第5章

图像视觉增强分析

图像视觉增强分析是图像模式识别中非常重要的图像预处理过程。图像视觉增强分析的目的是通过对图像中的信息进行处理，使得有利于模式识别的信息得到增强，不利于模式识别的信息被抑制，扩大图像中不同物体特征之间的差别，为图像视觉的信息提取及其识别奠定良好的基础。

5.1 图像增强方法

图像增强按实现方法不同可分为点增强、空域增强和频域增强。

1. 点增强

点增强主要指图像灰度变换和几何变换。

图像的灰度变换也称为点运算、对比度增强或对比度拉伸，它是图像数字化软件和图像显示软件的重要组成部分。

(1) 灰度变换是一种既简单又重要的技术，它能让用户改变图像数据占据的灰度范围。一幅输入图像经过灰度变换后将产生一幅新的输出图像，由输入像素点的灰度值决定相应的输出像素点的灰度值。灰度变换不会改变图像内的空间关系。

(2) 图像的几何变换是图像处理中的另一种基本变换。它通常包括图像的平移、图像的镜像变换、图像的缩放和图像的旋转。通过图像的几何变换可以实现图像的最基本的坐标变换及缩放功能。

2. 空域增强

图像的空间信息可以反映图像中物体的位置、形状、大小等特征，而这些特征可以通过一定的物理模式来描述。例如，物体的边缘轮廓由于灰度值变化剧烈一般出现高频率特征，而一个比较平滑的物体内部由于灰度值比较均一则呈现低频率特征。因此，根据需要可以分别增强图像的高频和低频特征。对图像的高频增强可以突出物体的边缘轮廓，从而起到锐化图像的作用。例如，对于人脸的比对查询，就需要通过高频增强技术来突出五官的轮廓。相应地，对图像的低频部分进行增强可以对图像进行平滑处理，一般用于图像的噪声消除。

3. 频域增强

图像的空域增强一般只是对数字图像进行局部增强，而图像的频域增强可以对图像进行全局增强。频域增强技术是在数字图像的频率域空间对图像进行滤波，因此需要将图像从空

间域变换到频率域，一般通过傅里叶变换实现。在频率域空间的滤波与空域滤波一样，可以通过卷积实现，因此傅里叶变换和卷积理论是频域滤波技术的基础。

对图像实现增强的主要目的在于：

（1）改善图像的视觉效果，提高图像的清晰度。

（2）针对给定图像的应用场合，突出某些感兴趣的特征，抑制不感兴趣的特征，以扩大图像中不同物体特征之间的差别，满足某些特殊分析的需要。

5.2　灰度变换

灰度变换主要针对独立的像素点进行处理，由输入像素点的灰度值决定相应的输出像素点的灰度值，通过改变原始图像数据所占的灰度范围而使图像在视觉上得到改善。

5.2.1　线性灰度变换

线性灰度增强，将图像中所有点的灰度按照线性灰度变换函数进行变换。在曝光不足或过度的情况下，图像的灰度可能局限在一个很小的灰度范围内，这时图像可能会很模糊不清。利用一个线性单值函数对图像内的每一个像素做线性拓展，将会有效地改善图像的视觉效果。

假设一幅图像 $f(x,y)$ 变换前的灰度范围是 $[a,b]$，希望变换后 $g(x,y)$ 灰度范围拓展或者压缩至 $[c,d]$，则灰度线性变换函数表达式为：

$$g(x,y)=\left[\frac{d-c}{b-a}\right](f(x,y)-a)+c$$

通过调整 a、b、c、d 的值可以控制线性变换函数的斜率，从而达到灰度范围的拓展或压缩。

【例 5-1】 对给定的原始图像进行线性灰度变换。

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
#读者在使用反色变换前请安装 NumPy 库和 pillow 库
plt.rcParams['font.sans-serif'] = ['SimHei']                    #用来正常显示中文标签
def image_inverse(x):                                           #定义反色变换函数
    value_max = np.max(x)
    y = value_max - x
    return y
if __name__ == '__main__':
    #模式"L"为灰色图像,它的每个像素用 8b 表示,0 表示黑,255 表示白,其他数字表示不同的灰度
    gray_img = np.asarray(Image.open(r'4.jpg').convert('L'))
    #Image.open 是打开图片,变量为其地址
    inv_img = image_inverse(gray_img)                  #将原图形作为矩阵传入函数中,进行反色变换
    fig = plt.figure()                                          #绘图
    ax1 = fig.add_subplot(121)   #解释一下 121,第一个 1 是一行,2 是两列,第二个 1 是第一个图
    ax1.set_title('原始图像')
    ax1.imshow(gray_img, cmap = 'gray', vmin = 0, vmax = 255)
    ax2 = fig.add_subplot(122)
    ax2.set_title('线性灰度变换')
    ax2.imshow(inv_img, cmap = 'gray', vmin = 0, vmax = 255)
    plt.show()
```

运行程序，效果如图 5-1 所示。

以上代码实现获取图片中的每个像素的原灰度值，再读取图片灰度值之中的最大值作为“L”，通过减法运算后再将其输出即可实现反色变换。

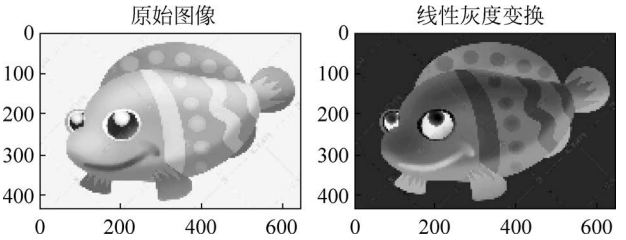


图 5-1　图像的线性灰度变换效果

5.2.2　分段线性灰度增强

分段线性灰度增强可将需要的图像细节灰度级扩展，增强对比度，将不需要的图像细节灰度级压缩。

假设输入图像 $f(x,y)$的灰度为 0 M 级，增强后图像 $g(x,y)$的灰度级为 0 N 级，区间$[a,b]$、$[c,d]$分别为原图像和增强图像的某一灰度区间。分段线性变换函数为：

$$g(x,y)=\begin{cases}\left(\dfrac{c}{a}\right)f(x,y), & 0\leqslant f(x,y)<a\\ \left[\dfrac{d-c}{b-a}\right][f(x,y)-a]+c, & a\leqslant f(x,y)\leqslant b\\ \left[\dfrac{N-d}{M-b}\right][f(x,y)-b]+d, & b<f(x,y)\leqslant M\end{cases}$$

a、b、c、d 取不同的值时，可得到不同的效果。

（1）如果 $a=c$，$b=d$，灰度变换函数为一条斜率为 1 的直线，增强图像与原图像相同。

（2）如果 $a>c$，$b<d$，原图像中灰度值在区间$[0,a]$与$[b,M]$中的动态范围减小，而原图像在区间$[a,b]$的动态范围增加，从而增强中间范围的对比度。

（3）如果 $a<c$，$b>d$，则原图像在区间$[0,a]$与$[b,M]$的动态范围增加，而原图像在区间$[a,b]$的动态范围减小。

由此可见，通过调整 a、b、c、d 可以控制分段的斜率，从而对任意灰度区间进行拓展或压缩。

【例 5-2】　对图像进行分段非线性分析。

```
import numpy as np
import cv2
def linear_transform(img):
    height,width = img.shape[:2]
    r1,s1 = 80,10
    r2,s2 = 140,200
    k1 = s1 / r1                          #第一段斜率
    k2 = (s2 - s1) / (r2 - r1)            #第二段斜率
    k3 = (255 - s2) / (255 - r2)          #第三段斜率
    img_copy = np.zeros_like(img)
    for i in range(height):
        for j in range(width):
            if img[i,j] < r1 :
                img_copy[i,j] = k1 * img[i,j]
            elif r1 <= img[i,j] <= r2:
                img_copy[i,j] = k2 * (img[i,j] - r1) + s1
            else:
                img_copy[i,j] = k3 * (img[i,j] - r2) + s2
    return img_copy
img = cv2.imread('img.png',0)
```

```
ret = linear_transform(img)
cv2.imshow('img',np.hstack((img,ret)))
cv2.waitKey()
cv2.destroyAllWindows()
```

运行程序，效果如图 5-2 所示。

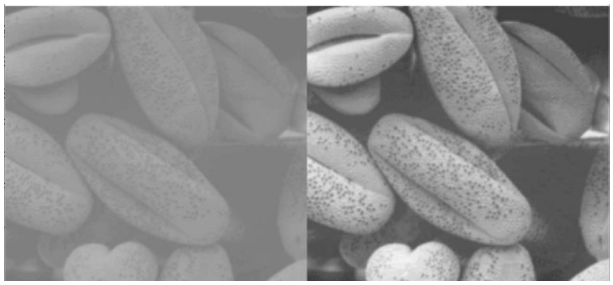


图 5-2　分段非线性分析效果

5.2.3　非线性灰度变换

显而易见，当用非线性函数对图像灰度进行映射时，可以实现图像的非线性灰度增强。常用的非线性灰度增强方法有对数函数非线性变换和指数函数非线性变换。

1. 对数函数非线性变换

对图像做对数非线性变换时，变换函数为：

$$g(x,y)=a+\frac{\ln[f(x,y)+1]}{b\cdot\ln c}$$

通过调整 a，b，c 可以调整曲线的位置与形状。利用此变换，可以使输入图像的低灰度范围得到扩展，高灰度范围得到压缩，以使图像分布均匀。

2. 指数函数非线性变换

对图像做指数函数非线性变换时，变换函数为：

$$g(x,y)=b^{c[f(x,y)-a]}-1$$

通过调整 a，b，c 可以调整曲线的位置与形状。利用此变换，可以使输入图像的低灰度范围得到扩展，高灰度范围得到压缩，以使图像分布均匀。

【例 5-3】　图像灰度对数变换。

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def log_plot(c):
    x = np.arange(0, 256, 0.01)
    y = c * np.log(1 + x)
    plt.plot(x, y, "r", linewidth=1)
    plt.rcParams["font.sans-serif"] = ["SimHei"]
    plt.title("对数变换函数")
    plt.xlim(0, 255)
    plt.ylim(0, 255)
    plt.show()

#对数变换
def log(c, img):
    output = c * np.log(1.0 + img)
    output = np.uint8(output)
```

```
    return output

img = cv2.imread("src.png")
log_plot(42)
result = log(42, img)
cv2.imshow("src", img)
cv2.imshow("result", result)
if cv2.waitKey() == 27:
    cv2.destroyAllWindows()
```

运行程序,效果如图 5-3 及图 5-4 所示。

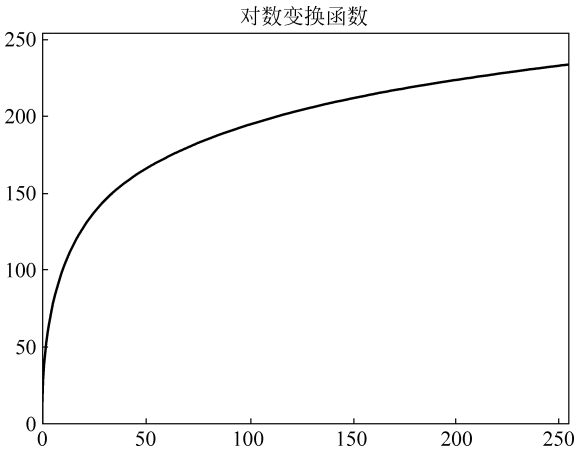


图 5-3　对数变换函数曲线

图 5-4　图像对数变换效果

5.3　空域增强

空域增强主要包括平滑线性滤波增强、非线性滤波增强、双边滤波增强等,下面对这几个增强进行介绍。

5.3.1　平滑线性滤波器

平滑线性空间滤波器的输出(响应)是包含在滤波器模板邻域内的像素的简单平均值。这些滤波器有时也称为均值滤波器,也可以把它们归入低通滤波器。

这种处理的结果降低了图像灰度的尖锐变化。由于典型的随机噪声由灰度级的急剧变化组成，因此常见的平滑处理的应用就是降噪。然而，由于图像边缘（几乎总是一幅图像希望有的特性）也是由图像灰度尖锐变化带来的特性，所以均值滤波器处理还是存在着不希望有的边缘模糊的负面效应。

图 5-5 这幅图中的是最为常见的简单平均的滤波器模板。

所有系数都相等的空间均值滤波器，有时也被称为盒状滤波器，如图 5-6 所示。

1	1	1
1	1	1
1	1	1

$$R=\frac{1}{9}\sum_{i=1}^{9}Z_i$$

图 5-5　简单平均的滤波器模板

1	2	1
2	4	2
1	2	1

图 5-6　盒状滤波器

图 5-6 所示的滤波器模板相比于图 5-5 所示的滤波器模板更加重要。该滤波器模板产生所谓的加权平均，使用这一术语是指，用不同的系数去乘以像素，即一些像素的重要性（权重）比另外一些像素的重要性更大。

在这个例子所示的模板中，中心位置的系数最大，因此在均值计算中可以为该像素提供更大的权重。其他像素离中心越近就赋予越大的权重。

这种加权重的策略的目的是，在平滑处理中，试图降低模糊。也可以选择其他权重来达到相同的目的。但是，这个例子中所有系数的和等于 16，这对于计算机来说是一个很有吸引力的特性，因为它是 2 的整数次幂。

在实践中，由于这些模板在一幅图像中的任何一个位置所跨越的区域很小，通常很难看出这两个模板或者类似方式进行平滑处理后的图像之间的区别。

一幅 $M\times N$ 的图像进行一个 $m\times n$ 的加权均值滤波器，滤波的过程可由下式给出：

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

当图像的细节与滤波器模板近似时，图像中的一些细节受到的影响比较大。邻域越大平滑的效果越好。但是邻域过大，平滑会使边缘信息损失越大，从而使输出的图像变得模糊，因此需要合理地选择邻域的大小。模板的大小由那些即将融入背景中的物体的尺寸来决定。

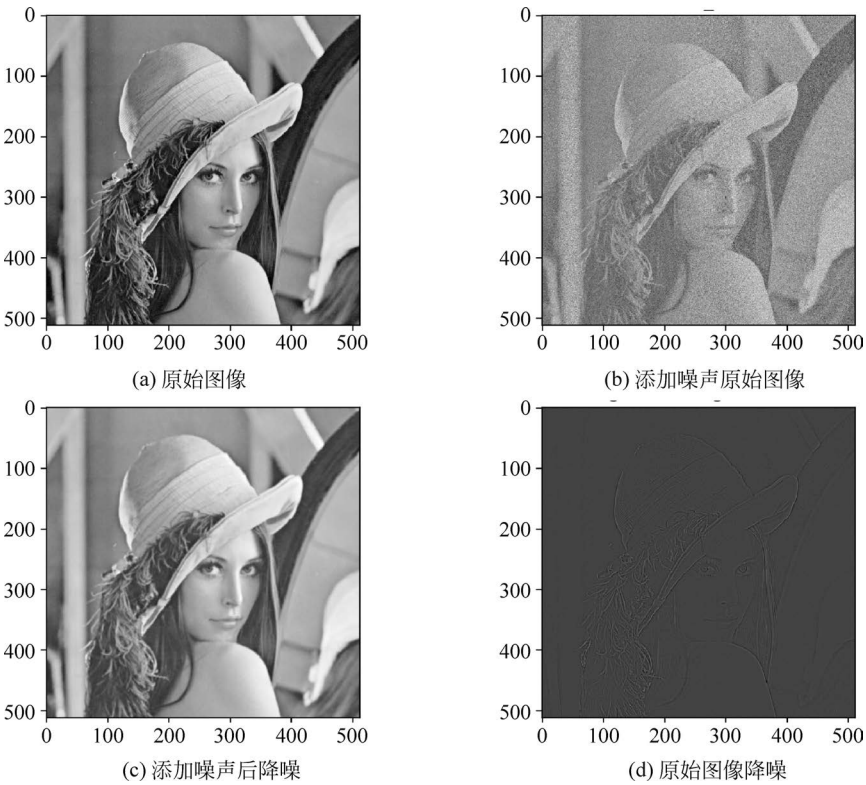
滤波后的图像中可能会有黑边。这是由于用 0（黑色）填充原图像的边界，经滤波后，再去除填充区域的结果，某些黑色混入了滤波后的图像。对于使用较大滤波器平滑的图像，这就成了问题。

【例 5-4】 对图像实现平滑线性滤波处理。

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

def show(f, s, a, b, c):
    plt.subplot(a, b, c)
    plt.imshow(f, "gray")
    plt.axis('on')
```

```
    plt.title(s)

#高斯噪声函数单行
def wgn(x, snr):
    snr = 10 ** (snr / 10.0)
    xpower = np.sum(x ** 2) / len(x)
    npower = xpower / snr
    return np.random.randn(len(x)) * np.sqrt(npower)

def main():
    original = plt.imread("lena.tiff", 0)
    rows, cols = original.shape
    original_noise = original.copy().astype(np.float64)
    #生成噪声图像,信噪比为 10
    for i in range(cols):
        original_noise[:, i] += wgn(original_noise[:, i], 10)
    mask = np.ones(9).reshape(3, 3)
    ImageDenoise = np.zeros(original.shape)
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            ImageDenoise[i, j] = np.mean(original[i - 1:i + 2, j - 1:j + 2] * mask)
    plt.figure()
    show(original, "original", 2, 2, 1)
    show(original_noise, "original_noise", 2, 2, 2)
    show(ImageDenoise, "ImageDenoise", 2, 2, 3)
    show(original - ImageDenoise, "original - ImageDenoise", 2, 2, 4)
    plt.show()
if __name__ == '__main__':
    main()
```

运行程序,效果如图 5-7 所示。



(a) 原始图像　(b) 添加噪声原始图像

(c) 添加噪声后降噪　(d) 原始图像降噪

图 5-7　图像平滑线性处理效果

5.3.2　统计排序(非线性)滤波器

统计排序滤波器是一种非线性空间滤波器,这种滤波器的响应以滤波器包围的图像的像素的排序为基础,然后使用统计排序结果决定的值代替中心像素的值。这一类中最知名的就要数中值滤波器了,它是将像素邻域内的灰度值的中值代替该像素的值。

中值滤波器的使用非常广泛,它对于一定类型的随机噪声提供了一种优秀的去噪能力。而且比同尺寸的线性平滑滤波器的模糊程度明显要低。不足之处就是中值滤波花费的时间是均值滤波的 5 倍以上。

中值滤波器对于处理脉冲噪声非常有效,这种噪声称为椒盐噪声,因为这种噪声是以黑白点的形式叠加在图像上的。

中值滤波器的主要功能是使拥有不同灰度的点看起来更接近于它们的相邻点。使用 $m \times m$ 中值滤波器来去除那些相对于其邻域像素更亮或更暗并且其区域小于(m^2)/2(滤波器区域一半)的孤立像素族。

【例 5-5】 对带噪声图像实现中值滤波处理。

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
def medianBlur(image,ksize = 2,):
    '''
    中值滤波,去除椒盐噪声
    args:
        image:输入图片数据,要求为灰度图片
        ksize:滤波窗口大小
    return:
        中值滤波之后的图片
    '''
    rows,cols = image.shape[:2]
    #输入校验
    half = ksize//2
    startSearchRow = half
    endSearchRow = rows - half - 1
    startSearchCol = half
    endSearchCol = cols - half - 1
    dst = np.zeros((rows,cols),dtype = np.uint8)
    #中值滤波
    for y in range(startSearchRow,endSearchRow):
        for x in range(startSearchCol,endSearchCol):
            window = []
            for i in range(y - half,y + half + 1):
                for j in range(x - half,x + half + 1):
                    window.append(image[i][j])
            #取中间值
            window = np.sort(window,axis = None)
            if len(window) % 2 == 1:
                medianValue = window[len(window)//2]
            else:
                medianValue = int((window[len(window)//2] + window[len(window)//2 + 1])/2)
            dst[y][x] = medianValue
    return dst

image = cv2.imread('lena.png')
```

```
med = medianBlur(image)
cv2.imwrite('med.png',med)
```

运行程序，效果如图 5-8 所示。

(a) 带噪声原图　　(b) 滤波后效果

图 5-8　中值滤波处理效果

5.3.3　双边滤波器

如何将图 5-9 中的点 A 和 B、C、D、E 区分开来，使得式(5-1)中计算 A 点的加权平均时，B、C、D、E 贡献的权值为 0(这是理想的情况)，就成了关键。那么一个很自然的想法是，同时考虑像素点的像素值，这样将像素点投到高维空间，那么有可能 B、C、D、E 将不再是 A 的相邻点。因为在低维空间中不易区分的点，在高维空间中可能比较容易区分开来。这点类似 SVM(支持向量机)中的核函数设置的思想。

$$\hat{I}(i,j)=f\circ I(i,j)=\frac{1}{\sum w(i',j')}\sum_{(i',j')\in N}w(i',j')I(i',j') \tag{5-1}$$

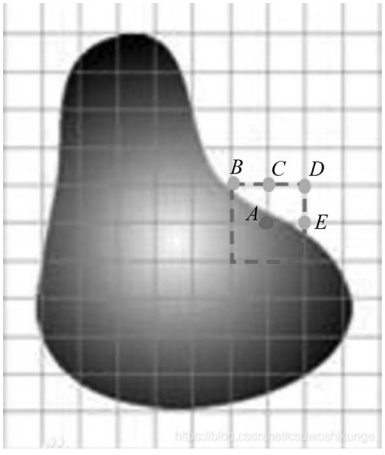


图 5-9　空间相邻像素点

同时统计像素点 A 的坐标和像素值，这其实就是图像函数 $I:(i,j)\to I(i,j)\in R$ 的图形，记作：

$$G=\{(i,j,I(i,j)):(i,j)\} \tag{5-2}$$

其中，(i,j)是图像像素坐标。

它是一个二维流形(嵌入在三维欧氏空间 R^3 中)，其实就是一个曲面，显然在这个图形 G 上，B,C,D,E 与 A 不再是相邻点。但是在直接二维流形 G 上寻找点的相邻点可能非常复杂，因为曲面的形状可能很不规则(曲率不为 0)，不像二维平面网格点(i,j)那样容易寻找相邻点。

为了计算方便，退而求其次，把二维曲面 G 嵌入在三维空间中考虑，利用三维欧氏空间 R^3 的曲率为 0，在三维网格点中寻找相邻点，这样就和在二维网格中寻找相邻点一样方便。但是考虑到嵌入的 G 实则是函数 I 的图像，本质上是二维的，所以双边滤波仍按定义域寻找相邻点即可，但是要减弱这些相邻点中颜色差异较大的点对滤波贡献的权重。

可以看到，在把 G 嵌入三维空间中考虑时，A 点对应(i_A,j_A,I_A)。以 D 点为例，D 对应(i_D,j_D,I_D)。不妨以欧氏范数为例，在三维空间中 A,D 之间的距离为：

$$d(A,D)=\sqrt{(i_D-i_A)^2+(j_D-j_A)^2+(I_D-I_A)^2}=\sqrt{1+1+(I_D-I_A)^2} \tag{5-3}$$

如式(5-3)所示，当 A，D 的像素值之差 $|I_D - I_A|$ 很大时，那么在 G 中 $d(A,D)$ 距离很远。而双边滤波正是通过考虑这种距离来弱化 D 对 A 的影响，这点可以从下面双边滤波的公式(5-4)中得到体现。

$$\hat{I}_A = \frac{1}{\sum w(q,A)} \sum_{q \in N} w(q,A) I_q \tag{5-4}$$

其中，N 是点 A 的邻域。

$$w(q,A) = \mathrm{e}^{-\frac{(i_q - i_A)^2 + (j_q - j_A)^2}{2\sigma_s^2} - \frac{(i_q - I_A)^2}{2\sigma_r^2}} \tag{5-5}$$

显然，当 q，A 之间的像素值相差很大时，$w(q,A)$ 就会变得很小，从而弱化了 q 点对 A 点的滤波的影响，容易看到图 5-9 中的 B，C，D，E 与 A 的距离很大(在 R^3 中考虑它们之间的距离时)。所以滤波时，B，C，D，E 对 A 的影响就大大减弱，从而大大减轻滤波后边界模糊的现象，达到了保边滤波的效果。

【例 5-6】 对给定图像实现双边滤波处理。

```
import numpy as np
from scipy import signal
import cv2
import random
import math
#双边滤波

def getClosenessWeight(sigma_g,H,W):
    r,c = np.mgrid[0:H:1,0:W:1]
    r -= (H - 1) // 2
    c -= int(W - 1) // 2
    closeWeight = np.exp(-0.5 * (np.power(r,2) + np.power(c,2))/math.pow(sigma_g,2))
    return closeWeight

def bfltGray(I,H,W,sigma_g,sigma_d):
    #构建空间距离权重模板
    closenessWeight = getClosenessWeight(sigma_g,H,W)
    #模板的中心点位置
    cH = (H - 1) // 2                    #"//"表示整数除法
    cW = (W - 1) // 2
    #图像矩阵的行数和列数
    rows,cols = I.shape
    #双边滤波后的结果
    bfltGrayImage = np.zeros(I.shape,np.float32)
    for r in range(rows):
        for c in range(cols):
            pixel = I[r][c]
            #判断边界
            rTop = 0 if r - cH < 0 else r - cH
            rBottom = rows - 1 if r + cH > rows - 1 else r + cH
            cLeft = 0 if c - cW < 0 else c - cW
            cRight = cols - 1 if c + cW > cols - 1 else c + cW
            #权重模板作用的区域
            region = I[rTop:rBottom + 1,cLeft:cRight + 1]
            #构建灰度值相似性的权重因子
similarityWeightTemp = np.exp(-0.5 * np.power(region - pixel,2.0)/math.pow(sigma_d,2))
closenessWeightTemp = closenessWeight[rTop - r + cH:rBottom - r + cH + 1,cLeft - c + cW:cRight - c +
cW + 1]
```

```
                #两个权重模板相乘
                weightTemp = similarityWeightTemp * closenessWeightTemp
                #归一化权重模板
                weightTemp = weightTemp/np.sum(weightTemp)
                #权重模板和对应的邻域值相乘求和
                bfltGrayImage[r][c] = np.sum(region * weightTemp)
    return bfltGrayImage

if __name__ == '__main__': #启动语句
    a = cv2.imread('2.png', cv2.IMREAD_UNCHANGED) #路径名中不能有中文,会出错
    image1 = cv2.split(a)[0] #蓝通道
    cv2.imshow(image1)
    image1 = image1/255.0
    #双边滤波
    bfltImage = bfltGray(image1,3,3,19,0.2)
    cv2.imshow(bfltImage)
    cv2.waitKey(0)
```

运行程序,效果如图 5-10 所示。

(a) 原始图像　　(b) 滤波后图像

图 5-10　双边滤波效果

5.4　空域锐化算子

在图像识别中,需要有边缘鲜明的图像,即图像锐化。图像锐化的目的是突出图像的边缘信息,加强图像的轮廓特征,以便于人眼的观察和机器的识别。在空间域进行图像锐化主要有以下方法。

(1) 梯度空间算子。

(2) 其他锐化算子。

(3) Laplacian 算子。

5.4.1　梯度空间算子

图像的边缘最直观的表现就是边缘两侧的灰度值相差比较大,在微积分中我们学过梯度的概念。梯度是一个列向量,可表示为:

$$G[f(x,y)]=\begin{bmatrix}\dfrac{\partial f}{\partial x}\\ \dfrac{\partial f}{\partial y}\end{bmatrix}=[G_x \quad G_y]^{\mathrm{T}}=\begin{bmatrix}\dfrac{\partial f}{\partial x} & \dfrac{\partial f}{\partial y}\end{bmatrix}^{\mathrm{T}}$$

而某点处梯度的模很好地反映了该点两侧的变化大小,所以,梯度值很大的点也就代表了

图像的边缘。而在实际计算中，为了降低运算量，一般用以下两种方法来代替模运算。

$$|G[f(x,y)]|=\sqrt{G_x^2+G_y^2}\approx|G_x|+|G_y|$$

$$|G[f(x,y)]|=\sqrt{G_x^2+G_y^2}\approx\max\{|G_x|,|G_y|\}$$

由于数字图像处理中处理的是数字离散信号，所以，用差分来等同于连续信号中的微分运算。典型的梯度运算有：

$$\begin{cases}G_x=f(i+1,j)-f(i,j)\\G_y=f(i,j+1)-f(i,j)\end{cases}$$

而另一种称为 Roberts 梯度的差分运算可由下式来表示：

$$\begin{cases}G_x=f(i+1,j+1)-f(i,j)\\G_y=f(i,j+1)-f(i+1,j)\end{cases}$$

在 Python 中，Roberts 算子主要通过 NumPy 定义模板，再调用 OpenCV 的 filter2D()函数实现边缘提取。该函数主要利用内核实现对图像的卷积运算，其函数语法格式为：

```
dst = filter2D(src, ddepth, kernel[, dst[, anchor[, delta[, borderType]]]])
```

各参数的含义如下。

- src：表示输入图像。
- dst：表示输出的边缘图，其大小和通道数与输入图像相同。
- ddepth：表示目标图像所需的深度。
- kernel：表示卷积核，一个单通道浮点型矩阵。
- anchor：表示内核的基准点，其默认值为(－1，－1)，位于中心位置。
- delta：表示在存储目标图像前可选的添加到像素的值，默认值为 0。
- borderType：表示边框模式。

【例 5-7】 对图像做 Roberts 算子处理。

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#读取图像
img = cv2.imread('lena.png')
lenna_img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
#灰度化处理图像
grayImage = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#Roberts 算子
kernelx = np.array([[-1,0],[0,1]], dtype=int)
kernely = np.array([[0,-1],[1,0]], dtype=int)
x = cv2.filter2D(grayImage, cv2.CV_16S, kernelx)
y = cv2.filter2D(grayImage, cv2.CV_16S, kernely)
#转 uint8
absX = cv2.convertScaleAbs(x)
absY = cv2.convertScaleAbs(y)
Roberts = cv2.addWeighted(absX,0.5,absY,0.5,0)
#用来正常显示中文标签
plt.rcParams['font.sans-serif'] = ['SimHei']
#显示图形
titles = [u'原始图像', u'Roberts 算子']
images = [lenna_img, Roberts]
for i in xrange(2):
 plt.subplot(1,2,i+1), plt.imshow(images[i], 'gray')
```

```
 plt.title(titles[i])
 plt.xticks([]),plt.yticks([])
plt.show()
```

运行程序,效果如图 5-11 所示。

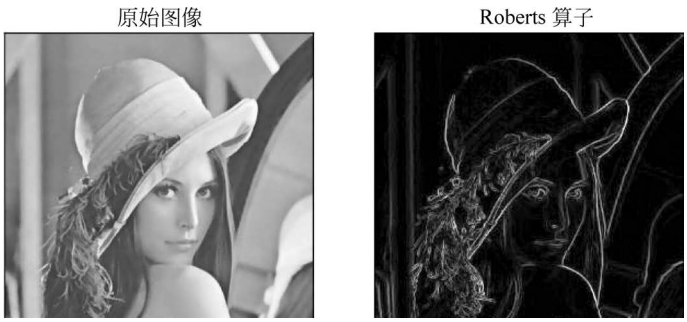


图 5-11　Roberts 算子效果

5.4.2　Prewitt 算子

Prewitt 是一种图像边缘检测的微分算子,其原理是利用特定区域内像素灰度值产生的差分实现边缘检测。由于 Prewitt 算子采用 3×3 模板对区域内的像素值进行计算,而 Robert 算子的模板为 2×2,故 Prewitt 算子的边缘检测结果在水平方向和垂直方向均比 Robert 算子更加明显。Prewitt 算子适合用来识别噪声较多、灰度渐变的图像,其计算公式如下。

$$\boldsymbol{D}_x=\begin{bmatrix}1 & 0 & -1\\ 1 & 0 & -1\\ 1 & 0 & -1\end{bmatrix},\quad \boldsymbol{D}_y=\begin{bmatrix}-1 & -1 & -1\\ 0 & 0 & 0\\ 1 & 1 & 1\end{bmatrix}$$

在 Python 中,Prewitt 算子的实现过程与 Roberts 算子比较相似。通过 NumPy 定义模板,再调用 OpenCV 的 filter2D()函数实现对图像的卷积运算,最终通过 convertScaleAbs()和 addWeighted()函数实现边缘提取。

【例 5-8】　对图像做 Prewitt 算子处理。

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#读取图像
img = cv2.imread('lena.png')
lenna_img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
#灰度化处理图像
grayImage = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#Prewitt 算子
kernelx = np.array([[1,1,1],[0,0,0],[-1,-1,-1]],dtype=int)
kernely = np.array([[-1,0,1],[-1,0,1],[-1,0,1]],dtype=int)
x = cv2.filter2D(grayImage, cv2.CV_16S, kernelx)
y = cv2.filter2D(grayImage, cv2.CV_16S, kernely)
#转 uint8
absX = cv2.convertScaleAbs(x)
absY = cv2.convertScaleAbs(y)
Prewitt = cv2.addWeighted(absX,0.5,absY,0.5,0)
#用来正常显示中文标签
plt.rcParams['font.sans-serif'] = ['SimHei']
#显示图形
titles = [u'原始图像', u'Prewitt 算子']
```

```
images = [lenna_img, Prewitt]
for i in xrange(2):
 plt.subplot(1,2,i + 1), plt.imshow(images[i], 'gray')
 plt.title(titles[i])
 plt.xticks([]),plt.yticks([])
plt.show()
```

输出结果如图 5-12 所示，左边为原始图像，右边为 Prewitt 算子图像锐化提取的边缘轮廓，其效果图的边缘检测结果在水平方向和垂直方向均比 Robert 算子更加明显。

图 5-12　Prewitt 算子效果

5.4.3　Sobel 算子

Sobel 算子是一种用于边缘检测的离散微分算子，它结合了高斯平滑和微分求导。该算子用于计算图像明暗程度近似值，根据图像边缘旁边的明暗程度把该区域内超过某个数的特定点记为边缘。Sobel 算子在 Prewitt 算子的基础上增加了权重的概念，认为相邻点的距离远近对当前像素点的影响是不同的，距离越近的像素点对应当前像素的影响越大，从而实现图像锐化并突出边缘轮廓。

Sobel 算子的边缘定位更准确，常用于噪声较多、灰度渐变的图像。其算法模板如以下公式所示，其中，$\boldsymbol{D}_x$ 表示水平方向，$\boldsymbol{D}_y$ 表示垂直方向。

$$\boldsymbol{D}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad \boldsymbol{D}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Sobel 算子根据像素点上下、左右邻点灰度加权差，在边缘处达到极值这一现象检测边缘，对噪声具有平滑作用，提供较为精确的边缘方向信息。因为 Sobel 算子结合了高斯平滑和微分求导(分化)，因此结果会具有更多的抗噪性，当对精度要求不是很高时，Sobel 算子是一种较为常用的边缘检测方法。在 Python 中，利用 Sobel()函数实现 Sobel 算法处理。函数的语法格式为：

```
dst = Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]])
```

各参数的含义如下。

- src：表示输入图像。
- dst：表示输出的边缘图，其大小和通道数与输入图像相同。
- ddepth：表示目标图像所需的深度，针对不同的输入图像，输出目标图像有不同的深度。
- dx：表示 x 方向上的差分阶数，取值 1 或 0。
- dy：表示 y 方向上的差分阶数，取值 1 或 0。

- ksize：表示 Sobel 算子的大小，其值必须是正数和奇数。
- scale：表示缩放导数的比例常数，默认情况下没有伸缩系数。
- delta：表示将结果存入目标图像之前，添加到结果中的可选增量值。
- borderType：表示边框模式。

注意，在进行 Sobel 算子处理之后，还需要调用 convertScaleAbs()函数计算绝对值，并将图像转换为 8 位图进行显示。其算法格式如下。

```
dst = convertScaleAbs(src[, dst[, alpha[, beta]]])
```

各参数的含义如下。

- src：表示原数组。
- dst：表示输出数组，深度为 8 位。
- alpha：表示比例因子。
- beta：表示原数组元素按比例缩放后添加的值。

【例 5-9】 对图像做 Sobel 算子处理。

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#读取图像
img = cv2.imread('lena.png')
lenna_img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
#灰度化处理图像
grayImage = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#Sobel 算子
x = cv2.Sobel(grayImage, cv2.CV_16S, 1, 0)                    #对 x 求一阶导
y = cv2.Sobel(grayImage, cv2.CV_16S, 0, 1)                    #对 y 求一阶导
absX = cv2.convertScaleAbs(x)
absY = cv2.convertScaleAbs(y)
Sobel = cv2.addWeighted(absX, 0.5, absY, 0.5, 0)
#用来正常显示中文标签
plt.rcParams['font.sans-serif'] = ['SimHei']
#显示图形
titles = [u'原始图像', u'Sobel 算子']
images = [lenna_img, Sobel]
for i in xrange(2):
 plt.subplot(1,2,i + 1), plt.imshow(images[i], 'gray')
 plt.title(titles[i])
 plt.xticks([]),plt.yticks([])
plt.show()
```

运行程序，效果如图 5-13 所示。

图 5-13 Sobel 算子效果

5.4.4　Laplacian 算子

拉普拉斯(Laplacian)算子是 n 维欧几里得空间中的一个二阶微分算子，常用于图像增强领域和边缘提取。它通过灰度差分计算邻域内的像素，基本流程是：判断图像中心像素灰度值与它周围其他像素的灰度值，如果中心像素的灰度值更高，则提升中心像素的灰度；反之降低中心像素的灰度，从而实现图像锐化操作。在算法实现过程中，Laplacian 算子通过对邻域中心像素的四方向或八方向求梯度，再将梯度相加起来判断中心像素灰度与邻域内其他像素灰度的关系，最后通过梯度运算的结果对像素灰度进行调整。

Laplacian 算子分为四邻域和八邻域，四邻域是对邻域中心像素的四方向求梯度，八邻域是对八方向求梯度。其中，四邻域模板如以下公式所示：

$$\boldsymbol{H}=\begin{bmatrix}0 & -1 & 0\\ -1 & 4 & -1\\ 0 & -1 & 0\end{bmatrix}$$

通过模板可以发现，当邻域内像素灰度相同时，模板的卷积运算结果为 0；当中心像素灰度高于邻域内其他像素的平均灰度时，模板的卷积运算结果为正数；当中心像素的灰度低于邻域内其他像素的平均灰度时，模板的卷积为负数。对卷积运算的结果用适当的衰弱因子处理并加在原中心像素上，就可以实现图像的锐化处理。

Laplacian 算子的八邻域模板公式如下：

$$\boldsymbol{H}=\begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix}$$

Python 和 OpenCV 将 Laplacian 算子封装在 Laplacian()函数中，其函数格式为：

```
dst = Laplacian(src, ddepth[, dst[, ksize[, scale[, delta[, borderType]]]]])
```

各参数的含义如下。

- src：表示输入图像。
- dst：表示输出的边缘图，其大小和通道数与输入图像相同。
- ddepth：表示目标图像所需的深度。
- ksize：表示用于计算二阶导数的滤波器的孔径大小，其值必须是正数和奇数，且默认值为 1。
- scale：表示计算拉普拉斯算子值的可选比例因子，默认值为 1。
- delta：表示将结果存入目标图像之前，添加到结果中的可选增量值，默认值为 0。
- borderType：表示边框模式。

注意：Laplacian 算子其实主要是利用 Sobel 算子的运算，通过加上 Sobel 算子运算得出的图像 X 方向和 Y 方向上的导数，得到输入图像的图像锐化结果。同时，在进行 Laplacian 算子处理之后，还需要调用 convertScaleAbs()函数计算绝对值，并将图像转换为 8 位图进行显示。

当 ksize=1 时，Laplacian()函数采用 3×3 的孔径(四邻域模板)进行变换处理。

【例 5-10】　采用 ksize=3 的 Laplacian 算子进行图像锐化处理。

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```
#读取图像
img = cv2.imread('lena.png')
lenna_img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
#灰度化处理图像
grayImage = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#拉普拉斯算法
dst = cv2.Laplacian(grayImage, cv2.CV_16S, ksize = 3)
Laplacian = cv2.convertScaleAbs(dst)
#用来正常显示中文标签
plt.rcParams['font.sans-serif'] = ['SimHei']
#显示图形
titles = [u'原始图像', u'Laplacian 算子']
images = [lenna_img, Laplacian]
for i in xrange(2):
 plt.subplot(1,2,i + 1), plt.imshow(images[i], 'gray')
 plt.title(titles[i])
 plt.xticks([]),plt.yticks([])
plt.show()
```

运行程序,效果如图 5-14 所示。

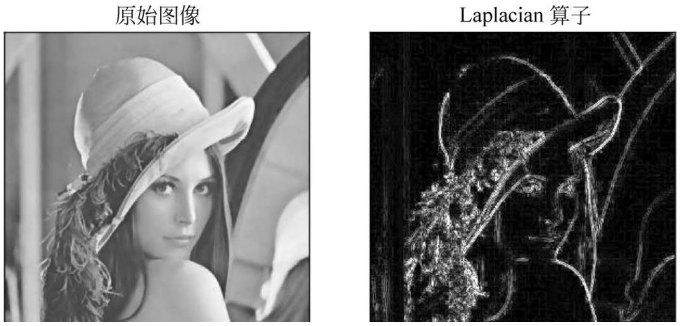


图 5-14　Laplacian 算子效果

边缘检测算法主要是基于图像强度的一阶和二阶导数,但导数通常对噪声很敏感,因此需要采用滤波器来过滤噪声,并调用图像增强或阈值化算法进行处理,最后再进行边缘检测。

【例 5-11】 采用高斯滤波去噪和阈值化处理之后,再进行边缘检测,并对比四种常见的边缘提取算法。

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#读取图像
img = cv2.imread('lena.png')
lenna_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
#灰度化处理图像
grayImage = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#高斯滤波
gaussianBlur = cv2.GaussianBlur(grayImage, (3,3), 0)
#阈值处理
ret, binary = cv2.threshold(gaussianBlur, 127, 255, cv2.THRESH_BINARY)
#Roberts 算子
kernelx = np.array([[-1,0],[0,1]], dtype=int)
kernely = np.array([[0,-1],[1,0]], dtype=int)
x = cv2.filter2D(binary, cv2.CV_16S, kernelx)
y = cv2.filter2D(binary, cv2.CV_16S, kernely)
absX = cv2.convertScaleAbs(x)
absY = cv2.convertScaleAbs(y)
```

```
Roberts = cv2.addWeighted(absX, 0.5, absY, 0.5, 0)
#Prewitt 算子
kernelx = np.array([[1,1,1],[0,0,0],[-1,-1,-1]], dtype=int)
kernely = np.array([[-1,0,1],[-1,0,1],[-1,0,1]], dtype=int)
x = cv2.filter2D(binary, cv2.CV_16S, kernelx)
y = cv2.filter2D(binary, cv2.CV_16S, kernely)
absX = cv2.convertScaleAbs(x)
absY = cv2.convertScaleAbs(y)
Prewitt = cv2.addWeighted(absX,0.5,absY,0.5,0)
#Sobel 算子
x = cv2.Sobel(binary, cv2.CV_16S, 1, 0)
y = cv2.Sobel(binary, cv2.CV_16S, 0, 1)
absX = cv2.convertScaleAbs(x)
absY = cv2.convertScaleAbs(y)
Sobel = cv2.addWeighted(absX, 0.5, absY, 0.5, 0)
#Laplacian 算子
dst = cv2.Laplacian(binary, cv2.CV_16S, ksize = 3)
Laplacian = cv2.convertScaleAbs(dst)
#效果图
titles = ['原始图像', '二值图像', 'Roberts 锐化图像',
 'Prewitt 锐化图像','Sobel 锐化图像', 'Laplacian 锐化图像']
images = [lenna_img, binary, Roberts, Prewitt, Sobel, Laplacian]
for i in np.arange(6):
 plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
 plt.title(titles[i])
 plt.xticks([]),plt.yticks([])
plt.show()
```

输出结果如图 5-15 所示。其中，Laplacian 算子对噪声比较敏感，由于其算法可能会出现双像素边界，常用来判断边缘像素位于图像的明区或暗区，很少用于边缘检测；Robert 算子对陡峭的低噪声图像效果较好，尤其是边缘正负 45°较多的图像，但定位准确率较差；Prewitt 算子对灰度渐变的图像边缘提取效果较好，而没有考虑相邻点的距离远近对当前像素点的影响；Sobel 算子考虑了综合因素，对噪声较多的图像处理效果更好。

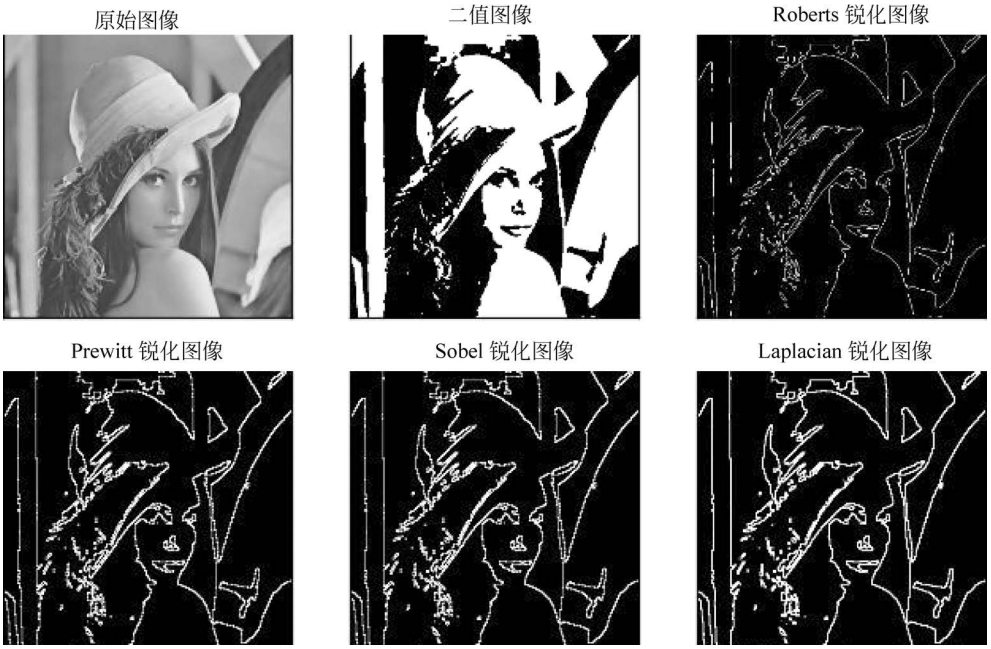


图 5-15　对比四种算子锐化图像效果

5.5 图像频域平滑处理

图像的平滑除了在空间域中进行外，也可以在频率域中进行。由于噪声主要集中在高频部分，为去除噪声改善图像质量，采用滤波器，然后再进行逆傅里叶变换获得滤波图像，就可达到平滑图像的目的。

在一幅图像中，低频部分对应图像变化缓慢的部分即图像大致外观和轮廓。高频部分对应图像变换剧烈的部分即图像细节。

低通滤波器的功能是让低频率通过而滤掉或衰减高频，其作用是过滤掉包含在高频中的噪声。即低通滤波的效果是图像去噪声平滑增强，但同时也抑制了图像的边界即过滤掉图像细节，造成图像不同程序上的模糊。对于大小为 $M\times N$ 的图像，频率点(u,v)与频域中心的距离为 $D(u,v)$，其表达式为：

$$D(u,v)=\left[\left(u-\frac{M}{2}\right)^2+\left(v-\frac{N}{2}\right)^2\right]^{\frac{1}{2}}$$

低通滤波器一共有三种，分别为理想低通滤波器、巴特沃斯低通滤波器和高斯低通滤波器。理想低通滤波器的滤波非常尖锐；高斯低通滤波器的滤波则非常平滑，巴特沃斯滤波器介于两者之间，当巴特沃斯低通滤波器的阶数较高时，接近于理想低通滤波器，当巴特沃斯低通滤波器的阶数较高时，则接近于高斯低通滤波器。

5.5.1 理想低通滤波器

理想低通滤波器的产生公式为：

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

图像如图 5-16 所示。

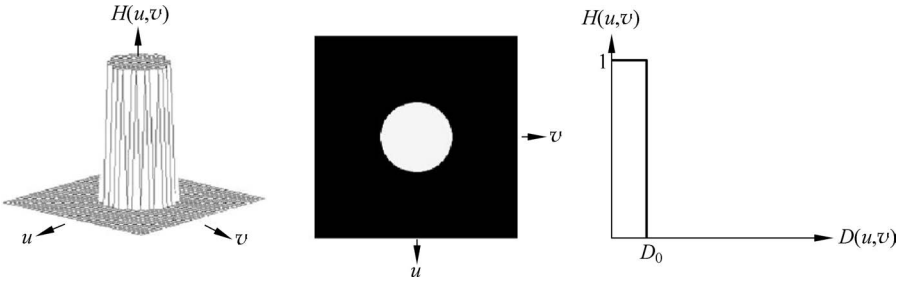


图 5-16 理想低通滤波器效果

在半径为 D_0 的圆内，所有频率没有衰减地通过滤波器，而在此半径的圆之外的所有频率完全被衰减掉。理想低通滤波器具有平滑图像的作用，但是有很严重的振铃现象。

提示：图像处理中，对一幅图像进行滤波处理，若选用的频域滤波器具有陡峭的变化，则会使滤波图像产生“振铃”。所谓“振铃”，即指输出图像的灰度剧烈变化处产生的震荡，就好像钟被敲击后产生的空气震荡，如图 5-17 所示。

(a) 原始图像　　(b) 振铃

图 5-17　图像振铃现象

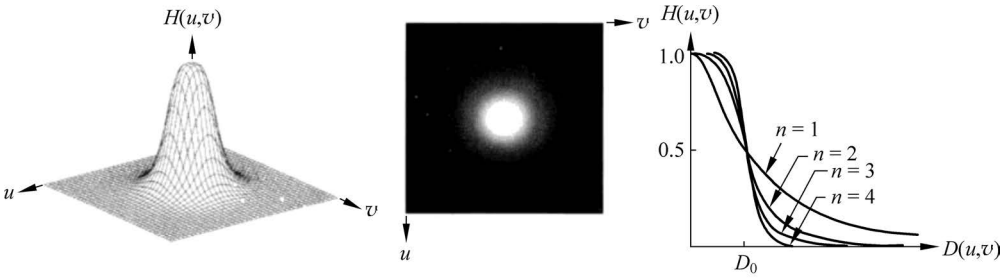
5.5.2　巴特沃思低通滤波器

巴特沃斯滤波器是电子滤波器的一种，它也被称作最大平坦滤波器。巴特沃斯滤波器的特点是通频带内的频率响应曲线最大限度平坦，没有纹波，而在阻频带则逐渐下降为零。

巴特沃斯低通滤波器可用如下振幅的平方对频率作用的公式表示：

$$H(u,v)=\frac{1}{1+\left[\dfrac{D(u,v)}{D_0}\right]^{2n}}$$

图像如图 5-18 所示。



(a) 巴特沃斯低通滤波器的透视图　　(b) 图像的滤波器　　(c) 阶数为1~4的滤波器径向剖面

图 5-18　巴特沃思低通滤波器效果

其中，D_0 为巴特沃斯低通滤波器的截止频率，参数 n 为巴特沃斯低通滤波器的阶数，n 越大则滤波器的形状越陡峭即振铃现象越明显。

5.5.3　高斯低通滤波器

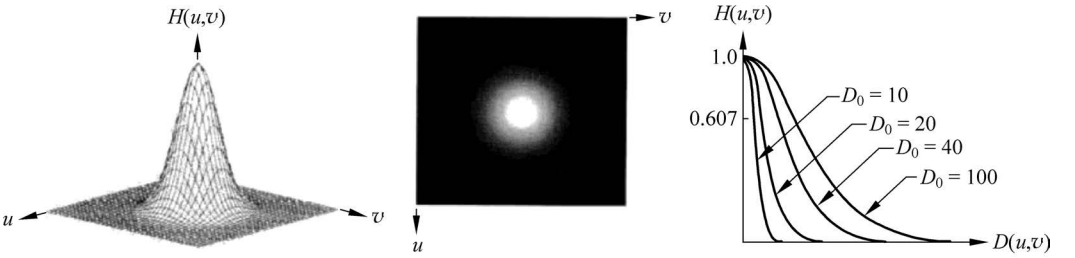
高斯滤波是一种线性平滑滤波，适用于消除高斯噪声，广泛应用于图像处理的减噪过程。通俗地讲，高斯滤波就是对整幅图像进行加权平均的过程，每一个像素点的值，都由其本身和邻域内的其他像素值经过加权平均后得到。

高斯低通滤波器的产生公式为：

$$H(u,v)=\mathrm{e}^{\frac{-D^2(u,v)}{2D_0^2}}$$

图像如图 5-19 所示。

其中，D_0 为高斯低通滤波器的截止频率，注意高斯低通滤波器不会产生振铃现象。



(a) 高斯低通滤波器的透视图　　(b) 图像的滤波器　　(c) 不同D_0值的滤波器径向剖面

图 5-19　高斯低通滤波器效果图

5.5.4　频域低通滤波器的应用

本节通过实例来实现：使用自生成图像(包含白色区域、黑色区域，并且部分区域添加椒盐噪声)进行傅里叶变换，然后分别使用理想低通滤波器、巴特沃斯低通滤波器、指数低通滤波器和梯度低通滤波器(至少使用两种低通滤波器)，显示滤波前后的频域能量分布图、空间图像。

【例 5-12】　实现图像频域平滑处理。

```
import random
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']                    #用来正常显示中文标签

def sp_noise(image, prob):
    """
    添加椒盐噪声
    prob为噪声比例
    """
    output = np.zeros(image.shape, np.uint8)
    thres = 1 - prob
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            rdn = random.random()
            if rdn < prob:
                output[i][j] = 0
            elif rdn > thres:
                output[i][j] = 255
            else:
                output[i][j] = image[i][j]
    return output

def ideal_low_filter(img, D0):
    #生成一个理想低通滤波器(并返回)
    h, w = img.shape[:2]
    filter_img = np.ones((h, w))
    u = np.fix(h / 2)
    v = np.fix(w / 2)
    for i in range(h):
        for j in range(w):
            d = np.sqrt((i - u) ** 2 + (j - v) ** 2)
            filter_img[i, j] = 0 if d > D0 else 1
    return filter_img
```

```
def butterworth_low_filter(img, D0, rank):
    #生成一个巴特沃斯低通滤波器(并返回)
    h, w = img.shape[:2]
    filter_img = np.zeros((h, w))
    u = np.fix(h / 2)
    v = np.fix(w / 2)
    for i in range(h):
        for j in range(w):
            d = np.sqrt((i - u) ** 2 + (j - v) ** 2)
            filter_img[i, j] = 1 / (1 + 0.414 * (d / D0) ** (2 * rank))
    return filter_img

def exp_low_filter(img, D0, rank):
    #生成一个指数低通滤波器(并返回)
    h, w = img.shape[:2]
    filter_img = np.zeros((h, w))
    u = np.fix(h / 2)
    v = np.fix(w / 2)
    for i in range(h):
        for j in range(w):
            d = np.sqrt((i - u) ** 2 + (j - v) ** 2)
            filter_img[i, j] = np.exp(np.log(1 / np.sqrt(2)) * (d / D0) ** (2 * rank))
    return filter_img

def filter_use(img, filter):
    #将图像 img 与滤波器 filter 结合,生成对应的滤波图像
    #首先进行傅里叶变换
    f = np.fft.fft2(img)
    f_center = np.fft.fftshift(f)
    #应用滤波器进行反变换
    S = np.multiply(f_center, filter)              #频率相乘——l(u,v) * H(u,v)
    f_origin = np.fft.ifftshift(S)                 #将低频移动到原来的位置
    f_origin = np.fft.ifft2(f_origin)              #使用 ifft2 进行傅里叶的逆变换
    f_origin = np.abs(f_origin)                    #设置区间
    return f_origin

def DFT_show(img):
    #对传入的图像进行傅里叶变换,生成频域图像
    f = np.fft.fft2(img)                           #使用 numpy 进行傅里叶变换
    fshift = np.fft.fftshift(f)                    #把零频率分量移到中间
    result = np.log(1 + abs(fshift))
    return result

#自生成实验图像,并添加椒盐噪声
src = np.zeros((300, 300), dtype = np.uint8)
salt_area1 = np.ones((130, 130), dtype = np.uint8)
salt_area1 = sp_noise(salt_area1, 0.04)
salt_area2 = np.zeros((130, 130), dtype = np.uint8)
salt_area2 = sp_noise(salt_area2, 0.04)
for i in range(10, 140):
    for j in range(10, 140):
        src[i, j + 75] = 255
        src[i + 150, j] = salt_area1[i - 10, j - 10] * 255
        src[i + 150, j + 150] = salt_area2[i - 10, j - 10]
my_img = src.copy()

'''理想低通滤波'''
```

```
ideal_filter = ideal_low_filter(my_img, D0 = 40)    #生成理想低通滤波器
ideal_img = filter_use(my_img, ideal_filter)        #将滤波器应用到图像,生成理想低通滤波图像
fre_img = DFT_show(my_img)                          #原图的频域图像
fre_ideal_img = DFT_show(ideal_img)                 #理想低通滤波图像的频域图像
plt.figure(dpi = 300)
plt.subplot(221)
plt.title('原始图')
plt.imshow(my_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(222)
plt.title("理想低通滤波图像")
plt.imshow(ideal_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(223)
plt.title('原始图频域图')
plt.imshow(fre_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(224)
plt.title("理想低通滤波图像的频域图")
plt.imshow(fre_ideal_img, cmap = plt.cm.gray)
plt.axis("off")
plt.show()

'''巴特沃斯低通滤波器'''
my_img = src.copy()
butterworth_filter = butterworth_low_filter(my_img, D0 = 10, rank = 2) #生成巴特沃斯低通
                                                                       #滤波器
butterworth_img = filter_use(my_img, butterworth_filter) #将滤波器应用到图像,生成巴特沃斯
                                                         #低通滤波图像
fre_butterworth_img = DFT_show(butterworth_img)    #巴特沃斯低通滤波图像的频域图像
plt.figure(dpi = 300)
plt.subplot(221)
plt.title('原始图')
plt.imshow(my_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(222)
plt.title("巴特沃斯低通滤波图像")
plt.imshow(butterworth_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(223)
plt.title('原始图频域图')
plt.imshow(fre_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(224)
plt.title("巴特沃斯低通滤波图像的频域图")
plt.imshow(fre_butterworth_img, cmap = plt.cm.gray)
plt.axis("off")
plt.show()

'''指数低通滤波器'''
my_img = src.copy()
exp_filter = exp_low_filter(my_img, D0 = 20, rank = 2)        #生成指数低通滤波器
exp_img = filter_use(my_img, exp_filter)          #将滤波器应用到图像,生成指数低通滤波图像
fre_exp_img = DFT_show(exp_img)                               #指数低通滤波图像的频域图像
plt.figure(dpi = 300)
```

```
plt.subplot(221)
plt.title('原始图')
plt.imshow(my_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(222)
plt.title("指数低通滤波图像")
plt.imshow(exp_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(223)
plt.title('原始图频域图')
plt.imshow(fre_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(224)
plt.title("指数低通滤波图像的频域图")
plt.imshow(fre_exp_img, cmap = plt.cm.gray)
plt.axis("off")
plt.show()
```

运行程序，效果如图 5-20～图 5-22 所示。

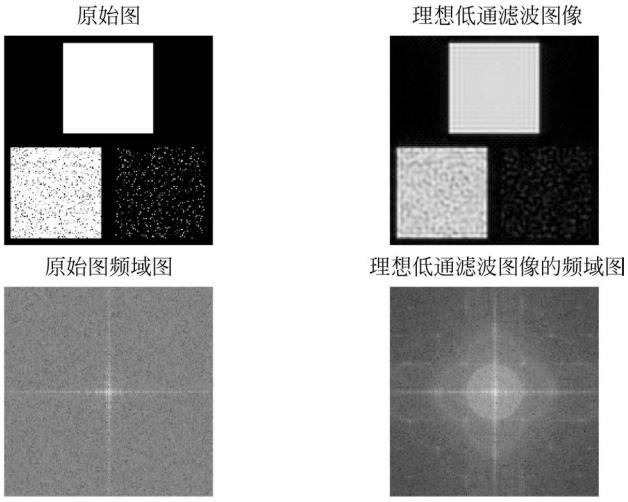


图 5-20　理想低通滤波效果

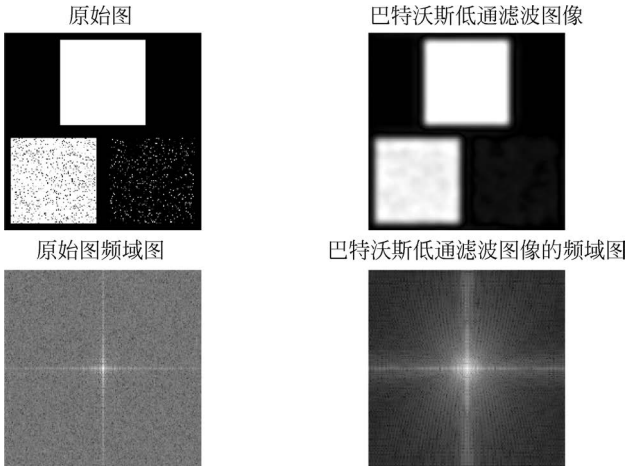


图 5-21　巴特沃斯低通滤波效果

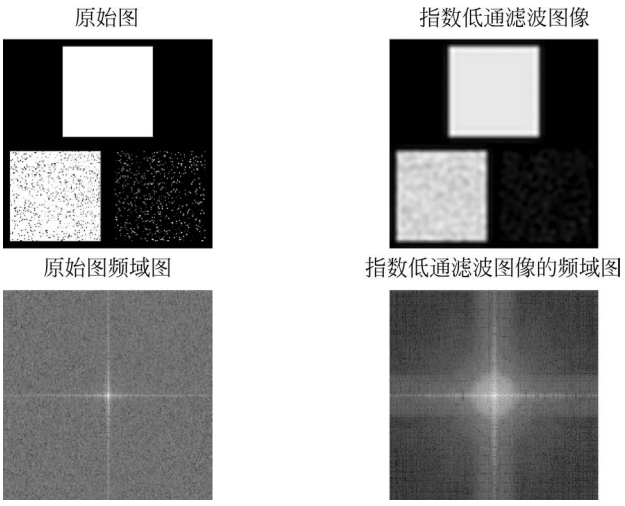


图 5-22　指数低通滤波效果

5.6　频域图像锐化

消除或减弱图像的低频分量从而增强图像中物体的边缘轮廓信息的过程称为图像锐化。图像锐化可以采用基于空间域的空域滤波的几种锐化方法或者基于频率域的高通滤波来处理。5.5 节介绍了几种空域滤波的锐化效果，本节将对高通滤波的锐化进行介绍。

首先，对一幅图像进行如下二维傅里叶变换：

$$F(u,v)=\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}f(x,y)\mathrm{e}^{-2\pi\left(\frac{u}{M}x+\frac{v}{N}y\right)}$$

将 $u=0$ 和 $v=0$ 代入上式，可以得到如下式子。

$$F(0,0)=MN\cdot\frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}f(x,y)=MN\cdot\bar{f}(x,y) \tag{5-6}$$

据式(5-6)，可以得到 $F(0,0)$的值是非常大的。此处，将 $F(0,0)$称为直流分量，直流分量比其他的成分要大好几个数量级。所以，这也就是傅里叶谱为什么要使用对数变换才能看清楚的原因。

对于高通滤波器而言，由于直流分量被衰减，所以，所得到的图像的动态范围是非常狭窄的，也就造成了图像偏灰。进一步而言，保持直流(DC)分量，对别的部分进行增强，可以增强图像的细节。这样的滤波器称为锐化滤波器。

5.6.1　理想高通滤波器

在以原点为圆心，以 D_0 为半径的圆内，无衰减地通过所有频率而在该圆外切断所有频率的二维高通滤波器，它由下面的函数来确定：

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

其中，D_0 是一个常数，$D(u,v)$是频率域中点(u,v)与频率矩形中心的距离，即

$$D(u,v)=\sqrt{u^2+v^2}$$

图 5-23 从左到右依次表示典型理想高通滤波器的透视图、图像表示和剖面图。

当把一个理想高通滤波器逆变换到空间域中后，会出现震荡曲线。用这样的模板做空间

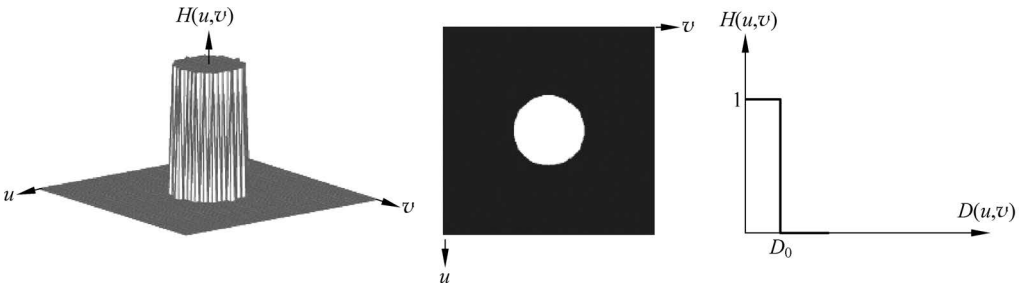


图 5-23　理想高通滤波器处理效果

域卷积时，就相当于与把这个模板复制到每个点周围（每个图像点就是一个冲激），如图 5-24 所示。

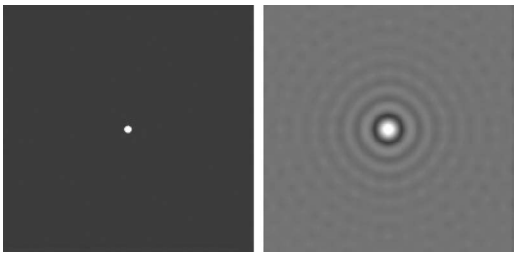


图 5-24　振铃现象

5.6.2　巴特沃斯高通滤波器

截止频率位于距原点 D_0 处的 n 阶巴特沃斯高通滤波器 BLPF 的传递函数为：

$$H(u,v)=\frac{1}{1+\left(\dfrac{D_0}{D(u,v)}\right)^{2n}}$$

图 5-25 显示了 BLPF 函数的透视图、图像显示和径向剖面图。

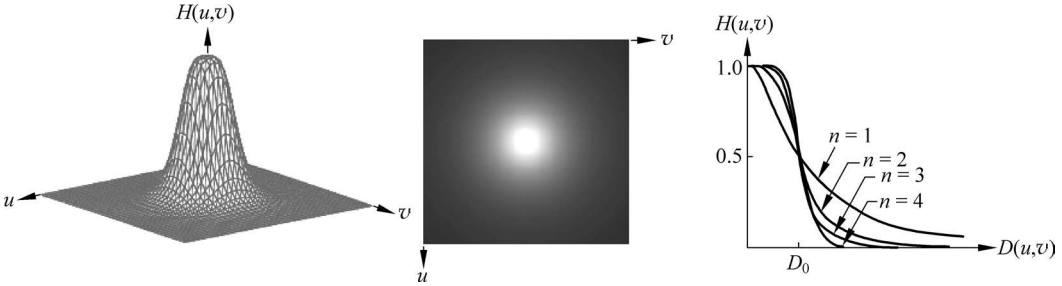


图 5-25　巴特沃斯高通滤波器处理效果

虽然巴特沃斯低阶时没有振铃现象，但是曲线太平缓，不能达到理想的分水岭效果，不能立即截止。想要提高截断的效果，就要提高滤波器的阶数。但是高阶的 BLPF 振铃现象会越来越明显。

5.6.3　指数高通滤波器

指数高通滤波器可用下述数学公式表示。

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

式中，n 为滤波器的增长速率因子，D_0 为截止频率，$D_0(u,v)$是点(u,v)到频率平面原点的距离，即

$$D(u,v)=\sqrt{u^2+v^2}$$

图 5-26 显示了指数高通滤波器函数的透视图、图像显示和径向剖面图。

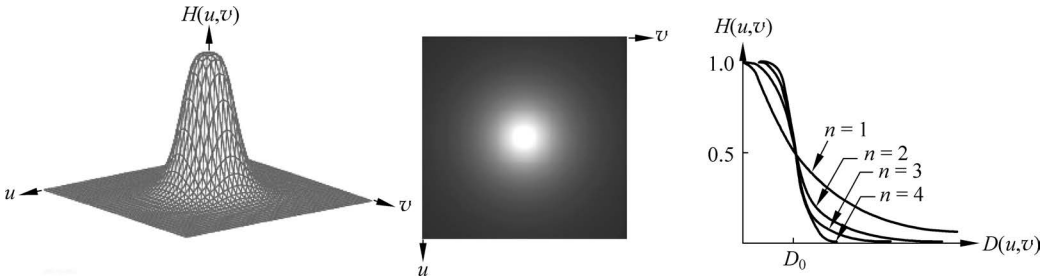


图 5-26　指数高通滤波器处理效果

5.6.4　频域高通滤波器的应用

本节通过实例来实现：选择一幅图像(rice.png)，分别使用理想高通滤波器、巴特沃斯高通滤波器、指数高通滤波器(至少使用两种高通滤波器)，显示滤波前后的频域能量分布图、空间图像。

【例 5-13】 对图像实现高通滤波处理。

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

def ideal_high_filter(img, D0):
    #生成一个理想高通滤波器(并返回)
    h, w = img.shape[:2]
    filter_img = np.zeros((h, w))
    u = np.fix(h / 2)
    v = np.fix(w / 2)
    for i in range(h):
        for j in range(w):
            d = np.sqrt((i - u) ** 2 + (j - v) ** 2)
            filter_img[i, j] = 0 if d < D0 else 1
    return filter_img

def butterworth_high_filter(img, D0, rank):
    #生成一个巴特沃斯高通滤波器(并返回)
    h, w = img.shape[:2]
    filter_img = np.zeros((h, w))
    u = np.fix(h / 2)
    v = np.fix(w / 2)
    for i in range(h):
        for j in range(w):
            d = np.sqrt((i - u) ** 2 + (j - v) ** 2)
            filter_img[i, j] = 1 / (1 + (D0 / d) ** (2 * rank))
    return filter_img

def exp_high_filter(img, D0, rank):
```

```
    #生成一个指数高通滤波器(并返回)
    h, w = img.shape[:2]
    filter_img = np.zeros((h, w))
    u = np.fix(h / 2)
    v = np.fix(w / 2)
    for i in range(h):
          for j in range(w):
              d = np.sqrt((i - u) ** 2 + (j - v) ** 2)
              filter_img[i, j] = np.exp((-1) * (D0 / d) ** rank)
    return filter_img

def filter_use(img, filter):
    #将图像 img 与滤波器 filter 结合,生成对应的滤波图像
    #首先进行傅里叶变换
    f = np.fft.fft2(img)
    f_center = np.fft.fftshift(f)
    #应用滤波器进行反变换
    S = np.multiply(f_center, filter)              #频率相乘——l(u,v) * H(u,v)
    f_origin = np.fft.ifftshift(S)                 #将低频移动到原来的位置
    f_origin = np.fft.ifft2(f_origin)              #使用 ifft2 进行傅里叶的逆变换
    f_origin = np.abs(f_origin)                    #设置区间
    f_origin = f_origin / np.max(f_origin.all())
    return f_origin

def DFT_show(img):
    #对传入的图像进行傅里叶变换,生成频域图像
    f = np.fft.fft2(img)                           #使用 NumPy 进行傅里叶变换
    fshift = np.fft.fftshift(f)                    #把零频率分量移到中间
    result = np.log(1 + abs(fshift))
    return result

src = cv.imread("wire.bmp", 0)
my_img = src.copy()
'''理想高通滤波'''
ideal_filter = ideal_high_filter(my_img, D0 = 40)  #生成理想高通滤波器
ideal_img = filter_use(my_img, ideal_filter)       #将滤波器应用到图像,生成理想高通滤波图像
fre_img = DFT_show(my_img)                         #原图的频域图像
fre_ideal_img = DFT_show(ideal_img)                #理想高通滤波图像的频域图像
plt.figure(dpi = 300)
plt.subplot(221)
plt.title('原图')
plt.imshow(my_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(222)
plt.title("理想高通滤波图像")
plt.imshow(ideal_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(223)
plt.title('原图频域图')
plt.imshow(fre_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(224)
plt.title("理想高通滤波图像的频域图")
```

```
plt.imshow(fre_ideal_img, cmap = plt.cm.gray)
plt.axis("off")
plt.show()

'''巴特沃斯高通滤波器'''
my_img = src.copy()
butterworth_filter = butterworth_high_filter(my_img, D0 = 40, rank = 2) #生成 Butterworth 高通
                                                                        #滤波器
butterworth_img = filter_use(my_img, butterworth_filter) #将滤波器应用到图像,生成 Butterworth
                                                         #高通滤波图像
fre_butterworth_img = DFT_show(butterworth_img)       #Butterworth 高通滤波图像的频域图像
plt.figure(dpi = 300)
plt.subplot(221)
plt.title('原图')
plt.imshow(my_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(222)
plt.title("Butterworth 高通滤波图像")
plt.imshow(butterworth_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(223)
plt.title('原图频域图')
plt.imshow(fre_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(224)
plt.title("Butterworth 高通滤波图像的频域图")
plt.imshow(fre_butterworth_img, cmap = plt.cm.gray)
plt.axis("off")
plt.show()

'''指数高通滤波器'''
my_img = src.copy()
exp_filter = exp_high_filter(my_img, D0 = 40, rank = 2) #生成指数高通滤波器
exp_img = filter_use(my_img, exp_filter)          #将滤波器应用到图像,生成指数高通滤波图像
fre_exp_img = DFT_show(exp_img)                      #指数高通滤波图像的频域图像
plt.figure(dpi = 300)
plt.subplot(221)
plt.title('原图')
plt.imshow(my_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(222)
plt.title("指数高通滤波图像")
plt.imshow(exp_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(223)
plt.title('原图频域图')
plt.imshow(fre_img, cmap = plt.cm.gray)
plt.axis("off")
plt.subplot(224)
plt.title("指数高通滤波图像的频域图")
plt.imshow(fre_exp_img, cmap = plt.cm.gray)
plt.axis("off")
plt.show()
```

运行程序,效果如图 5-27～图 5-29 所示。

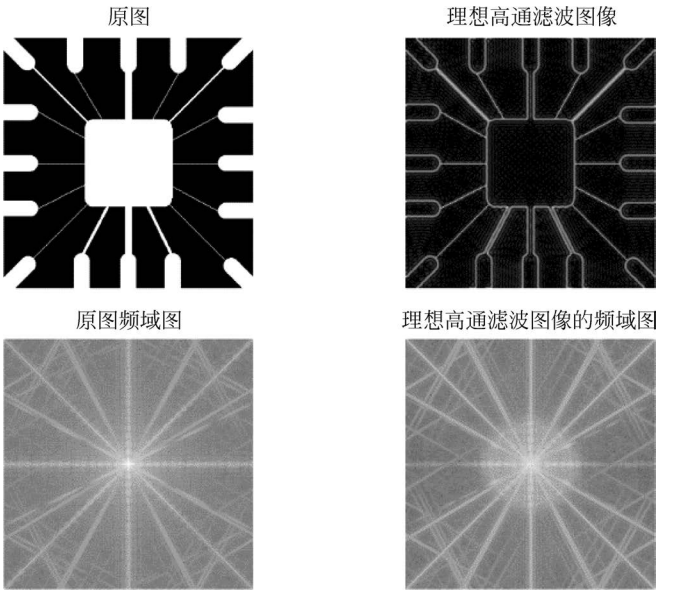


图 5-27　理想高通滤波效果

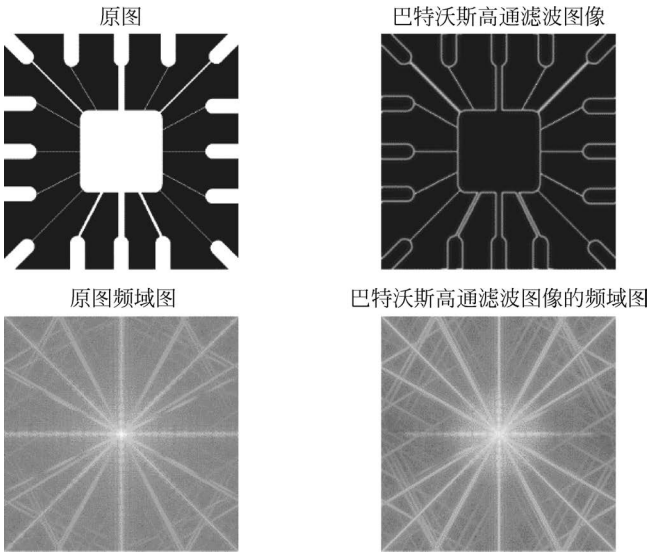


图 5-28　巴特沃斯高通滤波效果

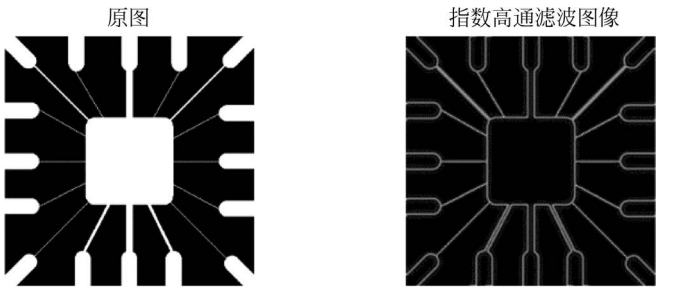


图 5-29　指数高通滤波效果

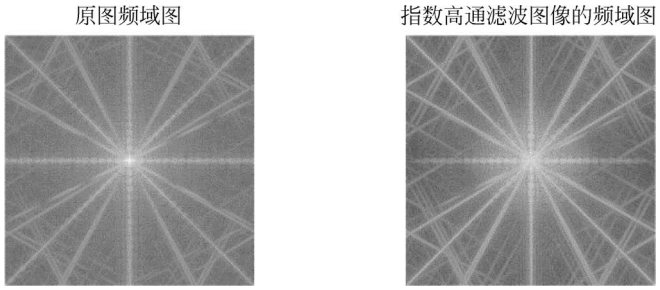


图 5-29 （续）

5.7 空/频域滤波的关系

频域滤波和空域滤波有着密不可分的关系。频域滤波器是通过对图像变化频率的控制来达到图像处理的目的，而空域滤波器是通过图像矩阵对模板进行卷积运算达到处理图像的效果。由卷积定理可知，空域上的卷积数值上等于图像和模板傅里叶变换乘积的反变换。

$$f(x,y)\times h(x,y)\Leftrightarrow F(u,v)H(u,v) \tag{5-7}$$

也就是说，如果将空域上的模板进行离散傅里叶变化得到频域上的模板，那么用空域模板进行空域滤波和用得到的频域模板进行频域滤波最后结果是一样的，两种方法有时可以互换。

```
p                    q
[[1 2 3 0 0]         [[1 1 1 0 0]
 [4 5 6 0 0]          [1 -8 1 0 0]
 [7 8 9 0 0]          [1 1 1 0 0]
 [0 0 0 0 0]          [0 0 0 0 0]
 [0 0 0 0 0]]         [0 0 0 0 0]]
```

图 5-30 填充后的 p 和 q

但需要注意的一点是，将原始图像与空域模板进行卷积运算，得到卷积结果的长度要比原来的图像长，就算对图像和模板进行填充，得到的卷积结果的第一位也不是模板在原始图像第一个像素处的卷积。例如，假设 p 位原始图像长度为 P，q 位卷积模板长度为 Q，则由卷积的运算公式(5-7)易得不产生混淆下图像的最小填充后尺寸为 $P+Q-1$，填充后的 p 和 q 如图 5-30 所示。

【例 5-14】 对给定的矩阵进行卷积运算。

```
import numpy as np
#保留小数点后三位
np.set_printoptions(precision = 3)
#不使用科学记数法
np.set_printoptions(suppress = True)

p = np.array([[1,2,3,0,0],[4,5,6,0,0],[7,8,9,0,0],[0,0,0,0,0],[0,0,0,0,0]])
q = np.array([[1,1,1,0,0],[1,-8,1,0,0],[1,1,1,0,0],[0,0,0,0,0],[0,0,0,0,0]])
pp = np.fft.fft2(p)
qq = np.fft.fft2(q)
tt = pp * qq
t = np.fft.ifft2(tt)
print('p\n', p)
print('q\n', q)
print('t\n', t.real)
```

运行程序，输出如下。

```
t
 [[  1.   3.   6.   5.   3.]
 [  5.   3.   3. -11.   9.]
 [ 12.  -9.   0. -21.  18.]
 [ 11. -39. -33. -53.  15.]
 [  7.  15.  24.  17.   9.]]
```

从上述运行结果可知，虽然进行零填充可以有效避免混淆，但无法改变的一点是，卷积后图像的尺寸会变大。可以看出卷积后的结果填满了整个 5×5 的矩阵。理论上，用模板在图像第一个像素处的卷积值（也就是 3）来替代图像原来的第一个像素更加恰当。实际上，真正在图像上的卷积结果位于 t 的中心处，即图 5-31 才是与原始图像相等大小的滤波结果。

```
 3.   3.  -11.
-9.   -0.  -21
-39.  -33.  -53.
```

图 5-31　滤波结果

因此要想得到和空域滤波器相同的结果，在填充和频域滤波之后提取图像时，就要将得到的处理结果的边缘去掉。假设模板大小为 $P \times Q$，则滤波后得到的边缘宽度为（floor(P/2)，floor(Q/2)）。

空域滤波和频域滤波的比较步骤如下。

（1）定义一个小尺寸的空域模板，用该模板进行空域滤波，获得滤波图像。

（2）根据空域滤波模板和原图像的大小计算频域模板的填充大小。

（3）将空域模板进行填充并乘以$(-x)+y$，然后进行傅里叶变换得到频域模板。

（4）用得到的频域模板进行频域滤波，并对滤波结果进行截取。

（5）空域滤波和频域滤波的结果显示比较。

【例 5-15】 对 Sobel 算子的比较演示。

```
import frequency_function as fre
import airspace_filter as air
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

original_image_test4 = cv.imread('1.png',0)
'''比较对应的频域滤波器和空域滤波器是否等效'''
#比较 Sobel 算子
#得到空域 Sobel 滤波函数
airspace_result_test1 = air.laplace_sharpen(original_image_test4, my_type = 'big')
#定义空域滤波模板
air_model = np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]])
#计算模板填充后的尺寸
shape = (2 * original_image_test4.shape[0], 2 * original_image_test4.shape[1])
#将空域模板填充到相应尺寸，变换为频域模板并将低频移至中心
fre_model = np.fft.fft2(fre.my_get_fp(fre.my_fill(air_model, shape)))
#用频域模板进行频域滤波
frequency_result_test1 = fre.myfunc_seldifine(original_image_test4, fre_model, output_offset = (1, 1))
#将滤波结果的像素值转换到 0～255
airspace_result_test1 = air.show_edge(airspace_result_test1)
frequency_result_test1 = air.show_edge(frequency_result_test1)
#结果显示
plt.subplot(131)
plt.imshow(original_image_test4)
plt.title('原图')
plt.subplot(132)
plt.imshow(airspace_result_test1)
plt.title('空域滤波')
plt.subplot(133)
plt.imshow(frequency_result_test1)
plt.title('频域滤波')
plt.show()
```

运行程序，效果如图 5-32 所示。

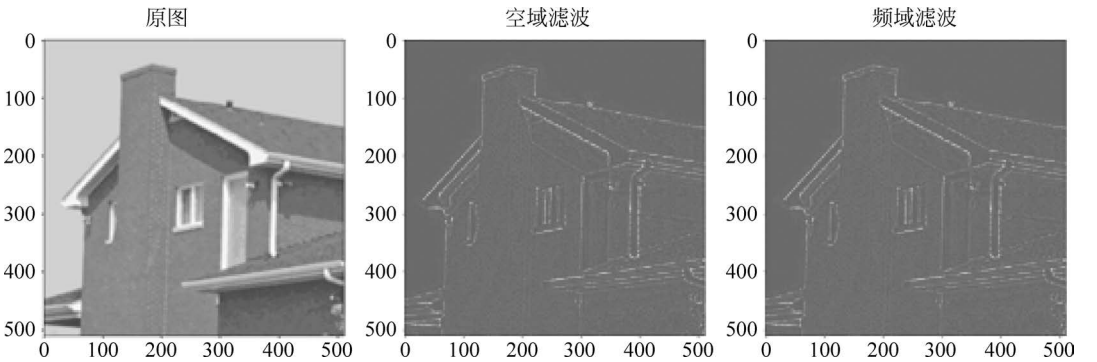


图 5-32　Sobel 算子比较效果

由图 5-32 可以看出两种方式效果是相同的。