图像处理与可视化 Homework 06

学号: 21307140051
姓名: 雍崔扬

Problem 1

实现基于 K-均值分类的分割算法或基于 Gauss 混合模型的分割算法.
并使用噪声污染过的图像 (例如 $p_{\text{pepper}} + p_{\text{salt}} = 0.1\%$ 的椒盐噪声) 进行测试:

- ① 测试二类分割，并与阈值算法 (例如 Ostu 算法) 二值化的结果进行比较.
- ② 测试多类分割.
- ③ 针对噪声图像，讨论为什么分割的结果不准确，有什么方法可以取得更好的分割结果.
  (备注: 不要求实现该方法，只是讨论)

Part (1)

Otsu 算法:

- ① 计算输入图像的归一化直方图 $p_i = \frac{n_i}{MN}\ (i = 0, \dots, L-1)$
- ③ 计算累计概率 $P_k = \sum_{i=0}^{k} p_i\ (k = 0, \dots, L-1)$
- ④ 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 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 $\sigma^2_{\text{global}} = \sum_{i=0}^{L-1} (i - \mu_{\text{global}})^2 p_i$
- ⑤ 计算类间方差 $(\sigma^{(k)}_{\text{between-class}})^2 = \frac{[P_k \cdot \mu_{\text{global}} - S_k]^2}{P_k(1-P_k)}\ (k = 0, \dots, L-1)$
  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 $\eta^\star = \frac{(\sigma^{(k^\star)}_{\text{between-class}})^2}{\sigma^2_{\text{global}}}$ 作为算法效果的评判依据

我们已经在 Homework 01 中讨论过 Otsu 算法的实现了 (详见附件中的 `HW01_P1_Otsu.py`)

K 均值算法:
给定灰度值 $z_1, \dots, z_n$

- ① 初始化: 规定一组初始均值 $\mu_1, \dots, \mu_K$
- ② 将每个样本分配给最接近的均值对应的聚类集合:
  遍历 $i = 1, \dots, n$
  灰度值 z_i 隶属的类别序号为 $\arg\min_{k \in \{1, \dots, K\}} \|z_i - \mu_k\|^2$，分配给对应的聚类集合.
  最终得到聚类集合 $S_1, \dots, S_K$
- ③ 更新聚类中心:

$$\mu_k = \text{average}(S_k) = \frac{1}{|S_k|} \sum_{z \in S_k} z_k\ (\forall\ k = 1, \dots, K)$$

- ④ 收敛检查:
  计算当前步骤和前几步中均值的残差的 Euclid 范数.
  若低于某个预设定的阈值，则停止迭代; 否则返回步骤 ②

该算法在有限次数的迭代后收敛到一个局部极小解 (不保证是全局极小解)
收敛的结果取决于 $\mu_1, \dots, \mu_K$ 的初值.
一般将均值 $\mu_1, \dots, \mu_K$ 的初值设置为随机选取的 k 个像素点的灰度值，重复多次，以检验解的稳定性.

K 均值算法的 Python 实现为:

```
def k_means_grayscale(image, K, initial_means=None, max_iter=100, tolerance=1e-5):
    """
    K-Means algorithm for clustering grayscale values in an image.

    Args:
        image (numpy.ndarray): 2D array representing the grayscale image.
        K (int): Number of clusters (i.e., number of means).
        initial_means (numpy.ndarray): Optional initial means (K values), if None, they are chosen randomly.
        max_iters (int): Maximum number of iterations to run the algorithm.
        tol (float): Convergence threshold based on the change in means.

    Returns:
        segmented_image (numpy.ndarray): 2D array representing the clustered image.
        means (numpy.ndarray): Final cluster means.
        num_iter: number of iterations
    """
    # Flatten the image to 1D array of grayscale values
    flat_image = image.flatten()

    # Initialize means (either given or randomly chosen)
    if initial_means is None:
        means = np.random.choice(flat_image, K, replace=False)
    else:
        means = initial_means

    # Store the previous means to check for convergence
    prev_means = np.zeros_like(means)
    num_iter = max_iter

    # Initialize cluster assignments
    labels = np.zeros(flat_image.shape, dtype=int)

    for iteration in range(max_iter):
        # Step 1: Assign each pixel to the nearest cluster
        for i, z in enumerate(flat_image):
            distances = np.abs(z - means)
            labels[i] = np.argmin(distances)

        # Step 2: Update the means based on the assigned clusters
        new_means = np.array([flat_image[labels == k].mean() if np.sum(labels == k) > 0 else means[k]
                              for k in range(K)])

        change = np.linalg.norm(new_means - prev_means)
        print(f"The {iteration + 1}-th iteration: {change}")

        # Check for convergence (if the change in means is below the tolerance)
        if change < tolerance:
            num_iter = iteration + 1
            print(f"Converged after {num_iter} iterations.")
            break
```

```
        prev_means = new_means

        # Update the cluster means
        means = new_means

    # Step 3: Create the segmented image based on the final labels
    segmented_image = np.reshape(means[labels], image.shape).astype(np.uint8)

    return segmented_image, means, num_iter
```

计算灰度直方图的函数:

```
def compute_histogram(image, num_bins=256):
    """
    Compute the grayscale histogram of an image.

    :param image: Grayscale image as a numpy array.
    :param num_bins: Number of bins for the histogram (default is 256).
    :return: The histogram and bin edges.
    """
    # Flatten the image and compute the histogram with the specified number of
bins
    histogram, bin_edges = np.histogram(image.ravel(), bins=num_bins, range=[0,
num_bins])
    return histogram, bin_edges
```

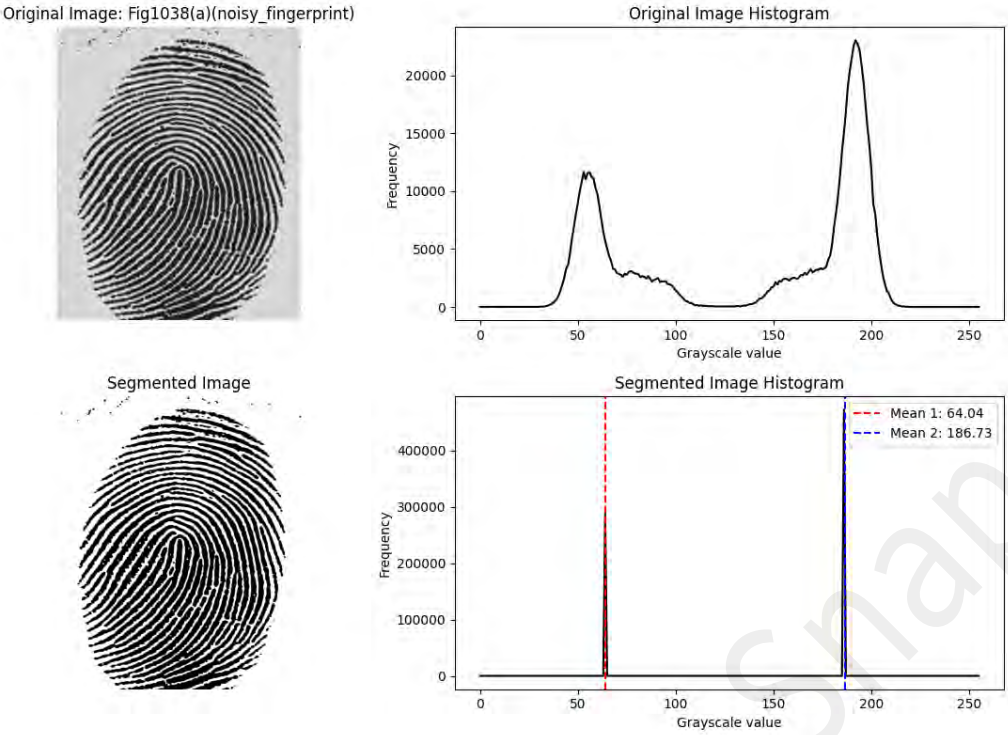
函数调用:

```
if __name__ == "__main__":

    option = 1
    if option == 1:
        image_path = 'Fig1038(a)(noisy_fingerprint).tif'
        image_name = 'Fig1038(a)(noisy_fingerprint)'
        K = 2
        initial_means = [60, 190]
        max_iter = 100
        tolerance = 1e-5
    else:
        image_path = 'noisy_Fig1038(a)(noisy_fingerprint).tif'
        image_name = 'noisy_Fig1038(a)(noisy_fingerprint)'
        K = 2
        initial_means = [60, 190]
        max_iter = 100
        tolerance = 1e-5

    # Load and convert the image to grayscale format
    image = Image.open(image_path).convert('L')
    image = np.array(image)  # Convert image to a numpy array for processing

    # Compute histogram of the original image
    original_histogram, bin_edges = compute_histogram(image)

    # Run K-means algorithm on the grayscale image
    segmented_image, final_means, num_iter = k_means_grayscale(image, K=K,
```

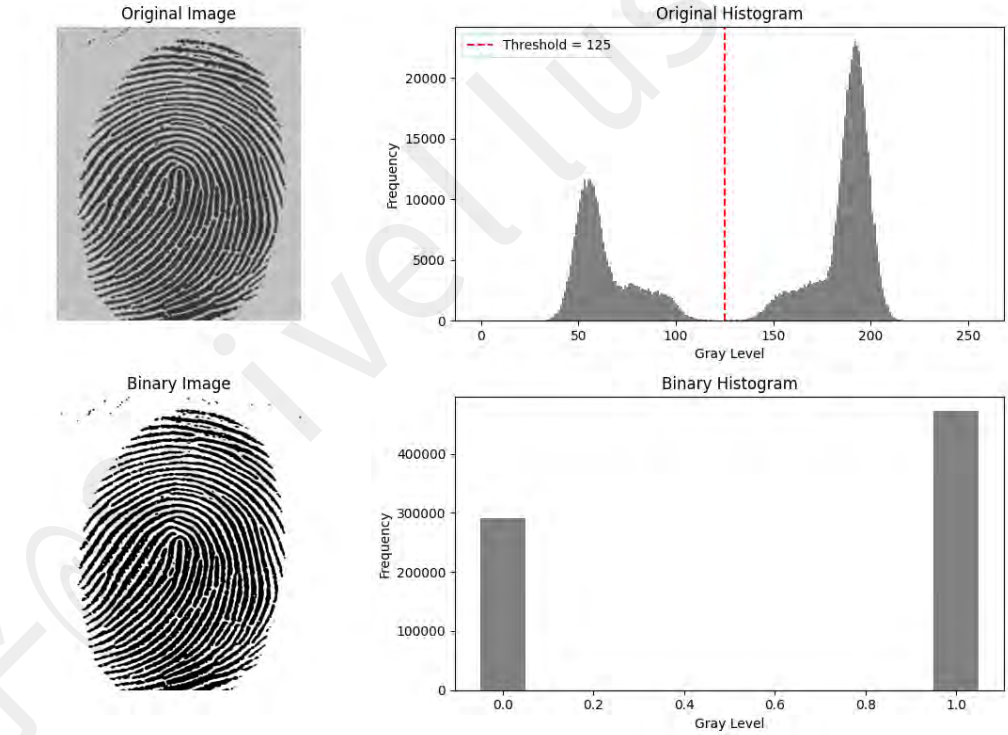
```
                                        initial_means=initial_means,
max_iter=max_iter, tolerance=tolerance)

    # Compute histogram of the segmented image
    segmented_histogram, _ = compute_histogram(segmented_image)

    # Display the results
    print("Final means (cluster centers):", final_means)

    plt.figure(figsize=(12, 8))

    # 1. Display original image
    plt.subplot(2, 2, 1)
    plt.title(f"Original Image: {image_name}")
    plt.imshow(image, cmap='gray')
    plt.axis('off')

    # 2. Display histogram of original image
    plt.subplot(2, 2, 2)
    plt.title("Original Image Histogram")
    plt.plot(bin_edges[:-1], original_histogram, color='black')
    plt.xlabel("Grayscale value")
    plt.ylabel("Frequency")

    # 3. Display segmented image
    plt.subplot(2, 2, 3)
    plt.title(f"Segmented Image (K-means: {num_iter} iterations)")
    plt.imshow(segmented_image, cmap='gray')
    plt.axis('off')

    # 4. Display histogram of segmented image
    plt.subplot(2, 2, 4)
    plt.title("Segmented Image Histogram")
    plt.plot(bin_edges[:-1], segmented_histogram, color='black')
    for i, mean in enumerate(final_means):
        plt.axvline(mean, linestyle='--', label=f"Mean {i+1}: {mean:.2f}")
    plt.xlabel("Grayscale value")
    plt.ylabel("Frequency")
    plt.legend()
    plt.tight_layout()
    save_path = f"{K}-means-{image_name}.png"
    plt.savefig(save_path)
    plt.show()
```

运行结果:
我们发现对于二值化问题来说，轻微的椒盐噪声的影响不是太大.

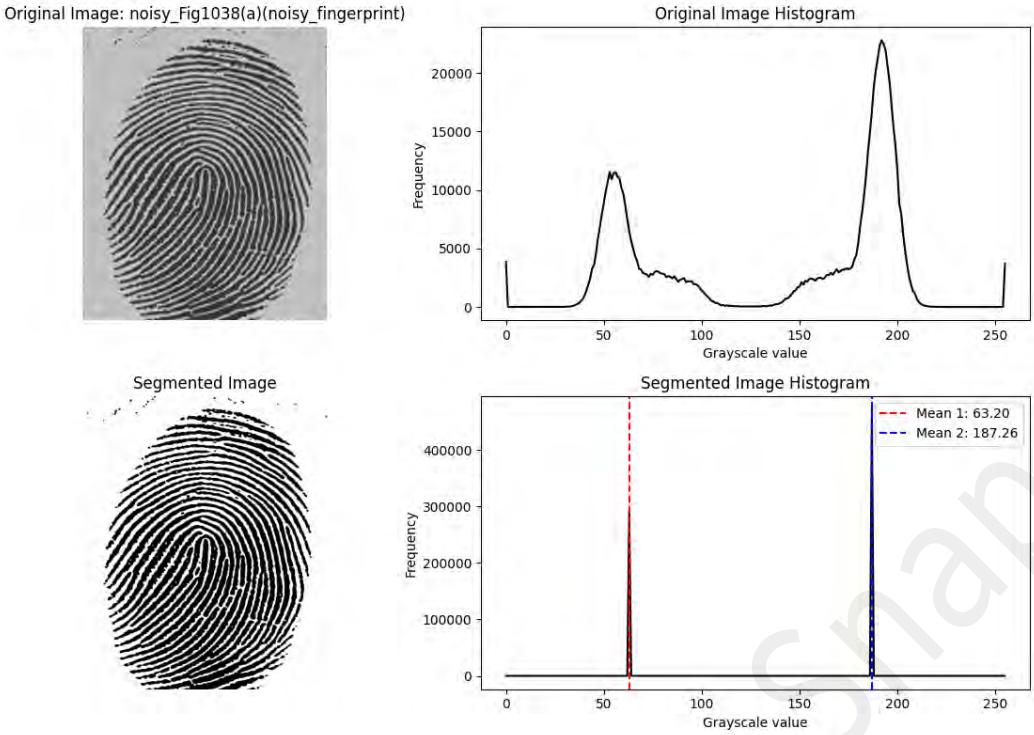
- **① 未添加椒盐噪声的指纹图像:**
  K-均值算法的结果:



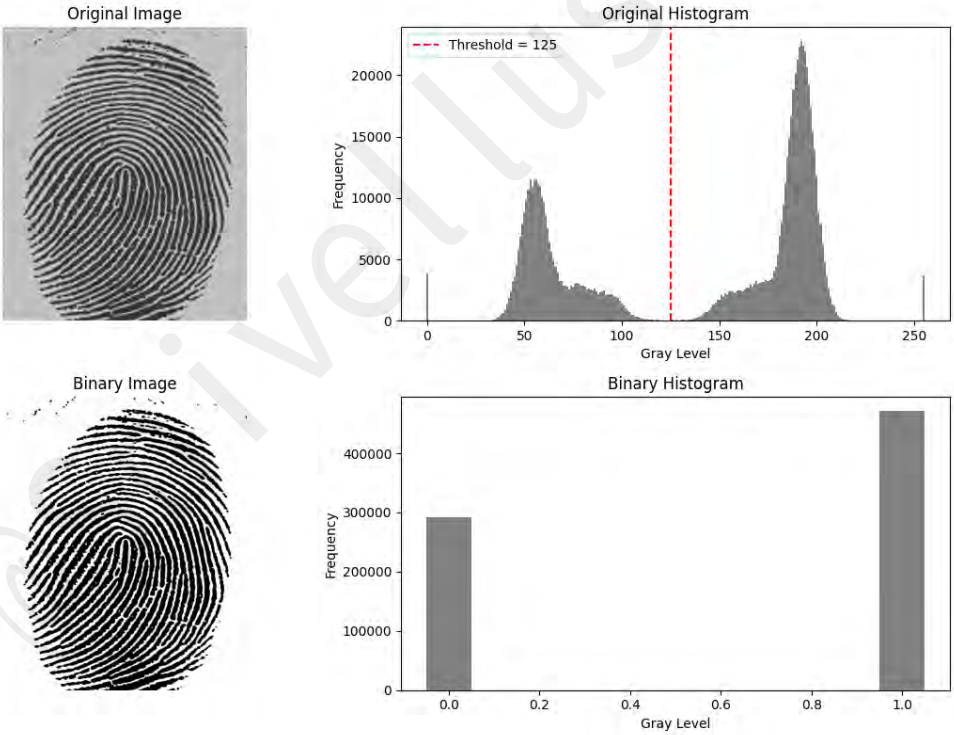
Ostu 算法的结果:



- **② 添加了椒盐噪声的指纹图像:**
  K-均值算法的结果:



Ostu 算法的结果:



Part (2)

测试多类分割.

Solution:
这里我们实现 GMM-EM 算法，并与 K-均值算法进行对比.

Gauss 混合模型参数估计的 EM 算法: (机器学习 周志华 图 9.6)
给定灰度值 $z_1, \ldots, z_n$

- **(1)** 初始化 Gauss 混合分布的模型参数 $\{(\alpha_k, \mu_k, \Sigma_k) : 1 \leq k \leq K\}$

- 可设置 $\alpha_1 = \cdots = \alpha_K = \frac{1}{K}$
- 可通过 K-means 聚类算法得到 $\mu_1, \ldots, \mu_K$ 的初值和初始聚类集合 $S_1, \ldots, S_K$
- 可设置 $S_1, \ldots, S_K$ 的样本协方差矩阵作为 $\Sigma_1, \ldots, \Sigma_K$ 的初值:

$$\Sigma_k := \frac{1}{|S_k|} \sum_{z \in S_k} (z - \mu_k)(z - \mu_k)^{\mathrm{T}}$$

- **(2)** 重复迭代直至模型参数的变化量小于某个预设定的阈值:
  - **① Expectation Step**
    遍历 $i = 1, \ldots, n$，计算 z_i 由第 $k = 1, \ldots, K$ 个 Gauss 混合成分生成的后验概率:

$$\gamma_i^{(k)} := \frac{\alpha_k \cdot p(z_i | \mu_k, \Sigma_k)}{\sum_{k=1}^{K} \alpha_k \cdot p(z_i | \mu_k, \Sigma_k)} \ (\forall\, k = 1, \ldots, K)$$

  - **② Maximization Step**
    遍历 $k = 1, \ldots, K$，更新模型参数 $(\alpha_k, \mu_k, \Sigma_k)$

$$\mu_k = \frac{\sum_{i=1}^{n} \gamma_i^{(k)} z_i}{\sum_{i=1}^{n} \gamma_i^{(k)}}$$

$$\Sigma_k^2 = \frac{\sum_{i=1}^{n} \gamma_i^{(k)} (z_i - \mu_k)(z_i - \mu_k)^{\mathrm{T}}}{\sum_{i=1}^{n} \gamma_i^{(k)}}$$

$$\alpha_k = \frac{\sum_{i=1}^{n} \gamma_i^{(k)}}{n}$$

- **(3)** 计算 $z_1, \ldots, z_n$ 的类标记，得到最终的 K 个聚类集合 $S_1, \ldots, S_K$

$$\mathrm{label}(z_i) = \arg \max_{k \in \{1, \ldots, K\}} \gamma_i^{(k)} \ (\forall\, i = 1, \ldots, n)$$

GMM-EM 算法的 Python 实现如下:

```
def gmm_em_grayscale(image, K, initial_means=None, max_iter=1000, tolerance=1e-5):
    """
    Gaussian Mixture Model (GMM) EM algorithm for clustering grayscale values in an image.

    Args:
        image (numpy.ndarray): 2D array representing the grayscale image.
        K (int): Number of clusters (i.e., number of Gaussian components).
        initial_means (numpy.ndarray): Optional initial means (K values), if None, they are chosen randomly.
        max_iters (int): Maximum number of iterations to run the algorithm.
        tolerance (float): Convergence threshold based on the change in log-likelihood.

    Returns:
        segmented_image (numpy.ndarray): 2D array representing the clustered image.
        means (numpy.ndarray): Final cluster means.
        covariances (numpy.ndarray): Final cluster covariance matrices.
        alphas (numpy.ndarray): Final mixture component weights.
        num_iter: number of iterations
```

```
    """
    # Flatten the image to 1D array of grayscale values
    flat_image = image.flatten()
    n = flat_image.shape[0]

    # Initialize means, covariances, and mixture coefficients
    if initial_means is None:
        means = np.random.choice(flat_image, K, replace=False)
    else:
        means = np.array(initial_means)

    # Initialize weights (alphas) and covariance matrices (Sigma)
    alphas = np.ones(K) / K  # Uniform initial weights
    covariances = np.array([np.var(flat_image)] * K)  # Initial covariances set
to the variance of the image

    # Initialize responsibilities (gamma values)
    gamma = np.zeros((n, K))
    num_iter = max_iter

    prev_means = np.copy(means)
    prev_covariances = np.copy(covariances)
    prev_alphas = np.copy(alphas)

    for iteration in range(max_iter):
        # E-step: Compute responsibilities (gamma values)
        for k in range(K):
            # Calculate the Gaussian probability density function for each pixel
            pdf = multivariate_normal.pdf(flat_image, mean=means[k],
cov=covariances[k])
            gamma[:, k] = alphas[k] * pdf

        # Normalize the responsibilities (soft assignment of pixels to clusters)
        gamma = gamma / np.sum(gamma, axis=1)[:, np.newaxis]

        # M-step: Update the model parameters (means, covariances, alphas)
        for k in range(K):
            # Update the mean for each cluster
            means[k] = np.sum(gamma[:, k] * flat_image) / np.sum(gamma[:, k])

            # Update the covariance for each cluster
            covariances[k] = np.sum(gamma[:, k] * (flat_image - means[k])**2) /
np.sum(gamma[:, k])

            # Update the weight (alpha) for each cluster
            alphas[k] = np.sum(gamma[:, k]) / n

        # Convergence check based on the variation of means, covariances, and
alphas
        mean_change = np.linalg.norm(means - prev_means)
        cov_change = np.linalg.norm(covariances - prev_covariances)
        alpha_change = np.linalg.norm(alphas - prev_alphas)

        # Combine changes to determine convergence
        total_change = mean_change + cov_change + alpha_change
        total_change = total_change / (np.linalg.norm(means) +
np.linalg.norm(covariances) + np.linalg.norm(alphas))
        print(f"The {iteration+1}-th iteration: {total_change}")
```

```
        if total_change < tolerance:
            num_iter = iteration + 1
            print(f"Converged after {num_iter} iterations.")
            break

        # Update previous values for next iteration
        prev_means = np.copy(means)
        prev_covariances = np.copy(covariances)
        prev_alphas = np.copy(alphas)

    # Assign the final cluster labels based on the highest responsibility
    labels = np.argmax(gamma, axis=1)

    # Step 3: Create the segmented image based on the final labels
    segmented_image = np.reshape(means[labels], image.shape).astype(np.uint8)

    return segmented_image, means, covariances, alphas, num_iter
```

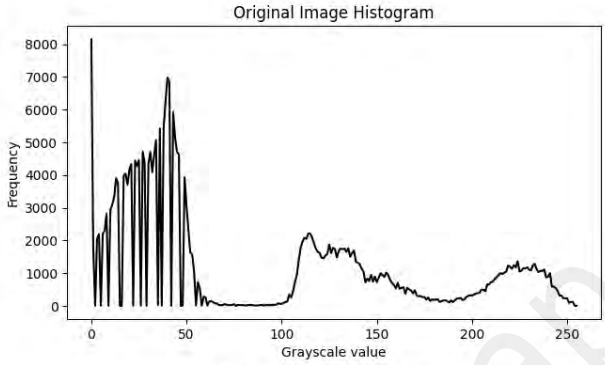
GMM-EM 算法的函数调用:

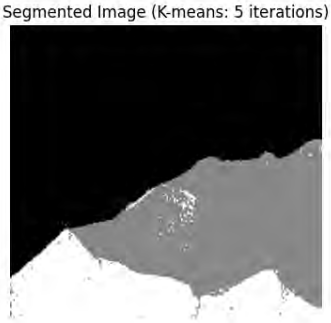
```
if __name__ == "__main__":

    option = 4
    if option == 1:
        image_path = 'Fig1045(a)(iceberg).tif'
        image_name = 'Fig1045(a)(iceberg)'
        K = 3
        initial_means = [28.74370715, 131.95402687, 221.96702829] # K-means
result
        max_iter = 1000
        tolerance = 1e-5
    elif option == 2:
        image_path = 'noisy_Fig1045(a)(iceberg).tif'
        image_name = 'noisy_Fig1045(a)(iceberg)'
        K = 3
        initial_means = [28.74370715, 131.95402687, 221.96702829] # K-means
result
        max_iter = 1000
        tolerance = 1e-5
    elif option == 3:
        image_path = 'Fig1016(a)(building_original).tif'
        image_name = 'Fig1016(a)(building_original)'
        K = 4
        initial_means = [47.21778267, 101.72042789, 166.50053224, 234.75427634]
# K-means result
        max_iter = 1000
        tolerance = 1e-5
    elif option == 4:
        image_path = 'noisy_Fig1016(a)(building_original).tif'
        image_name = 'noisy_Fig1016(a)(building_original)'
        K = 4
        initial_means = [47.21778267, 101.72042789, 166.50053224, 234.75427634]
# K-means result
        max_iter = 1000
        tolerance = 1e-5
```

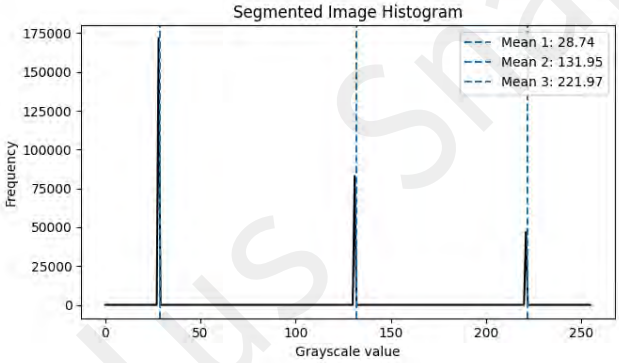
```
    # Load and convert the image to grayscale format
    image = Image.open(image_path).convert('L')
    image = np.array(image)  # Convert image to a numpy array for processing

    # Compute histogram of the original image
    original_histogram, bin_edges = compute_histogram(image)

    # Run GMM-EM algorithm on the grayscale image
    segmented_image, final_means, final_covariances, final_alphas, num_iter =
gmm_em_grayscale(image,
                                          K=K, initial_means=initial_means,
max_iter=max_iter, tolerance=tolerance)

    # Compute histogram of the segmented image
    segmented_histogram, _ = compute_histogram(segmented_image)

    # Display the results
    print("Final means (cluster centers):", final_means)
    print("Final covariances:", final_covariances)
    print("Final alphas (weights):", final_alphas)

    plt.figure(figsize=(12, 8))

    # 1. Display original image
    plt.subplot(2, 2, 1)
    plt.title(f"Original Image: {image_name}")
    plt.imshow(image, cmap='gray')
    plt.axis('off')

    # 2. Display histogram of original image
    plt.subplot(2, 2, 2)
    plt.title("Original Image Histogram")
    plt.plot(bin_edges[:-1], original_histogram, color='black')
    plt.xlabel("Grayscale value")
    plt.ylabel("Frequency")

    # 3. Display segmented image
    plt.subplot(2, 2, 3)
    plt.title(f"Segmented Image (GMM-EM: {num_iter} iterations)")
    plt.imshow(segmented_image, cmap='gray')
    plt.axis('off')

    # 4. Display histogram of segmented image
    plt.subplot(2, 2, 4)
    plt.title("Segmented Image Histogram")
    plt.plot(bin_edges[:-1], segmented_histogram, color='black')
    for i, mean in enumerate(final_means):
        plt.axvline(mean, linestyle='--', label=f"Mean {i+1}: {mean:.2f}")
    plt.xlabel("Grayscale value")
    plt.ylabel("Frequency")
    plt.legend()

    plt.tight_layout()

    # Save the results to file
    save_path = f"{K}-GMM-{image_name}.png"
    plt.savefig(save_path)
    plt.show()
```

运行结果:

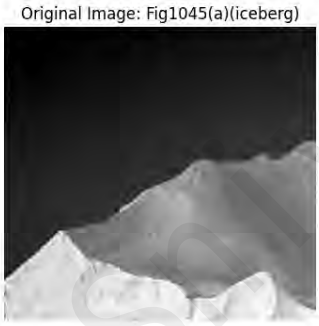
- **① 未添加椒盐噪声的冰山图像:**
  K-均值算法的结果:

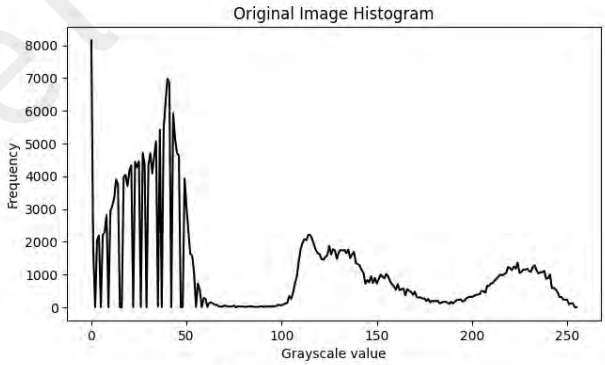


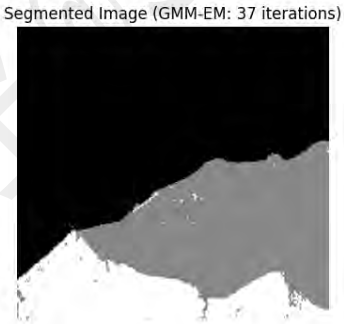


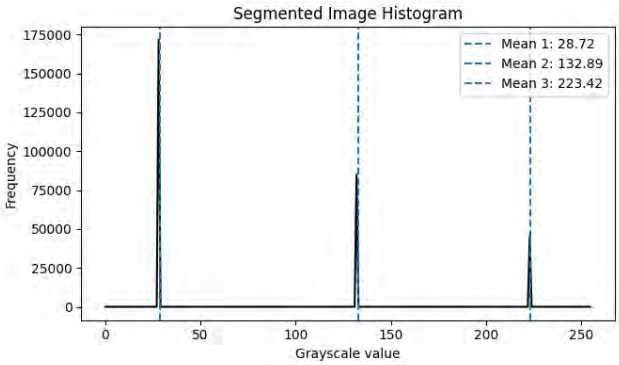


  GMM-EM 算法的结果:

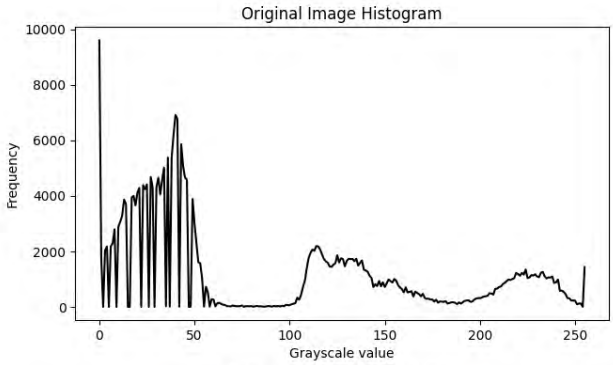


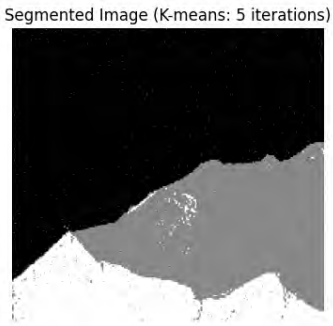


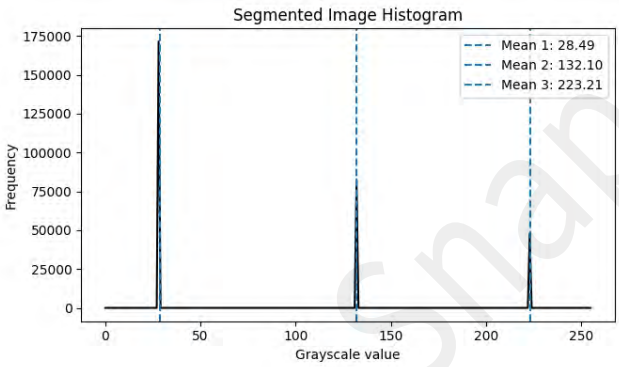




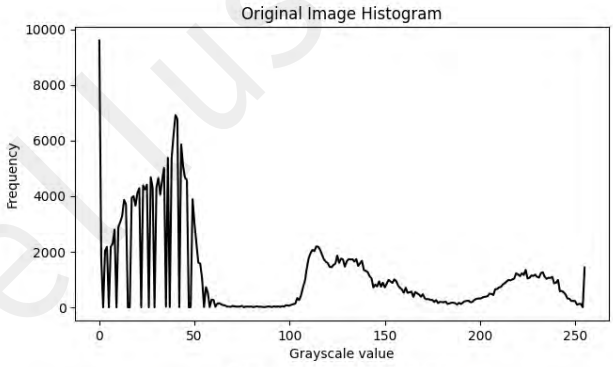
- **② 添加了椒盐噪声的冰山图像:**
  K-均值算法的结果:

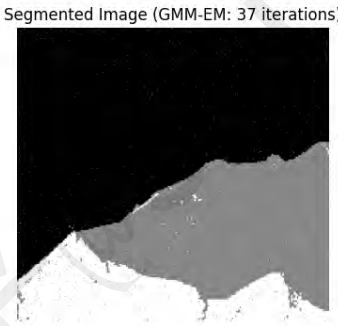


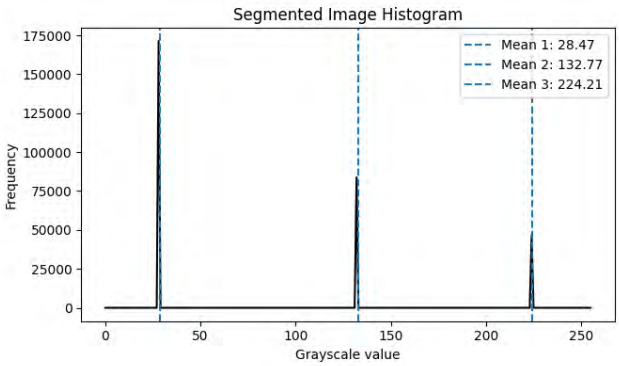




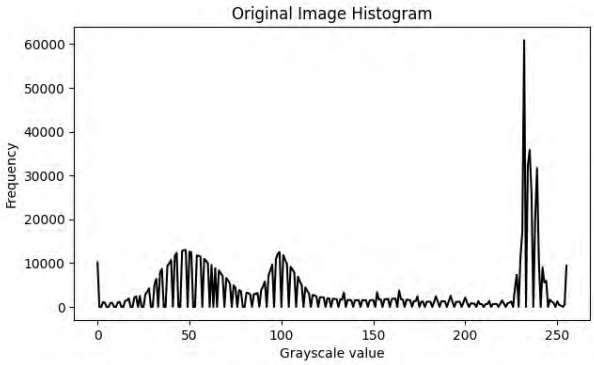
GMM-EM 算法的结果:

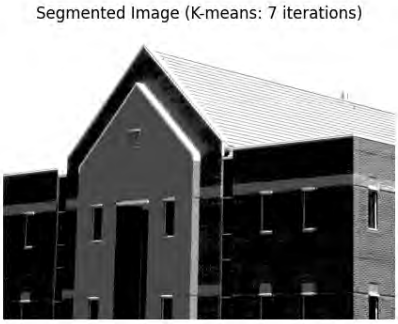


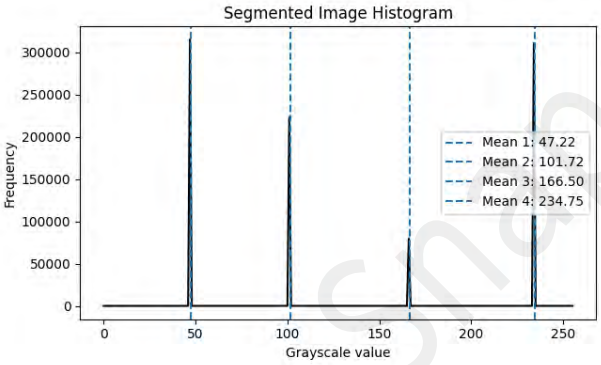




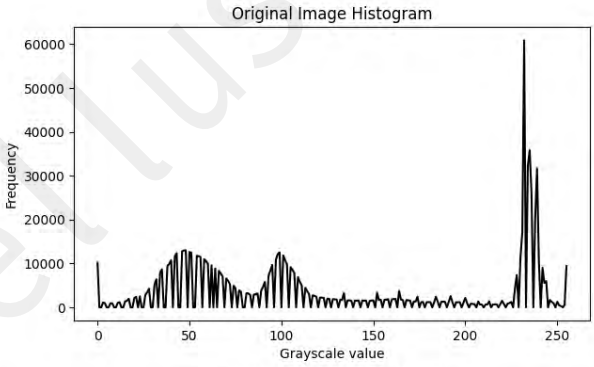
- **③ 未添加椒盐噪声的建筑图像:**
  K-均值算法的结果:

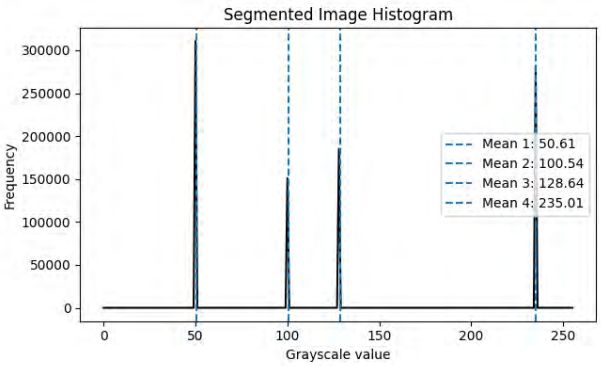






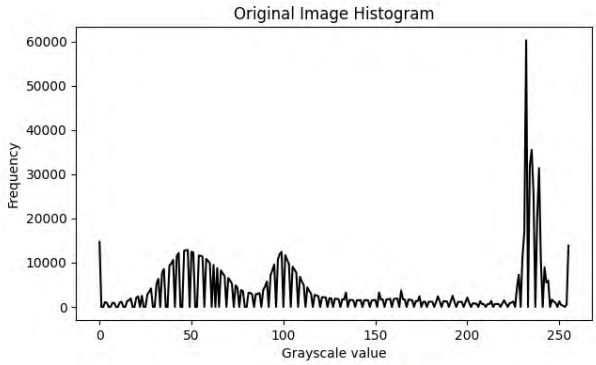
GMM-EM 算法的结果:

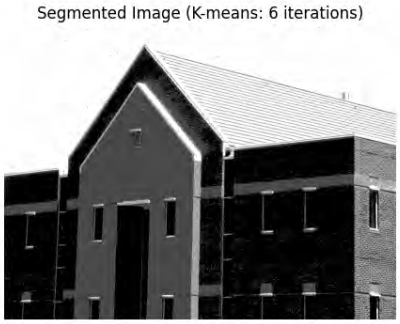


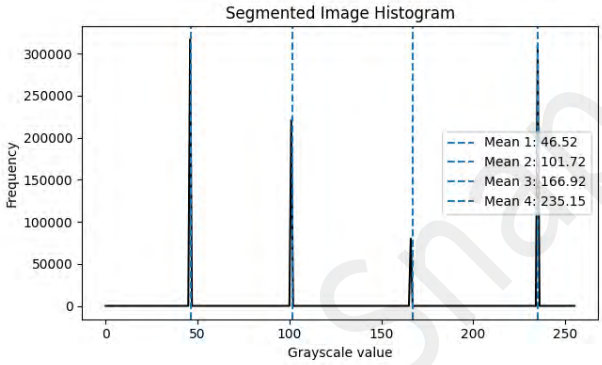


- **④ 添加了椒盐噪声的建筑图像:**

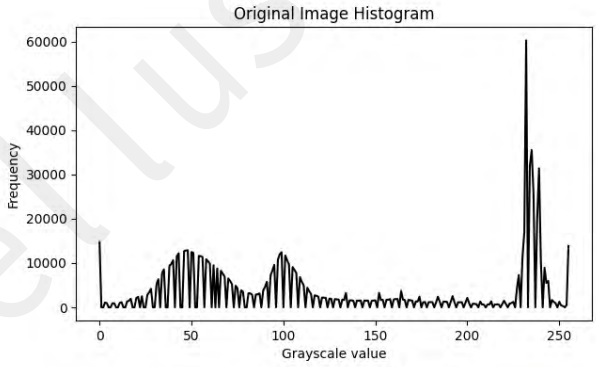
  K-均值算法的结果:

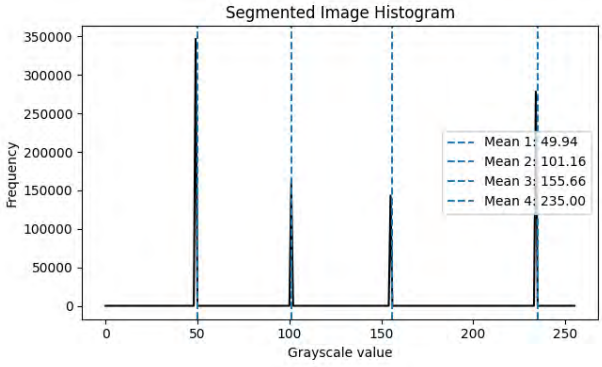






GMM-EM 算法的结果:





Part (3)

针对噪声图像，讨论为什么分割的结果不准确，有什么方法可以取得更好的分割结果.
(备注: 不要求实现该方法，只是讨论)

Solution:

由于 K-均值算法和 GMM-EM 算法在进行图像分割时都不可避免地把噪声考虑进去，
而噪声又通常不能提供有价值的信息，因而造成算法的图像分割效果变差.

对于被椒盐噪声污染的图像，我们可以先使用中值滤波器进行去噪，然后再进行图像分割.

- **中值滤波器:**
  记 N_{xy} 为以 (x,y) 为中心，大小为 $m \times n$ (设 m,n 均为奇数) 的矩形窗口中的坐标集合.
  基于中值滤波器的复原图像 $\hat{f}(x,y)$ 的定义为:

$$\hat{f}(x,y) := \underset{(s,t)\in N_{xy}}{\text{median}}\{g(s,t)\}$$

  与大小相同的线性平滑滤波器相比，它能有效地降低某些随机噪声，且模糊度要小得多.
  而且它可以很好地消除椒盐噪声 (无论是单极冲激噪声还是双极冲激噪声)
  (不过这要求椒盐噪声的空间密度比较低，具体来说 $p_{\text{pepper}}, p_{\text{salt}} < 0.2$)
  它是在图像处理中最常使用的次序滤波器.
  中值可以从一个位置到下一个位置迭代地更新，从而减少计算开销.

如果椒盐噪声的强度很大 (具体来说 $p_{\text{pepper}}, p_{\text{salt}} > 0.2$)，则我们可以使用自适应中值滤波器进行去噪.

- **自适应中值滤波器:**
  记矩形邻域 N_{xy} 的最小灰度值为 $z_{\min}$，最大灰度值为 $z_{\max}$，灰度中值为 z_{med}.
  - ① 若 $z_{\min} < z_{\text{med}} < z_{\max}$，则移步 ②
    否则增大 N_{xy} 的尺寸，再进行 ①，直至移步 ② 或达到最大尺寸.
    达到最大尺寸时输出 z_{med}
    (也可以输出 z_{xy}，这样做的结果更清晰，但无法检测到常数背景中嵌入的盐粒或胡椒噪声)
  - ② 若 $z_{\min} < z_{xy} < z_{\max}$，则输出 z_{xy}
    否则输出 z_{med}

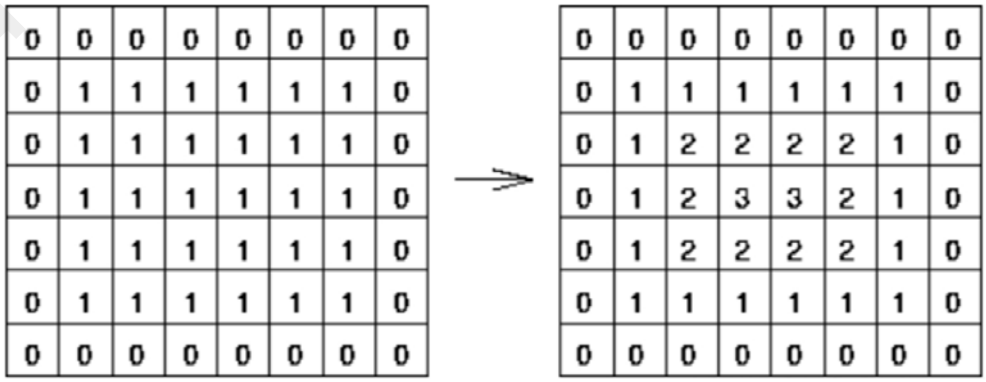
  最值和中值可以从一个位置到下一个位置迭代地更新，从而减少计算开销.

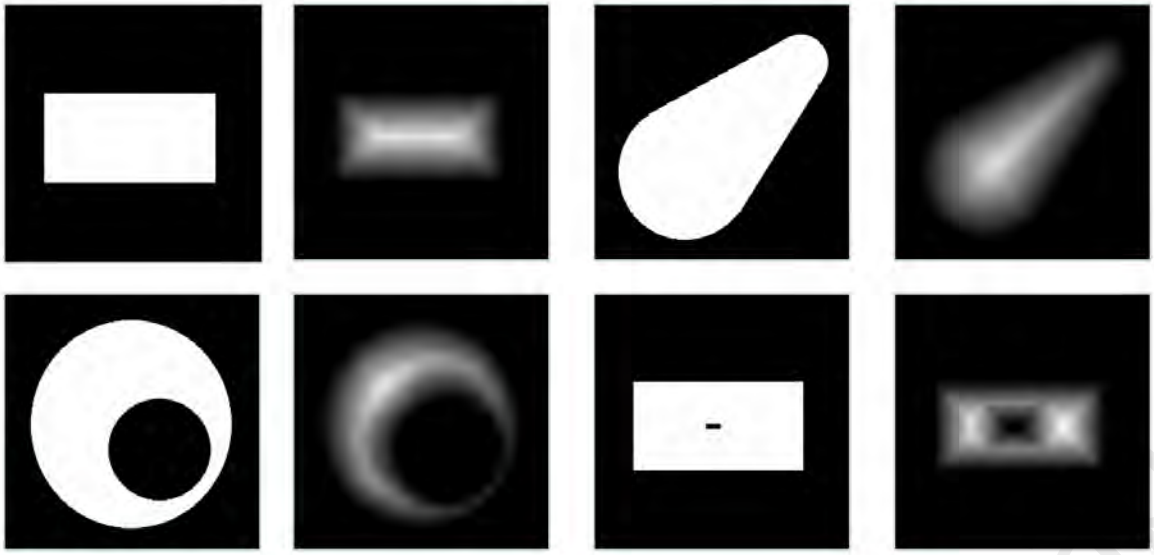
Problem 2

请使用课程学习的形态学操作实现以下二值图像的补洞和离散点去除:

(0) 距离变换

距离变换 (Distance Transform) 算法:
距离变换是一种图像处理技术，用于计算图像中每个前景像素到边缘的距离.





实现距离变换的方法有很多，常见的方式包括基于**动态规划**的算法.
它通过逐步更新每个像素的距离信息来避免直接计算每个像素到所有前景像素的距离，从而提高计算效率.

- **前向和后向扫描**: 通过前向扫描计算每个像素点到最近前景的距离，再通过后向扫描进一步优化.
  通常来说，前向扫描在一个方向上进行 (如从左上到右下)，后向扫描在反方向上进行 (如从右下到左上)
- **水平和垂直扫描**: 将图像分解成行和列的扫描，逐行逐列进行距离计算和更新，
  确保每个像素的距离基于其周围邻域的已有计算结果.

其时间复杂度通常为 $O(\text{Width} \cdot \text{Height})$
这是因为每个像素都要参与计算，且每次计算依赖于前一个像素的信息，因此处理复杂度是线性的.

OpenCV 库提供了距离变换算法的一个实现:

```
cv2.distanceTransform(src, distanceType, maskSize, dst=None)
```

- `src`: 二值输入图像
- `distanceType`: 距离度量类型，常见的有:
  - `cv2.DIST_L2`: Euclid 距离
  - `cv2.DIST_L1`: Manhattan 距离
  - `cv2.DIST_C`: Chebyshev 距离
- `maskSize`: 用于计算距离的掩膜大小，决定计算时邻域的大小，常用值为 `3`、`5`、`7`
- `dst`: 输出图像，它将保存计算得到的距离图，每个像素值表示该像素到最近前景像素的距离.

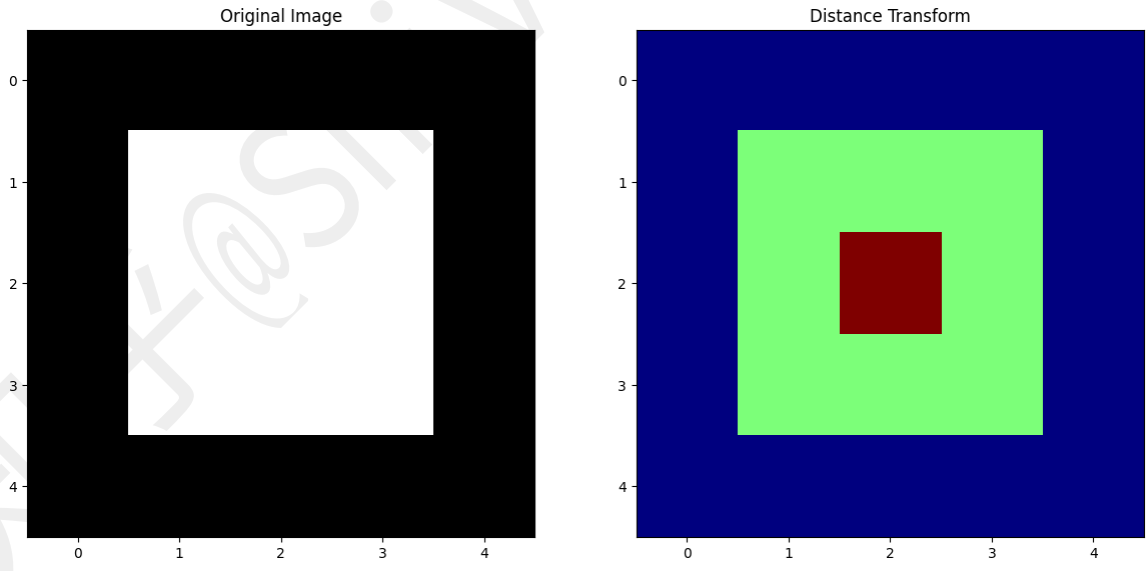
函数调用:

```
# 创建一个简单的0-1矩阵 (5x5的二值矩阵)
image = np.array([
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
], dtype=np.uint8)
```

```python
# 计算距离变换 (使用 Chebyshev 距离)
distance = cv2.distanceTransform(image, cv2.DIST_C, 5).astype(np.uint8)

# 显示距离变换结果
plt.subplot(121), plt.imshow(image, cmap='gray'), plt.title('Original Image')
plt.subplot(122), plt.imshow(distance, cmap='jet'), plt.title('Distance
Transform')
plt.show()

# 打印距离变换矩阵
print("Original Image:")
print(image)
print("\nDistance Transform Result:")
print(distance)
```

运行结果:

```
Original Image:
[[0 0 0 0 0]
 [0 1 1 1 0]
 [0 1 1 1 0]
 [0 1 1 1 0]
 [0 0 0 0 0]]

Distance Transform Result:
[[0 0 0 0 0]
 [0 1 1 1 0]
 [0 1 2 1 0]
 [0 1 1 1 0]
 [0 0 0 0 0]]
```



(1) 腐蚀

我们定义结构元 B 对目标集合 (前景像素集) A 的**腐蚀** (erosion) 为:

$$A \ominus B := \{(x, y) : (x, y) \in A \text{ such that } B_{\text{translate}(x,y)} \subseteq A\}$$

它是一种收缩、细化的运算.
腐蚀运算还有以下等价定义:

$$A \ominus B := \{(x,y) \in \mathbb{Z}^2 : B_{\text{translate}(x,y)} \cap A^c = \emptyset\}$$
$$A \ominus B := \{(x,y) \in \mathbb{Z}^2 : (x+x_0, y+y_0) \in A \text{ for all } (x_0,y_0) \in B\}$$
$$A \ominus B := \bigcap_{(x,y)\in B} A_{\text{translate}(-x,-y)}$$

简单起见，我们可以使用距离来实现腐蚀操作，即将背景边界像素的邻域内的像素都置为背景像素.
通常来说，在二值图像中我们使用白色 (255) 来标记前景像素，使用黑色 (0) 来标记背景像素.
但本题使用的测试图片违背了这个约定，因此需要将其逆转.

- **① 不使用库函数的实现 (速度相对较慢):**

```
# Erosion function
@njit
def erode(image: np.array, threshold: int):
    """
    Parameters:
    image: Grayscale array of the image
    threshold: Threshold value to determine erosion effect (distance
threshold)
    Returns: Grayscale array of the image after erosion
    """
    eroded_image = np.zeros(image.shape)
    M, N = image.shape
    for i in range(M):
        for j in range(N):
            eroded_image[i, j] = np.min(
                image[max(0,i-threshold):min(M+1,i+threshold+1),
                      max(0,j-threshold):min(N+1,j+threshold+1)])
    return eroded_image
```

- **② 使用 `openCV` 库的 `distanceTransform` 的实现 (速度相对较快):**

```
# Erosion function
def erode(image: np.array, threshold: int):
    """
    Parameters:
    image: Grayscale array of the image
    threshold: Threshold value to determine erosion effect (distance
threshold)
    Returns: Grayscale array of the image after erosion
    """
    distance_transform = compute_distance_transform(image)
    image[(distance_transform > 0) & (distance_transform <= threshold)] = 0
    return image
```

(2) 膨胀

我们定义结构元 B 对目标集合 (前景像素集) A 的**膨胀** (dilation):

$$A \oplus B := \{(x,y) \in \mathbb{Z}^2 : [(B_{\text{reflect}})_{\text{translate}(x,y)} \cap A] \subseteq A\}$$

它是一种增长、粗化的运算.
与腐蚀一样，将结构元 B 视为卷积核时，上述定义将更加直观.
但要记住，膨胀是以集合运算为基础的，因此是非线性运算; 而卷积是乘积求和，是线性运算.

其等效定义为:

$$A \oplus B := \{(x,y) \in \mathbb{Z}^2 : (x,y) = (a_1,a_2) + (b_1,b_2) \text{ for some } (a_1,a_2) \in A \text{ and } (b_1,b_2) \in B\}$$

$$A \oplus B := \bigcup_{(x,y) \in B} A_{\text{translate}(x,y)}$$

简单起见，我们可以使用距离来实现膨胀操作，即将前景边界像素的邻域内的像素都置为前景像素.
通常来说，在二值图像中我们使用白色 (255) 来标记前景像素，使用黑色 (0) 来标记背景像素.
但本题使用的测试图片违背了这个约定，因此需要将其逆转.

- **① 不使用库函数的实现 (速度相对较慢):**

```python
# Dilation function
@njit
def dilate(image: np.array, threshold: int):
    """
    Parameters:
    image: Grayscale array of the image
    threshold: Threshold value to determine dilation effect (distance
threshold)
    Returns: Grayscale array of the image after dilation
    """
    dilated_image = np.zeros(image.shape)
    M, N = image.shape
    for i in range(M):
        for j in range(N):
            dilated_image[i, j] = np.max(
                image[max(0,i-threshold):min(M+1, i+threshold+1),
                      max(0,j-threshold):min(N+1, j+threshold+1)])
    return dilated_image
```

- **② 使用 `openCV` 库的 `distanceTransform` 的实现 (速度相对较快):**

```python
# Dilation function
def dilate(image: np.array, threshold: int):
    """
    Parameters:
    image: Grayscale array of the image
    threshold: Threshold value to determine dilation effect (distance
threshold)
    Returns: Grayscale array of the image after dilation
    """
    distance_transform = compute_distance_transform(255 - image)
    image[(distance_transform > 0) & (distance_transform <= threshold)] =
255
    return image
```

(3) 开运算 & 闭运算

膨胀扩展集合的组成部分，而腐蚀缩小集合的组成部分.
现在我们定义另外两个重要的形态学运算: **开运算** (opening) 和**闭运算** (closing)

- 开运算通常平滑物体的轮廓，断开狭窄的狭颈，消除细长的突出物.
  结构元 B 对目标集合 A 的开运算定义为:

  $$A \circ B := (A \ominus B) \oplus B$$

  即结构元 B 首先对 A 腐蚀，再对 A 膨胀.

- 闭运算同样平滑轮廓，但与开运算相反，它通常弥合狭窄的断裂和细长的沟壑，消除小孔，并填补轮廓中的缝隙.
  结构元 B 对目标集合 A 的闭运算定义为:

  $$A \bullet B := (A \oplus B) \ominus B$$

  即结构元 B 首先对 A 膨胀，再对 A 腐蚀.

通常来说，在二值图像中我们使用白色 (255) 来标记前景像素，使用黑色 (0) 来标记背景像素.
但本题使用的测试图片违背了这个约定，因此需要将其逆转.
Python 代码为:

```python
# Opening operation, used for removing noise
def opening(image: np.array, threshold: int):
    """
    Parameters:
    image: Grayscale array of the image
    block_size: Size of the structuring element, measured according to the
Chebyshev distance;
    Returns: Grayscale array of the image after performing "opening" (erosion
followed by dilation)
    """
    image = erode(image, threshold)   # First erosion
    image = dilate(image, threshold)  # Then dilation
    return image

# Closing operation, used for filling holes
def closing(image: np.array, threshold: int):
    """
    Parameters:
    image: Grayscale array of the image
    threshold: Threshold value to determine closing effect (distance threshold);
    Returns: Grayscale array of the image after performing "closing" (dilation
followed by erosion)
    """
    image = dilate(image, threshold)  # First dilation
    image = erode(image, threshold)   # Then erosion
    return image
```

(4) 运行结果

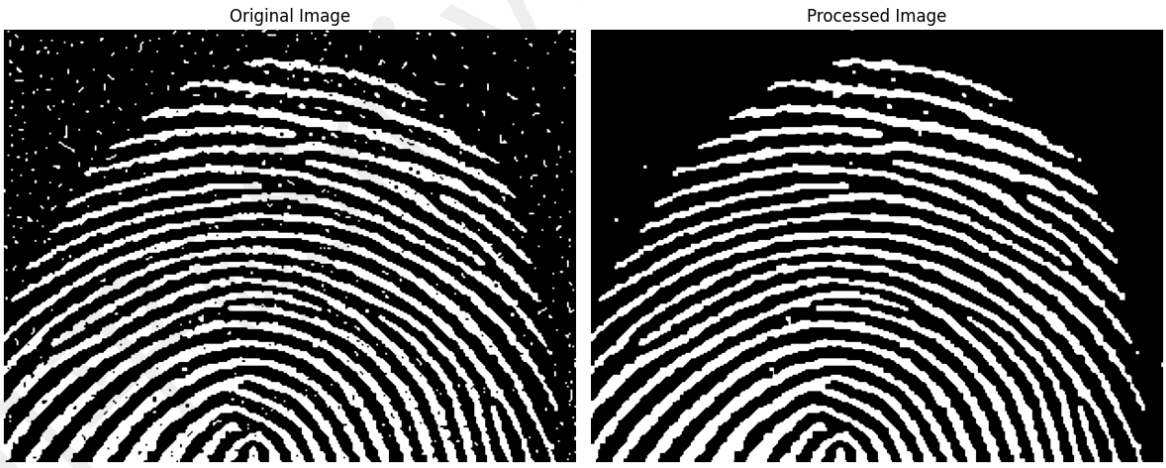
通常来说，在二值图像中我们使用白色 (255) 来标记前景像素，使用黑色 (0) 来标记背景像素.
但本题使用的测试图片违背了这个约定，因此需要将其逆转.
函数调用:

```
if __name__ == '__main__':

    # First, import the image and convert it to a numpy array
    option = 1
    if option == 1:
        image_path = 'zmic_fdu_noise.bmp'
        image_name = 'zmic_fdu_noise'
        open_threshold = 3
        close_threshold = 3
        inverse = True
    else:
        image_path = 'DIP 09.11(a)(noisy_fingerprint).bmp'
        image_name = 'DIP 09.11(a)(noisy_fingerprint)'
        open_threshold = 3
        close_threshold = 4
        inverse = False

    # Open the image, convert it to grayscale and then convert to a numpy array
    image = Image.open(image_path).convert('L')  # Convert the image to
grayscale ('L' mode)
    image = np.array(image)  # Convert the grayscale image to a numpy array
    print(np.where((image > 0) & (image < 255)))
    if inverse == True:
        image = 255 - image # Invert the pixel values (background becomes
foreground and vice versa)
    processed_image = np.copy(image)

    if inverse == True:
        # Perform closing operation on the image to fill holes
        processed_image = closing(processed_image, threshold=close_threshold)
        # Perform opening operation on the image to remove noise
        processed_image = opening(processed_image, threshold=open_threshold)
    else:
        # Perform opening operation on the image to remove noise
        processed_image = opening(processed_image, threshold=open_threshold)
        # Perform closing operation on the image to fill holes
        processed_image = closing(processed_image, threshold=close_threshold)

    # Invert the image back to its original form (foreground becomes background
and vice versa)
    if inverse == True:
        image = 255 - image
        processed_image = 255 - processed_image

    # Save result
    save_path = f"processed_{image_name}.bmp"
    Image.fromarray(processed_image.astype(np.uint8)).save(save_path,
format="BMP")

    # Plot the original and processed images side by side for comparison
    plt.figure(figsize=(12, 6))
```

```
    # Plot the original image
    plt.subplot(1, 2, 1)
    plt.imshow(image, cmap='gray')
    plt.title("Original Image")
    plt.axis('off')  # Hide axis ticks

    # Plot the processed image
    plt.subplot(1, 2, 2)
    plt.imshow(processed_image, cmap='gray')
    plt.title("Processed Image")
    plt.axis('off')  # Hide axis ticks

    # Display the comparison
    plt.tight_layout()
    save_path = f"Comparision-{image_name}.png"
    plt.savefig(save_path)
    plt.show()
```

运行结果: (设置 `option = 1`)

我们测试另一幅图像 (设置 `option = 2`)
这副图像符合惯例: 使用白色 (255) 来标记前景像素，使用黑色 (0) 来标记背景像素
因此不需要逆转.
(图片来源: Digital Image Process (3rd Edition, R. Gonzalez, R. Woods) Figure 9.11(a))



The End