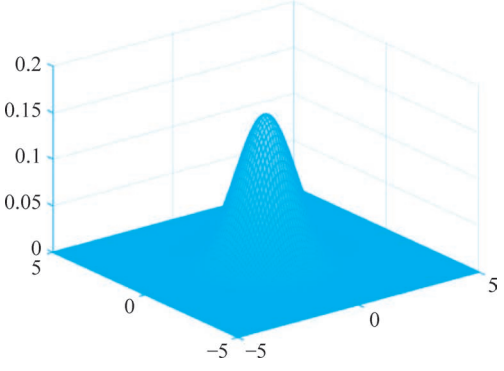


图 5-18　零均值、标准差为 1 的二维高斯函数

高斯滤波以某一像素为中心，选择一个局部

邻域，按高斯分布给邻域内像素分配相应的权值系数，再用邻域内所有像素值的加权平均值来代替原来像素值，降低噪声点与周围像素的差值，抑制噪声。

高斯滤波也可以表示为卷积模板运算，按照正态分布的统计，为模板上不同位置赋予不同的加权系数值。标准差 σ 代表数据的离散程度。σ 值越小，分布越集中，生成的高斯模板的中心系数值远远大于周围的系数值，对图像的平滑效果就越不明显；反之，σ 值越大，分布越分散，生成的高斯模板中不同系数值差别不大，类似均值模板，对图像的平滑效果越明显。标准差 σ 为 0.8 及 σ 为 1 时，5×5 的高斯模板如式(5-23)所示，矩阵前面的系数是将模板系数归一化。

$$\boldsymbol{H}_{\sigma=0.8}=\frac{1}{2070}\begin{bmatrix}1&10&22&10&1\\10&108&237&108&10\\22&237&518&237&22\\10&108&237&108&10\\1&10&22&10&1\end{bmatrix}\quad \boldsymbol{H}_{\sigma=1}=\frac{1}{330}\begin{bmatrix}1&4&7&4&1\\4&20&33&20&4\\7&33&54&33&7\\4&20&33&20&4\\1&4&7&4&1\end{bmatrix}\tag{5-23}$$

OpenCV 中 getGaussianKernel 函数可以生成一维高斯滤波器，GaussianBlur 函数可以实现高斯滤波，调用格式如下：

```
cv.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType[, hint]]]]) -> dst
cv.getGaussianKernel(ksize, sigma[, ktype]) -> retval
```

参数 src 是输入的原图，可以包含多个色彩通道；ksize 为模板尺寸，宽高均为正奇数，如果不设置，根据高斯核在 x 和 y 方向上的标准差 sigmaX 和 sigmaY 计算；如果 sigmaX 或 sigmaY 设为 0，表示二者相同；如果两者都设为 0，各自根据 ksize 的宽和高计算：sigma＝0.3×[(ksize－1)×0.5－1]＋0.8；ktype 指明滤波器系数类型，可以是 CV_32F 或 CV_64F。

【例 5.10】 编写程序，给图像添加高斯噪声并进行高斯滤波。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
noisedI = Image / 255 + np.random.normal(0, 0.05, np.shape(Image))
H1 = cv.getGaussianKernel(ksize = 7, sigma = 0.5)                    #生成一维高斯滤波器
H2 = cv.getGaussianKernel(ksize = 7, sigma = 1)
H3 = cv.getGaussianKernel(ksize = 7, sigma = 5)
np.set_printoptions(precision = 2)
print('一维高斯滤波器\nsigma = 0.5:', H1.T)
print('sigma = 1:', H2.T)
print('sigma = 5:', H3.T)
result1 = cv.GaussianBlur(noisedI, ksize = (7, 7), sigmaX = 0.5, sigmaY = 0,
                          borderType = cv.BORDER_REPLICATE)    #高斯滤波 σ = 0.5
result2 = cv.GaussianBlur(noisedI, ksize = (7, 7), sigmaX = 1, sigmaY = 0,
                          borderType = cv.BORDER_REPLICATE)    #高斯滤波 σ = 1
result3 = cv.GaussianBlur(noisedI, ksize = (7, 7), sigmaX = 5, sigmaY = 0,
                          borderType = cv.BORDER_REPLICATE)    #高斯滤波 σ = 5
cv.imshow("Original image", Image)
cv.imshow("Noisy image", noisedI)
cv.imshow("Image smoothed by Gaussian filter(sigma = 0.5)", result1)
cv.imshow("Image smoothed by Gaussian filter(sigma = 1)", result2)
cv.imshow("Image smoothed by Gaussian filter(sigma = 5)", result3)
cv.waitKey()
```

运行程序，输出 3 个不同 sigma 对应的一维高斯滤波器系数：

```
sigma = 0.5: [[1.20e-08  2.64e-04  1.06e-01  7.87e-01  1.06e-01  2.64e-04  1.20e-08]]
sigma = 1: [[0.     0.05  0.24  0.4   0.24  0.05  0.   ]]
sigma = 5: [[0.13  0.14  0.15  0.15  0.15  0.14  0.13]]
```

σ 值越大，高斯模板中不同系数值差别越小，当 $\sigma=5$ 时已经很接近于均值滤波模板了。采用 3 个高斯滤波的结果如图 5-19 所示，σ 越大，平滑力度越大，模糊程度也越大。

(a) $\sigma=0.5$　　(b) $\sigma=1$　　(b) $\sigma=5$

图 5-19　高斯滤波效果

5.3.4　双边滤波

高斯滤波仅考虑了位置对中心像素的影响，会较明显地模糊边缘。为了能够在消除噪声的同时很好地保留边缘信息，可以采用双边滤波方法，在平滑滤波的同时考虑邻域内像素的空间邻近性以及灰度相似性进行局部加权平均。

设输入图像为 $f(x,y)$，滤波输出图像为 $g(x,y)$，双边滤波如式(5-24)所示。

$$g(x,y)=\frac{\sum_{i,j} f(i,j)w(x,y,i,j)}{\sum_{i,j} w(x,y,i,j)} \tag{5-24}$$

其中，(i,j)是(x,y)邻域内的点，$w(x,y,i,j)$是综合考虑了相邻两点的距离和像素值差的加权系数，即

$$w(x,y,i,j)=\mathrm{e}^{-\left[\frac{(i-x)^2+(j-y)^2}{2\sigma_s^2}\right]}\times\mathrm{e}^{-\left[\frac{|f(i,j)-f(x,y)|^2}{2\sigma_r^2}\right]} \tag{5-25}$$

可知，与高斯滤波相比，在边缘附近，距离较远的像素对应的加权系数 w 第一项取值很小，不会太多影响到边缘上的像素值，能够很好地保留边缘信息。

双边滤波中参数 σ_s 和 σ_r 的选择直接影响双边滤波的输出结果。σ_s 控制空间邻近度，当 σ_s 变大时，距离远的像素对中心像素的影响增大，平滑程度增高。σ_r 用来控制灰度邻近度，当 σ_r 变大时，则灰度差值较大的点也能影响中心点的像素值，平滑程度增大，但灰度差值大于 σ_r 的像素几乎不参与运算，使得能够保留图像边缘的灰度信息。而当 σ_s、σ_r 取值都很小时，图像几乎不会产生平滑的效果。

OpenCV 中的 bilateralFilter 函数可以实现双边滤波，调用格式如下：

```
cv.bilateralFilter(src, d, sigmaColor, sigmaSpace[, dst[, borderType]]) -> dst
```

参数 src 是原图像数据，要求是 8 位或浮点数据，可以是 1 个或 3 个通道；d 表示每个像素的邻域直径，如果为非正数，根据参数 sigmaSpace 计算；sigmaSpace 是控制空间邻近度的标准差 σ_s；sigmaColor 是控制灰度邻近度的标准差 σ_r。

【例 5.11】 编写程序，给图像添加高斯噪声并进行双边滤波。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("girl.bmp", cv.IMREAD_GRAYSCALE)
noisedI = (Image / 255 + np.random.normal(0, 0.05, np.shape(Image)))
                                    .astype(np.float32)              # 生成高斯噪声图像
result1 = cv.GaussianBlur(noisedI, ksize = (7, 7), sigmaX = 5, sigmaY = 0,
                          borderType = cv.BORDER_REPLICATE)      # 高斯滤波 σ = 5
result2 = cv.bilateralFilter(noisedI, d = 7, sigmaColor = 0.1, sigmaSpace = 5,
                             borderType = cv.BORDER_REPLICATE)   # 双边滤波 σs = 5
result3 = cv.bilateralFilter(noisedI, d = 11, sigmaColor = 0.1, sigmaSpace = 9,
                             borderType = cv.BORDER_REPLICATE)   # 双边滤波 σs = 9
cv.imshow("Original image", Image)
cv.imshow("Noisy image", noisedI)
cv.imshow("Image smoothed by Gaussian filter(sigma = 5)", result1)
cv.imshow("Image smoothed by bilateral filter(sigma = 5, 0.1)", result2)
cv.imshow("Image smoothed by bilateral filter(sigma = 9, 0.1)", result3)
cv.waitKey()
```

程序运行结果如图 5-20 所示。图 5-20(a)为添加高斯噪声的图像，图 5-20(b)为高斯滤波效果，模糊明显；图 5-20(c)为双边滤波效果，和高斯滤波尺寸、空间标准差均相同，双边滤波在抑制噪声的同时较好地保留了边缘信息；图 5-20(d)是修改滤波尺寸为 11×11、$\sigma_s=9$ 时的双边滤波效果，窗口增大、空间标准差增大，滤波强度增大。

第 9 集
微课视频

(a) 噪声图像　(b) 高斯滤波　(c) 双边滤波1　(d) 双边滤波2

图 5-20　双边滤波效果

关于双边滤波的实现，请扫描二维码，查看讲解。

5.3.5　中值滤波

中值是指数字序列中取值在中间的值，通过将数字序列从小到大排序，奇数个数的序列取正中间的值，偶数个数的序列取中间两个数的平均值。中值滤波以图像中某一点为中心，选择周围一个邻域，把邻域内所有像素值排序，取中值代替该像素的值。

图像中噪声的出现，使该点像素比周围像素暗(亮)许多，若把其周围像素值排序，噪声点的值必然位于序列的前(后)端，序列的中值一般为未受到噪声污染的像素值，所以可以用中值取代原像素的值来滤除噪声。

【例 5.12】 设原图像为 $\boldsymbol{f}=\begin{bmatrix}1&2&1&4&3\\1&2&2&3&4\\5&7&6&8&9\\5&7&6&8&9\\5&6&7&8&9\end{bmatrix}$，对该图像进行中值滤波。

解：采用像素坐标系，对其进行基于 3×3 邻域的中值滤波处理。

以点(1,1)为例，即对图中模板所覆盖像素进行运算：$\boldsymbol{f}=\begin{bmatrix}\begin{bmatrix}1&2&1\\1&2&2\\5&7&6\end{bmatrix}&\begin{matrix}4&3\\3&4\\8&9\end{matrix}\\\begin{matrix}5&7&6\\5&6&7\end{matrix}&\begin{matrix}8&9\\8&9\end{matrix}\end{bmatrix}$

$g(1,1)=\text{med}\{1,2,1,1,2,2,5,7,6\}=2$

每个像素进行同样运算后，得最终结果：$\boldsymbol{g}=\begin{bmatrix}1&2&1&4&3\\1&2&3&4&4\\5&5&6&6&9\\5&6&7&8&9\\5&6&7&8&9\end{bmatrix}$

OpenCV 中的 medianBlur 函数可以实现中值滤波，调用格式如下：

```
cv.medianBlur(src, ksize[, dst]) -> dst
```

参数 src 是多通道的图像数据；ksize 是邻域尺寸，是大于 1 的奇数，取 3 或 5 时，图像深度为 CV_8U、CV_16U、CV_32F，取更大的尺寸时，图像深度只能为 CV_8U。

【例 5.13】 编写程序，对椒盐噪声图像和高斯噪声图像进行中值滤波。

解：程序如下。

```
import cv2 as cv
import numpy as np
noisedI_sp = cv.imread("noise_sp.jpg", cv.IMREAD_GRAYSCALE)
noisedI_g = cv.imread("noise_g.jpg", cv.IMREAD_GRAYSCALE)
result1 = cv.medianBlur(noisedI_sp.astype(np.uint8), ksize = 3)  # 对椒盐噪声图像进行中值滤波
result2 = cv.medianBlur(noisedI_sp.astype(np.uint8), ksize = 5)
result3 = cv.medianBlur(noisedI_g, ksize = 3)                    # 对高斯噪声图像进行中值滤波
result4 = cv.medianBlur(noisedI_g, ksize = 5)
cv.imshow("Salt and pepper noise", noisedI_sp)
cv.imshow("Gaussian noise", noisedI_g)
cv.imshow("Image smoothed by median filter 3(salt & pepper)", result1)
cv.imshow("Image smoothed by median filter 5(salt & pepper)", result2)
cv.imshow("Image smoothed by median filter 3(gauss)", result3)
cv.imshow("Image smoothed by median filter 5(gauss)", result4)
cv.waitKey()
```

程序运行结果如图 5-21 所示。从图中可以看出，中值滤波对椒盐噪声的抑制效果较好，对高斯噪声的抑制效果较差；随着模板尺寸增大，也有一定的模糊效应，但比均值滤波轻微。

对于椒盐噪声，中值滤波比均值滤波、高斯滤波效果好，模糊轻微，边缘信息保留较好。因为受椒盐噪声污染的图像中还存在干净点，中值滤波是选择适当的值来替代污染点的值。

对于高斯噪声，均值滤波、高斯滤波比中值滤波效果好。因为受高斯噪声污染的图像中每个点都是污染点。若噪声正态分布的均值为 0，则均值滤波、高斯滤波可以消除噪声。

中值滤波不适用于直接处理点线细节多的图像。因为中值滤波在滤除噪声的同时，可能把有用的细节信息滤掉，如图 5-22 所示。

(a) 椒盐噪声　(b) 3×3模板抑制椒盐噪声　(c) 5×5模板抑制椒盐噪声

(d) 高斯噪声　(e) 3×3模板抑制高斯噪声　(f) 5×5模板抑制高斯噪声

图 5-21　中值滤波效果

(a) 原图　(b) 1×3中值滤波　(c) 3×1中值滤波

(d) 4邻域中值滤波　(e) 8邻域中值滤波　(f) 5×5中值滤波

图 5-22　对点线细节多的图像进行中值滤波

5.4　空间域锐化滤波

对人眼视觉系统的研究表明，人类对形状的感知一般通过识别边缘、轮廓、前景和背景而形成。在图像处理中，边缘信息也十分重要。边缘通常定义为图像中亮度突变的位置，通过计算图像局部区域的亮度差异，从而检测出不同目标或场景各部分之间的边缘，是图像锐化、图像分割、区域形状特征提取等技术的重要基础。图像锐化（Image Sharpening）的目的是加强图像中景物的边缘和轮廓，突出图像中的细节或者增强被模糊了的细节。

5.4.1　边缘分析

图像中的边缘主要有以下 3 种类型：突变型边缘、细线型边缘和渐变型边缘，如图 5-23 所示。

图 5-23　图像中边缘类型示意

把图 5-23 中标注的三种类型边缘放在同一图像中，并绘制灰度变化曲线以及曲线的一阶和二阶微分，如图 5-24 所示。

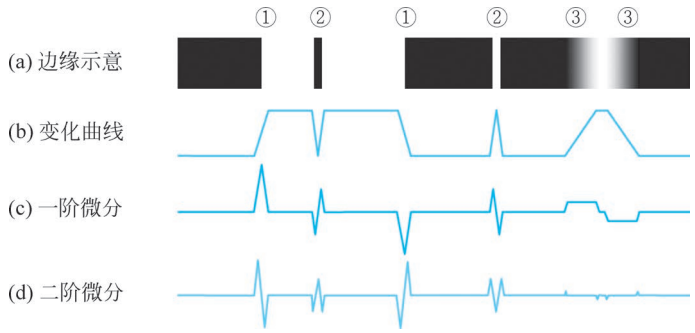


图 5-24　边缘和微分示意图

突变型边缘位于图像中两个具有不同灰度值的相邻区域之间，灰度曲线有阶跃变化，对应于一阶微分的极值和二阶微分的过零点；细线型边缘灰度变化曲线存在局部极值，对应于一阶微分过零点和二阶微分的极值点；渐变型边缘因灰度变化缓慢，没有明确的边界点。

通过分析边缘变化曲线和其一二阶微分曲线，可知图像中的边缘对应微分的特殊点，因此可以利用求微分去检测图像中的边缘所在。

5.4.2　一阶微分算子

由上节分析可知，一阶微分的极值或过零点与边缘存在对应的关系。本节分析常用的检测边缘的一阶微分算子，包括梯度算子、Roberts 算子、Sobel 算子以及 Prewitt 算子等。

1. 梯度算子

在图像处理中应用微分最常用的方法是计算梯度，梯度是方向导数取最大值的方向的向量。对于图像函数 $f(x,y)$，在(x,y)处的梯度为

$$G[f(x,y)]=\left[\frac{\partial f}{\partial x}\quad\frac{\partial f}{\partial y}\right]^{\mathrm{T}} \tag{5-26}$$

其中，G 表示对二维函数 $f(x,y)$计算梯度。

用梯度幅度值来代替梯度，得

$$G[f(x,y)]=\left[\left(\frac{\partial f}{\partial x}\right)^2+\left(\frac{\partial f}{\partial y}\right)^2\right]^{\frac{1}{2}} \tag{5-27}$$

为计算方便，也常用绝对值运算来代替式(5-27)。

$$G[f(x,y)]=\left|\frac{\partial f}{\partial x}\right|+\left|\frac{\partial f}{\partial y}\right| \tag{5-28}$$

因为图像为离散的数字矩阵，可用差分来代替微分，得梯度图像 $g(x,y)$：

$$\begin{cases}\dfrac{\partial f}{\partial x}=\dfrac{\Delta f}{\Delta x}=\dfrac{f(x+1,y)-f(x,y)}{x+1-x}=f(x+1,y)-f(x,y)\\ \dfrac{\partial f}{\partial y}=\dfrac{\Delta f}{\Delta y}=\dfrac{f(x,y+1)-f(x,y)}{y+1-y}=f(x,y+1)-f(x,y)\\ g(x,y)=|f_x|+|f_y|=|f(x+1,y)-f(x,y)|+|f(x,y+1)-f(x,y)|\end{cases} \tag{5-29}$$

式中，f_x、f_y 分别表示图像 f 在水平、垂直方向上的差分。

梯度图像表示图像局部的细节，图像锐化实质是原图像和梯度图像相加(或加权求和)，增强图中的变化。

边缘检测需要进一步判断梯度图像中的局部极值点，一般通过对梯度图像进行阈值化来实现：设定一个阈值，凡是梯度值大于该阈值的变为 1，表示边缘点；小于该阈值的变为 0，表示非边缘点。可以看出，检测效果受到阈值的影响：阈值越低，能够检测出的边线越多，结果也就越容易受到图像噪声的影响；相反，阈值越高，检测出的边线越少，有可能会遗失较弱的边线。实际中可以在边缘检测前进行滤波，降低噪声的影响，也可以采用不同的方法选择合适的阈值。

【例 5.14】 设原图像 $\boldsymbol{f}=\begin{bmatrix}3&3&3&3&3\\3&7&7&7&3\\3&7&7&7&3\\3&7&7&7&3\\3&3&3&3&3\end{bmatrix}$，对该图像进行处理，生成梯度图像。

解： 按照梯度算子公式，计算图中每一像素和其右邻点、下邻点差值的绝对值和，并赋给该像素，不存在右邻点和下邻点的直接赋背景值 0。

$$\begin{bmatrix}3&\boxed{3}&\boxed{3}&3&3\\3&\boxed{7}&7&7&3\\3&7&7&7&3\\3&7&7&7&3\\3&3&3&3&3\end{bmatrix}$$

$$\begin{aligned}g(0,0)&=|f(1,0)-f(0,0)|+|f(0,1)-f(0,0)|\\&=|3-3|+|3-3|=0\\&\cdots\\g(4,0)&=0\\g(0,1)&=|f(1,1)-f(0,1)|+|f(0,2)-f(0,1)|\\&=|7-3|+|3-3|=4\\&\cdots\\g(4,1)&=0\\&\cdots\\g(0,4)&=g(1,4)=g(2,4)=g(3,4)=g(4,4)=0\end{aligned}$$

得最终结果：

$$\boldsymbol{g}=\begin{bmatrix}0&4&4&4&0\\4&0&0&4&0\\4&0&0&4&0\\4&4&4&8&0\\0&0&0&0&0\end{bmatrix}$$

【例 5.15】　编写程序，实现基于梯度算子的图像处理。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
Image = Image / 255
height, width = np.shape(Image)
gradI = np.zeros([height, width])
for y in range(height - 1):
    for x in range(width - 1):
        gradI[y, x] = np.abs(Image[y, x + 1] - Image[y, x])
                    + np.abs(Image[y + 1, x] - Image[y, x]) #梯度运算
sharpI = Image + gradI                                      #图像锐化
thresh = np.mean(gradI) + 2 * np.std(gradI)                 #设置阈值
edgeI = np.zeros([height, width])
edgeI[gradI >= thresh] = 1                                  #边缘检测
cv.imshow("Original image", Image)
cv.imshow("Gradient image", gradI)
cv.imshow("Sharp image", sharpI)
cv.imshow("Edge image", edgeI)
cv.waitKey()
```

程序运行效果如图 5-25 所示。

(a) 原图　(b) 梯度图像

(c) 锐化图像　(d) 边缘检测

图 5-25　梯度算子的处理效果

2. 其他一阶微分算子

如果修改差分运算中的 Δx 和 Δy，综合像素邻域内多个邻点的差分运算结果估计像素的梯度值，可以得到不同的一阶微分算子。

1）Roberts 算子

Roberts 算子是通过交叉求微分检测局部变化，其运算公式为

$$g(x,y)=|f(x,y)-f(x+1,y+1)|+|f(x+1,y)-f(x,y+1)| \tag{5-30}$$

用模板表示为

$$\boldsymbol{H}_1=\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} \quad \boldsymbol{H}_2=\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} \tag{5-31}$$

【例 5.16】 编写程序，采用模板运算实现 Roberts 滤波。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
Image = Image / 255
height, width = np.shape(Image)
H1, H2 = np.array([[1, 0], [0, -1]]), np.array([[0, 1], [-1, 0]])  #Roberts 模板
G1 = cv.filter2D(Image, ddepth=-1, kernel=H1)                      #模板运算
G2 = cv.filter2D(Image, ddepth=-1, kernel=H2)
gradI = np.abs(G1) + np.abs(G2)                                    #Roberts 滤波图像
cv.imshow("Original image", Image)
cv.imshow("Robets gradient image", gradI)
cv.waitKey()
```

程序运行结果如图 5-26 所示。

(a) 原图　　(b) Roberts滤波图像

图 5-26　Roberts 滤波效果

2）Sobel 算子

Sobel 算子是一种 3×3 模板下的微分算子，定义为

$$\begin{aligned} f_y&=|f(x-1,y+1)+2f(x,y+1)+f(x+1,y+1)|-\\ &\quad |f(x-1,y-1)+2f(x,y-1)+f(x+1,y-1)|\\ f_x&=|f(x+1,y-1)+2f(x+1,y)+f(x+1,y+1)|-\\ &\quad |f(x-1,y-1)+2f(x-1,y)+f(x-1,y+1)|\\ g&=|f_x|+|f_y| \end{aligned} \tag{5-32}$$

用模板表示为

$$\boldsymbol{H}_x=\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix} \quad \boldsymbol{H}_y=\begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix} \tag{5-33}$$

Sobel 算子引入平均因素，对图像中随机噪声有一定的平滑作用；相隔两行或两列求差分，故边缘两侧的元素得到了增强，边缘显得粗而亮。

Scharr 算子是 Sobel 算子的改进，其模板表示为

$$\boldsymbol{H}_x=\begin{bmatrix}-3&0&3\\-10&0&10\\-3&0&3\end{bmatrix}\quad \boldsymbol{H}_y=\begin{bmatrix}-3&-10&-3\\0&0&0\\3&10&3\end{bmatrix} \tag{5-34}$$

进一步加重了四邻点在差异计算中的权重。

3）Prewitt 算子

Prewitt 算子与 Sobel 算子思路类似，但模板系数不一样，如式(5-35)所示：

$$\boldsymbol{H}_x=\begin{bmatrix}-1&0&1\\-1&0&1\\-1&0&1\end{bmatrix}\quad \boldsymbol{H}_y=\begin{bmatrix}-1&-1&-1\\0&0&0\\1&1&1\end{bmatrix} \tag{5-35}$$

4）其他微分算子

以上 3 种微分算子都是在 3×3 邻域内定义的，微分算子也可以扩展到更大的邻域，如 4×4 邻域，模板为

$$\boldsymbol{H}_x=\begin{bmatrix}-3&-1&1&3\\-3&-1&1&3\\-3&-1&1&3\\-3&-1&1&3\end{bmatrix}\quad \boldsymbol{H}_y=\begin{bmatrix}-3&-3&-3&-3\\-1&-1&-1&-1\\1&1&1&1\\3&3&3&3\end{bmatrix} \tag{5-36}$$

如 5×5 邻域，模板为

$$\boldsymbol{H}_x=\begin{bmatrix}-2&-1&0&1&2\\-2&-1&0&1&2\\-2&-1&0&1&2\\-2&-1&0&1&2\\-2&-1&0&1&2\end{bmatrix}\quad \boldsymbol{H}_y=\begin{bmatrix}-2&-2&-2&-2&-2\\-1&-1&-1&-1&-1\\0&0&0&0&0\\1&1&1&1&1\\2&2&2&2&2\end{bmatrix} \tag{5-37}$$

以上微分算子均可以表示为模板，可以采用模板运算获取对应的梯度图像。

3. 边缘方向

边缘除了具有强度特征，还有方向特征。梯度方向是函数最大增长的方向，例如从黑到白，一般取为(−180°,180°]。边缘方向与梯度方向垂直，可以按式(5-38)计算。

$$\theta(x,y)=\arctan(f_x/f_y) \tag{5-38}$$

由于微分算子可以用模板表示，通过旋转将模板扩展为 8 个或者更多模板，如 Sobel 算子扩展为 8 个模板：

$$\boldsymbol{H}_1=\begin{bmatrix}-1&-2&-1\\0&0&0\\1&2&1\end{bmatrix}\quad \boldsymbol{H}_2=\begin{bmatrix}0&-1&-2\\1&0&-1\\2&1&0\end{bmatrix}\quad \boldsymbol{H}_3=\begin{bmatrix}1&0&-1\\2&0&-2\\1&0&-1\end{bmatrix}\quad \boldsymbol{H}_4=\begin{bmatrix}2&1&0\\1&0&-1\\0&-1&-2\end{bmatrix}$$

$$\boldsymbol{H}_5=\begin{bmatrix}1&2&1\\0&0&0\\-1&-2&-1\end{bmatrix}\quad \boldsymbol{H}_6=\begin{bmatrix}0&1&2\\-1&0&1\\-2&-1&0\end{bmatrix}\quad \boldsymbol{H}_7=\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}\quad \boldsymbol{H}_8=\begin{bmatrix}-2&-1&0\\-1&0&1\\0&1&2\end{bmatrix}$$

每个模板作用在图像上，取最大响应作为梯度幅度

$$g=\max_i(\boldsymbol{H}_i\otimes f) \tag{5-39}$$

梯度的方向为对应模板确定的方向，进而确定边缘方向。

Prewitt 算子模板也可以通过旋转扩展到 8 个，同 Sobel 算子一样，这里不再赘述。

4. 仿真实现

在OpenCV中，Sobel函数可以实现Sobel滤波，Scharr函数可以实现Scharr滤波，spatialGradient函数可以利用Sobel算子计算水平、垂直差分图像，getDerivKernels函数可以获取滤波器系数，调用格式如下：

```
cv.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]]) -> dst
cv.Scharr(src, ddepth, dx, dy[, dst[, scale[, delta[, borderType]]]]) -> dst
cv.spatialGradient(src[, dx[, dy[, ksize[, borderType]]]]) -> dx, dy
cv.getDerivKernels(dx, dy, ksize[, kx[, ky[, normalize[, ktype]]]]) -> kx, ky
```

Sobel函数的参数src是原图像，可以是灰度或彩色图像数据；dx、dy分别是x和y方向的导数阶数；ksize指定Sobel滤波器的尺寸，可选FILTER_SCHARR、1、3、5、7；scale是差分值的比例因子，默认为1；delta是叠加在计算结果上的偏移值。

spatialGradient函数的参数dx和dy分别是水平和垂直差分图像，ksize取值为3。

getDerivKernels函数的参数normalize指定是否对滤波器系数进行归一化，ktype是滤波器系数类型，可以是CV_32f或CV_64F，kx和ky是滤波器行、列系数。

【例5.17】 编写程序，实现Sobel滤波并统计边缘方向。

解：程序如下。

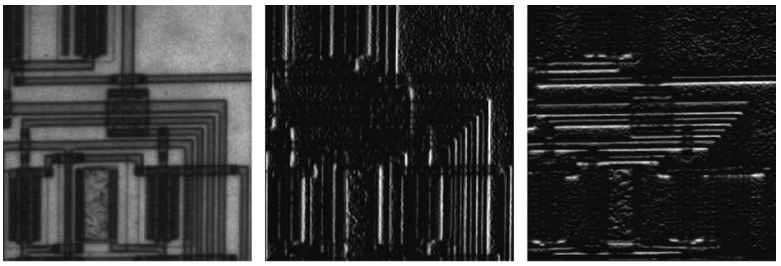
```
import cv2 as cv
import numpy as np
Image = cv.imread("circuit.tif", cv.IMREAD_GRAYSCALE)
Image = Image / 255
grad_h = cv.Sobel(Image, -1, 1, 0)                          #水平差分
grad_v = cv.Sobel(Image, -1, 0, 1)                          #垂直差分
gradI = np.abs(grad_h) + np.abs(grad_v)                     #差分绝对值和
thresh = np.mean(gradI) + 2 * np.std(gradI)
dirI = np.zeros(np.shape(Image))                            #计算梯度值较大处的边缘方向
dirI[gradI > thresh] = np.arctan(grad_h[gradI > thresh] /
                       (grad_v[gradI > thresh] + 0.001)) / np.pi + 0.5
kx, ky = cv.getDerivKernels(dx = 0, dy = 1, ksize = cv.FILTER_SCHARR, normalize = False)
print('行系数:', kx.T)
print('列系数:\n', ky)
cv.imshow("Original image", Image)
cv.imshow("Sobel gradient image fx", grad_h)
cv.imshow("Sobel gradient image fy", grad_v)
cv.imshow("Sobel gradient image", gradI)
cv.imshow("Edge direction image", dirI)
cv.waitKey()
```

运行程序，输出Scharr垂直差分滤波器的行和列系数：

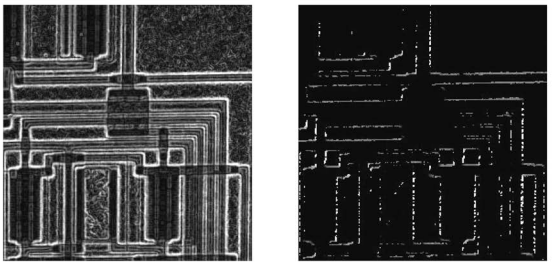
```
行系数 kx: [[ 3. 10. 3.]]
列系数 ky:
    [[-1.]
    [ 0.]
    [ 1.]]
```

Scharr垂直差分滤波器等于ky×kx。滤波结果如图5-27所示。

OpenCV中ximgproc模块的EdgeDrawing类可用于边缘检测，梯度算子变量GradientOperator可取Prewitt、Sobel、Scharr和LSD(直线段检测)，默认情况下为Prewitt算子，detectEdges函数可用于灰度图像的边缘检测，getEdgeImage函数可以用于获取检测的边缘图像，getGradientImage函数可以用于获取梯度图像，createEdgeDrawing函数可以用于创建类实例对象，各函数调用格式如下：



(a) 原图　(b) 水平差分图像　(c) 垂直差分图像



(d) 水平、垂直差分绝对值和图像　(e) 边缘方向图像

图 5-27　Sobel 滤波效果

```
cv.ximgproc.EdgeDrawing.detectEdges(src) -> None
cv.ximgproc.EdgeDrawing.getEdgeImage([, dst]) -> dst
cv.ximgproc.EdgeDrawing.getGradientImage([, dst]) -> dst
cv.ximgproc.createEdgeDrawing() -> retval
```

【例 5.18】　编写程序，实现基于 Prewitt 算子的滤波与边缘检测。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
edge = cv.ximgproc.createEdgeDrawing()
edge.detectEdges(Image)
edgeI = edge.getEdgeImage()                          #获取边缘图像
gradI = edge.getGradientImage().astype(np.uint8)     #获取梯度图像
cv.imshow("Original image", Image)
cv.imshow("Prewitt gradient image", gradI)
cv.imshow("Prewitt edge image", edgeI)
cv.waitKey()
```

程序运行结果如图 5-28 所示。

(a) 原图　(b) Prewitt梯度图像　(c) Prewitt边缘图像

图 5-28　Prewitt 算子的滤波及边缘检测效果

在微分运算中，模板大时，边缘两侧的像素得到增强，边缘较粗，定位不精确但方向比较精确；模板小时，边缘较细，定位精确但方向精确度低。

启发：

较大模板和较小模板在检测边缘时各有优势，同理，“人不同能，而任之以一事，不可责遍成”，认识自己，发挥个人的优势和特长，正确看待不足，切勿求全责备。

5.4.3 二阶微分算子

拉普拉斯算子是二阶微分算子，定义如下：

$$\nabla^2 f=\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2} \tag{5-40}$$

用差分代替

$$\frac{\partial^2 f}{\partial x^2}=\Delta_x f(x+1,y)-\Delta_x f(x,y)=[f(x+1,y)-f(x,y)]-[f(x,y)-f(x-1,y)]$$

$$=f(x+1,y)+f(x-1,y)-2f(x,y)$$

$$\frac{\partial^2 f}{\partial y^2}=\Delta_y f(x,y+1)-\Delta_y f(x,y)=[f(x,y+1)-f(x,y)]-\{f(x,y)-f(x,y-1)]$$

$$=f(x,y+1)+f(x,y-1)-2f(x,y)$$

所以

$$\nabla^2 f=f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)-4f(x,y) \tag{5-41}$$

用模板表示为

$$\boldsymbol{H}_1=\begin{bmatrix}0 & 1 & 0\\ 1 & -4 & 1\\ 0 & 1 & 0\end{bmatrix} \quad 或 \quad \boldsymbol{H}_1=\begin{bmatrix}0 & -1 & 0\\ -1 & 4 & -1\\ 0 & -1 & 0\end{bmatrix} \tag{5-42}$$

可以扩展为

$$\boldsymbol{H}_1=\begin{bmatrix}1 & 1 & 1\\ 1 & -8 & 1\\ 1 & 1 & 1\end{bmatrix} \quad 或 \quad \boldsymbol{H}_1=\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix} \tag{5-43}$$

拉普拉斯锐化模板表示为

$$\boldsymbol{H}=\begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix} \quad 或 \quad \boldsymbol{H}=\begin{bmatrix}-1 & -1 & -1\\ -1 & 9 & -1\\ -1 & -1 & -1\end{bmatrix} \tag{5-44}$$

【例 5.19】 利用式(5-41)对例 5.14 中的图像 $\boldsymbol{f}$ 进行运算。

解： 按照拉普拉斯算子公式，对图中每个像素进行计算，模板罩不住的像素直接赋背景值 0。

$$\begin{bmatrix}3 & 3 & 3 & 3 & 3\\ 3 & 7 & 7 & 7 & 3\\ 3 & 7 & 7 & 7 & 3\\ 3 & 7 & 7 & 7 & 3\\ 3 & 3 & 3 & 3 & 3\end{bmatrix}$$

以(1,1)点为例，即对图中模板所覆盖的像素进行运算：

$$\nabla^2 f=f(2,1)+f(0,1)+f(1,2)+f(1,0)-4f(1,1)=-8$$

每个点进行同样运算后，得最终结果：

$$\boldsymbol{g}=\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & -8 & -4 & -8 & 0\\ 0 & -4 & 0 & -4 & 0\\ 0 & -8 & -4 & -8 & 0\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

如果显示结果，图像像素值不能为负，可以采用如下两种处理方法将像素值负值转换为正值：

(1) 取绝对值，得到类似于梯度图像的效果；

(2) 整体加一个正整数(图中最小值的绝对值)，得到类似浮雕的效果。

【例 5.20】 编写程序，实现基于拉普拉斯算子的处理。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
Image = Image / 255
H = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
G = cv.filter2D(Image, ddepth = -1, kernel = H)                    #拉普拉斯滤波
result1 = np.abs(G)                                                #取绝对值
result2 = G - G.min() if G.min() < 0 else G.copy()                 #整体加正整数
result3 = Image + result1                                          #锐化图像
cv.imshow("Original image", Image)
cv.imshow("Absolute value of Laplacian", result1)
cv.imshow("Adding an integer", result2)
cv.imshow("Sharp image", result3)
cv.waitKey()
```

第 10 集
微课视频

程序运行结果如图 5-29 所示。

(a) 滤波图像　(b) 浮雕效果　(c) 锐化图像

图 5-29　拉普拉斯算子的处理效果

OpenCV 中 Laplacian 函数可以实现拉普拉斯滤波，其调用格式如下：

```
cv.Laplacian(src, ddepth[, dst[, ksize[, scale[, delta[, borderType]]]]]) -> dst
```

当参数 ksize 为 1 时，滤波模板如式(5-42)所示；参数 src 可以是灰度或彩色图像数据。采用 Laplacian 函数改写例 5.20，只需要将程序中加粗的两行代码修改为

```
G = cv.Laplacian(Image, -1)
```

关于二阶微分过零点检测，请扫描二维码，查看讲解。

5.4.4 高斯滤波与微分运算

1. 高斯函数

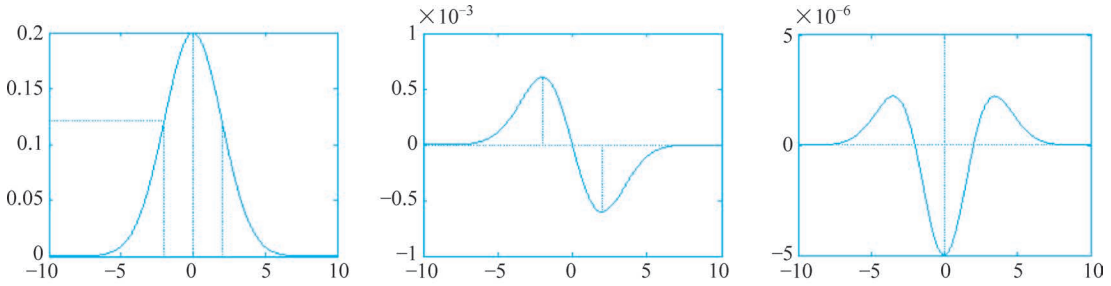
二维高斯函数如式(5-22)所示，其一阶导数为

$$\nabla h(x,y)=\frac{\partial h}{\partial x}+\frac{\partial h}{\partial y}=\left(-\frac{x+y}{2\pi\sigma^4}\right)\mathrm{e}^{\left(-\frac{x^2+y^2}{2\sigma^2}\right)} \tag{5-45}$$

二维高斯函数的二阶导数为

$$\nabla^2 h(x,y)=\frac{\partial^2 h}{\partial x^2}+\frac{\partial^2 h}{\partial y^2}=\frac{1}{2\pi\sigma^2}\left(\frac{x^2+y^2-2\sigma^2}{\sigma^4}\right)\mathrm{e}^{\left(-\frac{x^2+y^2}{2\sigma^2}\right)} \tag{5-46}$$

高斯函数及其一阶、二阶导数在滤波运算中非常重要，图 5-30 所示为均值为 0、标准差为 2 的一维高斯函数及其一阶和二阶导数图形。



(a) 一维高斯函数　(b) 一阶导数　(c) 二阶导数

图 5-30　均值为 0 标准差为 2 的一维高斯函数及其一阶、二阶导数图形

对一维高斯函数及其一阶、二阶导数进行分析，得到高斯函数的相关特性如下。

(1) 随着远离原点，权值逐渐减小到零，离中心较近的图像值比远处的图像值更重要；标准差 σ 决定邻域范围，总权值的 95%包含在 2σ 的中间范围内。这个特性使得高斯函数常被用来作为权值，高斯滤波就是利用了这个特性。

(2) 一维高斯函数的二阶导数具有光滑的中间突出部分，该部分函数值为负，还有两个光滑的侧边突出部分，该部分值为正。零交叉位于 $-\sigma$ 和 $+\sigma$ 处，与 $h(x)$ 的拐点和 $h'(x)$ 的极值点对应。

(3) 一维高斯函数绕垂直轴旋转可得到各向同性的二维高斯函数形式(在任意过原点的切面上具有相同的一维高斯截面)，其二阶导数形式好像一个宽边帽或称为墨西哥草帽。

从数学推导上看，帽子的空腔口沿 $z=h(x,y)$ 轴向上，但在显示和滤波应用中空腔口一般朝下，即中间突起的部分为正，帽边为负。

2. LoG 算子

图像常常受到随机噪声干扰，进行边缘检测时常把噪声当作边缘点而检测出来。针对这个问题，Marr 和 Hildreth 提出了一种解决思路：首先对原始图像作最佳平滑处理，再求边缘。这样就需要解决以下两个问题。

(1) 选择什么样的滤波作平滑处理；

(2) 选择什么算子来检测边缘。

Marr 用高斯函数先对图像作平滑处理，即将高斯函数 $h(x,y)$ 与图像函数 $f(x,y)$ 作卷积，得到一个平滑的图像函数，再对该函数做拉普拉斯运算，提取边缘。

可以证明$\nabla^2[f(x,y)*h(x,y)]=f(x,y)*\nabla^2h(x,y)$，即卷积运算和求二阶导数的顺序可以交换，$\nabla^2h(x,y)$如式(5-46)所示。

$\nabla^2h(x,y)$称为 LoG 滤波器(Laplacian of Gaussian Algorithm)，也称为 Marr-Hildrech 算子。σ 称为尺度因子，大的值可用来检测模糊的边缘，小的值可用来检测聚焦良好的图像细节。当图像边缘模糊或噪声较大时，检测过零点能提供较可靠的边缘位置。LoG 滤波器的形状如图 5-31 所示。

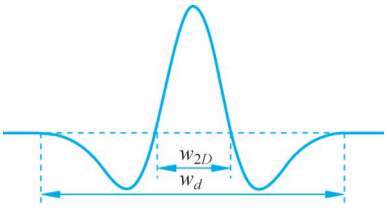


图 5-31　∇^2h 的横截面

LoG 滤波器的大小由 σ 的数值或等价地由 w_{2D} 的数值来确定。为了不使函数被过分地截短，应在足够大的窗口内作计算，窗口宽度通常取 $w_d\geqslant 3.6w_{2D}$，而 $w_{2D}=2\sigma$。

LoG 滤波器可以采用模板形式，设定标准差 σ 和模板尺寸，根据式(5-46)计算模板中各点的值。关于 LoG 滤波器模板生成，请扫描二维码，查看讲解。

对图像进行 LoG 滤波后，再采用过零点检测，则实现边缘检测。

SciPy 库 ndimage 模块的 gaussian_laplace 函数可以实现 LoG 滤波，其调用格式如下：

```
scipy.ndimage.gaussian_laplace(input, sigma, output = None, mode = 'reflect', cval = 0.0, **kwargs)
```

参数 input 可以是灰度或彩色图像数据；mode 指明边界像素的填塞方法，可取'reflect'、'constant'、'nearest'、'mirror'、'wrap'等值，cval 是当边界填塞方法为'constant'时要填充的值。

LoG 滤波也可以通过先进行高斯滤波再进行拉普拉斯滤波实现。

第 11 集
微课视频

【例 5.21】　编程实现 LoG 滤波。

解：程序如下。

```
import cv2 as cv
import numpy as np
from scipy.ndimage import gaussian_laplace
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
Image = Image / 255
height, width = np.shape(Image)
LG = gaussian_laplace(Image, 1)                                    #进行 LoG 滤波
result1 = LG - LG.min() if LG.min() < 0 else LG                    #整体加正整数,方便观察
temp1 = cv.GaussianBlur(Image, ksize = (7, 7), sigmaX = 1, sigmaY = 0)     #先进行高斯滤波
temp2 = cv.Laplacian(temp1, -1)                                    #再进行拉普拉斯滤波
result2 = temp2 - temp2.min() if temp2.min() < 0 else temp2
cv.imshow("Original image", Image)
cv.imshow("LoG", result1)
cv.imshow("G-L", result2)
cv.waitKey()
```

程序运行结果如图 5-32 所示。图 5-32(b)和图 5-32(c)分别是 LoG 滤波和先进行高斯滤波再进行拉普拉斯滤波的结果，两者一致(在计算误差范围内)。

3. DoG 算子

二维高斯函数对 σ 求偏导，得

$$\frac{\partial h}{\partial\sigma}=\frac{1}{2\pi\sigma^3}\left(\frac{x^2+y^2}{\sigma^2}-2\right)e^{\left(-\frac{x^2+y^2}{2\sigma^2}\right)} \tag{5-47}$$

由式(5-46)可知

(a) 原图　　(b) LoG滤波　　(c) 先进行高斯滤波再进行拉普拉斯滤波

图 5-32　LoG 算子处理效果

$$\frac{\partial h}{\partial \sigma}=\sigma \nabla^2 h \tag{5-48}$$

根据导数定义，可得

$$\nabla^2 h \approx \frac{h(k\sigma)-h(\sigma)}{\sigma(k\sigma-\sigma)}=\frac{h(k\sigma)-h(\sigma)}{\sigma^2(k-1)} \tag{5-49}$$

用高斯差分(Difference of Gaussians，DoG)

$$h(k\sigma)-h(\sigma)=\sigma^2(k-1)\nabla^2 h \tag{5-50}$$

代替 LoG。其中，$k>1$ 是尺度参数。

因此，DoG 算子是用两个不同标准差的高斯函数平滑图像，将结果相减，实现滤波，再利用过零点检测等方式处理滤波图像，实现边缘检测。如果在图像处理中已经建立了高斯差分金字塔，采用 DoG 算子更节省计算资源。

4. Canny 算子

Canny 边缘检测算法是 Canny 于 1986 年提出的一个多级边缘检测算法，被很多人认为是边缘检测的最优算法。

最优边缘检测的三个主要评价标准如下。

(1) 低错误率。标识出尽可能多的实际边缘，同时尽可能地减少噪声产生的误报。

(2) 对边缘的定位准确。标识出的边缘要与图像中的实际边缘尽可能接近。

(3) 最小响应。图像中的边缘最好只标识一次，并且可能存在的图像噪声部分不应标识为边缘。

Canny 算子结合了这三个准则，采用高斯滤波器对图像做平滑处理，在平滑处理后计算图像的每个像素的梯度幅值和方向；利用梯度方向，采用非极大抑制(Nonmaximum Suppression)方法细化边缘；用双阈值算法检测和连接边缘，最终得到细化的边缘图像。

利用 Canny 算子进行边缘检测的主要步骤如下。

(1) 使用高斯平滑滤波器卷积降噪。

(2) 计算平滑后图像的梯度幅值和方向，可以采用不同的梯度算子。

(3) 对梯度幅值应用非极大抑制，其过程是找出图像梯度中的局部极大值点，把其他非局部极大值点置零。

(4) 使用双阈值算法检测和连接边缘。

高阈值 T_{high} 被用来找到每一条线段：如果某像素位置的梯度幅值超过 T_{high}，表明找到了一条线段的起始。

低阈值 T_{low} 被用来确定线段上的点：以上一步找到的线段起始出发，在其邻域内搜寻梯

度幅值大于 T_{low} 的像素，保留为边缘点；梯度幅值小于 T_{low} 的像素被置为背景。

OpenCV 中的 Canny 函数可以实现基于 Canny 算子的边缘检测，调用格式如下：

```
cv.Canny(image, threshold1, threshold2[, edges[, apertureSize[, L2gradient]]]) -> edges
cv.Canny(dx, dy, threshold1, threshold2[, edges[, L2gradient]]) -> edges
```

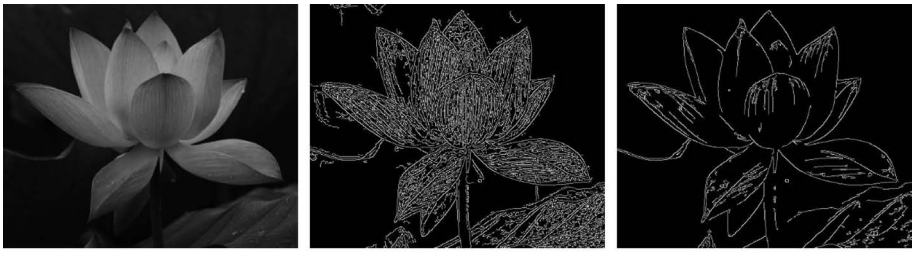
参数 image 是 8 位的灰度或彩色图像，threshold1 和 threshold2 是双阈值，先后顺序无关，自动检测两者中较小的值作为低阈值；dx 和 dy 是 16 位的水平和垂直方向上的差分图像。L2gradient 是计算梯度幅值的方法，设为 True 时选择 L_2 范数(式(5-27))，设为 False 时选择 L_1 范数(式(5-28))。

【例 5.22】 编写程序，实现基于 Canny 算子的边缘检测。

解： 程序如下。

```
import cv2 as cv
import numpy as np
from scipy.ndimage import gaussian_laplace
Image = cv.imread("lotus.jpg", cv.IMREAD_GRAYSCALE)
low_T, high_T = 10, 30                                    # 设置高低阈值
result1 = cv.Canny(Image, low_T, high_T)                  # Canny 边缘检测
result2 = cv.Canny(Image, high_T * 4, low_T * 4)          # 提高阈值进行边缘检测
cv.imshow("Original image", Image)
cv.imshow("Canny edge image(low threshold)", result1)
cv.imshow("Canny edge image(high threshold)", result2)
cv.waitKey()
```

Canny 边缘检测效果如图 5-33 所示。当阈值较小时，检测到更多的边缘。



(a) 原图　(b) 阈值小　(c) 阈值大

图 5-33　Canny 算子边缘检测

第 12 集
微课视频

关于 Canny 算子分析与实现，请扫描二维码，查看讲解。

5.5　频域滤波

图像信号在频域表示，变换系数反映了某些图像特征，可以在频域进行滤波，达到图像增强的目的。频域滤波可以表示为

$$G(u,v)=H(u,v)F(u,v) \tag{5-51}$$

其中，$F(u,v)$为图像 $f(x,y)$的正交变换，$H(u,v)$为频域滤波器传递函数。频域滤波就是选择合适的 $H(u,v)$对 $F(u,v)$进行调整，经逆变换得到滤波输出图像 $g(x,y)$。

5.5.1　低通滤波

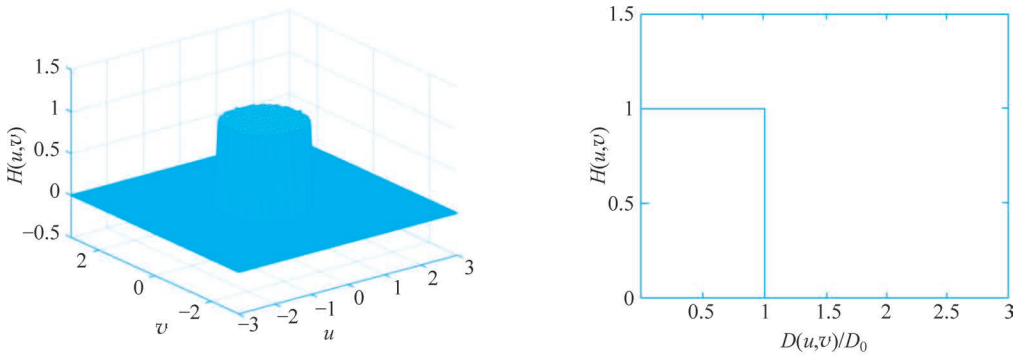
由于噪声表现为高频成分，因此可以通过构造一个频域低通滤波器，滤除噪声。

1. 理想低通滤波

理想低通滤波器的传递函数为

$$H(u,v)=\begin{cases}1, & D(u,v)\leqslant D_0\\ 0, & D(u,v)>D_0\end{cases} \tag{5-52}$$

其中，$D_0>0$ 为理想低通滤波器的截止频率，$D(u,v)=\sqrt{u^2+v^2}$ 为点(u,v)到频域原点的距离。理想低通滤波器传递函数及其剖面图如图 5-34 所示。



(a) 传递函数　　(b) 剖面图

图 5-34　理想低通滤波器

经过理想低通滤波，小于 D_0 的频率被无损保留，大于 D_0 的频率被滤除掉。但由于滤除的高频分量中含有大量的边缘信息，因此在抑制噪声的同时会出现图像边缘模糊的现象。

【例 5.23】　设计截断频率不同的理想低通滤波器，对图像进行低通滤波。

解：程序如下。

```
import cv2 as cv
import numpy as np
Image = (cv.imread("noise_sp.jpg", cv.IMREAD_GRAYSCALE)).astype(np.float32)
cv.imshow("Original image", Image / 255)                    #打开一幅高斯噪声图像
height, width = np.shape(Image)
DFT = np.fft.fftshift(cv.dft(Image, flags = cv.DFT_COMPLEX_OUTPUT))     #DFT
H = np.zeros([height, width])
diag = np.sqrt(height ** 2 + width ** 2)
D0 = [0.05 * diag, 0.1 * diag]                              #设置截止频率
X, y = np.linspace(0, width, width), np.linspace(0, height, height)
xv, yv = np.meshgrid(x, y)
centerx, centery = width // 2, height // 2
d = np.sqrt((xv - centerx) ** 2 + (yv - centery) ** 2)   #各点对应的频率
for i in range(2):
    H = H * 0
    H[d <= D0[i]] = 1                                       #设置低通滤波器
    G0, G1 = H * DFT[:, :, 0], H * DFT[:, :, 1]             #低通滤波,DFT为复数矩阵
    G = np.stack((G0, G1), 2)
    g = cv.idft(np.fft.ifftshift(G), flags = cv.DFT_REAL_OUTPUT + cv.DFT_SCALE)
    cv.imshow("D0 = %.2f" % D0[i], g / 255)
cv.waitKey()
```

程序运行结果如图 5-35 所示。可以看出，在截止频率较小时，噪声滤除较好，但有较严重的模糊现象；随着截止频率增大，保留的信息越来越多，滤波后的图像也越来越清晰。

(a) 噪声图像　(b) D_0为最大频率的0.05　(c) D_0为最大频率的0.1

图 5-35　理想低通滤波的效果

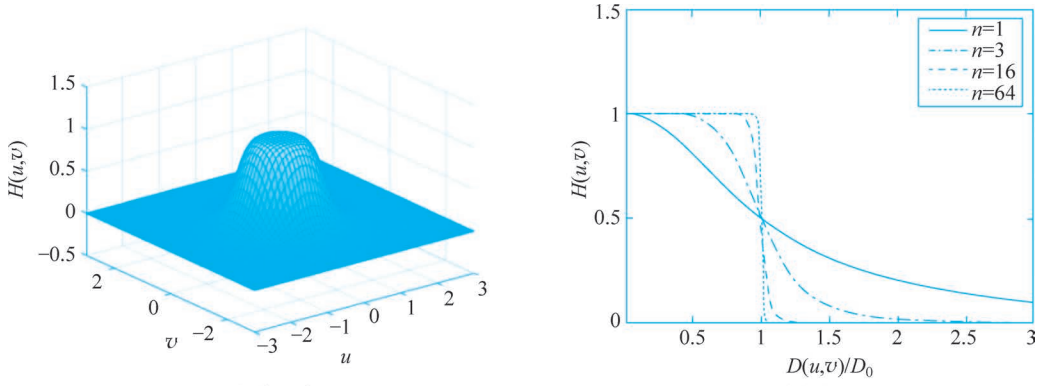
2. 巴特沃斯低通滤波

一个截止频率为 D_0 的 n 阶巴特沃斯低通滤波器的传递函数为

$$H(u,v)=\frac{1}{1+[D(u,v)/D_0]^{2n}} \quad 或 \quad H(u,v)=\frac{1}{1+(\sqrt{2}-1)[D(u,v)/D_0]^{2n}} \tag{5-53}$$

其中，n 为阶数，取正整数，用来控制曲线的衰减速度。

在 $D_0=10$、$n=3$ 时，巴特沃斯低通滤波器传递函数如图 5-36(a)所示，系统函数过渡平滑：通频带内的频率响应曲线最大限度平坦，没有起伏，阻频带内的频率响应曲线则逐渐下降为零，因此采用该滤波器在抑制噪声的同时，图像边缘的模糊程度大大减小且振铃效应减弱。图 5-36(b)为 $n=1,3,16,64$ 时变换函数径向剖面图，n 改变滤波器的形状：n 越大，滤波器越接近于理想滤波器。



(a) 传递函数　(b) 剖面图

图 5-36　巴特沃斯低通滤波器

【例 5.24】　设计截断频率不同的巴特沃斯低通滤波器，对图像进行低通滤波。

解：修改例 5.23 中设置低通滤波器的语句(加粗的代码)为

```
n = 3
H = 1 / (1 + (d / D0[i]) ** (2 * n))
```

即可实现巴特沃斯低通滤波。

程序运行结果如图 5-37 所示。和图 5-35 对比，同样的截止频率，巴特沃斯低通滤波相对理想低通滤波效果清晰。

3. 指数低通滤波

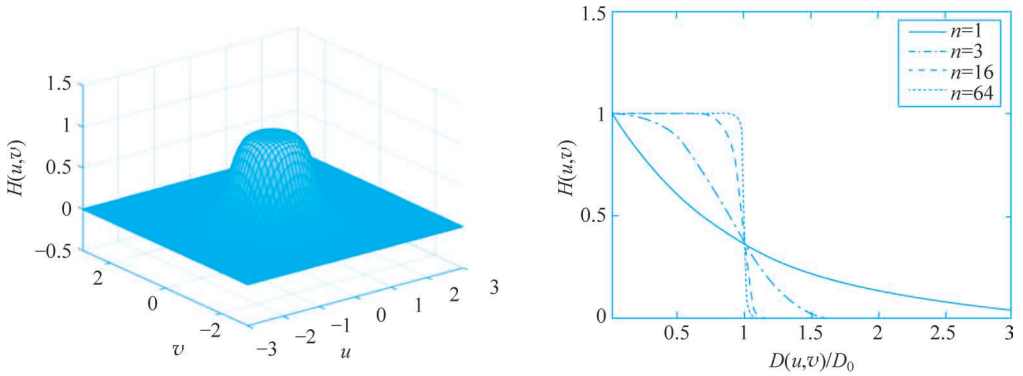
一个截止频率为 D_0 的 n 阶指数低通滤波器的传递函数 $H(u,v)$定义为

(a) 噪声图像　(b) D_0为最大频率的0.05　(c) D_0为最大频率的0.1

图 5-37　巴特沃斯低通滤波的效果

$$H(u,v)=\mathrm{e}^{-\left[\frac{D(u,v)}{D_0}\right]^n} \tag{5-54}$$

在 $D_0=10$、$n=3$ 时，指数低通滤波器传递函数如图 5-38(a)所示，系统函数过渡平滑。图 5-38(b)为 $n=1,3,16,64$ 时传递函数径向剖面图，n 改变滤波器的形状：n 越大，滤波器越接近于理想滤波器。



(a) 传递函数　(b) 剖面图

图 5-38　指数低通滤波器

4. 梯形低通滤波

梯形低通滤波器的传递函数介于理想低通滤波器和具有平滑过渡带的低通滤波器之间，为

$$H(u,v)=\begin{cases}1, & D(u,v)\leqslant D_0\\ \dfrac{D(u,v)-D_1}{D_0-D_1}, & D_0<D(u,v)\leqslant D_1\\ 0, & D(u,v)>D_1\end{cases} \tag{5-55}$$

其中，D_0 为截止频率，D_0、D_1 需满足 $D_0<D_1$。图 5-39 所示为梯形低通滤波器的传递函数及径向剖面图。

5.5.2　高通滤波

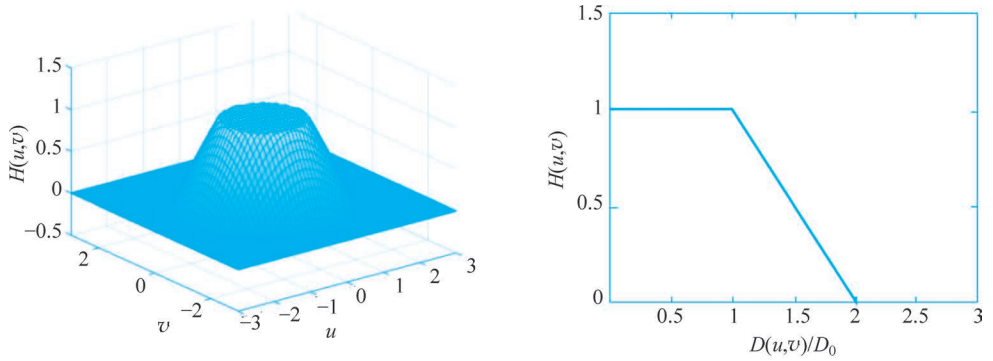
图像中的边缘对应于高频分量，所以图像锐化增强可以采用高通滤波器实现。

1. 理想高通滤波器

理想高通滤波器的传递函数为

$$H(u,v)=\begin{cases}0, & D(u,v)\leqslant D_0\\ 1, & D(u,v)>D_0\end{cases} \tag{5-56}$$

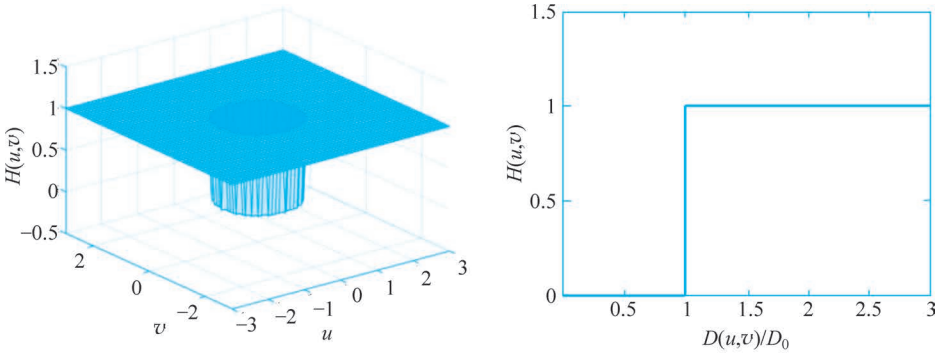
其中，$D_0>0$ 为截止频率。



(a) 传递函数　　(b) 剖面图

图 5-39　梯形低通滤波器

理想高通滤波器传递函数及其剖面图如图 5-40 所示，与理想低通滤波器正好相反。通过高通滤波器把以 D_0 为半径的圆内频率成分衰减掉，圆外的频率成分则无损通过。



(a) 传递函数　　(b) 剖面图

图 5-40　理想高通滤波器

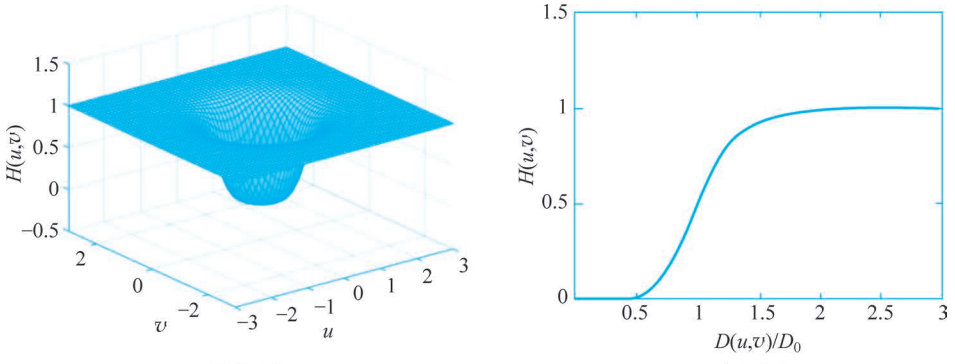
2. 巴特沃斯高通滤波器

一个截止频率为 D_0 的 n 阶巴特沃斯高通滤波器的传递函数为

$$H(u,v)=\frac{1}{1+[D_0/D(u,v)]^{2n}} \tag{5-57}$$

其中，D_0、$D(u,v)$的含义与理想高通滤波器中的含义相同。

在 $n=3$ 时，巴特沃斯高通滤波器传递函数及其径向剖面图如图 5-41 所示。



(a) 传递函数　　(b) 剖面图

图 5-41　巴特沃斯高通滤波器

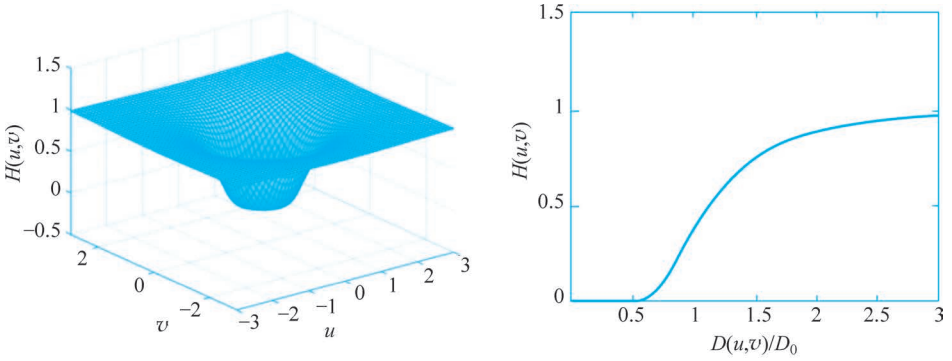
3. 指数高通滤波器

指数高通滤波器的传递函数为

$$H(u,v)=\exp\left\{-\left[\frac{D_0}{D(u,v)}\right]^n\right\} \tag{5-58}$$

其中，D_0、$D(u,v)$的含义与理想高通滤波器中的含义相同。

在 $n=3$ 时，指数高通滤波器传递函数及其径向剖面图如图 5-42 所示。



(a) 传递函数　　(b) 剖面图

图 5-42　指数高通滤波器

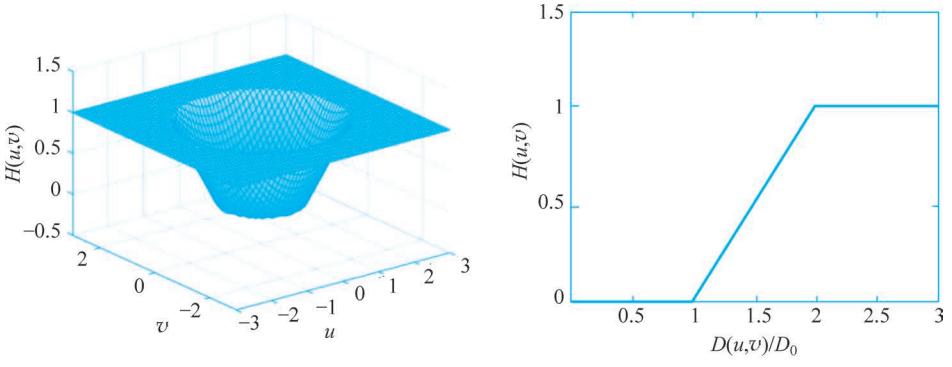
4. 梯形高通滤波器

梯形高通滤波器的传递函数为

$$H(u,v)=\begin{cases}0, & D(u,v)<D_0\\ \dfrac{1}{D_1-D_0}[D(u,v)-D_0], & D_0\leqslant D(u,v)\leqslant D_1\\ 1, & D(u,v)>D_1\end{cases} \tag{5-59}$$

其中，$D(u,v)$的含义与理想高通滤波器中的含义相同，D_1、D_0 为上下限截止频率。

梯形高通滤波器传递函数及其径向剖面图如图 5-43 所示。



(a) 传递函数　　(b) 剖面图

图 5-43　梯形高通滤波器

5.5.3　同态滤波

一般情况下，自然景物图像 $f(x,y)$可以表示为光源照度场(照明函数)$i(x,y)$和场景中物体反射光的反射场(反射函数)$r(x,y)$的乘积，称为图像的照度-反射模型，如式(5-60)所示。

$$f(x,y)=i(x,y)\cdot r(x,y) \tag{5-60}$$

其中，$0<i(x,y)<\infty$，$0<r(x,y)<1$。

近似认为，照明函数 $i(x,y)$ 描述景物的照明，其性质取决于照射源，与景物无关。反射函数 $r(x,y)$ 描述景物内容，其性质取决于成像物体的特性，而与照明无关。照明亮度一般是缓慢变化的，所以认为照明函数的频谱集中在低频段。反射函数随图像细节不同在空间快速变化，所以认为反射函数的频谱集中在高频段。这样，就可以根据式(5-60)将图像理解为高频分量与低频分量的乘积的结果。

同态滤波的基本原理是根据图像的照度-反射模型，对原始图像 $f(x,y)$ 中的反射分量 $r(x,y)$ 进行扩展，照明分量 $i(x,y)$ 进行压缩，以获得所要求的增强图像。照明函数以低频为主，反射函数以高频为主，因此，可在高通滤波函数的基础上设计同态滤波函数，即

$$H(u,v)=(\gamma_H-\gamma_L)\,\mathrm{High}(u,v)+\gamma_L \tag{5-61}$$

其中，γ_H 和 γ_L 分别表示高频分量频率场和低频分量频率场滤波特性，且当 $\gamma_H>1$、$0<\gamma_L<1$ 时，照明分量受到抑制，反射分量得到增强，从而突出图像的轮廓细节。同态滤波函数特性曲线如图 5-44 所示。

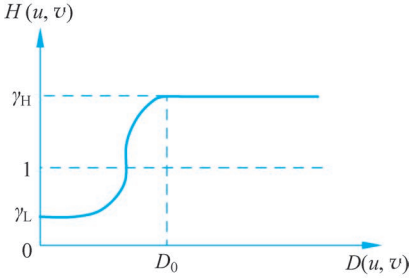


图 5-44　同态滤波函数特性曲线

同态滤波的具体算法步骤如下。

(1) 对图像函数 $f(x,y)$ 进行对数变换

$$\begin{aligned} z(x,y)&=\ln[f(x,y)]=\ln[i(x,y)\cdot r(x,y)]\\ &=\ln[i(x,y)]+\ln[r(x,y)] \end{aligned} \tag{5-62}$$

(2) 进行傅里叶变换

$$\begin{aligned} Z(u,v)&=\mathcal{F}\{z(x,y)\}=\mathcal{F}\{\ln i(x,y)\}+\mathcal{F}\{\ln r(x,y)\}\\ &=I(u,v)+R(u,v) \end{aligned} \tag{5-63}$$

(3) 进行同态滤波

$$\begin{aligned} S(u,v)&=H(u,v)Z(u,v)\\ &=H(u,v)I(u,v)+H(u,v)R(u,v) \end{aligned} \tag{5-64}$$

(4) 进行傅里叶逆变换

$$\begin{aligned} s(x,y)&=\mathcal{F}^{-1}\{S(u,v)\}\\ &=\mathcal{F}^{-1}\{H(u,v)I(u,v)\}+\mathcal{F}^{-1}\{H(u,v)R(u,v)\}\\ &=i'(x,y)+r'(x,y) \end{aligned} \tag{5-65}$$

(5) 进行指数变换，得到经同态滤波处理的图像

$$\begin{aligned} g(x,y)&=\exp\{s(x,y)\}=\exp(i'(x,y)+r'(x,y))\\ &=i_0(x,y)\cdot r_0(x,y) \end{aligned} \tag{5-66}$$

其中，$i_0(x,y)$ 是处理后的照明分量，$r_0(x,y)$ 是处理后的反射分量。

【例 5.25】 编写程序，对图像进行同态滤波增强。

解： 程序如下。

```
import cv2 as cv
import numpy as np
Image = (cv.imread("scene.jpg", cv.IMREAD_GRAYSCALE)).astype(np.float32)
cv.imshow("Original image", Image / 255)
height, width = np.shape(Image)
```

```
logI = np.log(Image + 1)                                          #取对数
DFT = np.fft.fftshift(cv.dft(logI, flags=cv.DFT_COMPLEX_OUTPUT))  #DFT
gammaH, gammaL = 2.0, 0.5
diag = np.sqrt(height ** 2 + width ** 2)
D0 = 0.6 * diag
x, y = np.linspace(0, width, width), np.linspace(0, height, height)
xv, yv = np.meshgrid(x, y)
centerx, centery = width // 2, height // 2
d = np.sqrt((xv - centerx) ** 2 + (yv - centery) ** 2)
n = 3
High = np.exp(-np.power(D0 / d, n))                               #设计高通滤波器
H = (gammaH - gammaL) * High + gammaL                             #设计同态滤波器
G0, G1 = H * DFT[:, :, 0], H * DFT[:, :, 1]
G = np.stack((G0, G1), 2)
g = cv.idft(np.fft.ifftshift(G), flags=cv.DFT_REAL_OUTPUT + cv.DFT_SCALE)
g = np.exp(g)
g = (g / np.max(g) * 255).astype(np.uint8)
cv.imshow("Homomorphic filtering", g)
cv.waitKey()
```

程序运行结果如图 5-45 所示。可以看出，原始的光照不均匀图像经过同态滤波增强处理后，暗区得到增强。

(a) 原图　　(b) 同态滤波

图 5-45　同态滤波效果

5.6　综合实例

【例 5.26】　编写程序，实现灰度图像背景虚化效果。

解：设计思路如下。

(1) 对原灰度图像进行高强度高斯滤波，实现图像虚化，用做背景。

(2) 对原灰度图像进行锐化滤波，实现图像增强，用做前景。

(3) 采用交互式方法，在图像上选定前景区域，生成模板。

(4) 将模板进行均值滤波，将边缘部分羽化，实现前景向背景的渐变过渡。

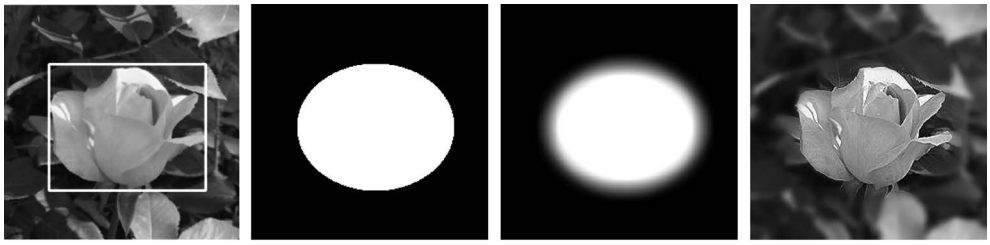
(5) 将模板和前景相乘，反色模板和背景相乘，两者相加，实现背景虚化、前景锐化增强的效果。

程序如下。

```
import cv2 as cv
import numpy as np
Image = cv.imread("flower.jpg", cv.IMREAD_GRAYSCALE)
height, width = np.shape(Image)
back = cv.GaussianBlur(Image, ksize=(21, 21), sigmaX=3, sigmaY=0) #背景虚化
```

```
H = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
fore = cv.filter2D(Image, ddepth = -1, kernel = H)                    #前景锐化
cv.imshow("Original image", Image)
rect = cv.selectROI("Original image", Image, showCrosshair = False)   #选择一个矩形区域
a, b = rect[2] // 2, rect[3] // 2                                     #椭圆长轴和短轴的一半长度
centerx, centery = rect[0] + a, rect[1] + b                           #椭圆中心
mask = np.zeros_like(Image, dtype = np.float32)
for y in range(height):
    for x in range(width):
        if ((x - centerx) ** 2 / (a ** 2) + (y - centery) ** 2 / (b ** 2)) < 1:
            mask[y, x] = 1                                            #如果点在椭圆内,模板中为白色
cv.imshow("Original mask", mask)
mask = cv.blur(mask, (25, 25))                                        #模板中椭圆区域边界羽化
cv.imshow("Mask with soft edges", mask)
result = (mask * fore + (1 - mask) * back).astype(np.uint8)           #前景、背景融合
cv.imshow("Result image", result)
cv.waitKey()
```

程序运行结果如图 5-46 所示。



(a) 选择矩形区域　(b) 生成模板　(c) 模板滤波　(d) 处理结果

图 5-46　背景虚化

习题

5.1　在对图像进行直方图均衡化时，为什么会产生简并现象？

5.2　一幅图像的直方图如图 5-47 所示，试分析图像的视觉效果，采用什么处理比较合适？

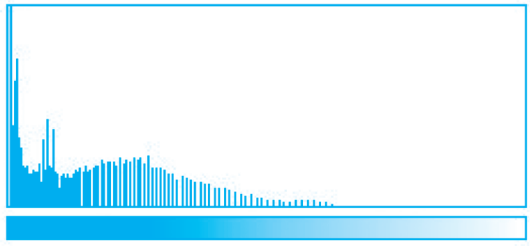


图 5-47　某图的直方图

5.3　一幅大小为 64×64 图像，8 个灰度级对应像素个数及概率 $p_r(r)$ 如表 5-5 所示，试对其进行直方图均衡化。

表 5-5　图像各灰度级对应的像素个数及概率

灰度级 r_k	0	1/7	2/7	3/7	4/7	5/7	6/7	1
像素数 n_k	560	920	1046	705	356	267	170	72
概率 $p_r(r)$	0.14	0.22	0.26	0.17	0.09	0.06	0.04	0.02

5.4 图像平滑的主要用途是什么？该操作对图像质量会带来什么负面影响？为什么？

5.5 双边滤波为什么能够在平滑图像的同时保留边缘信息？

5.6 请简述如何检测图像中的边缘。

5.7 已知一幅图像经过均值滤波之后变得模糊了，用锐化算法是否可以将其变得清晰一些？请说明观点，并编程验证。

5.8 编写程序，用 autoscaling、convertScaleAbs 函数对灰度图像进行处理。

5.9 编写程序实现习题 5.2 中设计的处理方法。

5.10 编写程序，采用模板运算实现图像的高斯滤波。

5.11 编写程序，采用 Prewitt 算子生成边缘方向图像。

5.12 编写程序，对灰度图像进行指数高通滤波。