FDU 数字图像处理 Homework 02

Due: Oct. 13, 2024
姓名: 雍崔扬
学号: 21307140051

Problem 1

Restate the Basic Global Thresholding (BGT) algorithm so that it uses the histogram of an image instead of the image itself. (Please refer to the statement of OTSU algorithm)

(1) BGT

基础全局阈值算法 (Basic Global Thresholding, BGT):
设输入图像的尺寸为 $M \times N$，灰度值是区间 $[0, L-1]$ 中的整数值.

- ① 计算输入图像的归一化直方图 $p_i = \frac{n_i}{MN}\ (i = 0, \dots, L-1)$
- ② 计算累计概率 $P_k = \sum_{i=0}^{k} p_i\ (k = 0, \dots, L-1)$
- ③ 计算累积灰度加权和 $S_k = \sum_{i=0}^{k} i \cdot p_i\ (k = 0, \dots, L-1)$
- ④ 取初始阈值 $\tau = \tau_0 = \mu_{\text{global}} = S_{L-1}$
- ⑤ 利用 $g(x,y) := \begin{cases} 1 & f(x,y) > \tau \\ 0 & f(x,y) \leq \tau \end{cases}$ 分割图像为两组像素 G_1, G_2，计算 G_1, G_2 的平均灰度值 μ_1, μ_2

$$\mu_1 = \sum_{i=0}^{\lfloor \tau \rfloor} i \cdot \frac{p_i}{P_{\lfloor \tau \rfloor}} = \frac{1}{P_{\lfloor \tau \rfloor}} \sum_{i=0}^{\lfloor \tau \rfloor} i \cdot p_i = \frac{S_{\lfloor \tau \rfloor}}{P_{\lfloor \tau \rfloor}}$$

$$\mu_2 = \sum_{i=\lfloor \tau \rfloor + 1}^{L-1} i \cdot \frac{p_i}{1 - P_{\lfloor \tau \rfloor}} = \frac{1}{1 - P_{\lfloor \tau \rfloor}} \sum_{i=\lfloor \tau \rfloor + 1}^{L-1} i \cdot p_i = \frac{S_{L-1} - S_{\lfloor \tau \rfloor}}{1 - P_{\lfloor \tau \rfloor}}$$

- ⑥ 计算新阈值 $\tau = \frac{1}{2}(\mu_1 + \mu_2)$，然后跳转至步骤 ⑤

重复迭代直至相邻两个 τ 值的绝对值差小于某个预定的值 ε 为止.
(事实上，无论模式是否可分，算法都会在有限步收敛)

---

① 计算窗口直方图的函数 `compute_histogram`:

```
def compute_histogram(image, num_bins=256):
    """
    计算图像的灰度直方图。

    :param image: 灰度图像的 numpy 数组
    :param num_bins: 直方图的 bins 数量
    :return: 直方图和 bins 边缘
    """
    histogram, bin_edges = np.histogram(image.ravel(), bins=num_bins, range=[0,
num_bins])
    return histogram, bin_edges
```

② 计算累计概率和累积灰度加权和的函数 `compute_cumulative_probabilities_and_weighted_sum`:

```
def compute_cumulative_probabilities_and_weighted_sum(histogram):
    """
    计算累计概率和累积灰度加权和。

    :param histogram: 图像的灰度直方图
    :return: 累计概率 P_k 和累积灰度加权和 S_k
    """
    total_pixels = np.sum(histogram)
    probabilities = histogram / total_pixels
    cumulative_probabilities = np.cumsum(probabilities)
    cumulative_weighted_sum = np.cumsum(np.arange(len(histogram)) *
probabilities)

    return cumulative_probabilities, cumulative_weighted_sum
```

③ 基础全局阈值算法的实现 `basic_global_thresholding`:

```
def basic_global_thresholding(image, epsilon=1e-5):
    """
    基础全局阈值算法的实现。

    :param image: 输入的灰度图像
    :param epsilon: 迭代停止的阈值
    :return: 二值化后的图像以及阈值
    """
    histogram, _ = compute_histogram(image)
    cumulative_probabilities, cumulative_weighted_sum =
compute_cumulative_probabilities_and_weighted_sum(histogram)

    # 初始阈值
    tau = cumulative_weighted_sum[-1]

    while True:
        # 计算 G1 和 G2 的平均灰度值
        lower_bound = int(np.floor(tau))
        if lower_bound >= len(histogram) - 1:
            lower_bound = len(histogram) - 1

        if lower_bound < 0:
            lower_bound = 0

        P1 = cumulative_probabilities[lower_bound]
        P2 = 1 - P1

        if P1 > 0:
            mu1 = cumulative_weighted_sum[lower_bound] / P1
        else:
            mu1 = 0

        if P2 > 0:
            mu2 = (cumulative_weighted_sum[-1] -
cumulative_weighted_sum[lower_bound]) / P2
        else:
```

```python
            mu2 = 0

        # 更新阈值
        new_tau = 0.5 * (mu1 + mu2)

        # 检查是否收敛
        if abs(new_tau - tau) < epsilon:
            break

        tau = new_tau

    # 生成二值化图像
    binary_image = (image > tau).astype(np.uint8)
    return binary_image, np.floor(tau).astype(int)
```

④ 绘制原始图像和二值化后的图像及其直方图的函数 `plot_image_and_histogram`:

```python
def plot_image_and_histogram(image, binary_image, threshold):
    """
    绘制原始图像和二值化后的图像及其直方图。

    :param image: 原始灰度图像的 2D numpy 数组
    :param binary_image: 二值化后的图像的 2D numpy 数组
    :param threshold: 用于二值化的阈值
    """
    fig, axes = plt.subplots(2, 2, figsize=(14, 10))

    # 绘制原始图像
    axes[0, 0].imshow(image, cmap='gray', vmin=0, vmax=255)
    axes[0, 0].set_title('Original Image')
    axes[0, 0].axis('off')

    # 绘制原始图像的直方图
    histogram, bin_edges = compute_histogram(image)
    axes[1, 0].bar(bin_edges[:-1], histogram, width=1, color='gray')
    axes[1, 0].set_title('Original Histogram')
    axes[1, 0].set_xlabel('Gray Level')
    axes[1, 0].set_ylabel('Frequency')

    # 添加阈值直线
    axes[1, 0].axvline(x=threshold, color='red', linestyle='--',
label='Threshold = {}'.format(int(threshold)))
    axes[1, 0].legend()

    # 绘制二值化后的图像
    axes[0, 1].imshow(binary_image, cmap='gray', vmin=0, vmax=1)
    axes[0, 1].set_title('Binary Image')
    axes[0, 1].axis('off')

    # 绘制二值化后的图像的直方图
    binary_histogram, _ = compute_histogram(binary_image, num_bins=2)
    axes[1, 1].bar(np.arange(2), binary_histogram, width=0.1, color='gray')
    axes[1, 1].set_title('Binary Histogram')
    axes[1, 1].set_xlabel('Gray Level')
    axes[1, 1].set_ylabel('Frequency')

    plt.tight_layout()
```

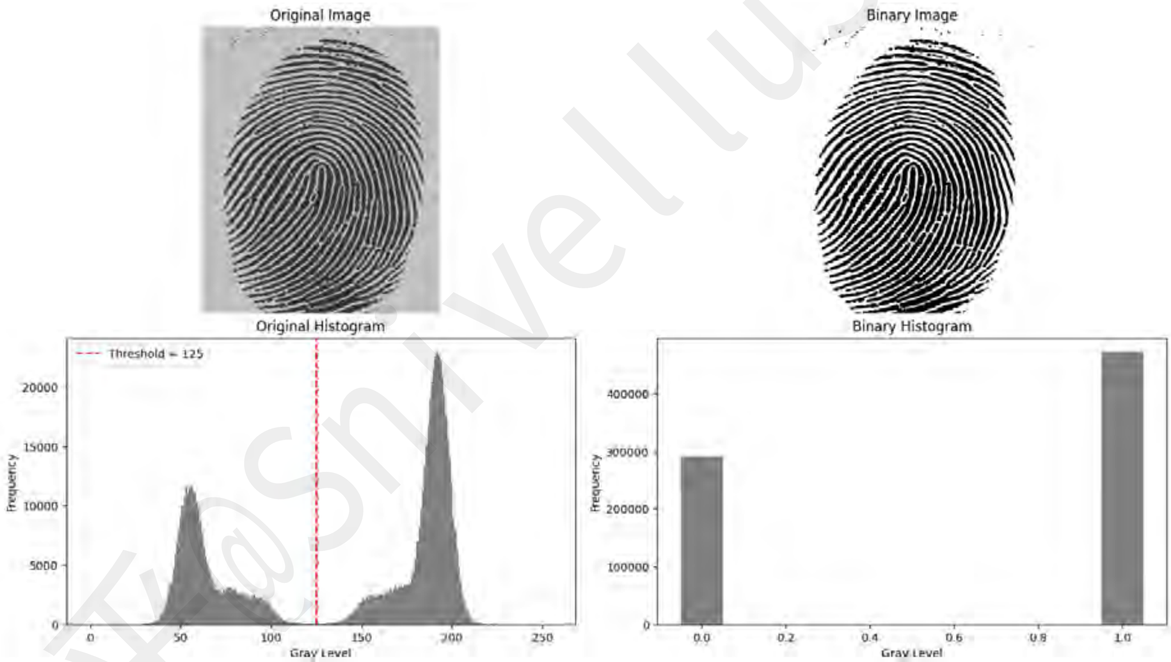
```
    plt.show()
```

⑤ **函数调用:**

```
if __name__ == "__main__":
    # 加载灰度图像
    image_name = 'DIP 10.38(a) (noisy_fingerprint).tif'
    image = Image.open(image_name).convert('L')
    image_array = np.array(image)

    # 应用基础全局阈值算法
    binary_result = basic_global_thresholding(image_array)

    # 保存二值化分割后的图像
    Image.fromarray(binary_result).save('Binary_separated_'+ str(image_name))

    # 绘制结果
    plot_image_and_histogram(image_array, binary_result)
```

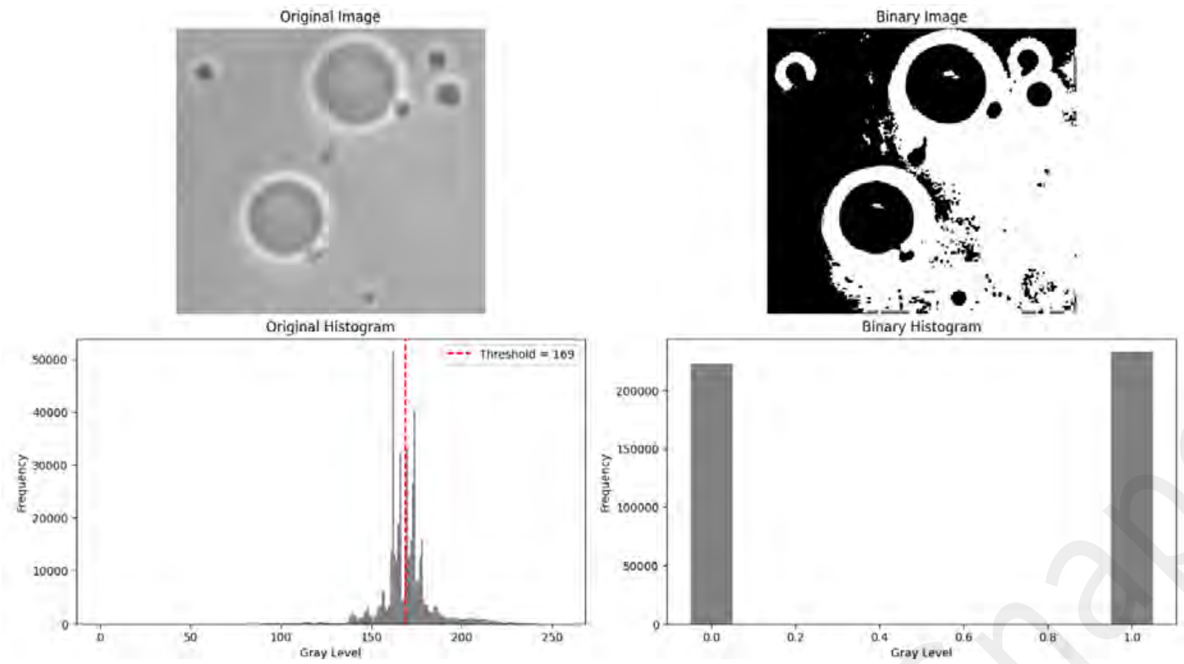
运行结果 1: (图片来源: [Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 10.38(a) noisy fingerprint]())

全局阈值为 $\tau_{\mathrm{BGT}} = 125$



运行结果 2: (图片来源: [Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 10.39(a) polymersomes]())

全局阈值为 $\tau_{\mathrm{BGT}} = 169$



(2) Otsu

Otsu 算法总结如下:
设输入图像的尺寸为 $M \times N$，灰度值是区间 $[0, L-1]$ 中的整数值.

- ① 计算输入图像的归一化直方图 $p_i = \frac{n_i}{MN}\ (i = 0, \ldots, L-1)$
- ③ 计算累计概率 $P_k = \sum_{i=0}^{k} p_i\ (k = 0, \ldots, L-1)$
- ④ 计算累积灰度加权和 $S_k = \sum_{i=0}^{k} i \cdot p_i\ (k = 0, \ldots, L-1)$
- ④ 全局灰度均值已经求出了: $\mu_{\text{global}} = \sum_{i=0}^{L-1} i \cdot p_i = S_{L-1}$
  我们只需再计算全局灰度方差 $\sigma_{\text{global}}^2 = \sum_{i=0}^{L-1} (i - \mu_{\text{global}})^2 p_i$
- ⑤ 计算类间方差 $(\sigma_{\text{between-class}}^{(k)})^2 = \frac{(P_k \cdot \mu_{\text{global}} - S_k)^2}{P_k(1-P_k)}\ (k = 0, \ldots, L-1)$
  并通过比较选取 Otsu 阈值 $\tau_{\text{otsu}} = k^\star$ (若最大点不唯一，则取平均值作为 $k^\star$)
- ⑥ 计算可分离性测度 $\eta^\star = \frac{(\sigma_{\text{between-class}}^{(k^\star)})^2}{\sigma_{\text{global}}^2}$ 作为算法效果的评判依据

---

① 计算窗口直方图的函数 `compute_histogram` **: (定义见 Problem 1 (1)①)**
② 计算累计概率和累积灰度加权和的函数
`compute_cumulative_probabilities_and_weighted_sum` **: (定义见 (1)②)**
③ Otsu 全局阈值算法的实现 `otsu_global_thresholding` **:**

```
def otsu_global_thresholding(image):
    """
    Otsu 算法的实现。

    :param image: 输入的灰度图像
    :return: 二值化后的图像、Otsu 阈值以及可分离性测度
    """
    histogram, _ = compute_histogram(image)
    cumulative_probabilities, cumulative_weighted_sum = 
compute_cumulative_probabilities_and_weighted_sum(histogram)

    # 计算全局均值
    mu_global = cumulative_weighted_sum[-1]
```

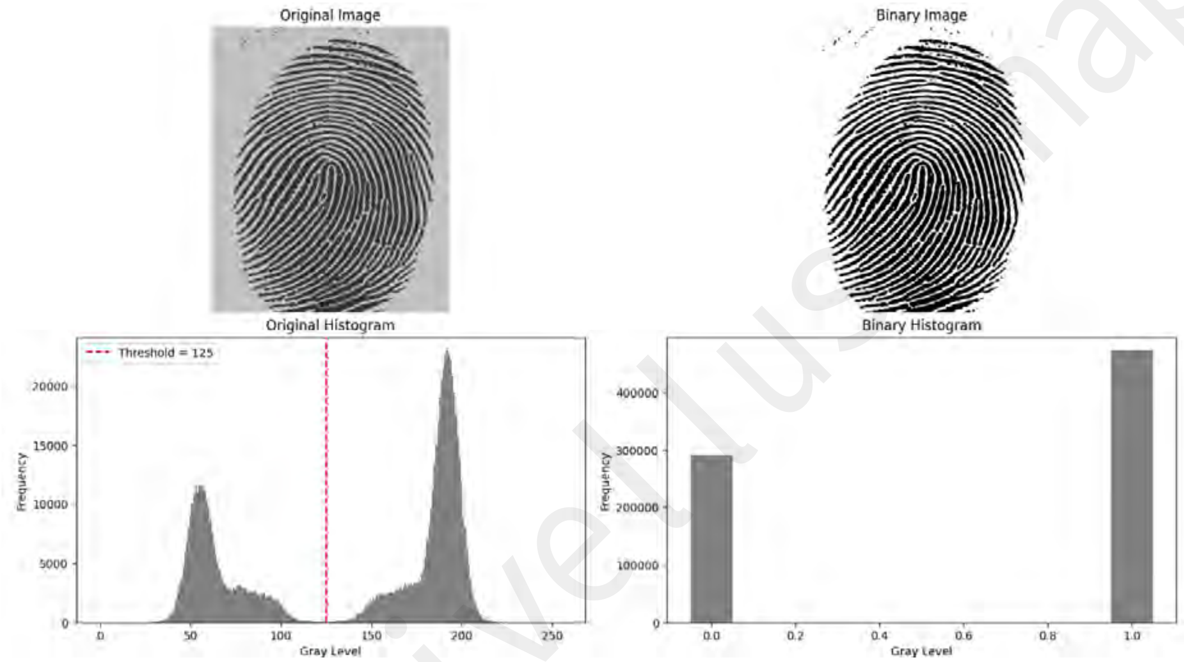
```
    # 计算全局方差
    sigma_global_squared = np.sum((np.arange(len(histogram)) - mu_global) ** 2 *
(histogram / np.sum(histogram)))
    print(f"Global mean is {mu_global} and global variance is
{sigma_global_squared}")

    # 类间方差的向量化计算
    P = cumulative_probabilities
    S = cumulative_weighted_sum

    # 避免除以 0 或无效计算的情况，先屏蔽掉 P_k 为 0 和 1 的值
    with np.errstate(divide='ignore', invalid='ignore'):
        sigma_between_class_squared = np.where(
            (P > 0) & (P < 1),
            (P * mu_global - S) ** 2 / (P * (1 - P)),
            0
        )

    # 寻找最大类间方差
    max_variance = np.max(sigma_between_class_squared)
    best_thresholds = np.where(sigma_between_class_squared == max_variance)[0]
 # 找到所有最大类间方差的阈值

    # 如果存在多个最大类间方差的阈值，取平均值
    best_threshold = np.mean(best_thresholds).astype(np.uint8)

    print(f"Max variance {max_variance} is reached at thresholds
{best_thresholds}, average threshold: {best_threshold}")
    # 生成二值化图像
    binary_image = (image > best_threshold).astype(np.uint8)

    # 计算最好阈值的可分离性测度
    separability_measure = max_variance / sigma_global_squared if
sigma_global_squared > 0 else 0

    return binary_image, best_threshold, separability_measure
```

④ 绘制原始图像和二值化后的图像及其直方图的函数 `plot_image_and_histogram`: (定义见 Problem 1 (1)④)
⑤ 函数调用:

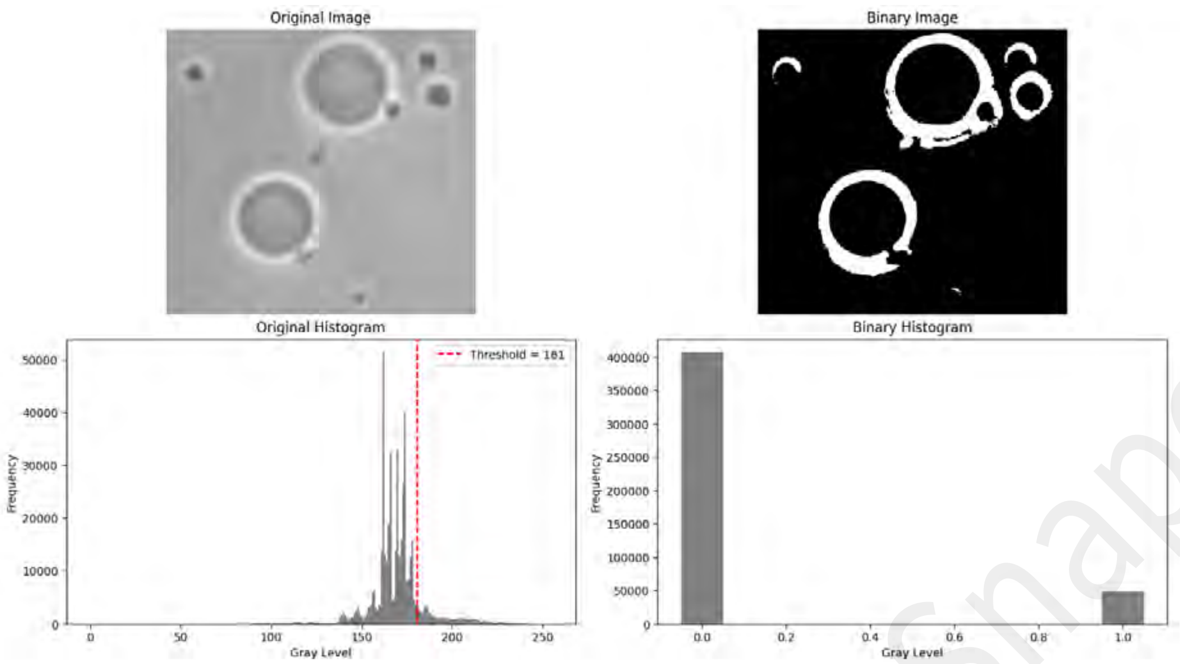
```
if __name__ == "__main__":
    # 加载灰度图像
    option = False # 更改图像选择
    if option is True:
        image_name = 'DIP 10.38(a) (noisy_fingerprint).tif'
    else:
        image_name = 'DIP 10.39(a) (polymersomes).tif'
    image = Image.open(image_name).convert('L')
    image_array = np.array(image)

    # 应用 Otsu 阈值算法
    binary_result, threshold, separability_measure =
otsu_global_thresholding(image_array)

    # 输出 Ostu 阈值的可分离性测度
    print(f"separability measure: {separability_measure}")
```

```
    # 保存二值化分割后的图像
    Image.fromarray(binary_result * 255).save('Otsu_Binary_separated_' + 
image_name)

    # 绘制结果
    plot_image_and_histogram(image_array, binary_result, threshold)
```

运行结果 1: (图片来源: Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 10.38(a) noisy fingerprint)
全局均值为 140.0，全局方差为 3762
最大累积方差为 3550，在阈值为 $\tau = 125$ 时取到.
Otsu 全局阈值 $\tau_{\text{otsu}} = 125$ 下的可分离测度 $\eta^\star = \frac{(\sigma^{(k^\star)}_{\text{between-class}})^2}{\sigma^2_{\text{global}}} = 0.944$



运行结果 2: (图片来源: Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 10.39(a) polymersomes)
全局均值为 169.6，全局方差为 190.7
最大累积方差为 88.93，在阈值为 $\tau = 181$ 时取到.
Otsu 全局阈值 $\tau_{\text{otsu}} = 181$ 下的可分离测度 $\eta^\star = \frac{(\sigma^{(k^\star)}_{\text{between-class}})^2}{\sigma^2_{\text{global}}} = 0.466$



(3) 最大熵

最大熵方法总结如下:

- ① 计算输入图像的归一化直方图 $p_i = \frac{n_i}{MN}\ (i = 0, \ldots, L-1)$
- ③ 计算累计概率 $P_k = \sum_{i=0}^{k} p_i\ (k = 0, \ldots, L-1)$
- ④ 计算累积信息熵 $E_k = -\sum_{i=0}^{k} p_i \log(p_i)\ (k = 0, \ldots, L-1)$
- ④ 计算阈值 $\tau = k\ (k = 0, \ldots, L-1)$ 下的 Shannon 信息熵:

$$H_k^{(1)} = \log(P_k) + \frac{E_k}{P_k}$$

$$H_k^{(2)} = \log(1 - P_k) + \frac{E_{L-1} - E_k}{1 - P_k}$$

$$H_k = H_k^{(1)} + H_k^{(2)}$$

选取 $H_k\ (k = 0, \ldots, L-1)$ 中最大值对应的阈值 $k^\star$ 作为阈值 (若最大点不唯一，则取平均值作为 $k^\star$)

---

① 计算窗口直方图的函数 `compute_histogram` **: (定义见 Problem 1 (1)①)**
② 计算累积概率和累积信息熵的函数 `compute_cumulative_probabilities_and_entropy` **:**

```
def compute_cumulative_probabilities_and_entropy(histogram):
    """
    计算累计概率和累积信息熵

    :param histogram: 图像的灰度直方图
    :return: 累计概率 P_k 和累积信息熵
    """
    total_pixels = np.sum(histogram)
    probabilities = histogram / total_pixels

    # 累计概率
```

```
    cumulative_probabilities = np.cumsum(probabilities)

    # 累积信息熵，跳过 p = 0 的项, 因为极限值 0 * log(0) 定义为 0
    cumulative_entropy = -np.cumsum(np.where(probabilities > 0, probabilities *
np.log(probabilities), 0))

    return cumulative_probabilities, cumulative_entropy
```

③ 最大熵全局阈值算法的实现 `max_entropy_global_thresholding`:

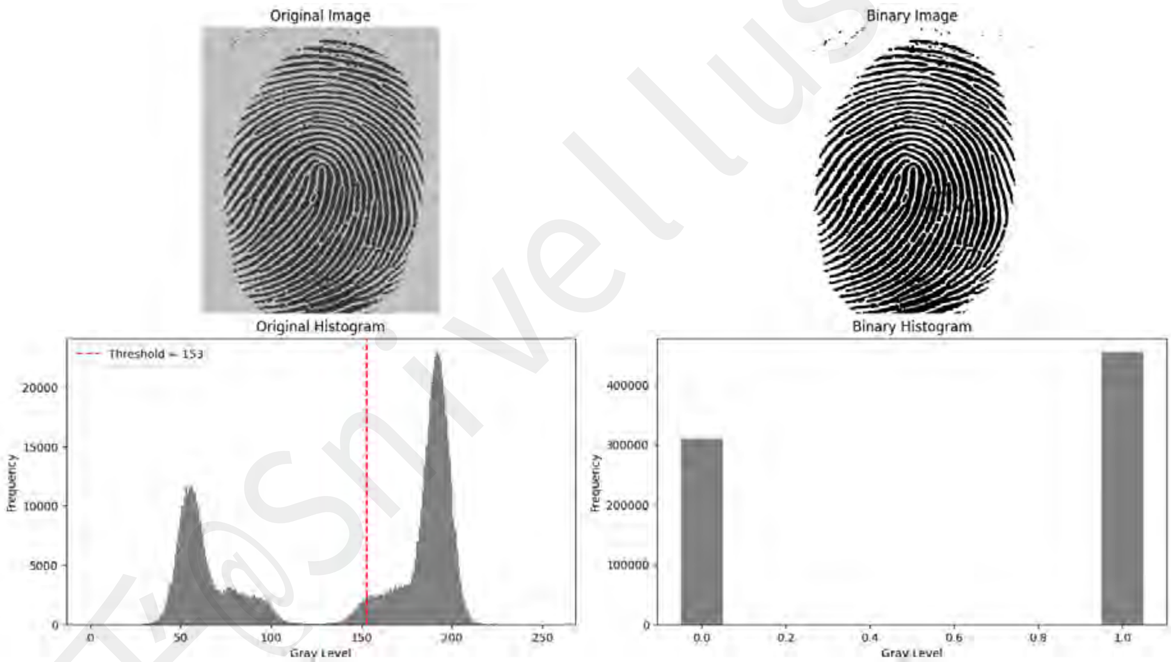
```
def max_entropy_global_thresholding(image):
    """
    最大熵分割算法的实现。

    :param image: 输入的灰度图像
    :return: 二值化后的图像、最大熵对应的阈值
    """
    # 计算灰度直方图
    histogram, _ = compute_histogram(image)

    # 计算累计概率和累积信息熵
    cumulative_probabilities, cumulative_entropy =
compute_cumulative_probabilities_and_entropy(histogram)

    # 计算 Shannon 信息熵 H
    with np.errstate(divide='ignore', invalid='ignore'):
        H_1 = np.where(cumulative_probabilities > 0,
                       np.log(cumulative_probabilities) + cumulative_entropy /
cumulative_probabilities,
                       0)
        H_2 = np.where(1 - cumulative_probabilities > 0,
                       np.log(1 - cumulative_probabilities) +
(cumulative_entropy[-1] - cumulative_entropy) / (1 - cumulative_probabilities),
                       0)

    H = H_1 + H_2

    # 寻找最大熵的所有阈值
    max_entropy = np.max(H)
    best_thresholds = np.where(H == max_entropy)[0]  # 找到所有最大熵对应的阈值

    # 如果存在多个最大熵阈值，取平均值
    best_threshold = np.mean(best_thresholds).astype(np.uint8)

    print(f"Max entropy {max_entropy} is reached at thresholds
{best_thresholds}, average threshold: {best_threshold}")

    # 生成二值化图像
    binary_image = (image > best_threshold).astype(np.uint8)

    return binary_image, best_threshold
```

④ 绘制原始图像和二值化后的图像及其直方图的函数 `plot_image_and_histogram`**: (定义见 Problem 1 (1)④)**

⑤ 函数调用:

```
if __name__ == "__main__":
    # 加载灰度图像
    option = True # 更改图像选择
    if option is True:
        image_name = 'DIP 10.38(a) (noisy_fingerprint).tif'
    else:
        image_name = 'DIP 10.39(a) (polymersomes).tif'
    image = Image.open(image_name).convert('L')
    image_array = np.array(image)

    # 应用最大熵分割算法
    binary_result, threshold = max_entropy_global_thresholding(image_array)

    # 保存二值化分割后的图像
    Image.fromarray(binary_result * 255).save('MaxEntropy_Binary_' + image_name)

    # 绘制结果
    plot_image_and_histogram(image_array, binary_result, threshold)
```

运行结果 1: (图片来源: [Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 10.38(a) noisy fingerprint](#))
最大熵 7.842 在阈值 $\tau = 153$ 下取得.



Problem 2

Design an algorithm of locally adaptive thresholding based on local OTSU or maximum of local entropy;
implement the algorithm and test it on example images.

Solution:
① 计算窗口直方图的函数 `compute_histogram` **: (定义见 Problem 1 (1)①)**
② 使用增量更新直方图的函数 `update_histogram` **:**

```
def update_histogram(old_hist, new_col=None, remove_col=None, num_bins=256):
    """
    使用增量更新直方图.
```

```
    :param old_hist: 当前的直方图.
    :param new_col: 要加入的新的列像素 (可以为 None).
    :param remove_col: 要移除的列像素 (可以为 None).
    :param num_bins: 直方图的 bins 数量.
    :return: 更新后的直方图.
    """
    if new_col is None:
        return old_hist - np.bincount(remove_col, minlength=num_bins)
    elif remove_col is None:
        return old_hist + np.bincount(new_col, minlength=num_bins)
    else:
        return old_hist - np.bincount(remove_col, minlength=num_bins) +
np.bincount(new_col, minlength=num_bins)
```

③ 计算局部直方图的函数 `compute_local_histograms`:

```
def compute_local_histograms(image, window_size=(9, 9), num_bins=256):
    """
    计算图像的局部直方图.
    :param image: 输入的灰度图像.
    :param window_size: 邻域窗口的尺寸 (height, width).
    :param num_bins: 直方图的 bins 数量.
    :return: 所有局部直方图的列表.
    """
    h, w = image.shape
    win_h, win_w = window_size
    half_win_h = win_h  // 2  # 使用整数除法, 得到窗口半径
    half_win_w = win_w  // 2

    # 初始化局部直方图列表
    local_histograms = np.zeros((h, w, num_bins), dtype=np.uint8)

    # 移动完整窗口
    for i in range(h):
        if i == 0:
            # 计算第一个完整窗口的直方图
            local_histograms[0, 0, :], _ =
compute_histogram(image[0:half_win_h+1,0:half_win_w+1], num_bins=num_bins)
        else:
        # 更新直方图 (垂直移动)
            if i <= half_win_h: # 首
                new_row = image[i+half_win_h, 0:half_win_w+1]
                remove_row = None
            elif i >= h - half_win_h: # 尾
                new_row = None
                remove_row = image[i-half_win_h-1, 0:half_win_w+1]
            else: # 中间部分
                new_row = image[i+half_win_h, 0:half_win_w+1]
                remove_row = image[i-half_win_h-1, 0:half_win_w+1]
            # 更新直方图 (垂直移动)
            local_histograms[i, 0, :] = update_histogram(local_histograms[i-1,
0, :], new_row, remove_row, num_bins=num_bins)

        i_safe_lower = max(i-half_win_h,0)
        i_safe_upper = min(i+half_win_h+1,h)

        for j in range(1, w):
```

```python
            # 更新直方图（水平移动）
            if j <= half_win_w: # 首
                new_col = image[i_safe_lower:i_safe_upper, j+half_win_w]
                remove_col = None
            elif j >= w - half_win_w: # 尾
                new_col = None
                remove_col = image[i_safe_lower:i_safe_upper, j-half_win_w-1]
            else: # 中间部分
                new_col = image[i_safe_lower:i_safe_upper, j+half_win_w]
                remove_col = image[i_safe_lower:i_safe_upper, j-half_win_w-1]

            # 更新直方图（水平移动）
            local_histograms[i, j, :] = update_histogram(local_histograms[i, j-
1, :], new_col, remove_col, num_bins=num_bins)

    return local_histograms
```

④ 计算累计概率和累积灰度加权和的函数 `compute_cumulative_probabilities_and_weighted_sum`**: (定义见 (1)②)**

⑤ Otsu 局部阈值算法 `otsu_local_thresholding`**:**

```python
def otsu_local_thresholding(image, window_size=(9, 9), num_bins=256):
    """
    Otsu 局部阈值算法的实现。

    :param image: 输入的灰度图像
    :param window_size: 局部窗口大小
    :param num_bins: 直方图的 bins 数量
    :return: 二值化后的图像和局部阈值矩阵
    """
    h, w = image.shape

    # 初始化二值化的图像和阈值矩阵
    binary_image = np.zeros_like(image, dtype=np.uint8)
    thresholds = np.zeros((h, w), dtype=np.uint8)

    # 计算每个局部窗口的直方图
    local_histograms = compute_local_histograms(image, window_size=window_size,
num_bins=num_bins)

    # 遍历图像中的每个像素
    for i in range(h):
        for j in range(w):
            # 使用该像素位置的局部直方图
            current_hist = local_histograms[i, j]

            # 计算累积概率和累积加权和
            cumulative_probabilities, cumulative_weighted_sum =
compute_cumulative_probabilities_and_weighted_sum(current_hist)

            # 全局均值
            mu_global = cumulative_weighted_sum[-1]

            # 类间方差的向量化计算
            P = cumulative_probabilities
            S = cumulative_weighted_sum
```

```
            with np.errstate(divide='ignore', invalid='ignore'):
                sigma_between_class_squared = np.where(
                    (P > 0) & (P < 1),
                    (P * mu_global - S) ** 2 / (P * (1 - P)),
                    0
                )

            # 寻找最大类间方差对应的阈值
            best_threshold = np.argmax(sigma_between_class_squared)

            # 记录最佳阈值
            thresholds[i, j] = best_threshold

            # 应用阈值进行二值化
            binary_image[i, j] = (image[i, j] > best_threshold).astype(np.uint8)

    return binary_image, thresholds
```

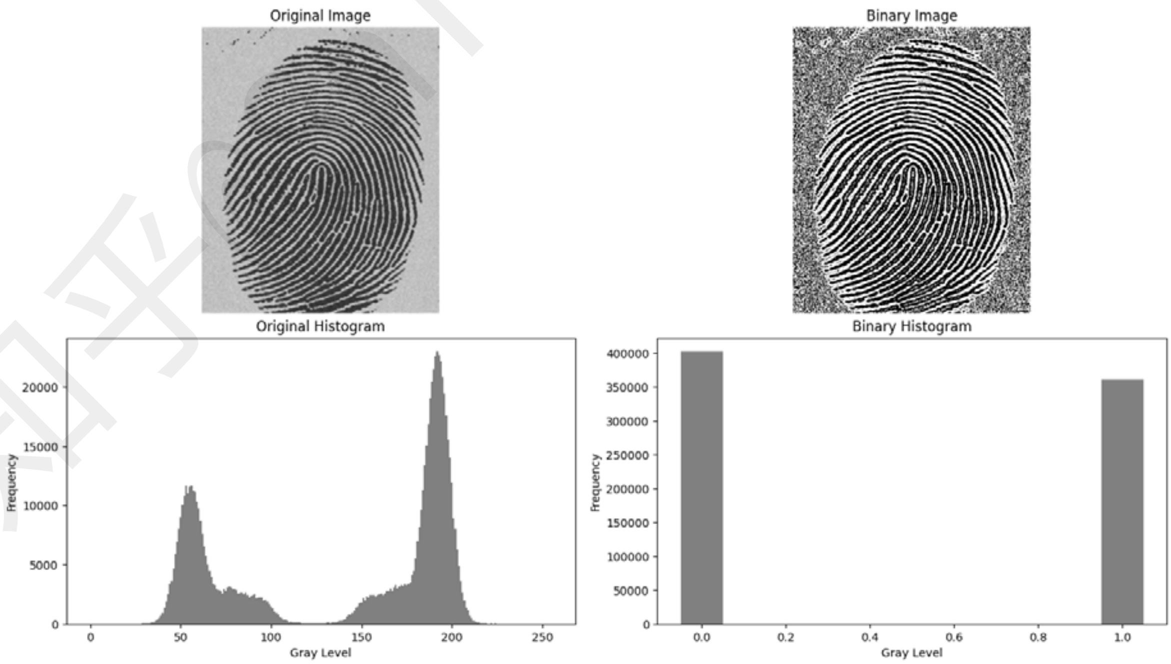
⑥ 绘制原始图像和二值化后的图像及其直方图的函数 `plot_image_and_histogram`**:**

```
def plot_image_and_histogram(image, binary_image):
    """
    绘制原始图像和二值化后的图像及其直方图。

    :param image: 原始灰度图像的 2D numpy 数组
    :param binary_image: 二值化后的图像的 2D numpy 数组
    """
    fig, axes = plt.subplots(2, 2, figsize=(14, 10))

    # 绘制原始图像
    axes[0, 0].imshow(image, cmap='gray', vmin=0, vmax=255)
    axes[0, 0].set_title('Original Image')
    axes[0, 0].axis('off')

    # 绘制原始图像的直方图
    histogram, bin_edges = compute_histogram(image)
    axes[1, 0].bar(bin_edges[:-1], histogram, width=1, color='gray')
    axes[1, 0].set_title('Original Histogram')
    axes[1, 0].set_xlabel('Gray Level')
    axes[1, 0].set_ylabel('Frequency')

    # 绘制二值化后的图像
    axes[0, 1].imshow(binary_image, cmap='gray', vmin=0, vmax=1)
    axes[0, 1].set_title('Binary Image')
    axes[0, 1].axis('off')

    # 绘制二值化后的图像的直方图
    binary_histogram, _ = compute_histogram(binary_image, num_bins=2)
    axes[1, 1].bar(np.arange(2), binary_histogram, width=0.1, color='gray')
    axes[1, 1].set_title('Binary Histogram')
    axes[1, 1].set_xlabel('Gray Level')
    axes[1, 1].set_ylabel('Frequency')

    plt.tight_layout()
    plt.show()
```

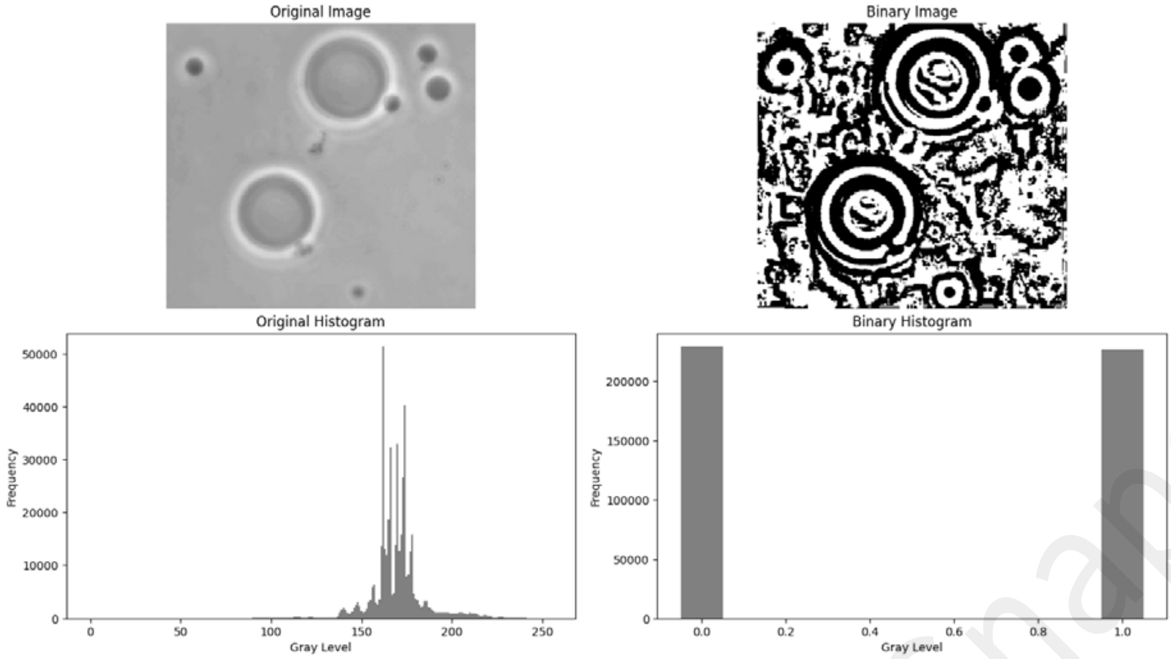
⑦ 函数调用:

```python
if __name__ == "__main__":
    # 加载灰度图像
    option = 3 # 更改图像选择
    if option == 1:
        image_name = 'DIP 10.38(a) (noisy_fingerprint).tif'
    elif option == 2:
        image_name = 'DIP 10.39(a) (polymersomes).tif'
    elif option == 3:
        image_name = 'DIP 2.22 (face).tif'
    else:
        image_name = 'DIP 10.43(a) (yeast_USC).tif'
    image = Image.open(image_name).convert('L')
    image_array = np.array(image)

    # 定义窗口大小
    window_size = (81, 81)

    # 应用 Otsu 局部阈值算法
    binary_image, thresholds = otsu_local_thresholding(image_array,
window_size=window_size)

    # 保存二值化分割后的图像
    Image.fromarray(binary_image * 255).save('Otsu_Binary_separated_' +
image_name)

    # 绘制结果
    plot_image_and_histogram(image_array, binary_image)
```

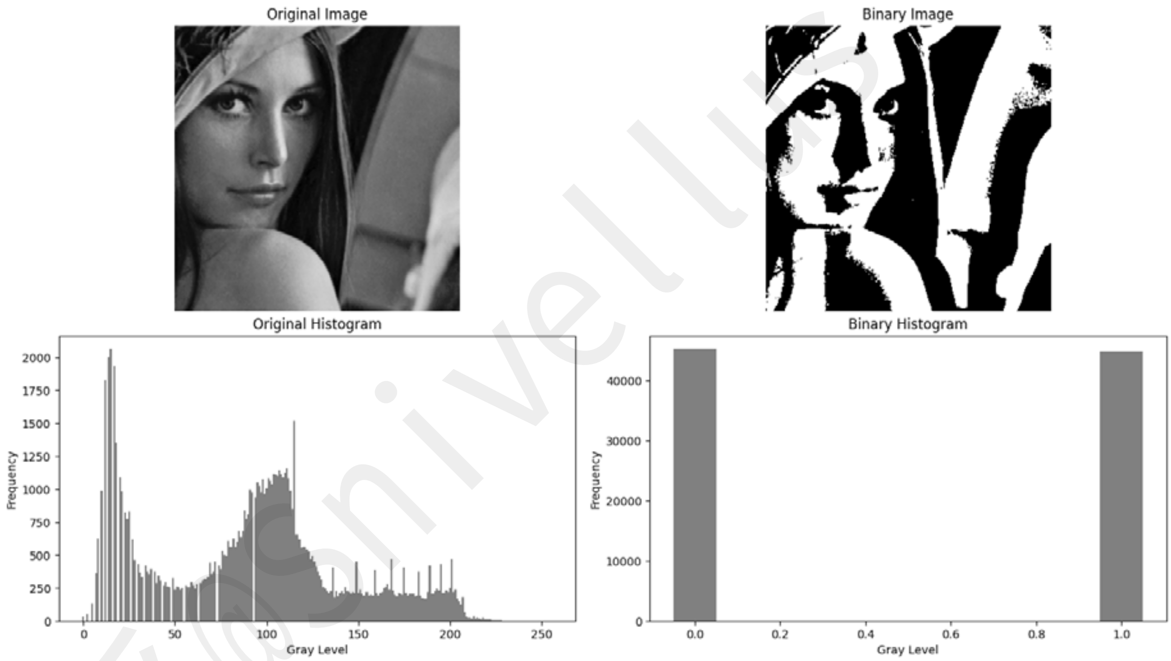
运行结果 1: (图片来源: [Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 10.38(a) noisy fingerprint]())
窗口大小: 9×9



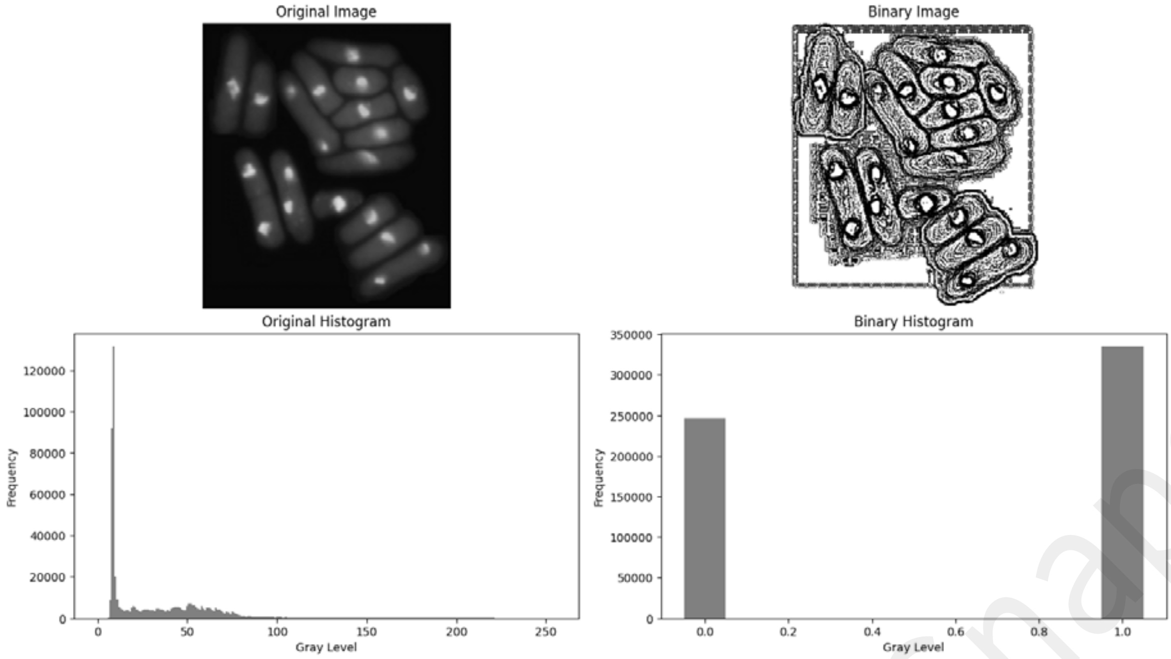
运行结果 2: (图片来源: [Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 10.39(a) polymersomes]())
窗口大小: 21×21



运行结果 3: (图片来源: [Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 2.22 face]())
窗口大小: 71×71



运行结果 4: (图片来源: [Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 10.43(a) yeast_USC]())
窗口大小: 9×9



Problem 3

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)
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.

Solution:
(图片来源: [Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 2.20(a) (chronometer 3600x2808)])
这是一幅大小为 3600×2808 像素的灰度图像 (显示时缩小为实际大小的 15%):

(1) 下采样

我们首先使用下采样将其缩小 18 倍，变为一幅 200×156 的图像.

```
def downsample_image(image, scale_factor):
    """
    将图像降采样，降低分辨率。

    :param image: 输入的灰度图像 (numpy数组)
    :param scale_factor: 缩小的倍数
    :return: 降采样后的图像 (numpy数组)
    """
    h, w = image.shape
    # 计算降采样后的尺寸
    new_h, new_w = int(h / scale_factor), int(w / scale_factor)

    # 创建降采样后的图像
    downsampled_image = np.zeros((new_h, new_w), dtype=np.uint8)

    for i in range(new_h):
        for j in range(new_w):
            # 对于下采样，简单选择最近邻的像素
            downsampled_image[i, j] = image[i * scale_factor, j * scale_factor]

    return downsampled_image
```

```python
if __name__ == "__main__":
    # 读取原图像
    image_path = 'DIP 2.20(a) (chronometer 3600x2808).tif'  # 输入图像路径
    img = Image.open(image_path).convert('L')  # 转换为灰度图像
    image_array = np.array(img)

    # 设置缩小倍数 N
    scale_factor = 18

    # 进行降采样
    downsampled_image = downsample_image(image_array, scale_factor)

    # 保存降采样后的图像
    downsampled_image_pil = Image.fromarray(downsampled_image)
    downsampled_image_pil.save('downsampled_image.tif')
    downsampled_image_pil.show()
```

运行结果: (downsampled_image.tif 显示的是实际大小)

(2) 上采样

现在我们想让下采样结果 (200×156) 放大 18 倍，恢复原尺寸 3600×2808

一种简单的放大方法是，创建一个大小为 3600×2808 像素的假想网格，网格的像素间隔与原图像的像素间隔相同.
然后收缩这个网格，使它完全与原图像重叠.
显然，收缩后的 3600×2808 网格的像素间隔要小于原图像的像素间隔.
我们可以基于原图像的灰度给新图像的像素赋值，最后将图像展开到指定的大小，得到放大后的图像.

赋值的方法有以下几种:

- **① 最邻近内插 (nearest neighbor interpolation):**
  将原图像中最近邻的灰度作为新图像中待求位置的灰度.
  这种方法简单，但会产生一些人为失真，例如严重的直边失真.

```python
def nearest_neighbor_interpolation(image, scale_factor):
    """
    对图像进行最邻近插值放大

    :param image: 输入的灰度图像 (numpy数组)
    :param scale_factor: 图像缩放的倍数
    :return: 放大后的图像 (numpy数组)
    """
    h, w = image.shape  # 原图像尺寸
    new_h, new_w = int(h * scale_factor), int(w * scale_factor)  # 新图像尺寸
```

```python
    # 创建目标图像
    new_image = np.zeros((new_h, new_w), dtype=np.uint8)

    # 生成新的像素坐标
    x_new = np.arange(new_h) / scale_factor
    y_new = np.arange(new_w) / scale_factor

    # 计算对应的原图像坐标
    orig_x = np.clip(np.floor(x_new).astype(int), 0, h - 1)
    orig_y = np.clip(np.floor(y_new).astype(int), 0, w - 1)

    # 使用广播机制将原图像的像素值赋给新图像
    new_image = image[orig_x[:, None], orig_y]

    return new_image
```

- **② 双线性内插 (bilinear interpolation):**
  使用尺寸为 $M \times N$ 的原图像 f 中的 4 个最近邻的灰度来计算新图像 g 中待求位置 (x, y) 的灰度. 取 x, y 的小数部分为 dx, dy，记四个邻近点的灰度值为 $I_{11}, I_{12}, I_{22}, I_{21}$ (左上, 右上, 右下, 左下)

$$
\begin{gathered}
dx = x - \lfloor x \rfloor \\
dy = y - \lfloor y \rfloor \\
I_{11} = f(\lfloor x \rfloor, \lfloor y \rfloor) \\
I_{12} = f(\lfloor x \rfloor, \min\{\lfloor y \rfloor + 1, N - 1\}) \\
I_{21} = f(\min\{\lfloor x \rfloor + 1, M - 1\}, \lfloor y \rfloor) \\
I_{22} = f(\min\{\lfloor x \rfloor + 1, M - 1\}, \min\{\lfloor y \rfloor + 1, N - 1\}) \\
\hline
g(x, y) = I_{11}(1 - dx)(1 - dy) + I_{12}(1 - dx)dy + I_{21}dx(1 - dy) + I_{22}dxdy
\end{gathered}
$$

```python
def bilinear_interpolation(image, scale_factor):
    """
    对图像进行双线性插值放大

    :param image: 输入的灰度图像 (numpy数组)
    :param scale_factor: 图像缩放的倍数
    :return: 放大后的图像 (numpy数组)
    """
    h, w = image.shape  # 原图像尺寸
    new_h, new_w = int(h * scale_factor), int(w * scale_factor)  # 新图像尺寸

    # 创建目标图像
    new_image = np.zeros((new_h, new_w), dtype=np.uint8)

    # 生成新图像中像素的浮点坐标
    x_new = np.arange(new_h) / scale_factor
    y_new = np.arange(new_w) / scale_factor

    # 获取整数部分和小数部分
    x1 = np.floor(x_new).astype(int)
    y1 = np.floor(y_new).astype(int)
    x2 = np.clip(x1 + 1, 0, h - 1)
    y2 = np.clip(y1 + 1, 0, w - 1)

    # 计算小数部分
    dx = x_new - x1
```

```
    dy = y_new - y1

    # 使用广播机制计算双线性插值
    I11 = image[x1[:, None], y1[None, :]]  # 左上角像素
    I12 = image[x1[:, None], y2[None, :]]  # 右上角像素
    I21 = image[x2[:, None], y1[None, :]]  # 左下角像素
    I22 = image[x2[:, None], y2[None, :]]  # 右下角像素

    # 计算插值结果
    new_image = ((I11 * (1 - dx)[:, None] * (1 - dy)[None, :] +
                  I12 * (1 - dx)[:, None] * dy[None, :] +
                  I21 * dx[:, None] * (1 - dy)[None, :] +
                  I22 * dx[:, None] * dy[None, :])).astype(np.uint8)

    return new_image
```

- **③ 双三次内插 (bicubic interpolation):**
  使用原图像 f 中的 16 个最近邻的灰度来计算新图像中待求位置 (x, y) 的灰度.

$$g(x, y) = \sum_{i,j=0}^{3} a_{ij} f(x_i, y_j)$$

其中 16 个系数 a_{ij} $(i, j = 0, 1, 2, 3)$ 由点 (x, y) 的 16 个最近邻点 (x_i, x_j) $(i, j = 0, 1, 2, 3)$ 的梯度和 Hessian 矩阵求出.
BiCubic 基函数为:

$$W(t) := \begin{cases} \frac{3}{2}|t|^3 - \frac{5}{2}|t|^2 + 1 & \text{if } 0 \le |t| \le 1 \\ -\frac{1}{2}|t|^3 + \frac{5}{2}|t|^2 - 4|t| + 2 & \text{if } 1 < |t| < 2 \\ 0 & \text{otherwise} \end{cases}$$

$$a_{ij} = W(x - x_i)W(y - y_j)\ (i, j = 0, 1, 2, 3)$$

```
def cubic_kernel(x):
    """
    双三次插值核函数
    """
    abs_x = np.abs(x)
    abs_x2 = abs_x ** 2
    abs_x3 = abs_x ** 3

    result = np.where(
        abs_x <= 1,
        (1.5 * abs_x3 - 2.5 * abs_x2 + 1),
        np.where(
            (abs_x > 1) & (abs_x <= 2),
            (-0.5 * abs_x3 + 2.5 * abs_x2 - 4 * abs_x + 2),
            0
        )
    )

    return result

def bicubic_interpolation(image, scale_factor):
    """
    对图像进行双三次插值放大

    :param image: 输入的灰度图像 (numpy数组)
```

```
    :param scale_factor: 图像缩放的倍数
    :return: 放大后的图像 (numpy数组)
    """
    h, w = image.shape  # 原图像尺寸
    new_h, new_w = int(h * scale_factor), int(w * scale_factor)  # 新图像尺寸

    # 创建目标图像，初始化为float64类型
    new_image = np.zeros((new_h, new_w), dtype=np.float64)

    # 生成新图像中像素的浮点坐标
    x_new = np.arange(new_h) / scale_factor
    y_new = np.arange(new_w) / scale_factor

    # 获取整数部分
    x_floor = np.floor(x_new).astype(int)
    y_floor = np.floor(y_new).astype(int)

    # 确保坐标不越界
    x_floor = np.clip(x_floor, 1, h - 3)
    y_floor = np.clip(y_floor, 1, w - 3)

    # 计算小数部分
    dx = x_new - x_floor
    dy = y_new - y_floor

    # 双三次插值
    for i in range(-1, 3):
        for j in range(-1, 3):
            # 获取插值权重
            weight_x = cubic_kernel(dx - i)
            weight_y = cubic_kernel(dy - j)

            # 获取原图像中的像素点
            patch = image[(x_floor + i)[:, None], (y_floor + j)[None, :]]

            # 对所有像素点进行加权求和
            new_image += (weight_x[:, None] * weight_y[None, :]) * patch

    # 将插值结果剪裁到合法范围并转换为uint8
    new_image = np.clip(new_image, 0, 255).astype(np.uint8)

    return new_image
```

函数调用:

```
if __name__ == "__main__":
    # 读取图像
    image_path = 'downsampled_image.tif'
    img = Image.open(image_path).convert('L')  # 转换为灰度图像
    image_array = np.array(img)

    # 设置缩放倍数 N
    N = 18

    # 进行最邻近插值
    resized_image = nearest_neighbor_interpolation(image_array, N)
```

```
    # 保存结果
    output_image = Image.fromarray(resized_image)
    output_image.save('resized_image_nearest_neighbor.png')
    output_image.show()

    # 进行双线性插值
    resized_image = bilinear_interpolation(image_array, N)

    # 保存结果
    output_image = Image.fromarray(resized_image)
    output_image.save('resized_image_bilinear.png')
    output_image.show()

    # 进行双三方插值
    resized_image = bicubic_interpolation(image_array, N)

    # 保存结果
    output_image = Image.fromarray(resized_image)
    output_image.save('resized_image_bicubic.png')
    output_image.show()
```

运行结果: (显示时缩放了 10 倍)
从左至右分别为最邻近内插、双线性内插和双三方内插的结果，可以看出清晰度逐渐提高.