图像细化与骨架实践报告

蒋颜丞，自动化(电气)1903，3190102563

1 实验内容和要求

自选一幅灰度图像（最好是数字、文字图像，或具有细线条的图像），编程实现以下功能：

⑴ 选用一种图像二值化方法，实现图像二值化，提取前景目标；

⑵ 基于二值形态学滤波方法，二值化s结果进行形态学滤波，实现小尺度噪点去除和孔洞填充；

⑶ 实践一种图像细化算法，实现前景目标细化。

我将使用以下灰度图片完成本次实践作业：

可以看到该图片的背景灰度不均匀，文字周边带有很多颗粒噪声，且文字中间有孔洞，适合用本次作业的方法进行处理。

2 实验原理

2.1 Otsu法（大津法）

Otsu法是阈值化中常用的自动确定阈值的方法之一。Otsu法确定最佳阈值的准则是使阈值分割后各个像素类的类内方差最小。另一种确定阈值的准则是使得阈值分割后的像素类的类间方差最大。这两种准则是等价的，因为类间方差与类内方差之和即整幅图像的方差，是一个常数。分割的目的就是要使类别之间的差别最大，类内之间的差别最小。

设图像总像素数为N，灰度级总数为L，灰度值为i的像素数为N_i。令$\omega(k)$和$\mu(k)$分别表示从灰度级0到灰度级k的像素的出现概率和平均灰度，分别表示为：

$$w(k)=\sum_{i=0}^{k}\frac{N_i}{N}$$

$$\mu(k)=\sum_{i=0}^{k}\frac{i\cdot N_i}{N}$$

由此可见，所有像素的总概率为$\omega(L-1)=1$，图像的平均灰度为$\mu_T=\mu(L-1)$。

设有M-1个阈值$(0 \leq t_1 < t_2 < \ldots < t_{M-1} \leq L-1)$，将图像分成$M$个像素类$C_j$($C_j \in [\, t_{j-1}+1, \ldots, t_j]$；$j = 1, 2, \ldots, M$；$t_0 = 0$，$t_M = L-1$)，则$C_j$的出现概率$\omega_j$、平均灰度$\mu_j$和方差$\sigma_j^2$为

$$\omega_j = \omega(t_j) - \omega(t_{j-1})$$

$$\mu_j = \frac{\mu(t_j) - \mu(t_{j-1})}{\omega(t_j) - \omega(t_{j-1})}$$

$$\sigma_j^2 = \sum_{i=t_{j-1}+1}^{t_j} (i - \mu_j)^2 \frac{\omega(i)}{\omega_j}$$

由此可得类内方差为

$$\sigma_W^2(t_1, t_2, \ldots, t_{M-1}) = \sum_{j=1}^{M} \omega_j * \sigma_j^2$$

各类的类间方差为

$$\sigma_B^2(t_1, t_2, \ldots, t_{M-1}) = \sum_{j=1}^{M} \omega_j * (\mu_j - \mu_T)^2$$

将使$(*)$式最小或使$(\#)$式最大的阈值组$(t_1, t_2, \ldots, t_{M-1})$作为M阈值化的最佳阈值组。若取$M$为2，即分割成2类，则可用上述方法求出二值化的最佳阈值。

2.2 形态学滤波

2.2.1 腐蚀

腐蚀可以用集合的方式定义，即

$$X \ominus S = \{x \mid S + x \subseteq X\}$$

上式表明，X用S腐蚀的结果是所有使S平移x后仍在X中的x的集合。换句话说，用S来腐蚀X得到的集合是S完全包括在X中时S的原点位置的集合。

腐蚀的编程实现将利用上述思想：将腐蚀核（模板）遍历全图，如果腐蚀核全部处在前景中，则将此时模板的中心点置为1。遍历结束后，所有为1的点的集合即为腐蚀运算的结果。

腐蚀在数学形态学运算中的作用是消除物体边界点。如果结构元素取3×3的像素块，腐蚀将使物体的边界沿周边减少一个像素。腐蚀可以把小于结构元素的物体(毛刺、 小凸起)去除，这样选取不同大小的结构元素，就可以在原图像中去掉不同大小的物体。如果两个物体之间有细小的连通，那么当结构元素足够大时， 通过腐蚀运算可以将两个物体分开。

2.2.2 膨胀

腐蚀可以看作是将图像X中每一与结构元素S全等的子集$S + x$收缩为点x。反之，也可以将X中的每一个点x扩大为$S + x$，这就是膨胀运算，记为$X \oplus S$。若用集合语言，它的定义为

$$X \oplus S = \{S + x \cap X \neq \emptyset\}$$

上式表明，X用S膨胀的结果是所有使S平移x后有部分在X中的x的集合。换句话说，用S来腐蚀X得到的集合是S有部分包括在X中时S的原点位置的集合。

膨胀的编程实现将利用上述思想：将膨胀核（模板）遍历全图，如果膨胀核有部分与前景相交，则将此时模板的中心点置为1。遍历结束后，所有为1的点的集合即为膨胀运算的结果。

与腐蚀运算相反，膨胀在数学形态学运算中的作用是扩充物体边界点。如果结构元素取3×3的像素块，膨胀将使物体的边界沿周边增加一个像素。膨胀可以把小于结构元素的物体的孔洞及图像边缘处的小凹陷部分填充，这样选取不同大小的结构元素，就可以在原图像中填充不同大小的孔洞和凹陷。

2.2.3 开运算

先对图像进行腐蚀然后膨胀，称为开运算。对于图像X及结构元素S，用符号$X \circ S$表示S对图像X作开运算，定义为

$$X \circ S = (X \ominus S) \oplus S$$

2.2.4 闭运算

先对图像进行膨胀然后腐蚀，称为闭运算。对于图像X及结构元素S，用符号$X \bullet S$表示S对图像X作闭运算，定义为

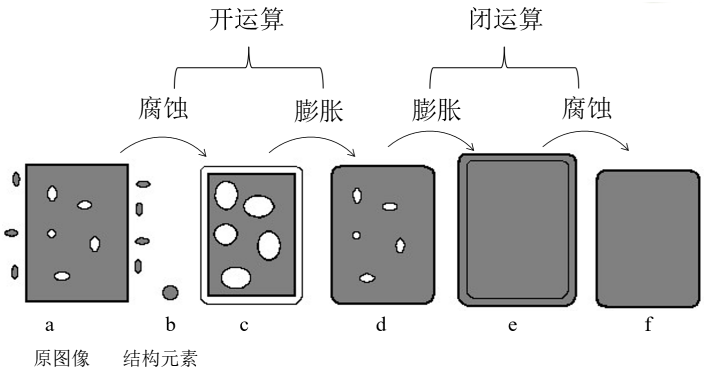
$$X \bullet S = (X \oplus S) \ominus S$$

2.2.5 形态学滤波

由于开、闭运算所处理的信息分别与图像的凸、凹处相关， 因此，它们本身都是单边算子，可以利用开、闭运算去除图像的噪声、恢复图像，也可交替使用开、闭运算以达到双边滤波目的。一般，可以将开、闭运算结合起来构成形态学噪声滤波器，例如$(X \circ S) \bullet S$或$(X \bullet S) \circ S$等。

下图给出消除噪声的一个图例。图中包括一个长方形的目标X，由于噪声的影响，在目标内部有一些噪声孔而在目标周围有一些噪声块。现在用如图(b)所示的结构元素S通过形态学操作来滤除噪声，这里的结构元素应当比所有的噪声孔和块都要大。先用S对X进行腐蚀得到图(c)，再用S对腐蚀结果进行膨胀得到图(d)，这两个操作的串行结合就是开运算，它将目标周围的噪声块消除掉了。再用S对图(d)进行一次膨胀得到图(e)，然后用S对膨胀结果进行腐蚀得到图(f)，这两个操作的串行结合就是闭运算，它将目标内部的噪声孔消除掉了。整个过程是先做开运算再做闭运算，可以写为

$$\{[(X \ominus S) \oplus S] \oplus S\} \ominus S = (X \circ S) \bullet S$$



形态学滤波示意图

2.3 图像细化

一个图像的“骨架”，是指图像中央的骨骼部分，是描述图像几何及拓扑性质的重要特征之一。求一幅图像骨架的过程就是对图像进行“细化”的过程。在文字识别、地质构造识别、工业零件形状识别或图像理解中，先对被处理的图像进行细化有助于突出形状特点和减少冗余信息量。在细化一幅图像X时应满足两个条件：第一，在细化的过程中，X应该有规律地缩小；第二，在X逐步缩小的过程中，应当使X的连通性质保持不变。下面介绍一个具体的细化算法：

设已知目标点标记为1，背景点标记为0。边界点是指本身标记为1而其8连通邻域中至少有一个标记为0的点。算法对一幅图像的所有边界点即一个3×3区域都进行如下检验和操作：

(1) 考虑以边界点为中心的8邻域，设p_1为中心点，对其邻域的8个点逆时针绕中心点分别标记为$p_2, p_3, \ldots, p_9$，其中p_2位于p_1的上方。如果$p_1 = 1$(即黑点)时，下面4个条件同时满足，则标记p_1为删除点(最后统一删除)：

① $2 \leq N(p_1) \leq 6$，其中$N(p_1)$是p_1的非零邻点的个数；

② $S(p_1) = 1$，其中$S(p_1)$是以$p_2, p_3, p_4, \ldots, p_9$为序时这些点的值从0到1变化的次数；

③ $p_2 p_6 p_8 = 0$；

④ $p_4 p_6 p_8 = 0$。

(2) 同第(1)步，仅将③中的条件改为$p_2 p_4 p_6 = 0$，④中的条件改为$p_2 p_4 p_8 = 0$。同样当对所有边界点都检验完毕后，将所有满足条件的点删除。

以上两步操作构成一次迭代。算法反复迭代，直至没有点再满足标记删除的条件，这时剩下的点就组成区域的骨架。

3 源代码

otsu.py（otsu法二值化）

```python
import numpy as np

def otsu(img, GrayScale):
    '''
    otsu法确定二值化阈值
    输入：图像，灰度级
    输出：二值化阈值
    '''
    pixel_sum=img.shape[0]*img.shape[1] # 总像素数目初始化
    hist=np.zeros(GrayScale) # 各个灰度值像素数目初始化

    w=np.zeros(GrayScale) # 出现概率值函数w(k)
    u=np.zeros(GrayScale) # 平均灰度值函数u(k)

    # 统计各个灰度值的像素个数
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            hist[img[i][j]] += 1 # 当前灰度值数目+1

```

```
    # 计算w(k)
    for i in range(GrayScale):
        w[i]=np.sum(hist[:i])*1.0/pixel_sum

    # 计算u(k)
    for i in range(GrayScale):
        sum_temp = 0
        for j in range(i+1):
            sum_temp += hist[j]*j
        u[i]=sum_temp*1.0/pixel_sum

    # 确定最大类间方差对应的阈值
    Max_var = 0 # 最大类间方差初始化
    for thi in range(1, GrayScale): # 遍历每一个阈值
        w1=w[thi]-w[0] # 前景像素的比例
        w2=w[-1]-w[thi] # 背景像素的比例

        if w1 != 0 and w2 != 0: # 确保分母不为0
            u1 = (u[thi]-u[0]) * 1.0 / w1 # 前景像素的平均灰度值
            u2 = (u[-1]-u[thi])* 1.0 / w2 # 背景像素的平均灰度值
            tem_var=w1*np.power((u1-u[-1]),2)+w2*np.power((u2-u[-1]),2) # 当前类间方差
            if Max_var<tem_var: # 判断当前类间方差是否为最大值
                Max_var=tem_var # 更新最大值
                th=thi # 更新阈值
    return th

def threshold(img, th):
    '''
    图像二值化
    输入：灰度图像，阈值
    输出：二值化后的图像
    '''
    th_img = np.zeros(img.shape, dtype=np.uint8) # 二值化图像初始化
    th_img[img>=th] = 255 # 将大于阈值的像素置为255
    th_img[img<th] = 0 # 将小于阈值的像素置为0
    return th_img
```

dilate_and_erode.py（形态学滤波，腐蚀与膨胀）

```
import numpy as np

def dilate(img, dilate_time=1):
    '''
    对图像进行腐蚀
    输入：二值化图像，腐蚀次数
    输出：腐蚀后的图像
    '''
    h, w = img.shape # 获取图像的高和宽
```

```
    kernal = np.array(((0, 1, 0),(1, 0, 1),(0, 1, 0)), dtype=int) # 创建腐蚀核

    out = img.copy() # 创建输出图像
    for i in range(dilate_time):
        tmp = out.copy()
        for x in range(1, h-1):
            for y in range(1, w-1):
                if np.sum(kernal * tmp[x-1:x+2, y-1:y+2]) >= 255: # 至少有一个像素点为背景
                    out[x, y] = 255 # 将中心点置为背景
    return out

def erode(img, erode_time=1):
    '''
    对图像进行膨胀
    输入：二值化图像，膨胀次数
    输出：膨胀后的图像
    '''
    h, w = img.shape # 获取图像的高和宽
    kernal = np.array(((0, 1, 0),(1, 0, 1),(0, 1, 0)), dtype=int) # 创建膨胀核

    out = img.copy() # 创建输出图像
    for i in range(erode_time):
        tmp = out.copy()
        for x in range(1, h-1):
            for y in range(1, w-1):
                if np.sum(kernal * tmp[x-1:x+2, y-1:y+2]) < 255*4: # 至少有一个像素点为前景
                    out[x, y] = 0 # 将中心点置为前景
    return out
```

thin.py（图像细化）

```
import numpy as np

def img_thin(img):
    '''
    图像细化，提取骨架
    输入：二值化图像
    输出：细化后的图像
    '''
    h, w = img.shape # 获取图像的高和宽

    out = np.zeros((h, w), dtype=int) # 初始化输出图像
    out[img > 0] = 1 # 将二值化图像转换为1和0
    out = 1-out # 取反

```

```
    while True:
        delet_node1 = []
        delet_node2 = [] # 创建删除点list

        # step 1
        for x in range(1, h-1):
            for y in range(1, w-1):
                if out[x, y] == 1: # 如果是前景点
                    num_of_one = np.sum(out[x-1:x+2, y-1:y+2])-1 # p1的非零邻点个数
                    if num_of_one >= 2 and num_of_one <= 6: # 如果p1的非零邻点个数在2到6之间
                        if count_0_to_1(out, x, y) == 1: # 若从0到1的变化次数为1
                            if out[x-1, y]*out[x+1, y]*out[x, y+1] == 0 and out[x, y-1]*out[x+1, y]*out[x, y+1] == 0:
                                delet_node1.append((x, y)) # 将当前点加入删除点list

        for node in delet_node1: # 对删除点list进行遍历
            out[node] = 0 # 将删除点设置为0

        # step 2
        for x in range(1, h-1):
            for y in range(1, w-1):
                if out[x, y] == 1: # 如果是前景点
                    num_of_one = np.sum(out[x-1:x+2, y-1:y+2])-1 # p1的非零邻点个数
                    if num_of_one >= 2 and num_of_one <= 6: # 如果p1的非零邻点个数在2到6之间
                        if count_0_to_1(out, x, y) == 1: # 若从0到1的变化次数为1
                            if out[x-1, y]*out[x, y-1]*out[x+1, y] == 0 and out[x-1, y]*out[x, y-1]*out[x, y+1] == 0:
                                delet_node2.append((x, y)) # 将当前点加入删除点list

        for node in delet_node2: # 对删除点list进行遍历
            out[node] = 0 # 将删除点设置为0

        if len(delet_node1) ==0 and len(delet_node2) == 0: # 如果没有点再满足标记删除的条件，则退出循环
            break

    out = 1 - out # 取反
    out = out.astype(np.uint8) * 255 # 转换为0和255

    return out

```

```
def count_0_to_1(img, x, y):
    '''
    计算从0到1的变化次数
    输入：二值化图像，坐标
    输出：从0到1的变化次数
    '''
    num = 0
    if (img[x-1, y-1] - img[x-1, y]) == 1: # p2到p3
        num += 1
    if (img[x, y-1] - img[x-1, y-1]) == 1: # p3到p4
        num += 1
    if (img[x+1, y-1] - img[x, y-1]) == 1: # p4到p5
        num += 1
    if (img[x+1, y] - img[x+1, y-1]) == 1: # p5到p6
        num += 1
    if (img[x+1, y+1] - img[x+1, y]) == 1: # p6到p7
        num += 1
    if (img[x, y+1] - img[x+1, y+1]) == 1: # p7到p8
        num += 1
    if (img[x-1, y+1] - img[x, y+1]) == 1: # p8到p9
        num += 1
    if (img[x-1, y] - img[x-1, y+1]) == 1: # p9到p2
        num += 1
    return num
```

main.py（主函数）

```
import cv2
from otsu import otsu,threshold
from dilate_and_erode import dilate,erode
from thin import img_thin

if "__main__" == __name__:
    img = cv2.imread('siling3.png', 0) # 读入图片
    origin_img = img.copy() # 备份原图
    cv2.imshow("origin_img", origin_img) # 显示原图

    th = otsu(img, 256) # 使用otsu法确定阈值
    print("otsu法确定的阈值为" + str(th))
    th_img = threshold(img, th) # 使用阈值th进行二值化
    cv2.imshow("th_img", th_img) # 显示二值化图像
    cv2.imwrite("th_img.png", th_img) # 保存二值化图像

    # 开运算
    dilate_result = dilate(th_img, dilate_time=2) # 腐蚀
    erode_result = erode(dilate_result, erode_time=2) # 膨胀
    cv2.imshow("open_result", erode_result)
    cv2.imwrite("open_result.png", erode_result)

```

```
    # 闭运算
    erode_result = erode(erode_result, erode_time=4) # 膨胀
    dilate_result = dilate(erode_result, dilate_time=4) # 腐蚀
    cv2.imshow("close_result", dilate_result)
    cv2.imwrite("close_result.png", dilate_result)

    out = img_thin(dilate_result) # 图像细化
    cv2.imshow("thin_result", out)
    cv2.imwrite("thin_result.png", out)

    cv2.waitKey(0)
```

4 实验结果与分析

4.1 二值化结果与分析

使用Otsu法进行图像二值化的结果中，文字外围带有很多毛刺和小突起，同时，文字笔画内部带有孔洞，需要进行进一步的处理。

4.2 开运算结果与分析

本次实践的开运算采用先腐蚀2次、后膨胀2次实现。从结果图中可以看出，文字外围的毛刺和小突起被很好地消除。

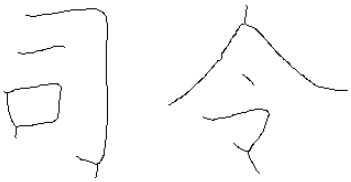
4.3 闭运算结果（形态学滤波结果）与分析

本次实践的闭运算采用先膨胀4次、后腐蚀4次实现。从结果图中可以看出，文字笔画内部的孔洞被很好地填充，已经具备进行图像细化的条件。

4.4 细化结果与分析

细化过程（如gif不能播放，请查看附件）：

细化结果：



从细化过程中可以看出，随着算法的迭代，文字均匀变细，同时保持连通关系不变。最终的结果也相当不错，反应出了文字的骨架和连通特征，还带有部分的字体特征，可以为后续的文字识别等工作带来很好的帮助。