2019 年秋季《图像处理与分析》编程作业-04

【图像增强处理】

姓名：　　学号：

【问题 1】编写一个图像直方图均衡化程序，g=histequal4e(I)，其中 I 是 8 比特图像

问题分析：

为了实现该问题，首先需要明白直方图均衡化的原理。对于灰度级范围为[0, L−1]的数字图像，其直方图可以表示为一个离散函数 $h(r_k)=n_k$，其中 r_k 是第 k 级灰度值（intensity value），n_k 是图像中灰度值为 r_k 的像素个数。假设灰度图像的维数是 M×N，MN 表示图像的像素总数，则归一化直方图可以表示为：

$$p(r_k) = n_k/MN,\ \ k = 0, 1, \ldots, L-1 \tag{1-1}$$

均衡化过程是一个灰度映射公式，该公式是将灰度级的分布进行累加并乘以灰度级最大值作为映射关系，公式如下：

$$\begin{aligned} s_k &= T(r_k) \\ &= (L-1)\sum_{j=0}^{k} p_r(r_j) = \frac{L-1}{MN}\sum_{j=0}^{k} n_j, k = 0, 1, 2, \cdots, L-1 \end{aligned} \tag{1-2}$$

其中，MN 为图像像素总数，n_k 是灰度为 r_k 的像素个数，L 是图像可能的灰度级数量(对 8 比特图像 L=256)。输出图像中像素的灰度值可由输入图像中像素灰度 r_k 映射为 s_k 后得到。

具体的实现步骤：

（1）主意扫过图像每一个像素，统计[0,255]的每一灰度值 r_k 对应的像素点数 n_k 存成一个长度为 256 的灰度分布数组 hist；

（2）将一维数组 hist 元素除以图像像素总数 M*N 归一化到[0,1]间，即 n_k/M*N；

（3）将数组元素进行累加，即从第 1 项开始令 hist[i]=hist[i-1]+ hist[i]（i=1…255）；

（4）将数组中的元素乘以 255 对应于均衡化后的灰度值 s_k，得到一个新的灰度分布；

（5）将原始图像像素灰度为 $r=r_k$ 映射到新灰度值 $s=s_k$；

（6）完成图像灰度直方图均衡化。

实现细节及代码：

- 编程语言：Python3.7+
- 所用模块：Numpy1.17.2（用于矩阵处理），PIL5.3.0（用于图像读取等基本操作），

matplotlib3.0.2（用于绘制图像等），opencv-python4.1.1（调用其灰度直方图均衡函数进行比较）

- 函数文件：histequal.py
- 主程序文件：histequal_test.py
- histequal4e(I)函数文件相关变量：

  I ： 输入图像矩阵

  hist ： 灰度分布一维数组

  m, n ： 输入图像的高度和宽度

  img ： 输出的均衡化图像矩阵

函数 histequal4e(I)实现代码：

```python
import numpy as np

def histequal4e(I):
    #对输入I的类型以及数据类型进行检测，如果不满足则程序运行终止
    assert(type(I)==np.ndarray)
    assert(I.dtype==np.uint8)#要求输入的图像是8比特图像
    #直方图统计
    hist=[0 for x in range(256)] #新建一个长度为256的0元素列表
    m,n=I.shape
    img=np.zeros((m,n))          #新建一个与输入同尺寸的零矩阵作为输出
    for i in range(m):
        for j in range(n):
            hist[I[i,j]]+=1
    #归一化处理
    for i in range(256):
        hist[i]=hist[i]/(n*m)
    #直方图累加
    for i in range(1,256):
        hist[i]+=hist[i-1]
    #均衡化得到新的灰度分布
    for i in range(256):
        hist[i]=np.uint8(hist[i]*255)
    #得到输出图像的对应像素灰度值
    for i in range(m):
        for j in range(n):
            img[i,j]=hist[I[i,j]]

    return img
```

- 主程序相关变量：

  I ： 输入图像’ assignment01_images\einstein.tif’得到的矩阵

  img ： 自定义函数 histequal4e(I)得到的图像矩阵

  img1 ： 调用 cv2.equalizeHist(I)函数得到的灰度直方图均衡化图像矩阵

主程序代码：

```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import cv2
import histequal                #导入自编写的函数模块

I=np.array(Image.open('assignment01_images\einstein.tif'))#读取图像
img=histequal.histequal4e(I)                                #直方图均衡化
img1= cv2.equalizeHist(I)      #使用Opencv库直方图均衡化函数

plt.gray()
plt.figure(figsize=(15,8))

plt.subplot(231)
plt.title('(a)Original Image')
plt.imshow(I)      #绘制原始图像
plt.subplot(232)
plt.title('(b)After My_Histogram_equalization')
plt.imshow(img)  #绘制自编直方图均衡化函数作用后的图像
plt.subplot(233)
plt.title('(c)After Opencv_Histogram_equalization')
plt.imshow(img1) #绘制opencv库函数均衡后的图像
plt.subplot(234)
plt.title('(e)The histogram of origin image')
plt.hist(I.ravel(),256,[0,256]) #绘制原始图像的灰度直方图(ravel函数将多维数组降为一维数组)
plt.subplot(235)
plt.title('(f)Histogram:Using My histequal_function')
plt.hist(img.ravel(),256,[0,256]) #绘制自编写函数均衡化后的直方图
plt.subplot(236)
plt.title('(g)Histogram:Using cv2.equalizeHist')
plt.hist(img1.ravel(),256,[0,256])#绘制opencv库函数均衡化后图像的直方图

plt.savefig('hist-equalization.jpg')
```

效果展示：

运行主程序文件 histequal_test.py，实现对图像’assignment01_images\einstein.tif’的灰度直方图均衡化。显示的图像结果如下图所示：

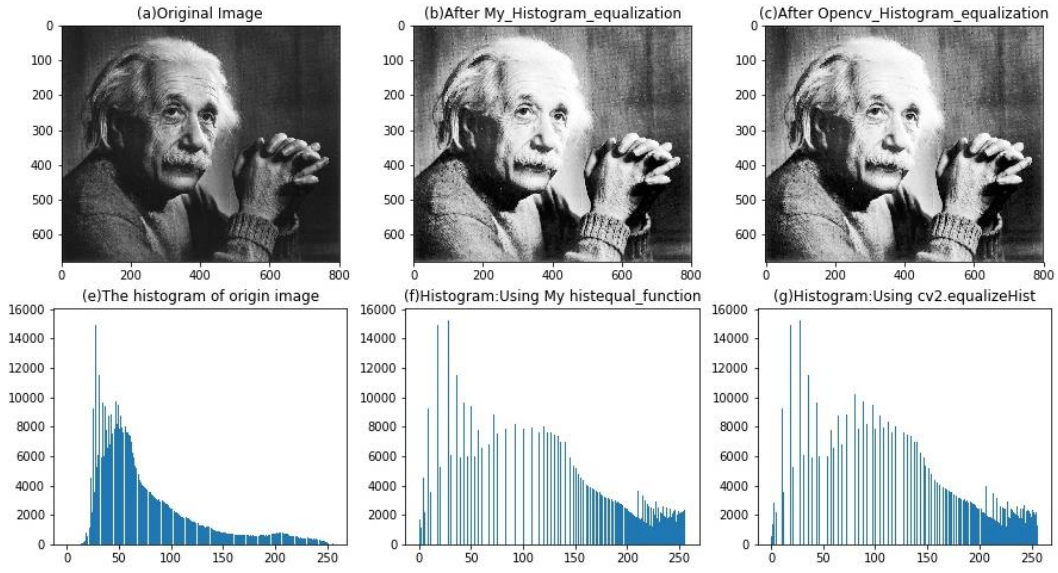

图 1.问题一程序运行结果图

图中（a）为原始图像，（b）为自编写函数实现的均衡化图像，（c）为 opencv-python 库

函数实现的灰度直方图均衡化图像，（d）（e）（f）为分别对应（a）（b）（c）的灰度直方图。可以发现按照基本原理编写的灰度直方图均衡化函数实现效果与 opencv 库函数实现效果基本一致。可以观察到，对图像做均衡化有助于图像直方图的延展，均衡化后图像的灰度级范围更宽，有效地增强了图像的对比度。但是，由于灰度直方图是概率分布函数的近似，并且处理后不产生新的灰度级，所以在实际的直方图均衡化很少能够得到完全平坦的直方图。在离散情况下，通常不能证明离散的直方图均衡化能得到均匀的直方图。

【问题 2】编写一个程序完成如下功能：读入清晰图像，加上椒盐噪声，采用 有选择保边缘平滑法对图像进行平滑。

问题分析：

本问题实现分为两部分：首先是对读入的清晰图像添加椒盐噪声；其次是编写“有选择保边缘平滑”函数对图像实现平滑滤波。

首先考虑生成椒盐噪声的函数。椒盐噪声也称为脉冲噪声，是图像中经常见到的一种噪声，它是一种随机出现的白点或者黑点，可能是亮的区域有黑色像素或是在暗的区域有白色像素（或是两者皆有）。生成（添加）椒盐噪声算法步骤如下：

（1）输入一幅图像并自定义信噪比 SNR（其取值范围在[0, 1]之间）；

（2）计算图像像素点个数 SP，进而得到椒盐噪声的像素点数目 NP = SP * (1-SNR)；

（3）随机获取要加噪的每个像素位置 img [i , j]；

（4）随机生成[0,1]之间的一个浮点数；

（5）判定浮点数是否大于 0.5，并指定像素灰度值为 255 或者 0；

（6）重复 3，4，5 三个步骤完成所有像素的 NP 个像素加粗样式；

（7）输出加人噪声以后的图像。

接着考虑设计平滑滤波的函数。有选择保边缘平滑滤波算法的原理是在一个 5×5 的掩膜区域内分 9 个小的掩膜区域计算灰度值方差，选择方差最小的掩膜区域的灰度均值作为输出图像的灰度。9 个掩膜包括其中包括一个 3×3 正方形、4 个五边形和 4 个六边形，如下图所示：

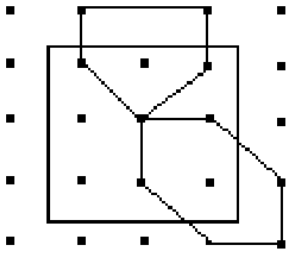

图 2.有选择保边缘掩膜示意图

具体的实现步骤如下所示：

（1）对输入图像边缘填充（零填充或复制填充）得到新的图像；

（2）对填充后的图像循环进行掩膜计算；

（3）掩膜计算内部包含 9 个子掩膜的均值及方差计算；

（4）比较方差大小，最小方差掩膜对应的均值作为输出像素的灰度值；

（5）结束循环操作，输出图像。

关于上述计算中的填充，分为零填充和复制填充。填充时包括 8 个部分，图像的上、下左、右 4 块以及四个角对应的矩阵，若使用复制填充则分别复制原图像的第一行、最后一行、第一列、最后一列以及四个角上 4 个像素点的灰度值，如下图所示：

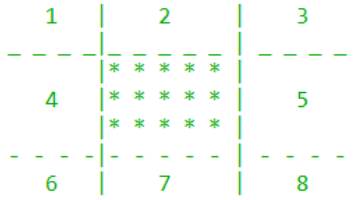

图 3.填充区域示意图

实现细节及代码：

- 编程语言：python3.7+
- 使用模块：Numpy1.17.2（用于矩阵处理），PIL5.3.0（用于图像读取等基本操作），matplotlib3.0.2（用于绘制图像等）
- 函数文件：smooth.py（平滑滤波函数），SaltPepper.py（产生椒盐噪声）
- 主程序文件：smooth_test.py
- smoothFunc(I, padding='zero')函数相关变量：

  I ：输入图像矩阵

  padding ：图像填充方式，默认值为’zero’（即零元素填充）

  m, n ：输入图像的高度和宽度

  I1 ：填充后的矩阵，用于掩膜计算

  img ：输出图像矩阵

  mean ：9 个子掩膜计算求得的均值组成的一维数组，长度为 9

  var ：9 个子掩膜计算求得的方差组成的一维数组，长度为 9

  A1…A9：9 个掩膜各自内部元素组成的一维数组

  a ：var 数组最小值对应的索引值

有选择保边缘平滑滤波函数 smoothFunc(I, padding='zero')代码：

```python
import numpy as np
def smoothFunc(I,padding='zero'):
    m,n=I.shape
    img=np.zeros((m,n))
    I1=np.zeros((m+4,n+4))
    #周围填充0保证输出图像尺寸不变
    for i in range(m):
        for j in range(n):
            I1[i+2,j+2]=I[i,j]

    #根据不同的填充方式来填充
    if padding=='zero':
        pass
    if padding=='replicate':#填充方式为复制

        for i in range(2):
            I1[i,2:2+n] = I[0,:]              #填充图片正上方矩阵（复制第一行）
            I1[m+2+i,2:2+n] = I[-1,:]     #填充图片正下方矩阵（复制最后一行）
        for j in range(2):
            I1[2:m+2,j]=I[:,0]                #填充图片正左方矩阵（复制第一列）
            I1[2:m+2,n+2+j]=I[:,-1]       #填充图片正右方矩阵（复制最后一列）

        I1[0:2,0:2] = I[0,0]                #左上角小矩阵填充
        I1[0:2,n+2:n+4] = I[0,-1]           #右上角小矩阵填充
        I1[m+2:m+4,0:2] = I[-1,0]           #左下角小矩阵填充
        I1[m+2:m+4,n+2:n+4] = I[-1,-1]      #右下角小矩阵填充
    #掩膜计算
    for i in range(2,m+2):
        for j in range(2,n+2):
            #3*3正方形掩膜
            mean=[]
            var=[]
            A1=I1[(i-1):(i+1),(j-1):(j+1)]
            mean.append(np.mean(A1))
            var.append(np.var(A1))
            #五边形掩膜-1（上）
            A2=[I1[i-2,j-1],I1[i-2,j],I1[i-1,j+1],I1[i-1,j-1],I1[i-1,j],I1[i-1,j+1],I1[i,j]]
            mean.append(np.mean(A2))
            var.append(np.var(A2))
            #五边形掩膜-2（下）
            A3=[I1[i+2,j-1],I1[i+2,j],I1[i+1,j+1],I1[i+1,j-1],I1[i+1,j],I1[i+1,j+1],I1[i,j]]
            mean.append(np.mean(A3))
            var.append(np.var(A3))
            #五边形掩膜-3（左）
            A4=[I1[i-1,j-2],I1[i,j-2],I1[i+1,j-2],I1[i-1,j-1],I1[i,j-1],I1[i+1,j-1],I1[i,j]]
            mean.append(np.mean(A4))
            var.append(np.var(A4))
            #五边形掩膜-4（右）
            A5=[I1[i-1,j+2],I1[i,j+2],I1[i+1,j+2],I1[i-1,j+1],I1[i,j+1],I1[i+1,j+1],I1[i,j]]
            mean.append(np.mean(A5))
            var.append(np.var(A5))
            #六边形掩膜-1（左上）
            A6=[I1[i-2,j-2],I1[i-1,j-2],I1[i-2,j-1],I1[i-1,j-1],I1[i,j-1],I1[i-1,j],I1[i,j]]
            mean.append(np.mean(A6))
            var.append(np.var(A6))
            #六边形掩膜-1（右上）
            A7=[I1[i-2,j+2],I1[i-1,j+2],I1[i-2,j+1],I1[i-1,j+1],I1[i,j+1],I1[i-1,j],I1[i,j]]
            mean.append(np.mean(A7))
            var.append(np.var(A7))
            #六边形掩膜-1（左下）
            A8=[I1[i+2,j-2],I1[i+1,j-2],I1[i+2,j-1],I1[i+1,j-1],I1[i,j-1],I1[i+1,j],I1[i,j]]
            mean.append(np.mean(A8))
            var.append(np.var(A8))
```

```
            #六边形掩膜-1（右下）
            A9=[I1[i+2,j+2],I1[i+1,j+2],I1[i+2,j+1],I1[i+1,j+1],I1[i,j+1],I1[i+1,j],I1[i,j]]
            mean.append(np.mean(A9))
            var.append(np.var(A9))
            #提取最小方差掩膜的均值
            a=var.index(min(var))
            img[i-2,j-2]=mean[a]
    return np.array(img,dtype='uint8')
```

- sp_noise(img, snr=0.9)函数（生成椒盐噪声）相关变量：

  img ： 输入图像矩阵

  snr ： 设置的信噪比，默认设置成 0.9

  m, n ： 输入图像的高度和宽度

  img1 ： 输出图像矩阵（添加噪声）

  sp ： 图像总像素点数

  NP ： 根据信噪比计算的产生椒盐噪声的像素点个数

  x, y ； 随机生成的噪声点横坐标与纵坐标

椒盐噪声生成函数代码：

```
import numpy as np
#椒盐噪声函数
def sp_noise(img,snr=0.9):
    #img:原图像; snr: 信噪比
    m,n=img.shape
    img1=img.copy()       # 复制原图形模板
    sp=m*n                # 计算图像像素点个数
    NP=int(sp*(1-snr))    # 计算图像椒盐噪声点个数
    for i in range (NP):
        x=np.random.randint(1,m-1)   # 生成一个 1 至 m-1 之间的随机整数
        y=np.random.randint(1,n-1)   # 生成一个 1 至 n-1 之间的随机整数
        if np.random.random()<=0.5:  #随机生成一个 0 至 1 之间的浮点数
            img1[x,y]=0              #椒噪声（黑色）
        else:
            img1[x,y]=255            #盐噪声（白色）
    return img1
```

- 主程序相关变量：

  I ： 输入图像’ assignment01_images\cameraman.tif’得到的矩阵

  I_sp ： 添加椒盐噪声（信噪比为 0.9）得到的图像矩阵

  I_smooth1 ： 使用零元素填充的平滑得到的图像矩阵

  I_smooth2 ： 使用复制填充的平滑得到的图像矩阵

  difference ： I_smooth1 与 I_smooth2 差值绝对值矩阵

主程序代码：

```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import smooth
import SaltPepper

I=np.array(Image.open('assignment01_images/cameraman.tif'))  #读取图像
I_sp=SaltPepper.sp_noise(I,snr=0.9)          #加入椒盐噪声
I_smooth1=smooth.smoothFunc(I_sp,'zero') #使用有选择保边缘平滑滤波(零元素填充)
I_smooth2=smooth.smoothFunc(I_sp,'replicate') #使用有选择保边缘平滑滤波(复制填充)
difference=abs(I_smooth2-I_smooth1)

plt.gray()
plt.figure(figsize=(15,10))
plt.subplot(231)
plt.title('(a)Original Image')
plt.imshow(I)
plt.subplot(232)
plt.title('(b)Add Salt_Pepper noise')
plt.imshow(I_sp)
plt.subplot(233)
plt.title('(c)After filtering(padding=zero)')
plt.imshow(I_smooth1)
plt.subplot(234)
plt.title('(d)After filtering(padding=replicate)')
plt.imshow(I_smooth2)
plt.subplot(235)
plt.title('(e)Difference between (c)and(d)')
plt.axis('off')
plt.imshow(difference)
plt.subplot(236)
plt.title('(f)Reverse image(e)')
plt.axis('off')
plt.imshow(255-difference)

plt.savefig('smooth_test.jpg')
```

效果展示：

运行主程序文件 smooth.py 得到下面的图像结果：

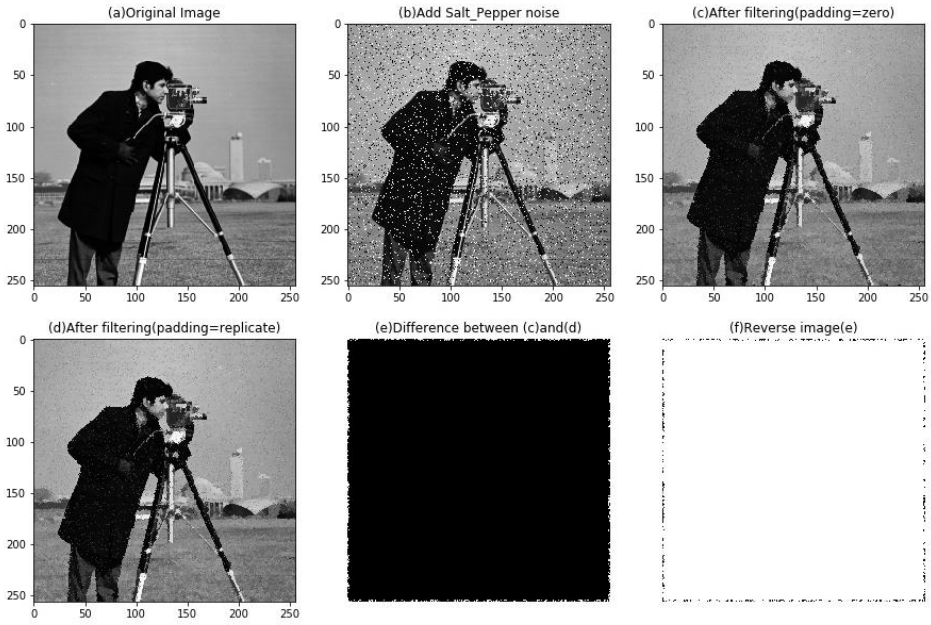

图 4.问题 2 程序运行结果图

其中图中（a）为原始图像，（b）为加入椒盐噪声的图像，（c）为使用补零填充平滑滤

波后的图像，（d）为复制填充平滑滤波后的图像，（e）为（c）与（d）图像差值的绝对值图像，（f）为（e）取反后的图像。可以观察到，有选择保边缘平滑对于椒盐噪声的滤除效果比较好，在滤除噪声的同时较好地保持了原始图像的边缘。而填充方式的不同仅仅在图像外边缘产生了也许差异。

分析来看，有选择保边缘平滑方法以方差作为各个区域灰度均匀性的测度：若区域含有尖锐的边缘，它的灰度方差必定很大，而不含边缘或灰度均匀的区域，它的方差就小，那么最小方差所对应的区域就是灰度最均匀区域。因此有选择保边缘平滑法既能够消除噪声，又不破坏区域边界的细节。另外，五边形和六边形在（x,y）处都有锐角，这样，即使像素（x,y）位于一个复杂形状区域的锐角处，也能找到均匀的区域。从而在平滑时既不会使尖锐边缘模糊，也不会破坏边缘形状。

【问题 3】编写一个程序完成拉普拉斯增强。

问题分析：

本问题主要涉及拉普拉斯算子的图像增强原理。对于一个图像的边缘来说，边缘两侧像素的灰度会有一个较大梯度的改变，如下图中“原信号”所示。对图像灰度求一阶导数，则边缘位置会有一个尖峰脉冲信号。再求二阶导数，其峰值位于原图像信号灰度梯度变化最快的位置，其检测边缘的能力比一阶导数要强。将二阶微分结果取反后乘以一定的锐化系数叠加到原图像上，则可以将图像的边缘增强。

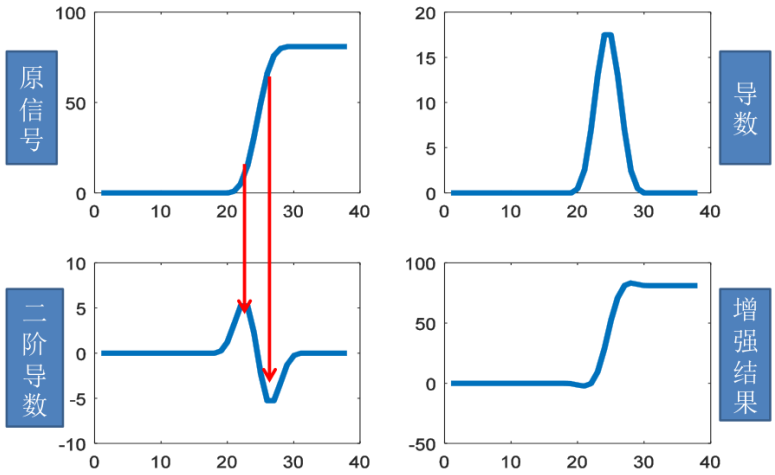

图 5.二阶微分算子增强图像的原理说明图

根据这个原理，Laplace 二阶微分算子公式即为：

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \quad (3\text{-}1)$$

对离散的数字图像而言，二阶偏导数可用二阶差分近似，可推导出 Laplacian 算子表达式为：

$$\nabla^2 f(x, y) = f(x+1, y)+f(x-1, y)+f(x, y+1)+f(x, y-1)-4f(x, y) \quad (3\text{-}2)$$

因此 Laplacian 增强算子为：

$$g(x, y) = f(x, y)-\nabla^2 f(x, y) = 5f(x, y) - [f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1)]$$

在实际应用中往往在叠加过程中加上一个锐化系数 C，即输出图像为

$$g(x, y) = f(x, y)-c\nabla^2 f(x, y) \quad (3\text{-}3)$$

实现步骤：

（1）对输入图像 I 边缘填充（零填充或复制填充）得到新的图像 I1；

（2）对填充后的图像 I1 循环进行拉普拉斯算子卷积运算；

（3）卷积得到的结果乘以锐化系数 c 加到 I1 上；

（4）将 I1 的填充边缘进行裁剪得到增强后的输出图像 img_out；

实现细节及代码：

- 编程语言：python3.7+
- 使用模块：Numpy1.17.2（用于矩阵处理），PIL5.3.0（用于图像读取等基本操作），matplotlib3.0.2（用于绘制图像等）
- 函数文件：laplace.py（拉普拉斯增强函数文件）
- 主程序文件：laplace_test.py
- Laplace_sharpen(I, c=0.1, padding='zero')函数相关变量：

  I ：输入图像矩阵

  Laplace_filter ：拉普拉斯算子

  padding ：图像填充方式，默认值为’zero’（即零元素填充）

  m, n ：输入图像的高度和宽度

  I1 ：填充后的矩阵，用于掩膜计算

  img_out ：输出图像矩阵

拉普拉斯增强函数详细代码：

```python
import numpy as np
def laplace_sharpen(I,c=0.1,padding='zero'):
    #c锐化系数，这里设置缺省值为0.1;padding为填充方式，默认为零元素填充
    # 拉普拉斯算子
    laplace_filter = np.array([
        [0, 1, 0],
        [1, -4, 1],
        [0, 1, 0],
    ])
    m,n=I.shape
    I1=np.zeros((m+2,n+2))
    #周围填充0保证输出图像尺寸不变
    for i in range(m):
        for j in range(n):
            I1[i+1,j+1]=I[i,j]
    #根据不同的填充方式来填充
    if padding=='zero':
        pass
    if padding=='replicate':#填充方式为复制
        I1[0,1:1+n] = I[0,:]          #复制第一行
        I1[-1,1:1+n] = I[-1,:]     #复制最后一行
        I1[1:m+1,0]=I[:,0]            #复制第一列
        I1[1:m+1,-1]=I[:,-1]       #复制最后一列
        I1[0,0] = I[0,0]              #左上角填充
        I1[0,-1] = I[0,-1]           #右上角填充
        I1[-1,0] = I[-1,0]           #左下角填充
        I1[-1,-1] = I[-1,-1]        #右下角填充
    I1=np.array(I1,dtype='uint8')

    img_out = np.copy(I1)  # 复制填充后的图像作为输出图像
    for i in range(1, m+1):
        for j in range(1, n+1):
            # 拉普拉斯算子在图像上卷积计算
            R = np.sum(laplace_filter * I1[i - 1:i + 2, j - 1:j + 2])
            #将算子计算结果乘以锐化系数叠加到原图像上作为输出
            img_out[i, j] = I1[i, j] - c * R
    img_out = img_out[1:m+1, 1:n+1]  # 把输出图像周围多余填充区域裁剪

    return img_out
```

效果展示：

为了对不同锐化系数的拉普拉斯增强效果进行比较，选择了锐化系数 c=0.1、0.5、1 三种情况，得到不同的拉普拉斯增强图片。运行主程序文件 laplace_test.py 可得到下面的图像结果：

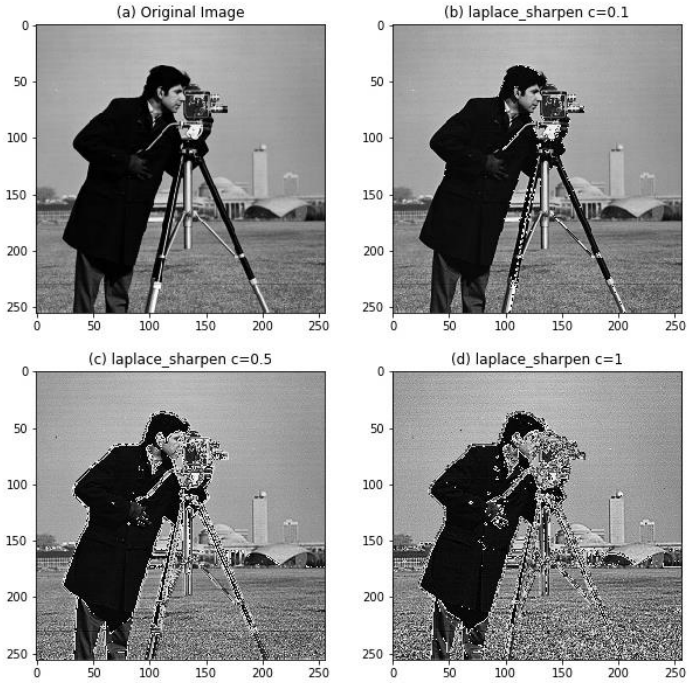

图 6.问题 3 程序运行结果图

如图所示，（a）为原始图像，（b）（c）（d）对应于锐化系数 c=0.1、0.5、1 的增强结果。当锐化系数为 1 时，实际上就是将原图像减去拉普拉斯算子的卷积结果得到的图像。这种情况用算子表示就是矩阵(0, -1, 0 ; -1, 5, -1 ; 0, -1, 0)，即课堂上老师所讲的“拉普拉斯增强算子”。在实际应用中加入锐化系数 c 即可对增强程度进行控制，得到想要的结果。本次实验结果中当锐化系数为 1 时，图像的边缘被很好地提取了，但是过度提取的多余边缘信息反而使得增强后的图像变得不清晰。