Contents

Objectives:

- To learn about fingerprints
- To learn about thresholding
- To acquire knowledge about morphological operations
- To learn about minutiae points of fingerprints
- To learn how to match two fingerprints

Introduction:

Fingerprint matching in image processing involves comparing two fingerprint images to determine if they are from the same finger. This process is essential in various applications like biometric authentication, forensic investigations, and access control systems. The fingerprint recognition systems are similar in all implementation steps which represented by pre and post processing of fingerprint images and extract features.

Morphological operations are a set of image processing techniques that process images based on their shapes. They are particularly useful for enhancing fingerprint images, which often contain complex ridge patterns.

Minutiae points are critical features in fingerprint images used for fingerprint recognition and matching. These points represent specific types of ridge discontinuities in fingerprint patterns, and the two most common types are ridge endings and bifurcations.

Fingerprint matching in raw image processing involves several stages: image acquisition, preprocessing, feature extraction, and matching.

Methodology:

The steps of fingerprint matching using morphological operations are:

- Image acquisition
- Preprocessing
- Feature extraction
- Matching

Image acquisition:

Two images are taken for matching using the cv2 function from the folder.

Fig 1: Input Image1

Fig 2: Input Image2

Preprocessing:

The steps of preprocessing for two fingerprint images matching:

- Image rotation
- Filtering
- Thresholding

- Morphological operations

Image rotation:

Rotating fingerprints by 90 degrees during the matching process is performed to handle the variability in the orientation of fingerprint images. When fingerprint images are captured, they can be oriented in different directions due to how the finger is placed on the scanner. Rotational variations can make direct matching of minutiae points challenging. These are the reasons why 90-degree rotation is considered during fingerprint matching.

Fig 3: Image rotation

Filtering:

In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. The equation of Gaussian function is:

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

Fig 4: Filtered Images

Thresholding:

This method is a form of adaptive thresholding, where the threshold value is determined based on the local mean of the pixel values in the neighborhood of each pixel. The function adaptive_threshold calculates the mean of a block of pixels centered around each pixel and subtracts a constant value C from it to determine the threshold for that particular pixel.

Fig 5: Thresholded Images

Morphological operations:

Morphological operations apply a structuring element to an input image, creating an output image of the same size. In a morphological operation, the value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors. The morphological operation here has been used:

- Closing
- Skeletonization

Closing:

It is a morphological operation that is a combination of dilation followed by erosion using the same structuring element. It is typically used to close small holes and gaps in the foreground (white) regions of a binary image and to connect disjointed elements.

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

Fig 6: Dilated Images

Fig 7: Closed Images after erosion

Skeletonization:

Skeletonization transforms a binary image into a skeleton, which is a thin version of the original object that is equidistant from its boundaries. The resulting skeleton retains the essential structure of the object but is one pixel wide.

The Zhang-Suen Thinning Algorithm is used here which is a widely used technique for skeletonization of binary images. It is an iterative algorithm that progressively removes edge pixels from the objects in a binary image while preserving their topological structure. The algorithm operates in two sub-iterations for each main iteration, refining the skeleton until it converges.

Let P(i,j) be a pixel in the binary image. The 8 neighbors of P(i,j) are labeled as follows:

P9 P2 P3

P8 P1 P4

P7 P6 P5

The method for extracting the skeleton of a picture consists of removing all contour pixels of the picture except those belonging to the skeleton. In the first sub-iteration, the contour point p1 is deleted from the pattern, if it satisfies the following conditions:

(a) $2 \leq B(p1) \leq 6$

(b) $A(p1)=1$

(c) $p2 \times p4 \times p6 = 0$

(d) $p4 \times p6 \times p8 = 0$

In the second sub-iteration, the contour point p1 is deleted from the pattern, if it satisfies the following conditions:

(a) $2 \leq B(p1) \leq 6$

(b) $A(p1)=1$

(c') $p2 \times p4 \times p8 = 0$

(d') $p2 \times p6 \times p8 = 0$

where: A(p1) is the number of "0-1" (white-black, in this order) pairs in the clock-wise traversal of the 8-neighborhood of p1, i.e. p2,p3,p4,...,p8, p9. B(p1) is the number of non-zero 8-neighbors.

$$B(p_1) = \sum_{i=2}^{9} P_i.$$

If any condition is not satisfied then p1 will not be deleted from the foreground.

Fig 8: Skeletonized Images

Feature extraction:

Minutiae points, or minutiae features, are the specific, detailed characteristics of fingerprint ridges and their structures. These points are crucial for the analysis and matching of fingerprints, as they provide the unique details necessary to differentiate between different prints.

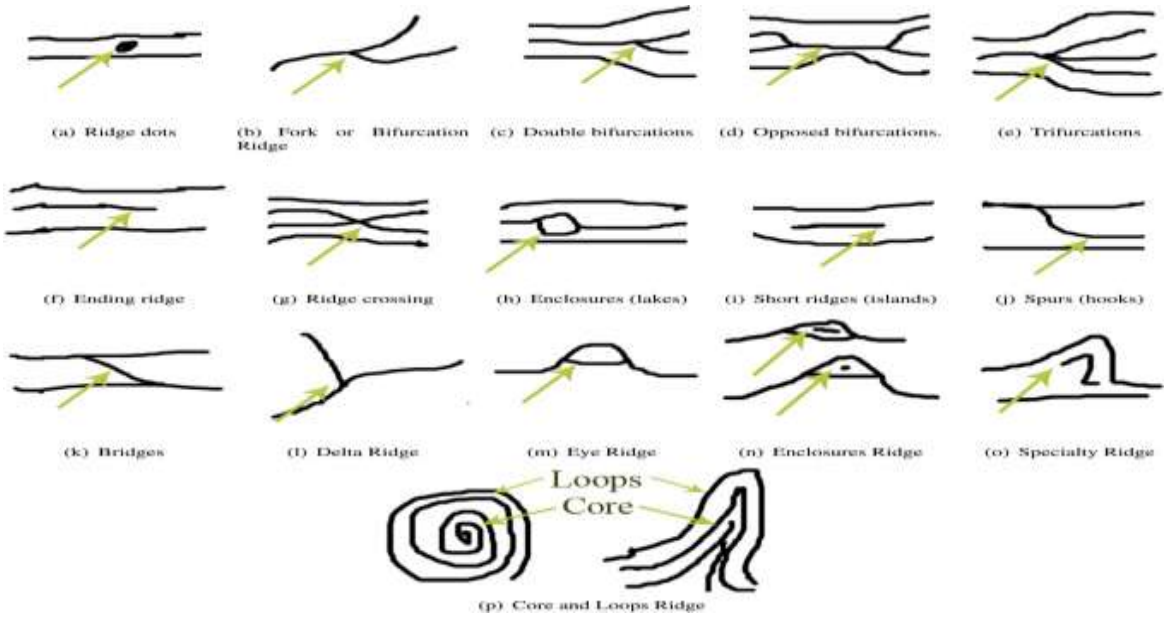

Fig 9: Minutiae features

Minutiae Detection:

- **Ridge Ending**: Identified by checking the pixel neighborhood. If a pixel has exactly one neighbor that is a ridge pixel, it is classified as a ridge ending.
- **Bifurcation**: Identified by checking the pixel neighborhood. If a pixel has more than two transitions from ridge to non-ridge pixels, it is classified as a bifurcation.

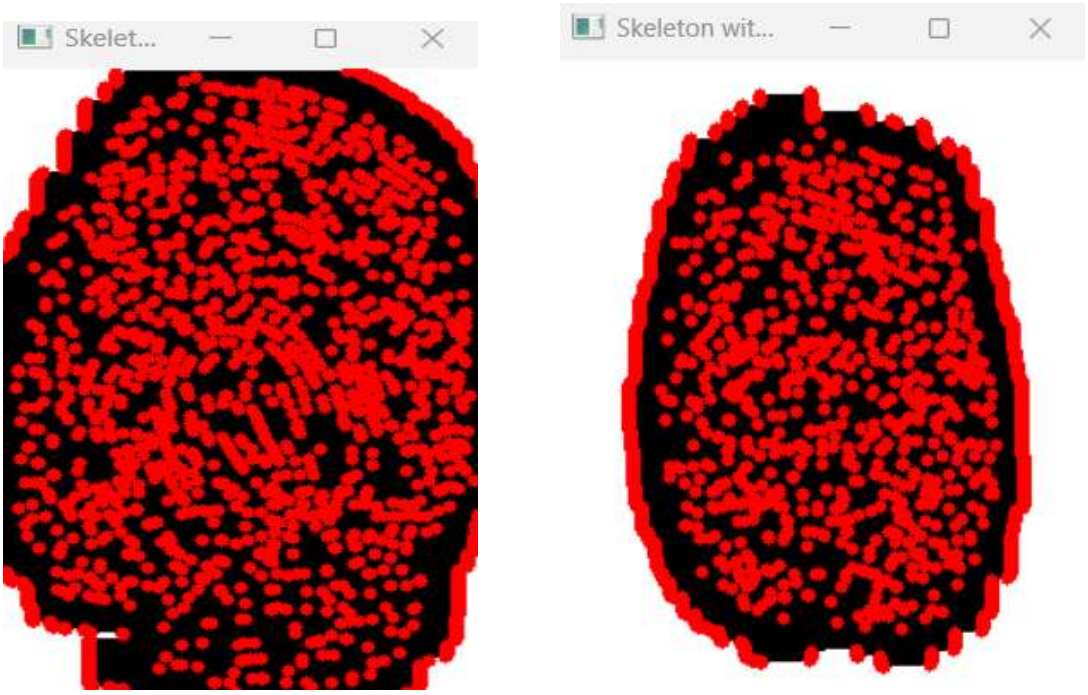

Fig 10: Minutiae Points

Matching:

Euclidean Distance: The function match_minutiae_points is used to match minutiae points between two sets of minutiae points. For each minutiae point in the first set (image 1), the function calculates the Euclidean distance to every minutiae point in the second set (image 2).

Set a Distance Threshold: A maximum distance threshold (e.g., 10 pixels) is defined. If the Euclidean distance between a pair of minutiae points from the two images is less than or equal to this threshold, the points are considered a match.

Count Matched Points: The function counts the number of matched minutiae points between the two images. Find out the percentage of matching

Discussion:

The fingerprint matching system effectively processes two fingerprint images through several steps: image filtering, adaptive thresholding, morphological operations, skeletonization, minutiae extraction, and minutiae matching. Initially, a Gaussian filter smooths the images to reduce noise. Adaptive thresholding then converts the images to binary format, enhancing ridge clarity. Morphological operations like dilation and erosion further refine these images by enhancing ridge structures and removing noise. The Zhang-Suen thinning algorithm skeletonizes the binary images, making ridge lines thinner and suitable for minutiae extraction.

Minutiae points, which include ridge endings and bifurcations, are extracted from these skeletonized images. The matching algorithm then compares these points between the two images, calculating the Euclidean distance between each pair and considering them matched if the distance is within a specified threshold.

The matching percentage is calculated based on the number of matched minutiae points relative to the total minutiae points in both images, considering the highest match percentage from all rotations. This system ensures accurate fingerprint identification and verification by addressing noise reduction, feature enhancement, and orientation variations, providing reliable results and insightful visualizations of the processing steps and minutiae points.

Conclusion:

This fingerprint matching system shows how to use advanced image processing techniques to identify people based on their fingerprints. The system improves the quality of the fingerprint images using

techniques like morphological operations, adaptive thresholding, and Gaussian filtering, making it easier to extract important details. By simplifying the images with the Zhang-Suen thinning algorithm, it accurately identifies specific points where ridges end or split. It matches these points even if the fingerprint is rotated, ensuring reliable results. The system calculates a match percentage to show how similar two fingerprints are, which is useful for security and authentication. This approach highlights the importance of careful image preparation and detail extraction for reliable fingerprint recognition, crucial for secure access control and identity verification.

References:

1. https://www.researchgate.net/publication/308822048_A_new_thinning_algorithm_for_binary_images
2. https://en.wikipedia.org/wiki/Closing_(morphology)
3. https://www.researchgate.net/publication/281449451_Performance_Improvement_for_Fingerprint_Recognition_System_using_Shape_and_Orientation_Descriptors
4. https://en.wikipedia.org/wiki/Gaussian_blur
5. https://www.mathworks.com/help/images/morphological-dilation-and-erosion.html

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.morphology import skeletonize
import math


PI = math.pi
sigma_x = 1
sigma_y = 1

# Gaussian function
def gauss_func(x, y):
    coeff = 1 / (2 * PI * sigma_x * sigma_y)
    val = (((x ** 2) / (sigma_x ** 2)) + ((y ** 2) / (sigma_y ** 2))) / 2
    value = math.exp(-val)
    value = value * coeff
    return value

# Gaussian kernel
def gaussian_kernel(size):
    k_row, k_col = size, size
    gauss_kernel = np.zeros((k_row, k_col), dtype=np.float32)
    center = (size - 1) / 2

    for i in range(k_row):
        for j in range(k_col):
            x = i - center
            y = j - center
            gauss_kernel[i, j] = gauss_func(x, y)

    # Normalize the kernel
    gauss_kernel /= np.sum(gauss_kernel)

    return gauss_kernel

def convolve(image, kernel):
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width,
pad_width)), mode='constant', constant_values=0)
    convolved_image = np.zeros_like(image)
    for i in range(image_height):
```

```
        for j in range(image_width):
            convolved_image[i, j] = np.sum(kernel *
padded_image[i:i+kernel_height, j:j+kernel_width])
    return convolved_image

def adaptive_threshold(image, block_size, C):
    height, width = image.shape
    new_image = np.zeros_like(image)
    pad = block_size // 2
    padded_image = np.pad(image, pad, mode='constant', constant_values=0)
    for i in range(height):
        for j in range(width):
            local_region = padded_image[i:i+block_size, j:j+block_size]
            local_mean = np.mean(local_region)
            threshold = local_mean - C
            new_image[i, j] = 255 if image[i, j] > threshold else 0
    return new_image


def erosion(image, kernel):
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width,
pad_width)), mode='constant', constant_values=255)
    eroded_image = np.zeros_like(image)
    for i in range(image_height):
        for j in range(image_width):
            local_region = padded_image[i:i+kernel_height, j:j+kernel_width]
            eroded_image[i, j] = np.min(local_region[kernel == 1])
    return eroded_image

def dilation(image, kernel):
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width,
pad_width)), mode='constant', constant_values=0)
    dilated_image = np.zeros_like(image)
    for i in range(image_height):
        for j in range(image_width):
            local_region = padded_image[i:i+kernel_height, j:j+kernel_width]
            dilated_image[i, j] = np.max(local_region[kernel == 1])
    return dilated_image

def thinning_iteration(image, iter):
    marker = np.zeros(image.shape, np.uint8)
```

```
    rows, cols = image.shape
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            P2, P3, P4 = image[i-1, j], image[i-1, j+1], image[i, j+1]
            P5, P6, P7 = image[i+1, j+1], image[i+1, j], image[i+1, j-1]
            P8, P9 = image[i, j-1], image[i-1, j-1]

            A = (P2 == 0 and P3 == 1) + (P3 == 0 and P4 == 1) + \
            (P4 == 0 and P5 == 1) + (P5 == 0 and P6 == 1) + \
            (P6 == 0 and P7 == 1) + (P7 == 0 and P8 == 1) + \
            (P8 == 0 and P9 == 1) + (P9 == 0 and P2 == 1)
            B = P2 + P3 + P4 + P5 + P6 + P7 + P8 + P9
            m1 = P2 * P4 * P6 if iter == 0 else P2 * P4 * P8
            m2 = P4 * P6 * P8 if iter == 0 else P2 * P6 * P8

            if A == 1 and (2 <= B <= 6) and m1 == 0 and m2 == 0:
                marker[i, j] = 1
    image[marker == 1] = 0

def zhang_suen_thinning(image, max_iter=10):
    binary_image = image.copy() // 255
    prev = np.zeros(binary_image.shape, np.uint8)
    diff = True

    for _ in range(max_iter):
        if not diff:
            break
        thinning_iteration(binary_image, 0)
        thinning_iteration(binary_image, 1)
        diff = not np.array_equal(binary_image, prev)
        prev = binary_image.copy()

    return binary_image * 255

def minutiae_extraction(skel):
    minutiae_points = []
    for i in range(1, skel.shape[0] - 1):
        for j in range(1, skel.shape[1] - 1):
            if skel[i, j] == 255:
                neighbors = skel[i-1:i+2, j-1:j+2].flatten()
                transitions = np.count_nonzero(np.diff(neighbors == 255))
                if transitions == 2:
                    minutiae_points.append((i, j))  # Ridge ending
                elif transitions > 2:
                    minutiae_points.append((i, j))  # Bifurcation
    return minutiae_points

def match_minutiae_points(points1, points2, max_distance=10):
```

```
    matched_points = 0
    for p1 in points1:
        for p2 in points2:
            distance = np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
            if distance <= max_distance:
                matched_points += 1
                break
    return matched_points


fingerprint_image1 = cv2.imread('input_img.jpg', cv2.IMREAD_GRAYSCALE)
fingerprint_image2 = cv2.imread('input_img1.jpg', cv2.IMREAD_GRAYSCALE)

r1 = cv2.rotate(fingerprint_image2, cv2.ROTATE_90_CLOCKWISE)
r2 = cv2.rotate(r1, cv2.ROTATE_90_CLOCKWISE)
r3 = cv2.rotate(r2, cv2.ROTATE_90_CLOCKWISE)




gaussian_kernel = gaussian_kernel(5)
filtered_img1 = convolve(fingerprint_image1, gaussian_kernel)
filtered_img2 = convolve(fingerprint_image2, gaussian_kernel)
filtered_r1 = convolve(r1, gaussian_kernel)
filtered_r2 = convolve(r2, gaussian_kernel)
filtered_r3 = convolve(r2, gaussian_kernel)

block_size = 11
C = 2
thresh1 = adaptive_threshold(filtered_img1, block_size, C)

thresh2 = adaptive_threshold(filtered_img2, block_size, C)
thresh_r1 = adaptive_threshold(filtered_r1, block_size, C)
thresh_r2 = adaptive_threshold(filtered_r2, block_size, C)
thresh_r3 = adaptive_threshold(filtered_r3, block_size, C)



kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))


dilated_img1 = dilation(thresh1, kernel)
dilated_img2 = dilation(thresh2, kernel)
dilated_r1 = dilation(thresh_r1, kernel)
dilated_r2 = dilation(thresh_r2, kernel)
dilated_r3 = dilation(thresh_r3, kernel)

closed_img1 = erosion(dilated_img1, kernel)
closed_img2 = erosion(dilated_img2, kernel)
```

```
closed_r1 = erosion(dilated_r1, kernel)
closed_r2 = erosion(dilated_r2, kernel)
closed_r3 = erosion(dilated_r3, kernel)

# Apply Zhang-Suen Thinning Algorithm
skeleton1 = zhang_suen_thinning(closed_img1)
skeleton2 = zhang_suen_thinning(closed_img2)
skeleton_r1 = zhang_suen_thinning(closed_r1)
skeleton_r2 = zhang_suen_thinning(closed_r2)
skeleton_r3 = zhang_suen_thinning(closed_r3)


# Minutiae Extraction
minutiae_points1 = minutiae_extraction(skeleton1)
minutiae_points2 = minutiae_extraction(skeleton2)
minutiae_pointsr1 = minutiae_extraction(skeleton_r1)
minutiae_pointsr2 = minutiae_extraction(skeleton_r2)
minutiae_pointsr3 = minutiae_extraction(skeleton_r3)


# Minutiae Matching
matched_points = match_minutiae_points(minutiae_points1, minutiae_points2)
total_points = len(minutiae_points1) + len(minutiae_points2)
match_percentage = (2 * matched_points / total_points) * 100

# Minutiae Matching
matched_pointsr1 = match_minutiae_points(minutiae_points1, minutiae_pointsr1)
total_pointsr1 = len(minutiae_points1) + len(minutiae_pointsr1)
match_percentager1 = (2 * matched_pointsr1 / total_pointsr1) * 100

# Minutiae Matching
matched_pointsr2 = match_minutiae_points(minutiae_points1, minutiae_pointsr2)
total_pointsr2 = len(minutiae_points1) + len(minutiae_pointsr2)
match_percentager2 = (2 * matched_pointsr2 / total_pointsr2) * 100

# Minutiae Matching
matched_pointsr3 = match_minutiae_points(minutiae_points1, minutiae_pointsr3)
total_pointsr3 = len(minutiae_points1) + len(minutiae_pointsr3)
match_percentager3 = (2 * matched_pointsr3 / total_pointsr3) * 100

match_per = []
match_per.append(match_percentage)
match_per.append(match_percentager1)
match_per.append(match_percentager2)
match_per.append(match_percentager3)

maxx=max(match_per)
```

```
print(f"Match Percentage: {maxx:.2f}%")

# Visualize Minutiae Points
skeleton_rgb1 = cv2.cvtColor(skeleton1, cv2.COLOR_GRAY2BGR)
skeleton_rgb2 = cv2.cvtColor(skeleton2, cv2.COLOR_GRAY2BGR)
for point in minutiae_points1:
    cv2.circle(skeleton_rgb1, point[::-1], 3, (0, 0, 255), -1)
for point in minutiae_points2:
    cv2.circle(skeleton_rgb2, point[::-1], 3, (0, 0, 255), -1)

cv2.imwrite('skeleton_minutiae1.jpg', skeleton_rgb1)
cv2.imwrite('skeleton_minutiae2.jpg', skeleton_rgb2)


cv2.imwrite('filtered_img1.jpg', filtered_img1)
cv2.imwrite('filtered_img2.jpg', filtered_img2)
cv2.imwrite('thresh1.jpg', thresh1)
cv2.imwrite('thresh2.jpg', thresh2)
cv2.imwrite('dilated_img1.jpg', dilated_img1)
cv2.imwrite('dilated_img2.jpg', dilated_img2)
cv2.imwrite('closed_img1.jpg', closed_img1)
cv2.imwrite('closed_img2.jpg', closed_img2)
cv2.imwrite('skeleton1.jpg', skeleton1)
cv2.imwrite('skeleton2.jpg', skeleton2)



# Define images and titles for plotting
images1 = [
    fingerprint_image1, filtered_img1, thresh1,  dilated_img1,  closed_img1,
skeleton1, skeleton_rgb1
]
titles1 = [
    'Original Image 1', 'Filtered Image 1', 'Thresholded Image 1', 'Dilated
Image 1', 'Closed Image 1', 'Skeleton Image 1', 'Skeleton with Minutiae 1'
]

images2 = [
    fingerprint_image2, filtered_img2, thresh2, dilated_img2, closed_img2,
skeleton2, skeleton_rgb2
]
titles2 = [
    'Original Image 2', 'Filtered Image 2', 'Thresholded Image 2',  'Dilated
Image 2', 'Closed Image 2', 'Skeleton Image 2', 'Skeleton with Minutiae 2'
]

def plot_images(images, titles, figure_title):
    fig, axes = plt.subplots(len(images) // 3 + (len(images) % 3 > 0), 3,
figsize=(12, len(images) * 2))
```

```
    fig.suptitle(figure_title, fontsize=26)

    for ax in axes.flat:
        ax.tick_params(axis='both', which='both', labelsize=12)

    for i, ax in enumerate(axes.flat):
        if i < len(images):
            if len(images[i].shape) == 2:
                ax.imshow(images[i], cmap='gray')
            else:
                ax.imshow(images[i])
            ax.set_title(titles[i], fontsize=22)
            ax.axis('off')

    plt.tight_layout(rect=[0, 0, 1, 0.97])
    plt.show()


# Plot images for both fingerprint images
plot_images(images1, titles1, "Processing Steps for Fingerprint Image 1")
plot_images(images2, titles2, "Processing Steps for Fingerprint Image 2")

cv2.imshow('Skeleton with Minutiae Points 1', skeleton_rgb1)
cv2.imshow('Skeleton with Minutiae Points 2', skeleton_rgb2)


cv2.imshow("r1",r1)
cv2.imshow("r2",r2)
cv2.imshow("r3",r3)

cv2.waitKey(0)
cv2.destroyAllWindows()
```