Homework 2

邓文 17307130171

系统环境：Windows 10

编程语言：Python 3.7

1. **Restate the Basic Global Thresholding (BGT) algorithm so that it uses the histogram of an image instead of the image itself.**

解答:

- 计算图像的直方图$Hist$.
- 选取一个初始化的阈值 T_0 (例如：图像整体的平均强度。). 其中T_0需要满足 $T_0 \in (IntensityMin,\ IntensityMax)$， 即$T_0$在图像的像素值最大值和最小值之间。
- 对于$i = 1, 2, 3, \cdots, Max\ Iterations(最大迭代次数)$
  (1) 根据选取的阈值T_{i-1}，将直方图划分$Hist$为R_1和R_2两部分。R_1包含了图像上强度小于T_{i-1}的像素点，R_2则包含了图像上强度大于T_{i-1}的像素点。
  (2) 根据上面得到的划分R_1和R_2分别计算R_1和R_2两部分的平均强度μ_1和μ_2.
  方式如下:

$$\mu_1 = \sum_{i=0}^{T_{i-1}} i \cdot Hist(i) \Big/ \sum_{i=0}^{T_{i-1}} Hist(i)$$

$$\mu_2 = \sum_{i=T_{i-1}}^{L-1} i \cdot Hist(i) \Big/ \sum_{i=T_{i-1}}^{L-1} Hist(i)$$

  其中$Hist(i)$代表直方图中像素强度为i的高度，也反应了图像中像素强度为i的像素个数。

  (3) 更新新的阈值，即令$T_i = \frac{(\mu_1+\mu_2)}{2}$.

  (4) 进行循环终止检查:

  如果$|T_i - T_{i-1}| < \varepsilon\ (e.g.\ \varepsilon = \frac{IntensityMax-IntensityMin}{1000})$，循环终止，得到阈值 $T_{final} = T_i$.

- 对于图像中的每一个点，进行阈值化操作，即:

$$g(x,y) = \begin{cases} 1, & f(x,y) \geq T_{final} \\ 0, & f(x,y) < T_{final} \end{cases}$$

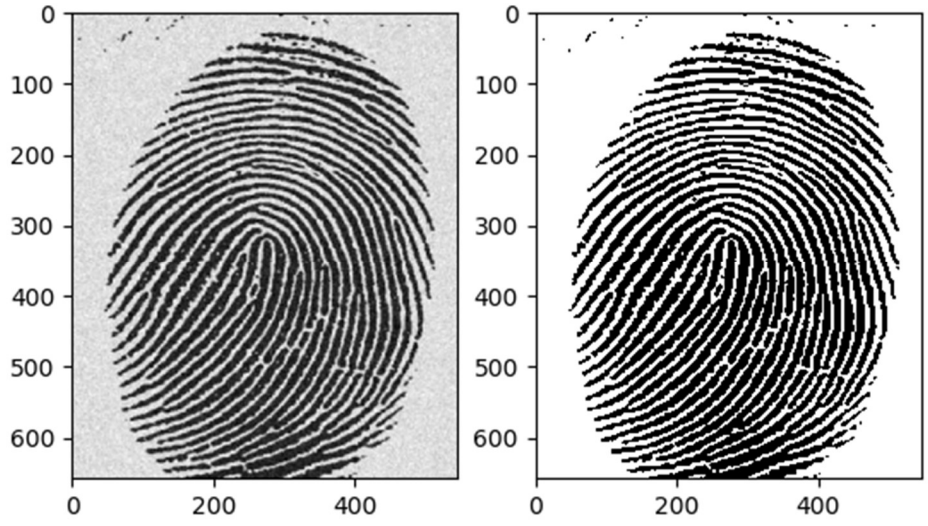
- 输出最终图像的结果。

另外，我用 python 实现了这个算法， 代码就不多做解释，如下:

```
def basic_global_thrsh(img):
    """
    compute the threshold based on bgt algorithm
    :param img:  input image
    :return:  threshold
    """
    x_len, y_len = img.shape  # the shape of image
    N = x_len * y_len  # the total number of pixel

    # compute the histogram
    pixel_sum = 0
    hist = np.zeros(256)
```

```
    for i in range(x_len):
        for j in range(y_len):
            pixel_sum += img[i][j]
            hist[img[i][j]] += 1

    # initial the threshold equal to mean intensity
    old_t = pixel_sum/N

    while True:
        # modify
        t = int(old_t) + 1

        n_bk = np.sum(hist[:t])  # the number of pixels in background
        n_fg = np.sum(hist[t:])  # the number of pixels in foreground

        u_bk = np.sum(hist[:t] * np.arange(t)) / n_bk  # mean intensity of background
        u_fg = np.sum(hist[t:] * np.arange(t, 256)) / n_fg  # mean intensity of foreground

        # update
        new_t = (u_bk + u_fg) / 2

        # terminal condition
        if abs(new_t - old_t) < (img.max() - img.min())/1000:
            break

        old_t = new_t

    return old_t
```

测试：



Original　　BGT

可以看到，在这个图像上，算法的二值化效果还是不错的。第一题完成！

2. **Give an explanation why the initial threshold in the Basic Global Thresholding (BGT) algorithm must be between the minimum and maximum values in the image.**

解答：

可以举一个简单的反例，假如一个图像像素强度为在区间$[100, 200]$的呈均值为 150 的均匀分布，即所有的像素都位于这个区间内， 这时候我们选择一个初始的阈值 $T_0 = 80 < 100 = minimum\ values$，这时形成的划分，所有的点都在$R_2$ 中，且均值就为 150，

而R_1中并没有点，均值为 0， 因此新的$T_1 = \frac{150+0}{2} = 75$，这时继续执行循环，仍然会得到同样的$R_1$和$R_2$的划分， T_i的值并不会随着迭代次数的增长而收敛。这时 Basic Global Thresholding 算法就失效了。

一句话概括就是，当选定的初始阈值在图像像素的最小值和最大值之外时，初始划分会导致所有的点都在其中一个划分，而当全局图像像素均值 $\frac{\mu_{global}}{2} \notin (minimum,\ maximum)$时，会导致这样的情况永远持续下去，阈值不会收敛。

因此，BGT 算法初始阈值的选择必须在图像像素的最大值和最小值之间。

3. **Design an algorithm with the function of locally adaptive thresholding (e.g. based on moving average or local OSTU); implement the algorithm and test it on exemplar image(s).**

解答:

这道题我分别用 moving average 和 local OSTU 的方法实现了局部自适应阈值化的算法。首先，定义了一个函数 board_detection() 的函数来确定邻域的边界，这个函数与第一次作业定义的 board_detection() 一模一样，就是根据图片和当前邻域的中心以及邻域窗口大小来确定领域的上下左右边界，具体就不解释了:

```
def board_detection(data, pos, size):
    # this is a function which will return the coordinates of window corner
    x_length, y_length = data.shape
    px, py = pos
    r = int(size / 2)
    ex = px + r + 1
    ey = py + r + 1
    while ex > x_length:
        ex -= 1
    while ey > y_length:
        ey -= 1
    l = size - r - 1
    sx = px - l
    sy = py - l
    while sx < 0:
        sx += 1
    while sy < 0:
        sy += 1
    return sx, ex, sy, ey
```

然后第一步先实现基于 移动平均 的 局部自适应阈值化:

1. **adpt_thrsh_MovAvg(ori_img, window_size, c=1.0)**

参数：

- ori_img: 输入的原始图像， 以 numpy.array 的形式储存。

- window_size: 指定邻域的窗口大小，为大于 0 的整数
- c: 关于调整阈值的超参，与算出的移动平均值相乘，来进行调整。
- return: 返回阈值化后的图片，同样以 np.array 的形式储存。

具体实现：

```
img = ori_img.copy().astype(int)
x_len, y_len = img.shape  # the shape of image
new_img = img.copy()  # copy a new variance to store the new_img

# initialize
cur_left, cur_right, cur_up, cur_down = board_detection(img, (0, 0), window_size)
neighborhood_num = (cur_right-cur_left)*(cur_down-cur_up)
neighborhood_sum = np.sum(img[cur_left:cur_right, cur_up:cur_down])
m = neighborhood_sum * c / neighborhood_num
new_img[0][0] = 1 if img[0][0] >= m else 0
```

首先是初始化操作:

- 创建一个 img 变量作为原始图像的副本，并将格式转换为 int 型，这是因为读入的图片可能是 uint8 的类型，这样在后面进行移动邻域更新时，可能会出现 overflow 的现象。
- 然后求出图片的长度 x_len 和宽度 y_len，并定义一个新的变量 new_img 用来保存新的图片。
- 由于第一次计算邻域不能用高效计算方法，所以需要初始化计算邻域的边界，当前领域的像素点的数量以及当前领域像素强度的和，然后根据 $g(x,y) = m(x,y) * c = neighborhood\_sum*c/neighborhood\_num$ 计算出当前的邻域的阈值，并更新 new_img 中当前的像素点。

这样有了变量 neighborhood_sum 以及 neighborhood_num 我们就可以接下来高效地对邻域的移动平均值进行更新了!

```
    # run a loop to update all position
    direction = (0, 1)  # initial direction : right
    i, j = 0, 0
    while True:
        # move the neighborhood
        i, j = i + direction[0], j + direction[1]
        # compute the new boarder
        boarder = board_detection(img, (i, j), window_size)
        # update the neighborhood number
        neighborhood_num = (boarder[1]-boarder[0])*(boarder[3]-boarder[2])
        if direction == (0, 1):  # move right
            add = np.sum(img[boarder[0]:boarder[1], cur_down:boarder[3]])
            delete = np.sum(img[boarder[0]:boarder[1], cur_up:boarder[2]])
            neighborhood_sum = neighborhood_sum + add - delete
        if direction == (1, 0):  # move down
            add = np.sum(img[cur_right:boarder[1], boarder[2]:boarder[3]])
            delete = np.sum(img[cur_left:boarder[0], boarder[2]:boarder[3]])
            neighborhood_sum = neighborhood_sum + add - delete
        if direction == (0, -1):  # move left
```

```
            add = np.sum(img[boarder[0]:boarder[1], boarder[2]:cur_up])
            delete = np.sum(img[boarder[0]:boarder[1], boarder[3]:cur_down])
            neighborhood_sum = neighborhood_sum + add - delete
        # update the direction of last step
        cur_left, cur_right, cur_up, cur_down = boarder
        # calculate current position's threshold
        m = neighborhood_sum * c / neighborhood_num
        # update the pixel
        new_img[i][j] = 1 if img[i][j] >= m else 0
```

上面的代码是每个领域的更新流程:

- 这次的邻域移动与第一次作业不同，我使用了 Z 字型的移动，即每次都向右移动到底，然后再向下移动一格，再向左移动，……，直到终止。为了能够支持这种移动，我定义了一个 direction 变量，可以取值为 (0, 1)， (0, -1), (1, 0) 分别代表向右，向左，向下移动。初始化进入循环前，我们定义方向为 (0, 1)，位置(i, j) 为 (0, 0).
- 每次循环的时候，首先邻域朝 direction 指定的方向移动一步，然后就需要计算一个新的边界保存在 boarder 中，根据 boarder 的边界计算新的邻域中点的总数。
- 由于邻域在移动的过程中，为了能够高效地计算，我们不会每次都重新计算每个领域的像素和和均值，而是通过删掉邻域移动后没覆盖的点的像素值，再加上邻域移动后新覆盖的点的像素值，来进行快速的迭代计算。
- 需要增加的像素点的值用 add 变量表示，需要删除的像素点的值用 delete 表示，在不同的方向下，这个两个变量可以根据上一步的邻域边界和这一步的邻域边界来对图像 img 进行切片求和得到。 然后新的 neighborhood_sum = neighborhood_sum + add – delete.
- 得到新的 neighborhood_sum 后，就可以根据之前的公式 g(x,y) = m(x,y) * c = neighborhood_sum*c/neighborhood_num 计算出当前的邻域的阈值，并更新 new_img 中当前的像素点。此外还需要用 boarder 变量来更新上一步的边界值。

```
    # this is a terminal condition
    if (j == y_len - 1 and i == x_len - 1 and x_len % 2 == 1) or (j == 0 and i == x_len - 1 and x_len % 2 == 0):
        # finish and exit
        break
    # direction judgement to support Z type move
    elif (j == y_len - 1 and i % 2 == 0) or (j == 0 and i % 2 == 1):  # reach to the end of a row of beginning
        direction = (1, 0)
    elif j == y_len - 1 and i % 2 == 1:  # after switching to next row change direction
        direction = (0, -1)
    elif j == 0 and i % 2 == 0:  # after switching to next row change direction
        direction = (0, 1)

return new_img
```

- 循环的最后一步，需要进行边界判断，当图像的行数为偶数时，邻域中心(i, j)移动到左下角就截止了，当图像的行数为奇数时，邻域中心(i, j)移动到右下角更新才会结束。
- 同时，为了能够支持 Z 字形的移动，当图像到达奇数行最左端或者偶数行最右端时，方向会变为 (1, 0) 即向下移动。
- 当图像到达偶数行最左端时，方向会变为(0, 1)即向右移动， 同理，当图像到达奇数行最右端时，方向会变为(0, -1)即向左移动

至此，基于移动平均的局部自适应阈值算法已经完成，只需要输出结果就好了。

接下来完成基于 OSTU 的局部自适应阈值化

2. 基于 OSTU 的局部自适应阈值化

首先， 我定义了一个基于直方图的全局的 OSTU 算法来计算阈值，希望能够到时候将图片的邻域切片取出来直接输入这个算法中进行计算。

```
def OTSU_threshold(img):
    """
    Use OTSU algorithm to calculate the threshold of an img
    :param img:  the input image
    :return:  a threshold
    """
    x_len, y_len = img.shape  # the shape of image
    N = x_len * y_len  # the total number of pixel

    # compute the histogram
    hist = np.zeros(256)
    for i in range(x_len):
        for j in range(y_len):
            hist[img[i][j]] += 1

    max_variance = 0  # to keep record of the maximum between-group variance
    threshold = 0  # to keep record of the threshold that maximize the between-group variance

    # run a loop to compute the best threshold
    for t in range(img.min()+1, img.max()):
        n_bk = np.sum(hist[:t])  # the number of pixels in background
        n_fg = np.sum(hist[t:])  # the number of pixels in foreground

        w_bk = n_bk / N  # freq of background
        w_fg = n_fg / N  # freq of foreground
        if n_bk == 0:
            u_bk = 0
        else:
            u_bk = np.sum(hist[:t] * np.arange(t)) / n_bk  # mean intensity of background
        if n_fg == 0:
            u_fg = 0
        else:
            u_fg = np.sum(hist[t:] * np.arange(t, 256)) / n_fg  # mean intensity of foreground

        var_bet = w_bk * w_fg * (u_fg - u_bk) ** 2  # between-group variance

        # to compare with the max_variance
        if var_bet > max_variance:
            max_variance = var_bet  # update variance
            threshold = t  # update threshold

    return threshold
```

以上代码即为基于直方图的 OSTU 算法：

- 首先利用两层的 for 循环计算出图像的直方图，并保存到变量 hist 中。
- 然后定义了 max_variance 和 threshold 用来存储目前为止最大的类间方差和对应的最佳阈值。
- 由于阈值必须在像素的最小值，和最大值之间，为了避免不必要的时间，我们将循环

的起始点和终止点设为了图像的像素最小值，和像素最大值。

- 然后对于之间的每个像素值 t，我们以它为界，对直方图进行划分，分别计算两个区域的类内均值和该类在图像所有像素点占的频率。循环中的 if 条件判断是为了防止 0 作为分母时导致的 overflow 的现象。
- 然后根据 $Var_{bet} = \omega_{bk}\omega_{fg}(\mu_{bk} - \mu_{fg})^2$，计算当前的类间方差，如果它大于目前记录的 max_variance，则将 max_variance 更新，并记录最佳的阈值 threshold 为当前的 t 值。
- 直到循环结束，返回最佳的阈值。

以上便是基于直方图的 OSTU 算法。但是在后面的实验中，对于一幅图像，可能有几十万个像素点甚至不止，如果对于每一个像素点，都运行上面的这样的 OSTU 算法，至少在对 t 值的遍历中，可能会重复计算很多值不同，但效果一样的 t 值，会导致跑完一次局部自适应阈值化花费很长的时间。因此，对于局部的 OSTU 算法，我想到了一种计算更加简洁的方法，因为邻域的设置一般都很小，大部分都是 3x3， 5x5，这样的小窗口，因此从小到大直接遍历每个像素点来进行划分，相对于遍历很多不可能的像素点 t 值要快速很多。
下面详细介绍新定义的 **local_OSTU** 的算法

```
def local_OTSU(img, c):
    """
    run OTSU on every neighborhood is expensive, a better idea is to go over every pixels in the local img
    :param img: input local img
    :param c: hyper parameter
    :return: local threshold
    """
    # initialize
    flat_img = img.flatten().astype(float)  # flatten operation to make later operation more convenience
    # sort the pixels
    flat_img.sort()
    # keep record of the maximum between-class variance
    max_variance = 0
    # keep record of the best threshold
    threshold = 0
    # the number of pixels in the background
    back_total = flat_img[0]
    # the number of pixels in the foreground
    fore_total = np.sum(flat_img[1:])
```

上面便是 local_OTSU 的算法，输入为两个参数 img：输入的图像 和 c: 用于调整阈值的超参。输出为一个阈值，能够对输入的 img 进行划分，使得类间方差最大。

- 首先初始化，将输入的 img 扁平化，这是为了方便切片操作，并转为 float 类型，防止后面计算时产生 overflow 的现象，保存在 flat_img 中。
- 然后对 flat_img 进行排序，这也是为了方便后面对前景和后景的统计。
- 与之前的 OTSU 相同，用 max_variance 和 threshold 来保存当前最大的类间方差和对应的最佳阈值。 用 back_total 来记录后景中的像素值的和， 初始化为第一个像素值的值，用 fore_total 来记录前景中像素值的和，初始化为后 n-1 像素点的值的和。这样做的目的于高效的局部直方图更新类似，是为了，每次在进行划分的移动时，都采用加上新像素点，或减去剔除的像素点的高效更新方法，避免每次都重新计算，浪费时间。

```
for i in range(1, flat_img.size):
    # to avoid repeat computation
    if flat_img[i] == flat_img[i-1]:
        back_total += flat_img[i]
        fore_total -= flat_img[i]
        continue
    # mean of background and foreground
    u_bk = back_total/i
    u_fg = fore_total/(flat_img.size-i)
    # freq of background and foreground
    w_bk = i/flat_img.size
    w_fg = 1-w_bk
    # between-class variance
    var_bet = w_fg * w_bk * (u_fg - u_bk) ** 2
    # to compare
    if var_bet > max_variance:
        # update
        threshold = flat_img[i]
        max_variance = var_bet
    # update the the number of pixels in the background and foreground
    back_total += flat_img[i]
    fore_total -= flat_img[i]
return threshold * c
```

- 然后进入循环，首先判断 i 位置的像素值是否与上一个位置的像素值相同，如果相同，就更新 back_total 和 fore_total 后直接进入下一次循环。
- 如果不相同，则代表这是一个新的划分。比较方便的是， i 就是后景的像素点的数量，flat_img.size – i 就是前景的像素点的数量，这样在计算前景均值和后景均值，只需要将我们记录的 fore_total 和 back_total 除以各自的像素点个数即可，前景和后景的像素点频率计算方法类似。
- 同样根据$Var_{bet} = \omega_{bk}\omega_{fg}(\mu_{bk} - \mu_{fg})^2$来计算类间方差，如果它大于 max_variance，则更新 max_variance 并记录下当前最优阈值 threshold 为 flat_img[i]的值。
- 循环结束后，输出 threshold 乘以 超参 c 后的值。

以上就是为了 基于 OSTU 的局部自适应阈值化设计的 局部 OSTU 算法。接下来完成主要的局部自适应阈值算法：

```
def adpt_thrsh_OTSU(img, window_size, c):
    """
    locally adaptive threshold based on OTSU
    :param img:  input image
    :param window_size:  window size
    :param c: hyper parameter
    :return: img
    """
    x_len, y_len = img.shape  # the shape of the img
    new_img = img.copy()  # to store the new img
```

```
    # begin of the loop
    for i in range(x_len):
        for j in range(y_len):
            # the boarder of the neighborhood
            boarder = board_detection(img, (i, j), window_size)
            # compute the local threshold
            local_threshold = local_OTSU(img[boarder[0]:boarder[1], boarder[2]:boarder[3]], c)
            # update
            new_img[i][j] = 1 if img[i][j] >= local_threshold else 0

    return new_img
```

参数:

- img: 输入的图像
- window_size: 指定的窗口大小
- c: 关于调整阈值的超参，与算出的移动平均值相乘，来进行调整。
- return: 输出阈值化后的图像

有了 local_OSTU()的实现，实现这个函数的步骤就很简单了，简单描述一下:

- 首先定义一个变量 new_image 来存储新的图像
- 然后对于图像的每一个像素点
  i. 计算它的邻域边界
  ii. 根据邻域边界将图像进行切片，并传递到之前定义好的 local_OSTU() 函数中，计算该点的阈值
  iii. 根据阈值，对 new_img 中对应的像素点进行更新
- 返回新的图像。

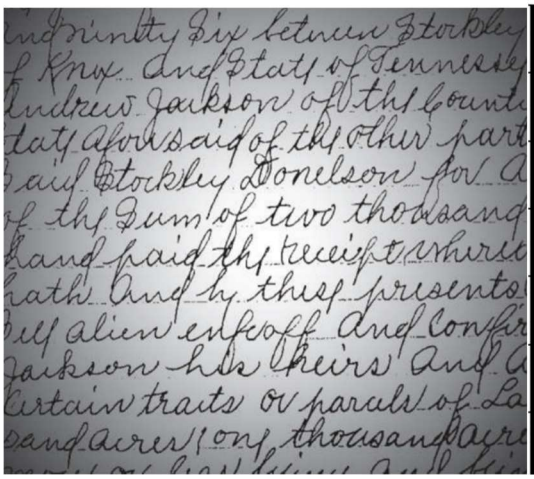
至此，关于第三题的所以函数编写已经完成!

测试：

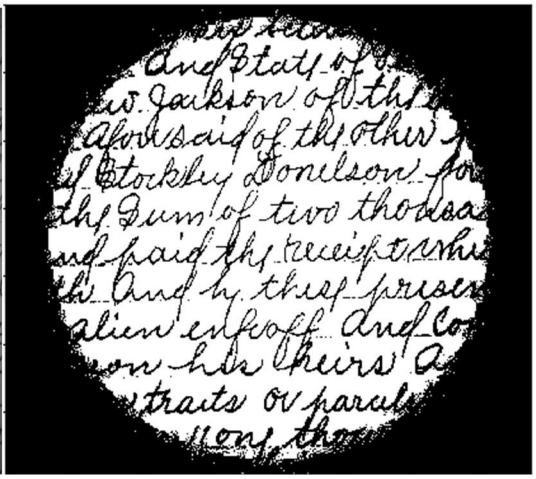
```
if __name__ == "__main__":
    from PIL import Image
    import matplotlib.pyplot as plt
    for case in ["test3.1.png", "test3.2.png"]:
        image = np.array(Image.open(case).convert("L"))
        threshold = OTSU_threshold(image)
        img2 = image.copy()
        img2[image >= threshold] = 1
        img2[image < threshold] = 0
        plt.clf()
        plt.imshow(img2, cmap=plt.get_cmap('gray'))
        plt.savefig("result" + case[4:8] + "1.png")
        plt.show()
        image3 = adpt_thrsh_MovAvg(image, 4, 0.9)
        plt.clf()
        plt.imshow(image3, cmap=plt.get_cmap('gray'))
        plt.savefig("result" + case[4:8] + "2.png")
        plt.show()
        image4 = adpt_thrsh_OTSU(image, 6, 0.9)
        plt.clf()
        plt.imshow(image4, cmap=plt.get_cmap('gray'))
        plt.savefig("result" + case[4:8] + "3.png")
        plt.show()
```

```
image = np.array(Image.open("test3.3.png").convert("L"))
threshold = OTSU_threshold(image)
img2 = image.copy()
img2[image >= threshold] = 1
img2[image < threshold] = 0
plt.clf()
plt.imshow(img2, cmap=plt.get_cmap('gray'))
plt.savefig("result3.3.1.png")
plt.show()
image3 = adpt_thrsh_MovAvg(image, 7, 0.95)
plt.clf()
plt.imshow(image3, cmap=plt.get_cmap('gray'))
plt.savefig("result3.3.2.png")
plt.show()
image4 = adpt_thrsh_OTSU(image, 6, 0.95)
plt.clf()
plt.imshow(image4, cmap=plt.get_cmap('gray'))
plt.savefig("result3.3.3.png")
plt.show()
```

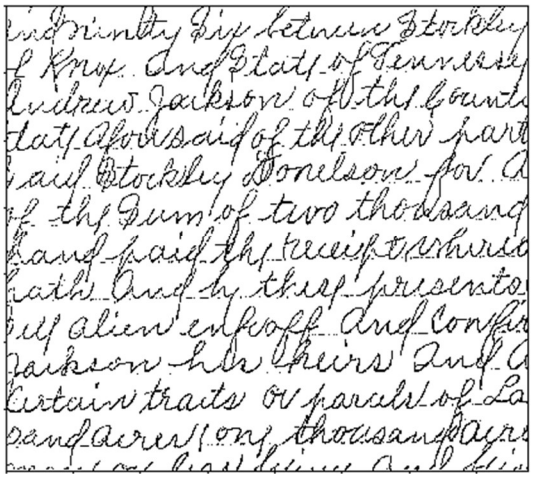
对于函数的测试，我用到了教材中给的两幅图片，和网络博客上找的一幅图片进行测试，分别用全局的 OTSU 算法，基于移动平均的局部自适应阈值化算法，和基于 local OSTU 的局部自适应阈值化算法进行了测试， 输出结果如下:



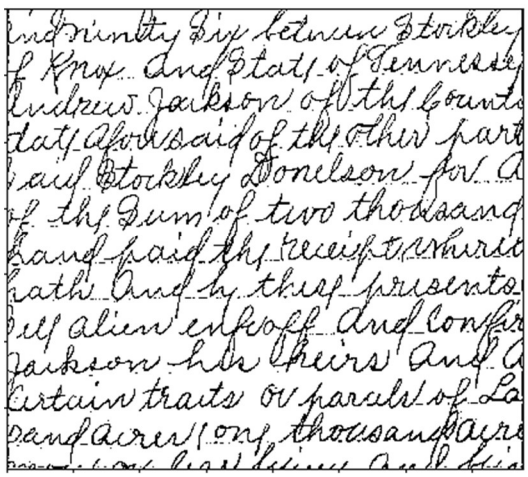
Oringinal



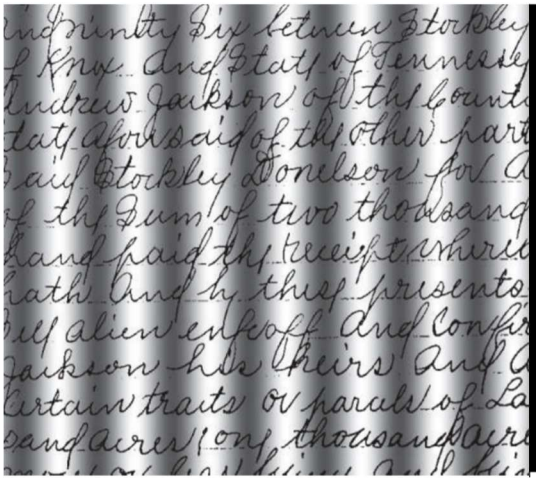
Global OSTU



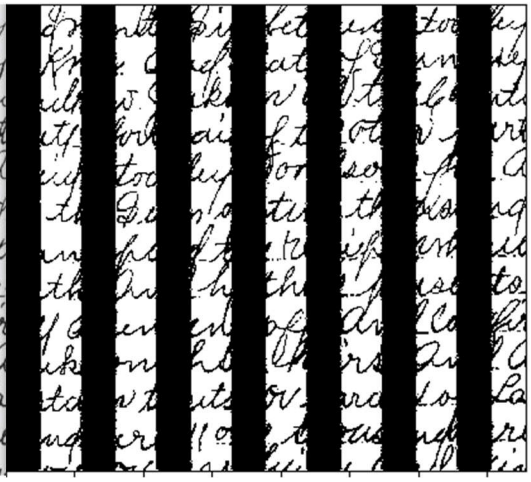
Moving average



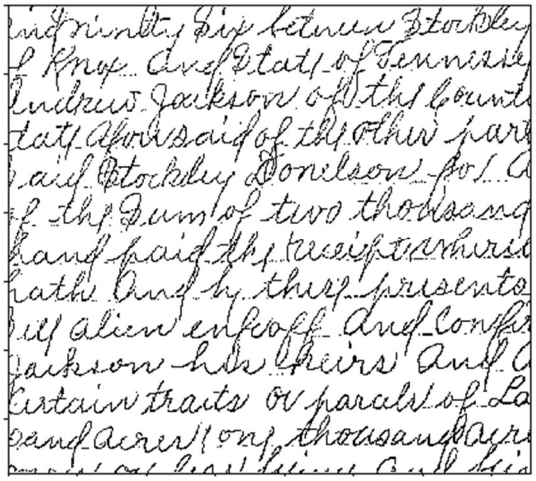
Local OSTU



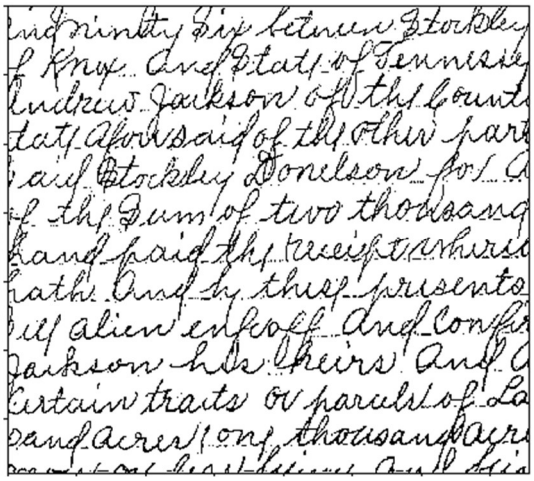
Original



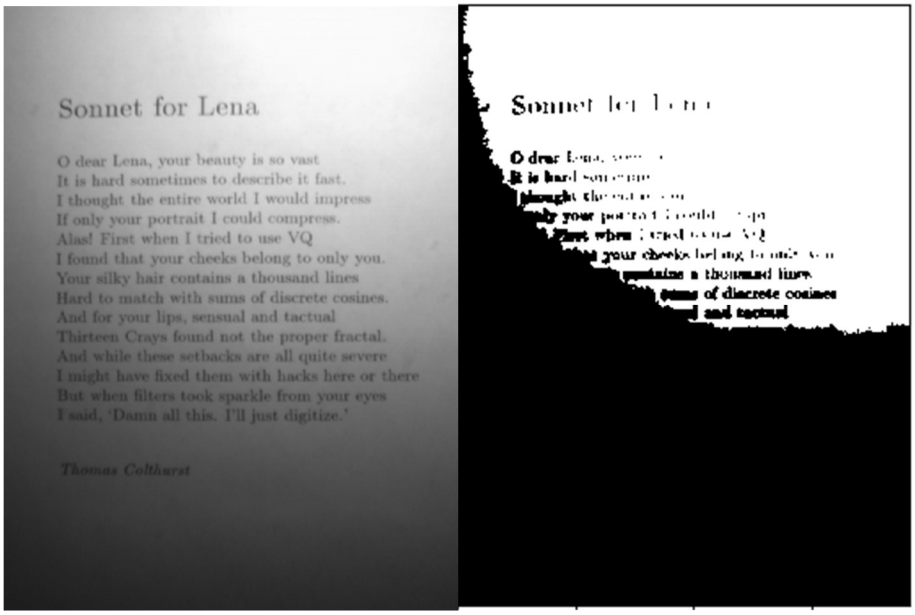
Global OSTU



Moving Average

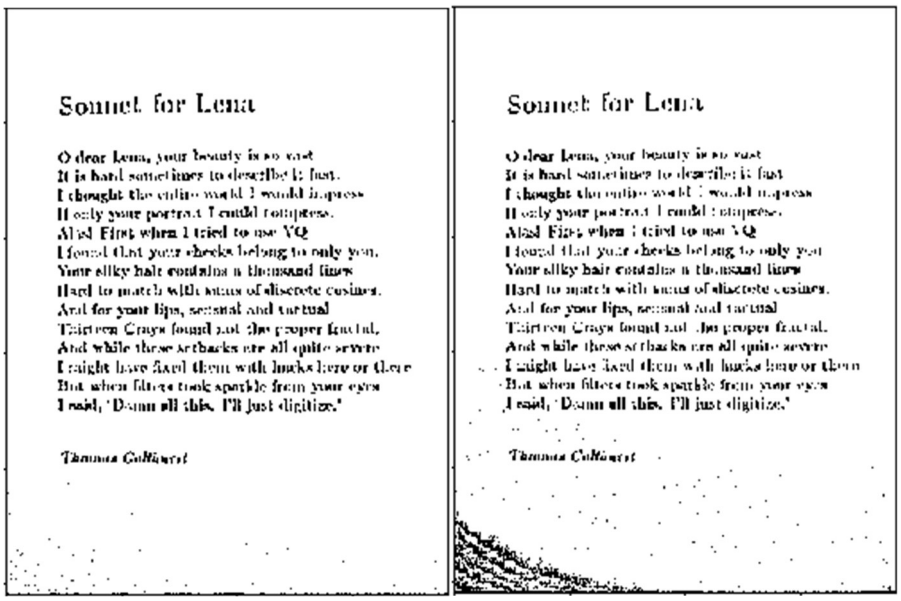


Local OSTU



Original

Global OSTU



Moving Average　　Local OSTU

- 我们可以看到，对于这种整体灰度有明暗变化的图像，全局阈值化并不能有效处理这种情况，而局部的自适应阈值就能够很好地处理这种情况。
- 在使用局部自适应阈值的时候，对于局部的阈值超参的设定，需要能够将前景和后景区分得更加清楚，对于窗口的选择，需要窗口能够不能太大，否则还是会不能区分整体的灰度变化，也不能够太小，这样容易捕捉不到局部的灰度差异。
- 其中在对第一，二幅图上面，我们对于移动平均方法选择了 4x4 的窗口和 0.9 的阈值矫正系数。 对于局部 OSTU 的方法选择了 6x6 的窗口和 0.9 的阈值矫正系数。
- 对于第三幅图，移动平均选择了 7x7 的窗口和 0.95 的阈值矫正系数，对于局部 OSTU 的方法选择了 6x6 的窗口和 0.95 的系数。

至此，第三题的任务全部完成！

4. 编程实现线性插值算法（不能调用某个算法库里面的插值函数），并应用：读出一幅图像，利用线性插值把图片空间分辨率放大 N 倍，然后保存图片。

解答:

linear_interp(img, n)

参数：

- Img: 输入的图像。
- n: 需要放大的倍数
- return: 返回线性插值后的图像

具体实现：

- 首先定义一个 single_interp() 函数，这个函数的输入为即为原始图片的 img， 和对于给定的新的图像坐标在原始图像上的映射 point
- 根据公式 $w_0 = (1-r)(1-s)$, $w_1 = r(1-s)$, $w_2 = (1-r)s$, $w_3 = rs$ $I(\mathbf{p}) = \sum_{i=0}^{n-1} w_i I(\mathbf{p}_i)$ 来计算在原始图像小数

（或整数）坐标的插值。整数就直接读取该位置的像素值，小数则根据公式计算。

```
def single_interp(pdata, point):
    """
    linear Interpolation  for a single point
    :param pdata: Given img data
    :param point: Given point which need to compute
    :return: the result of Interpolation
    """
    x, y = point
    x -= 1
    y -= 1
    try:
        return pdata[x][y]
    except IndexError:
        x1, y1 = int(x), int(y)
        x2, y2 = x1 + 1, y1 + 1
        result = pdata[x1][y1] * (x2 - x) * (y2 - y) + pdata[x2][y1] * (x - x1) * (y2 - y) + pdata[x1][y2] * (
                    x2 - x) * (y - y1) + pdata[x2][y2] * (x - x1) * (y - y1)  # based on the linear interp formula
        return int(result)
```

以上便是这个 single_interp() 函数的实现。

```
# the shape of original image
x_len, y_len = img.shape
# the shape of new image
new_width, new_height = x_len*n, y_len*n
# define a variable to store new image
new_img = np.zeros((new_width, new_height))
# begin a loop
for i in range(new_width):
    for j in range(new_height):
        # i, j corresponding to the i/n, j/n point in original image
        point = (i/n, j/n)
        # use the function pre-defined to compute the pixel value
        new_img[i][j] = single_interp(img, point)

return new_img
```

- 接下来则先获取原始图像的行数和列数，然后根据放大尺度 n 进行相乘得到新的图像的行数和列数 new_width 和 new_height，并根据它们创建一个用 0 填充的 new_image 用来保存新的图像。
- 然后对于 new_image 中的每一个像素点
  - 先根据该点的位置算出它在原图中的映射位置。
  - 然后将其输入到之前定义的 single_interp() 函数中去，得到该店在原始图像中的插值。并将新图像 new_image 对应(i, j)的像素值更换为得到的这个插值
- 循环结束后，输出新的图像。

至此， 图像的线性插值算法完成。

测试：

在测试阶段，我截取了 ppt 上的一张图片，和在网上选择了一张 lena 的图片作为测试用例。

```
if __name__ == "__main__":
    from PIL import Image
    for case in ["test4.1.png", "test4.2.png"]:
        image = np.array(Image.open(case).convert("L"))
        image2 = linear_interp(image, 4)
        image2 = Image.fromarray(image2).convert("RGB")
        image2.save("result4." + case[6] + ".png")
```

结果如下：

- 可以看到两幅图像在进行 4 倍的插值后，图像都相比原来的图像扩大了，并且分辨率也相比原来的图像提高了。其中第二幅图像，页面放不下，自动缩小了。

至此，线性插值的算法完成。本次作业也完成了!