数字图像处理实验报告

项目名称：图像空域滤波和基本边缘检测

姓名：XXX

班级：XXXXXXXX

学号：XXXXXXXXXX

摘　要

该实验基于 OpenCV 计算机视觉库和 scikit-image 数字图像处理库，对图像进行了多种空域滤波处理。

本文对比使用了 Gaussian 滤波和中值滤波技术对图像进行平滑处理，得出两种策略对于不同的噪声有着不同的处理能力的结论。除此以外，还使用了 Unsharp Masking、Sobel Edge Detection、Laplace Edge Detection 和 Canny Algorithm 等算法对图像进行了边缘检测，并分别分析了四种策略的优缺点。

一 题目 1

1.1 题目

使用空域低通滤波器，分别用高斯滤波器和中值滤波器去平滑测试图像 test1 和 test2，模板大小分别是$3\times3,5\times5,7\times7$，并分析各自优缺点，图片如下。

图1　test1

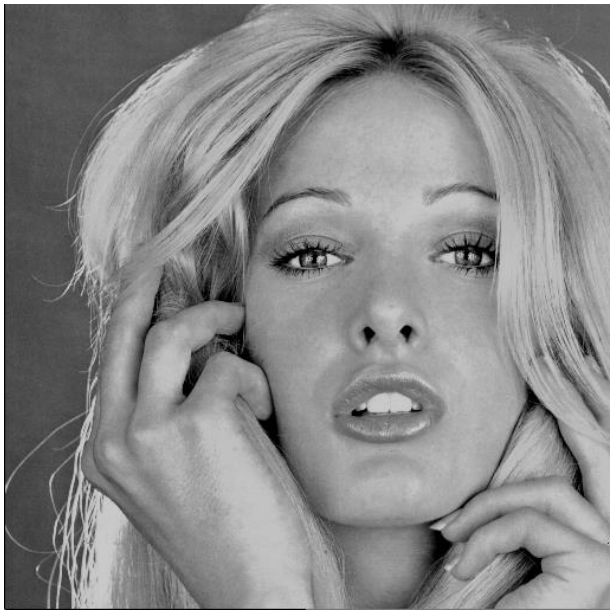

图2　test2

1.2 解答

1.2.1 高斯滤波

在图像处理中，高斯模糊（也称为高斯平滑）是通过高斯函数模糊图像的过程。高斯模糊是图像模糊滤波器的一种，它使用高斯函数——也就是概率统计中的正态分布——来计算应用于图像中每个像素的变换。注意到一维高斯函数的公式是

$$G(x)=\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{x^2}{2\sigma^2}}$$

在两个维度中，它是两个这样的高斯函数的乘积，其中每个维度一个高斯函数：

$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

其中x是水平轴上距原点的距离，y是垂直轴上距原点的距离，σ是高斯分布的标准偏差。当此公式应用于二维时，该公式生成的曲面的轮廓是同心圆，从中心点开始呈高斯分布。

对于高斯滤波器的卷积核（以下用 kernel 代替）而言，将 kernel 中心点坐标设为$(0,0)$，可得到 kernel 中每个位置的权重：

$$\omega(i,\ j)=\frac{1}{2\pi\sigma^2}e^{-\frac{(i-d-1)^2+(j-d-1)^2}{2\sigma^2}}$$

其中d为 kernel 半径，kernel 矩阵$\boldsymbol{W}\in\mathbb{R}^{(2d+1)\times(2d+1)}$。例如，对于$3\times3$的 kernel，可得出

$$\boldsymbol{W}=\frac{1}{2\pi\sigma^2}\begin{bmatrix}e^{-\frac{2}{2\sigma^2}} & e^{-\frac{1}{2\sigma^2}} & e^{-\frac{2}{2\sigma^2}}\\ e^{-\frac{1}{2\sigma^2}} & 1 & e^{-\frac{1}{2\sigma^2}}\\ e^{-\frac{2}{2\sigma^2}} & e^{-\frac{1}{2\sigma^2}} & e^{-\frac{2}{2\sigma^2}}\end{bmatrix}$$

在 OpenCV 中，我们使用 GaussianBlur()函数进行滤波，在该函数中，可设定高斯核标准差σ为 1.5，kernel 大小分别设定为$3\times3,5\times5,7\times7$，结果如下所示。

对图片 test1 高斯滤波：

图3　test1 高斯滤波3×3

图4　test1 高斯滤波5×5

图5　test1 高斯滤波7×7

对图片 test2 高斯滤波（为适应文档尺寸，图片已进行缩小）：

图6　test2 高斯滤波3×3

图7　test2 高斯滤波5×5

图8　test2 高斯滤波7×7

由如上滤波结果可见，高斯滤波器对图像噪声进行了有效去除，其中高斯 kernel 越大（OpenCV 中表示为参数 dsize 越大），滤波后图像越模糊。

1.2.2 中值滤波

对于中值滤波器，以3×3的 kernel 为例，若其掩膜区域如下所示：

$I(x-1,y-1)$	$I(x,y-1)$	$I(x+1,y-1)$
$I(x-1,y)$	$I(x,y)$	$I(x+1,y)$
$I(x-1,y+1)$	$I(x,y+1)$	$I(x+1,y+1)$

其中$I(x,y)$表示灰度图像(x,y)像素点的灰度值，则中值滤波计算函数可表示为：

$$f(x,y)=\mathbf{median}\begin{Bmatrix} I(x-1,y-1),I(x,y-1),I(x+1,y-1), \\ I(x-1,y),I(x,y),I(x+1,y), \\ I(x-1,y+1),I(x,y+1),I(x+1,y+1) \end{Bmatrix}$$

其中**median**表示取中位数。

在 OpenCV 中，我们使用 medianBlur()函数进行滤波，该函数使用具有ksize × ksize kernel 孔径的中值滤波器平滑图像，多通道图像的每个通道都是独立处理的。以下为滤波结果。

对图片 test1 中值滤波：

图9　test1 中值滤波 ksize=3

图10　test1 中值滤波 ksize=5

图11　test1 中值滤波 ksize=7

对图片 test2 中值滤波（为适应文档尺寸，图片已进行缩小）：

图12　test2 中值滤波 ksize=3

图13　test2 中值滤波 ksize=5

图14　test2 中值滤波 ksize=7

以上结果可以表明，中值滤波同样对噪声滤除作用较好，kernel 的尺寸越大，滤波作用越明显，图片越模糊。

1.2.3 比较

所有输出结果一并显示如下：

图15　输出结果对比

而高斯滤波的优势在于对高斯型噪声（Gaussian noise）有非常好的降噪效果，但是其计算相对复杂。需要注意的是，高斯滤波对整幅图像所有细节进行了模糊化处理，这样的一个特质需具体问题具体分析其利弊。

而中值滤波的优势是计算简单迅速，能够有效的去除图像中的椒盐噪音（Salt-and-pepper noise）。其软肋是对于不能有效地处理高斯噪声。

对于本题而言，采用中值滤波的效果略微好于高斯滤波的效果。kernel 的尺寸越大，滤波效果越明显，但图像的细节则会有所损失，需要合理权衡图像模糊程度与图像细节保留之间的关系。

二 题目 2

2.1 题目

利用固定方差$\sigma=1.5$产生高斯滤波器，分析各自优缺点。

2.2 解答

本题中方差$\sigma=1.5$的滤波器已在题目 1 中通过 GaussianBlur()函数使用过，本题中仅阐述高斯核的产生方法，以下均以3×3高斯核为例。

一种方法是根据定义，使用高斯函数对 kernel 矩阵中每个点进行计算，基于 Python 的 numpy 库，可编写如下代码：

```
def myGaussianKernel(size, sigma):
    ax = np.linspace(-(size - 1) / 2., (size - 1) / 2., size)
    gauss = np.exp(-0.5 * np.square(ax) / np.square(sigma))
    my_kernel = np.outer(gauss, gauss)
    return my_kernel / np.sum(my_kernel)
```

（注意计算后应归一化）

另一种更简单的写法是使用 numpy 中的 fromfunction()函数，使得代码更简洁，其原理与以上方法一致：

```
kernel = np.fromfunction(lambda x, y: (1 / (2 * math.pi *
(sigma ** 2))) * math.e ** ((-1 *((x - (size - 1) / 2) ** 2 +
(y - (size - 1) / 2) ** 2)) / (2 * (sigma ** 2))) , (size,
size))
kernel /= np.sum(kernel)  # 归一化
```

除此之外，还可使用 OpenCV 中的 getGaussianKernel()函数产生高斯核，该函数计算并返回高斯滤波器系数的ksize$\times1$矩阵：

$$G_i=\alpha\cdot e^{-\frac{\left(i-\frac{\text{ksize}-1}{2}\right)^2}{2\cdot\sigma^2}}$$

其中$i=0,...,\text{ksize}-1$，α为归一化比例因子，使得$\sum_i G_i=1$。将此ksize$\times1$矩阵

与其转置相乘，即可得到高斯核。代码如下：

```
kernel2 = cv2.getGaussianKernel(ksize=size, sigma=sigma)
kernel = kernel2 * kernel2.T
```

以上三种代码均可得到3×3高斯核为：

$$\begin{bmatrix} 0.0947 & 0.1183 & 0.0947 \\ 0.1183 & 0.1477 & 0.1183 \\ 0.0947 & 0.1183 & 0.0947 \end{bmatrix}$$

三 题目 3

3.1 题目

利用高通滤波器滤波测试图像 test3 和 test4，包括 unsharp masking、Sobel edge detector、Laplace edge detection 和 Canny algorithm，并分析各自优缺点。图片如下。

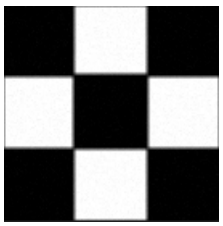

图16　test3

图17　test4

3.2 解答

3.2.1 Unsharp Masking

反锐化掩模（Unsharp Masking）是一种图像锐化技术，常用于数字图像处理软件。该技术使用模糊或者不锐化的负片图像来创建原始图像的遮罩。然后将反锐化遮罩与原始正片图像结合，生成比原始图像模糊的图像。

在 Unsharp Masking 中，图像清晰的细节被识别为原始（original）图像和模糊（blurred）版本之间的差异。然后缩放这些细节，并将其添加回原始图像，具体公式为：

$$\text{enhanced image} = \text{original} + \text{amount} \times (\text{original} - \text{blurred})$$

模糊步骤可以使用任何图像滤波方法，例如中值滤波，但传统上使用高斯滤波器。反锐化掩蔽滤波器中的半径（radius）参数指的是高斯滤波器的σ参数。

我们使用 Python 的图像处理库 scikit-image 中的 unsharp_mask 模块对图像进行处理，我们将 radius 参数设置为 3，将 amount 参数设置为 1，可得到如下结果。

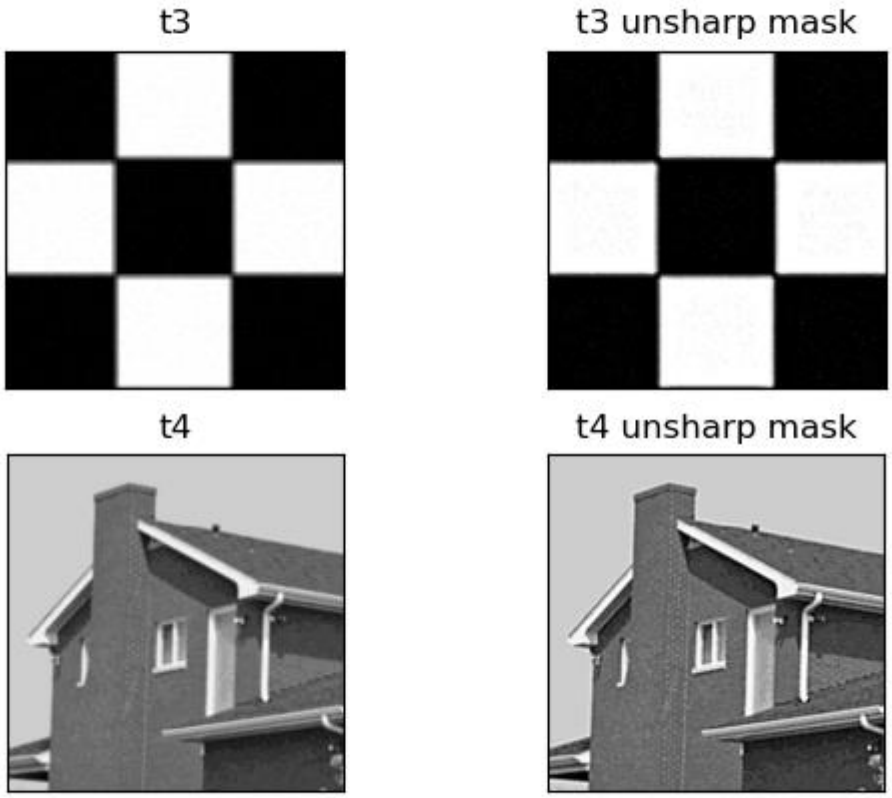

图18　test3 和 test4 的 Unsharp Masking 结果

Unsharp Masking 算法可以使得图像的边缘轮廓和高频成分增强。

反锐化掩蔽是一种简单的线性图像操作——由狄拉克增量减去高斯模糊核的核进行卷积。在不知道获取图像的方式的情况下，反锐化掩模会增加图像的明显清晰度。

通过反褶积，可以大致恢复“丢失”的图像细节，尽管通常无法验证任何恢复的细节是否准确。从统计学上讲，经过锐化的图像与正在成像的实际场景之间可以达到某种程度的对应。如果将来要捕捉的场景与经过验证的图像场景足够相似，则可以评估恢复的细节的准确程度。

3.2.2 Sobel Edge Detection

Sobel 算子（Sobel 滤波器），常用于图像处理的边缘检测算法中，它创建强调边缘的图像。它是一种离散微分算子，计算图像强度函数梯度的近似值。在图像中的每个点上，Sobel 算子的结果要么是相应的梯度向量，要么是该向量的范数。Sobel 算子基于在水平和垂直方向上用一个小的、可分离的整数值滤波器卷积图像，因此计算成本相对较低。另一方面，它产生的梯度近似相对粗糙，尤其是对于图像中的高频变化。

该算子使用两个3×3内核与原始图像卷积计算导数的近似值，一个用于水平变化，一个用于垂直变化。如果我们将$\mathbf{A}$定义为源图像，并将$\mathbf{G}_x$和$\mathbf{G}_y$定义为两个图像，这两个图像在每个点上分别包含水平导数和垂直导数近似，则计算如下：

$$\mathbf{G}_x=\begin{bmatrix}-1 & 0 & +1\\ -2 & 0 & +2\\ -1 & 0 & +1\end{bmatrix}*\mathbf{A}\ ,\quad \mathbf{G}_y=\begin{bmatrix}+1 & +2 & +1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix}*\mathbf{A}$$

其中$*$代表二维信号处理的卷积运算。

由于 Sobel 核可以分解为平均核和微分核的乘积，因此它们通过平滑计算梯度。例如$\mathbf{G}_x$可以被表示为

$$\mathbf{G}_x=\begin{bmatrix}1\\ 2\\ 1\end{bmatrix}*\left(\begin{bmatrix}-1 & 0 & +1\end{bmatrix}*\mathbf{A}\right)\quad ,\quad \mathbf{G}_y=\begin{bmatrix}+1\\ 0\\ -1\end{bmatrix}*\left(\begin{bmatrix}1 & 2 & 1\end{bmatrix}*\mathbf{A}\right)$$

此处，x 坐标定义为右方向上的增量，y 坐标定义为下方向上的增量。在图像中的每个点上，可以使用以下方法组合得到的梯度近似值，以给出梯度大小：

$$\mathbf{G}=\sqrt{\mathbf{G}_x{}^2+\mathbf{G}_y{}^2}$$

也可以使用更为简单的公式：

$$\mathbf{G}=|\mathbf{G}_x|+|\mathbf{G}_y|$$

利用这些信息，我们还可以计算梯度的方向：

$$\mathbf{\Theta}=\operatorname{atan}2(\mathbf{G}_y,\mathbf{G}_x)$$

在 OpenCV 中，我们使用 Sobel()函数分别计算 x 和 y 方向的 Sobel 算子，得出结果如下。

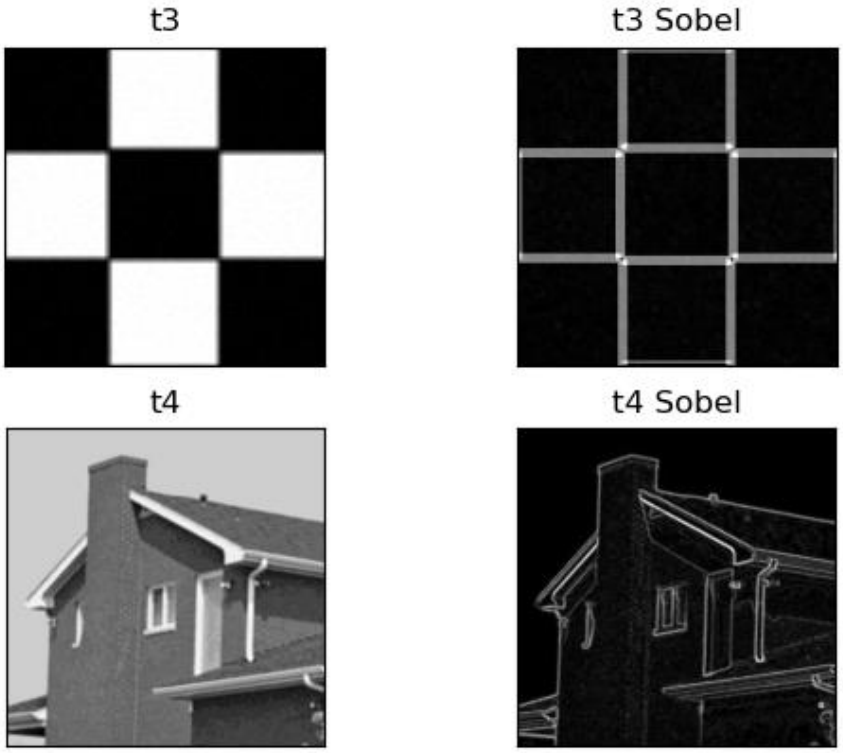

图19　test3 和 test4 的 Sobel Edge Detection 结果

综上，可以应用 Sobel 算子从图像中检测出水平和垂直边缘。此外，Sobel 运算符比其他边缘检测方法找到更多边，或者使边更为可见。这是因为在 Sobel 算子中，我们为边缘周围的像素强度分配了更多权重。

3.2.3 Laplacian Edge Detection

拉普拉斯边缘提取的原理是对二维函数求二阶导。对二元函数，其拉普拉斯变换为

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

对于图像的离散函数，其二阶差分可表示为

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

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

在 OpenCV 中，我们使用 Laplacian()函数进行边缘提取，该函数通过将使用 Sobel 运算符计算的二阶 x 和 y 偏导数相加，来计算源图像的拉普拉斯函数：

$$\mathrm{dst} = \Delta \mathrm{src} = \frac{\partial^2 \mathrm{src}}{\partial x^2} + \frac{\partial^2 \mathrm{src}}{\partial y^2}$$

注意到当 kernel 的尺寸为3×3时，其拉普拉斯算子为

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

拉普拉斯边缘检测的结果如下所示，我们将 dsize 设为 3。

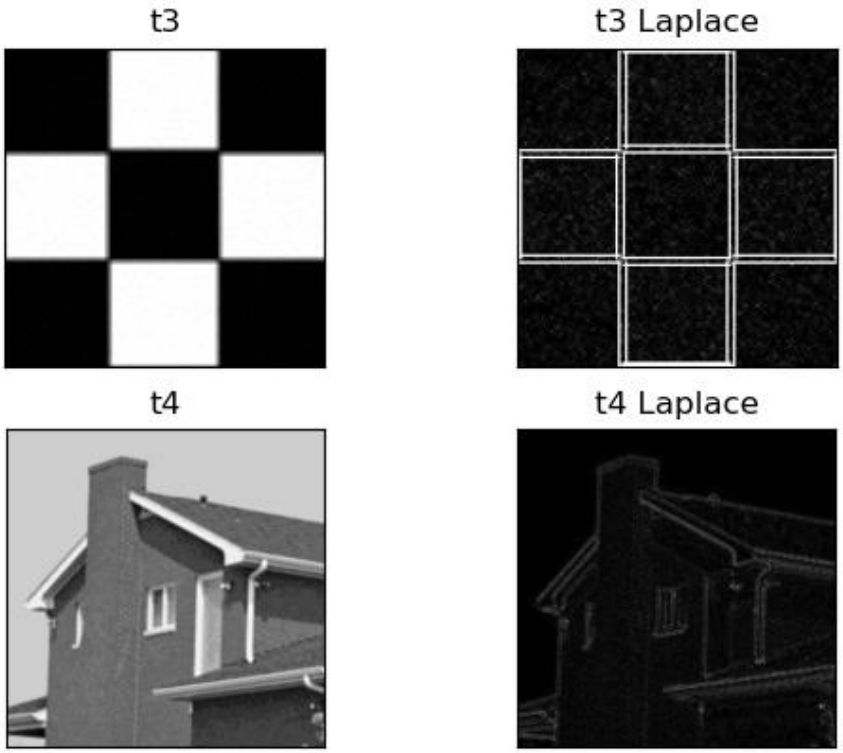

图20　test3 和 test4 的 Laplacian Edge Detection 结果

以上结果可以表明，采用拉普拉斯边缘检测算法，可以当图像的 salt-and-pepper noise 较少时有较好的边缘检测作用，且其精度明显高于 Sobel 算子。但是 Laplacian 对于噪声过于敏感，在噪声干扰下容易得出 salt-and-pepper noise 较多的结果。

3.2.4 Canny Algorithm

Canny 边缘检测算法的过程可分为五个不同的步骤：

（1） 应用高斯滤波器平滑图像以去除噪声；

（2） 找到图像的灰度梯度；

（3） 采用梯度幅度阈值或下限截止抑制来消除边缘检测的杂散响应；

（4） 应用双阈值来确定潜在的边缘；

（5） 通过滞后跟踪边缘，即通过抑制所有其他弱边缘和未连接到强边缘的边缘，完成边缘检测。

由于所有的边缘检测结果都很容易受到图像中噪声的影响，因此有必要滤除噪声，以防止由噪声引起的误检测。为了平滑图像，将高斯滤波器核与图像卷积。这一步将略微平滑图像，以减少明显噪声对边缘检测器的影响。大小为$(2k+1)\times(2k+1)$的高斯滤波器核的方程如下所示：

$$H_{ij}=\frac{1}{2\pi\sigma^2}e^{-\frac{(i-(k+1))^2+(j-(k+1))^2}{2\sigma^2}};\ 1\le i,j\le(2k+1)$$

图像中的边缘可能指向不同的方向，因此 Canny 算法使用四个过滤器来检测模糊图像中的水平、垂直和对角边缘。边缘检测操作符返回水平方向$\mathbf{G}_x$和垂直方向$\mathbf{G}_y$的一阶导数值。由此可确定边缘梯度和方向：

$$\mathbf{G}=\sqrt{\mathbf{G}_x{}^2+\mathbf{G}_y{}^2}$$
$$\mathbf{\Theta}=\operatorname{atan2}(\mathbf{G}_y,\mathbf{G}_x)$$

其中，atan2 是具有两个参数的反正切函数。边缘方向角四舍五入为代表垂直、水平和两条对角线$(0°,45°,90°,135°)$的四个角之一。每个颜色区域中的边缘方向将设置为特定角度值。

在减少噪声并计算强度梯度后，这一步中的算法使用一种称为非最大边缘抑制（non-maximum suppression of edges）的技术来过滤掉不需要的像素（实际上可能并不构成边缘）。为了实现这一点，将每个像素与其相邻像素在正梯度和负梯度方向上进行比较。如果当前像素的梯度大小大于其相邻像素，则保持不变。否则，当前像素的大小将设置为零。

在 Canny 边缘检测的最后一步中，将梯度大小与两个阈值进行比较，一个阈值小于另一个阈值。

如果“渐变幅值”值高于较大的阈值，则这些像素与强边缘关联，并包含在最终的

边缘贴图中。如果梯度幅值小于较小的阈值，则会抑制像素，并从最终的边缘贴图中排除。梯度大小介于这两个阈值之间的所有其他像素被标记为“弱”边缘（即它们成为最终边缘映射中的候选对象）。如果“弱”像素与那些与强边关联的像素相连，那么它们也会包含在最终的边缘映射中。

我们使用 OpenCV 的 Canny()函数，接收两个阈值分别为 100 和 200 进行计算。

采用 Canny 得到的边缘检测图像如下所示。

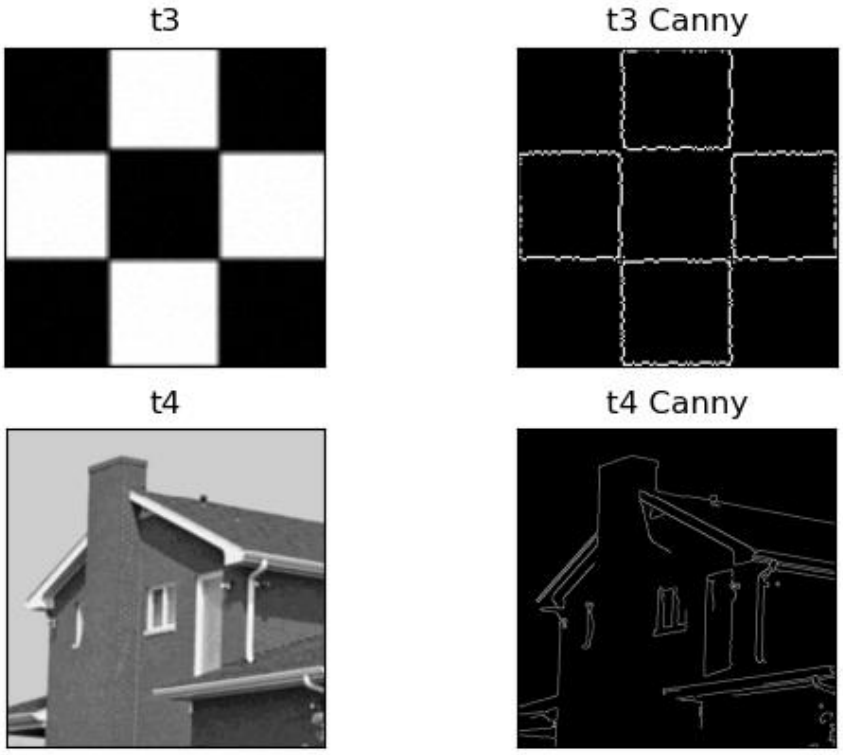

图21　test3 和 test4 的 Canny 结果

Canny 算法几乎不受噪音干扰，且得到的边缘精度很高，其边缘检测结果优于其他三种算法。但是 Canny 算法的计算较为复杂，步骤繁多。

四 附录：所有代码

注：以下代码在 Jupyter Notebook 中按顺序执行。

4.1 导入所需库

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import math
from skimage import io
from skimage.filters import unsharp_mask
from skimage.util import img_as_float
```

4.2 题目 1

```
img1 = cv2.imread("Resource/test1.pgm", cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread("Resource/test2.tif", cv2.IMREAD_GRAYSCALE)

# 高斯滤波
img1Gaussian3 = cv2.GaussianBlur(img1, ksize=(3, 3), sigmaX=1.5)
img1Gaussian5 = cv2.GaussianBlur(img1, ksize=(5, 5), sigmaX=1.5)
img1Gaussian7 = cv2.GaussianBlur(img1, ksize=(7, 7), sigmaX=1.5)

img2Gaussian3 = cv2.GaussianBlur(img2, ksize=(3, 3), sigmaX=1.5)
img2Gaussian5 = cv2.GaussianBlur(img2, ksize=(5, 5), sigmaX=1.5)
img2Gaussian7 = cv2.GaussianBlur(img2, ksize=(7, 7), sigmaX=1.5)

# 中值滤波
img1Median3 = cv2.medianBlur(img1, ksize=3)
img1Median5 = cv2.medianBlur(img1, ksize=5)
img1Median7 = cv2.medianBlur(img1, ksize=7)

img2Median3 = cv2.medianBlur(img2, ksize=3)
img2Median5 = cv2.medianBlur(img2, ksize=5)
img2Median7 = cv2.medianBlur(img2, ksize=7)

titles = ['t1 Gaussian 3x3', 't1 Gaussian 5x5', 't1 Gaussian 7x7',
```

```
          't2 Gaussian 3x3', 't2 Gaussian 5x5', 't2 Gaussian 7x7',
          't1 median 3x3', 't1 median 5x5', 't1 median 7x7',
          't2 median 3x3', 't2 median 5x5', 't2 median 7x7']
images = [img1Gaussian3, img1Gaussian5, img1Gaussian7,
          img2Gaussian3, img2Gaussian5, img2Gaussian7,
          img1Median3, img1Median5, img1Median7,
          img2Median3, img2Median5, img2Median7]

# 请使用 Python console（将代码粘贴至 Python 控制台中）
# 而非 jupyter notebook 来执行图形显示代码
# 因为 jupyter 输出过小，会导致图片显示有误
for i in range(12):
    plt.subplot(4, 3, i + 1), plt.imshow(images[i], 'gray')
    plt.title(titles[i])
    plt.xticks([]), plt.yticks([])

plt.show()

cv2.imwrite("Output/img1Gaussian3.jpg", img1Gaussian3)
cv2.imwrite("Output/img1Gaussian5.jpg", img1Gaussian5)
cv2.imwrite("Output/img1Gaussian7.jpg", img1Gaussian7)
cv2.imwrite("Output/img2Gaussian3.jpg", img2Gaussian3)
cv2.imwrite("Output/img2Gaussian5.jpg", img2Gaussian5)
cv2.imwrite("Output/img2Gaussian7.jpg", img2Gaussian7)
cv2.imwrite("Output/img1Median3.jpg", img1Median3)
cv2.imwrite("Output/img1Median5.jpg", img1Median5)
cv2.imwrite("Output/img1Median7.jpg", img1Median7)
cv2.imwrite("Output/img2Median3.jpg", img2Median3)
cv2.imwrite("Output/img2Median5.jpg", img2Median5)
cv2.imwrite("Output/img2Median7.jpg", img2Median7)
```

4.3 题目 2

```python
# 得到高斯核：方法一

sigma = 1.5
size = 3
kernel = np.fromfunction(lambda x, y: (1 / (2 * math.pi * (sigma ** 2))) *
math.e ** ((-1 *((x - (size - 1) / 2) ** 2 + (y - (size - 1) / 2) ** 2)) / (2 *
(sigma ** 2))), (size, size))
kernel /= np.sum(kernel)  # 归一化
```

```python
# 得到高斯核：方法二

kernel2 = cv2.getGaussianKernel(ksize=size, sigma=sigma)
kernel2 * kernel2.T
```

```python
# 得到高斯核：方法三

def myGaussianKernel(size, sigma):
    ax = np.linspace(-(size - 1) / 2., (size - 1) / 2., size)
    gauss = np.exp(-0.5 * np.square(ax) / np.square(sigma))
    my_kernel = np.outer(gauss, gauss)
    return my_kernel / np.sum(my_kernel)

myGaussianKernel(size, sigma)
```

4.4 题目 3

```
img3 = cv2.imread("Resource/test3_corrupt.pgm", cv2.IMREAD_GRAYSCALE)
img4 = cv2.imread("Resource/test4.tif", cv2.IMREAD_GRAYSCALE)

img31 = img_as_float(img3)
img41 = img_as_float(img4)

# unsharp masking
img3Unsharped = unsharp_mask(img31, radius=3, amount=1.0)
img4Unsharped = unsharp_mask(img41, radius=3, amount=1.0)

titles = ['t3', 't3 unsharp mask', 't4', 't4 unsharp mask']
images = [img3, img3Unsharped, img4, img4Unsharped]

# 请使用 Python console（将代码粘贴至 Python 控制台中）
# 而非 jupyter notebook 来执行图形显示代码
# 因为 jupyter 输出过小，会导致图片显示有误
for i in range(4):
    plt.subplot(2, 2, i + 1), plt.imshow(images[i], 'gray')
    plt.title(titles[i])
    plt.xticks([]), plt.yticks([])

plt.show()

io.imsave("Output/img3Unsharped.jpg", img3Unsharped)
io.imsave("Output/img4Unsharped.jpg", img4Unsharped)
```

```
img3SobelX = cv2.convertScaleAbs(cv2.Sobel(img3, ddepth=cv2.CV_64F, dx=1, dy=0))
img3SobelY = cv2.convertScaleAbs(cv2.Sobel(img3, ddepth=cv2.CV_64F, dx=0, dy=1))
# img3SobelCombined = cv2.bitwise_or(img3SobelX, img3SobelY)
img3SobelCombined = cv2.addWeighted(img3SobelX, 0.5, img3SobelY, 0.5, gamma=0)

img4SobelX = cv2.convertScaleAbs(cv2.Sobel(img4, ddepth=cv2.CV_64F, dx=1, dy=0))
img4SobelY = cv2.convertScaleAbs(cv2.Sobel(img4, ddepth=cv2.CV_64F, dx=0, dy=1))
# img4SobelCombined = cv2.bitwise_or(img4SobelX, img4SobelY)
img4SobelCombined = cv2.addWeighted(img4SobelX, 0.5, img4SobelY, 0.5, gamma=0)


titles = ['t3', 't3 Sobel', 't4', 't4 Sobel']
images = [img3, img3SobelCombined, img4, img4SobelCombined]
```

```
# 请使用 Python console（将代码粘贴至 Python 控制台中）
# 而非 jupyter notebook 来执行图形显示代码
# 因为 jupyter 输出过小，会导致图片显示有误
for i in range(4):
    plt.subplot(2, 2, i + 1), plt.imshow(images[i], 'gray')
    plt.title(titles[i])
    plt.xticks([]), plt.yticks([])

plt.show()

cv2.imwrite("Output/img3Sobel.jpg", img3SobelCombined)
cv2.imwrite("Output/img4Sobel.jpg", img4SobelCombined)
```

```
# 拉普拉斯边缘提取
img3Laplace = cv2.convertScaleAbs(cv2.Laplacian(img3, ddepth=cv2.CV_64F,
ksize=3))
img4Laplace = cv2.convertScaleAbs(cv2.Laplacian(img4, ddepth=cv2.CV_64F,
ksize=3))

titles = ['t3', 't3 Laplace', 't4', 't4 Laplace']
images = [img3, img3Laplace, img4, img4Laplace]

# 请使用 Python console（将代码粘贴至 Python 控制台中）
# 而非 jupyter notebook 来执行图形显示代码
# 因为 jupyter 输出过小，会导致图片显示有误
for i in range(4):
    plt.subplot(2, 2, i + 1), plt.imshow(images[i], 'gray')
    plt.title(titles[i])
    plt.xticks([]), plt.yticks([])

plt.show()

cv2.imwrite("Output/img3Laplace.jpg", img3Laplace)
cv2.imwrite("Output/img4Laplace.jpg", img4Laplace)
```

```
# Canny 边缘提取
img3Canny = cv2.Canny(img3, threshold1=100, threshold2=200)
img4Canny = cv2.Canny(img4, threshold1=100, threshold2=200)

titles = ['t3', 't3 Canny', 't4', 't4 Canny']
images = [img3, img3Canny, img4, img4Canny]
```

```
# 请使用 Python console（将代码粘贴至 Python 控制台中）
# 而非 jupyter notebook 来执行图形显示代码
# 因为 jupyter 输出过小，会导致图片显示有误
for i in range(4):
    plt.subplot(2, 2, i + 1), plt.imshow(images[i], 'gray')
    plt.title(titles[i])
    plt.xticks([]), plt.yticks([])

plt.show()

cv2.imwrite("Output/img3Canny.jpg", img3Canny)
cv2.imwrite("Output/img4Canny.jpg", img4Canny)
```