Digital Image Processing Lab Report 4
CSE 4734

Abdullah Al Jubaer Gem
ID: 210041226
Section: 2B

January 5, 2026

1 Task 1: Dilation and Erosion

1.1 Problem Statement

Implement two functions that simulate the behaviour of morphological Dilation and Erosion. For each of these functions, a parameter SE (structuring element) is used.

1.2 Solution Approach

Morphological operations like dilation and erosion modify the shape of objects in binary or grayscale images using a structuring element. Dilation expands bright regions by adding pixels to the boundaries, while erosion shrinks them by removing pixels. Both operations use padding to handle image boundaries and iterate over each pixel, applying the max (dilation) or min (erosion) operation within the neighborhood defined by the structuring element.

1.3 Implementation

```
import numpy as np

def dilation(image, SE):
    img_h, img_w = image.shape
    se_h, se_w = SE.shape

    pad_h = se_h // 2
    pad_w = se_w // 2

    padded = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
    dilated_image = np.zeros_like(image)

    for i in range(img_h):
        for j in range(img_w):
          region = padded[i:i+se_h, j:j+se_w]
          dilated_image[i, j] = np.max(region[SE == 1])

    return dilated_image

def erosion(image, SE):
  img_h, img_w = image.shape
  se_h, se_w = SE.shape

  pad_h = se_h // 2
  pad_w = se_w // 2

  padded = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
  eroded_image = np.zeros_like(image)

  for i in range(img_h):
      for j in range(img_w):
        region = padded[i:i+se_h, j:j+se_w]
        eroded_image[i, j] = np.min(region[SE == 1])
```

```
34
35    return eroded_image
```

1.4 Output

The following figures show the original image and eroded images.

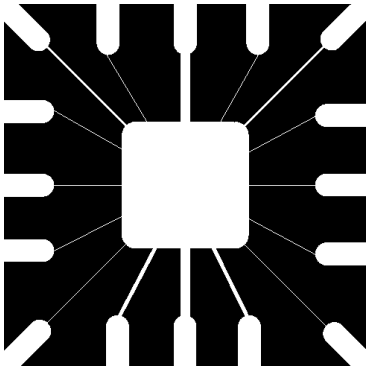

Figure 1: Original Image

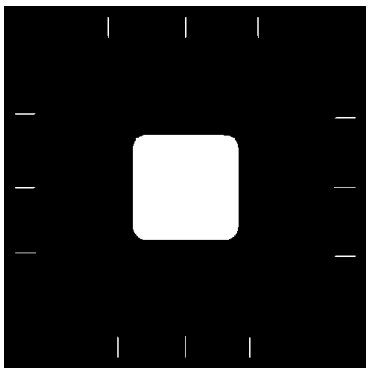

Figure 2: Eroded Image (1 time)

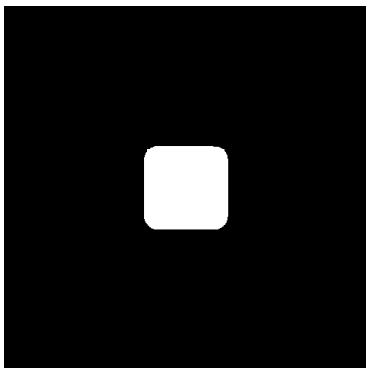

Figure 3: Eroded Image (2 times)

1.5 Analysis

1. **What happens when you apply dilation multiple times:** Dilation expands bright regions by adding pixels to their boundaries. Applying it multiple times progressively enlarges objects, filling small gaps and connecting nearby structures. This can lead to significant morphological changes, such as merging separate components into larger blobs.

2. **What happens when you apply erosion multiple times:** Erosion shrinks bright regions by removing boundary pixels. Repeated erosion reduces object sizes, eliminates thin structures, and can disconnect connected components. Over-application may cause objects to disappear entirely, leaving only the most robust shapes.

2 Task 2: Granulometry

2.1 Problem Statement

Determine the sizes of granules in an image of wood dowel plugs using granulometry, which involves opening with circular structuring elements of variable sizes and plotting the surface area vs radius.

2.2 Solution Approach

Granulometry analyzes particle size distribution by successively opening the image with structuring elements of increasing size and measuring the remaining area. Opening combines erosion followed by dilation, removing small objects while preserving larger ones. Circular structuring elements ensure isotropic processing. The average intensity decreases as larger particles are preserved, and the difference in intensity indicates the presence of particles of certain sizes. Smoothing is applied first to reduce noise.

2.3 Implementation

```
1  def opening(image, SE):
2    eroded = cv2.erode(image, SE)
3    open_image = cv2.dilate(eroded, SE)
4    return open_image
5
6  def circular_se(radius):
7      size = 2 * radius + 1
8      y, x = np.ogrid[-radius:radius+1, -radius:radius+1]
9      mask = x**2 + y**2 <= radius**2
```

```
    return mask.astype(np.uint8)

def granulometry(image):
    radii = []
    diff_intensity = []
    avg_intensity = []
    prev_avg = None

    for r in range(5, 50, 5):
        SE = circular_se(r)
        open_image = opening(image, SE)

        if r in [5, 25, 45]:
            cv2.imwrite(f'output_images/task2_opened_r{r}.png', open_image)

        avg = np.mean(open_image)
        radii.append(r)
        avg_intensity.append(avg)

        if prev_avg is not None:
            diff_intensity.append(prev_avg - avg)
        prev_avg = avg

    plt.figure(figsize=(6,4))
    plt.plot(radii, avg_intensity, marker='o')
    plt.xlabel("Structuring Element Radius")
    plt.ylabel("Average Intensity")
    plt.title("Granulometry Curve")
    plt.grid(True)
    plt.savefig('output_images/task2_granulometry_curve.png')
    plt.close()

    plt.figure(figsize=(6,4))
    plt.plot(radii[1:], diff_intensity, marker='o')
    plt.xlabel("Structuring Element Radius")
    plt.ylabel("Average Intensity difference")
    plt.title("Granulometry intensity difference Curve")
    plt.grid(True)
    plt.savefig('output_images/task2_intensity_difference_curve.png')
    plt.close()

    return radii, avg_intensity, diff_intensity
```

2.4 Output

The following figures show the original image, opened images for selected radii, and the granulometry curves.

Figure 4: Original Image

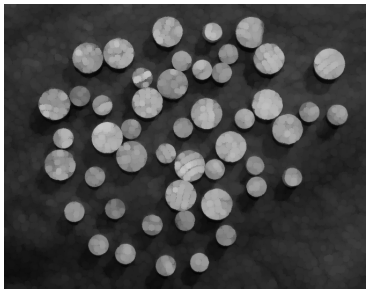

Figure 5: Opened (r=5)

Figure 6: Opened (r=25)

Figure 7: Opened (r=45)

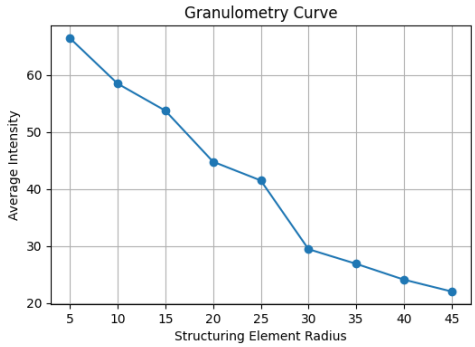

Figure 8: Granulometry Curve

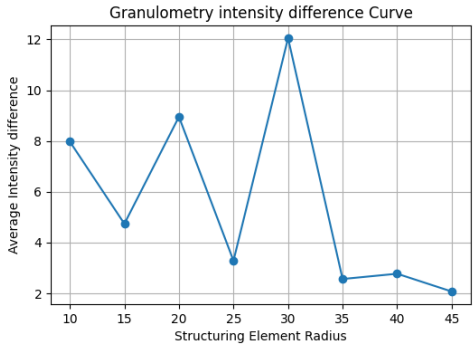

Figure 9: Intensity Difference Curve

2.5 Analysis

1. **Why is it recommended to apply smoothing first before performing the opening:** Smoothing reduces high-frequency noise and small-scale variations in the image. Without smoothing, these artifacts could be mistaken for small particles during granulometry, leading to inaccurate size distribution measurements. Smoothing ensures that only meaningful structures are analyzed, improving the reliability of the particle size estimation.

3 Task 3: Locate Boundary

3.1 Problem Statement

Locate the boundary between distinct texture regions in an image of small and large circles.

3.2 Solution Approach

The image contains small circles enclosed by larger ones. To locate the boundary, openings are performed with varying radii to isolate different scales, followed by closings to fill gaps. The boundary is detected by subtracting a further eroded version from the processed image.

3.3 Implementation

```
def locateBoundary(image):
    binary_img = (image < 128).astype(np.uint8)
    radii = list(range(14, 26, 2))

    plt.figure(figsize=(12, 8))
    for idx, r in enumerate(radii):
        SE = circular_se(r)
        opened_img = opening(binary_img, SE)
        plt.subplot(2, 3, idx + 1)
        plt.imshow(opened_img, cmap='gray')
        plt.title(f"Opening (r = {r})")
```

```
        plt.axis('off')
    plt.tight_layout()
    plt.savefig('output_images/task3_openings.png')
    plt.close()

    SE = circular_se(23)
    opened_img = opening(binary_img, SE)
    radii_outer = list(range(32, 44, 2))

    plt.figure(figsize=(12, 8))
    for idx, r in enumerate(radii_outer):
        SE = circular_se(r)
        dilated = dilation(opened_img, SE)
        closed = erosion(dilated, SE)
        plt.subplot(2, 3, idx + 1)
        plt.imshow(closed, cmap='gray')
        plt.title(f"Closing (r = {r})")
        plt.axis('off')
    plt.tight_layout()
    plt.savefig('output_images/task3_closings.png')
    plt.close()

    inner_radius = 23
    outer_radius = 38
    required_radius = outer_radius

    SE = circular_se(inner_radius)
    opened_img = opening(binary_img, SE)

    SE = circular_se(required_radius)
    closed = closing(opened_img, SE)

    SE = circular_se(5)
    final_close = erosion(closed, SE)
    boundary = final_close - closed

    plt.imshow(boundary, cmap='gray')
    plt.title('Boundary')
    plt.axis('off')
    plt.savefig('output_images/task3_boundary.png')
    plt.close()

    return boundary
```

3.4 Output

The following figures show the original image, intermediate openings and closings, and the final boundary.

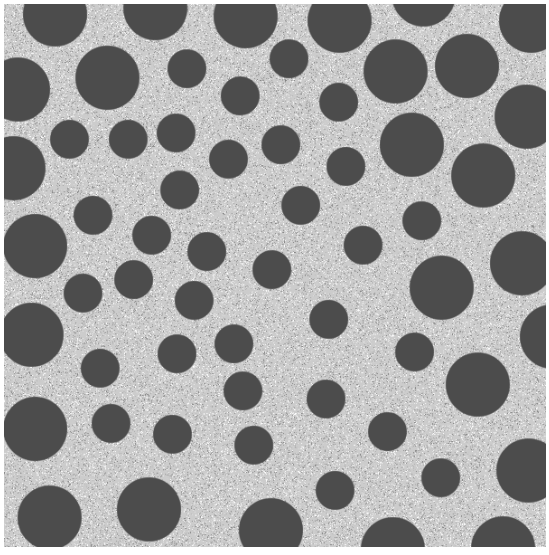

Figure 10: Original Image

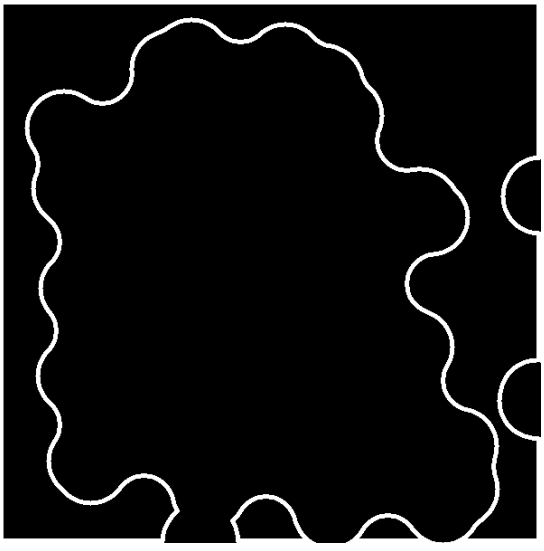

Figure 11: Detected Boundary

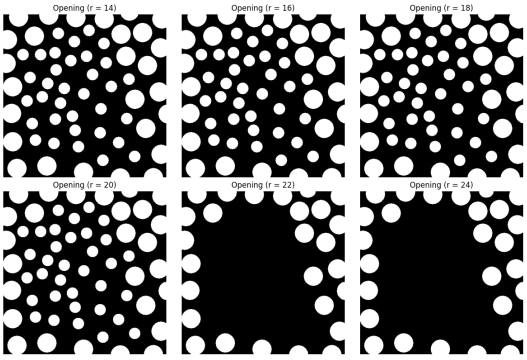

Figure 12: Intermediate Openings

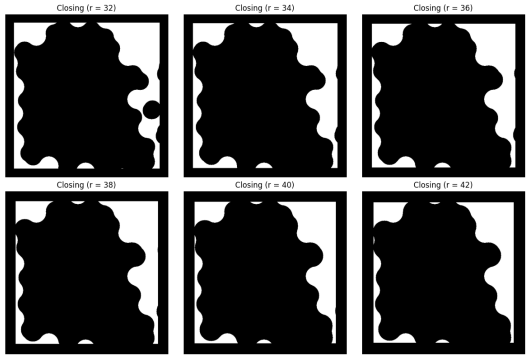

Figure 13: Intermediate Closings

3.5 Analysis

The boundary detection algorithm effectively locates the texture boundary by leveraging morphological operations at different scales. Opening with smaller structuring elements removes the inner small circles, while subsequent closings with larger elements fill the outer regions. The final erosion and subtraction isolate the boundary pixels, providing a clear demarcation between the two texture regions. This approach is robust to noise and variations in circle sizes, making it suitable for texture segmentation in microscopy images.

4 Task 4: Microscopy Preprocessing

4.1 Problem Statement

Isolate individual particles from overlapping ones in a microscopy image, producing images of boundary-touching particles, overlapping particles, and non-overlapping particles.

4.2 Solution Approach

Particles touching the image boundary are identified and removed using iterative dilation. The remaining image is labeled to find connected components, and area thresholding distinguishes overlapping (larger area) from non-overlapping particles.

4.3 Implementation

```
from scipy.ndimage import label

def particles_touching_boundary(binary_img):
    h, w = binary_img.shape
    boundary = np.zeros_like(binary_img)
    boundary[0, :] = 1
    boundary[-1, :] = 1
    boundary[:, 0] = 1
    boundary[:, -1] = 1
    touching = binary_img * boundary
    SE = circular_se(3)
    prev = np.zeros_like(binary_img)
    curr = touching.copy()
    while not np.array_equal(prev, curr):
        prev = curr.copy()
        curr = dilation(curr, SE) * binary_img
    return curr

def microscopy(image):
    binary_img = (image > 128).astype(np.uint8)
    cv2.imwrite('output_images/task4_binary.png', binary_img * 255)
    boundaryParticles = particles_touching_boundary(binary_img)
    binary_img_clean = binary_img - boundaryParticles
    cv2.imwrite('output_images/task4_clean_binary.png', binary_img_clean * 255)
```

```
    labeled, num = label(binary_img_clean)
    overlappingParticle = np.zeros_like(binary_img)
    nonoverlappingParticles = np.zeros_like(binary_img)
    AREA_THRESHOLD = 530
    for lab in range(1, num + 1):
        component = (labeled == lab)
        area = np.sum(component)
        if area > AREA_THRESHOLD:
            overlappingParticle[component] = 255
        else:
            nonoverlappingParticles[component] = 255
    boundaryParticles = boundaryParticles * 255
    return boundaryParticles, overlappingParticle, nonoverlappingParticles
```

4.4 Output

The following figures show the original image, binary processing steps, and separated particle types.

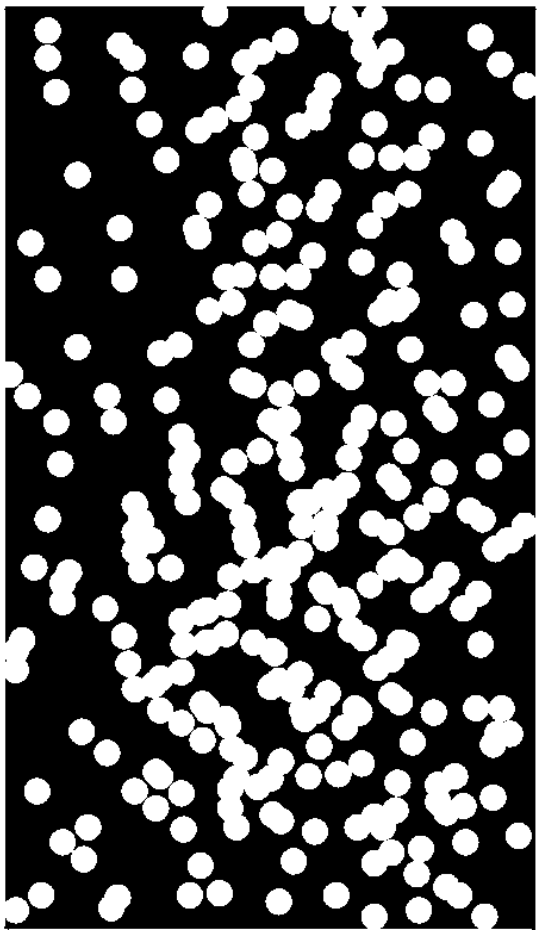

Figure 14: Original Image

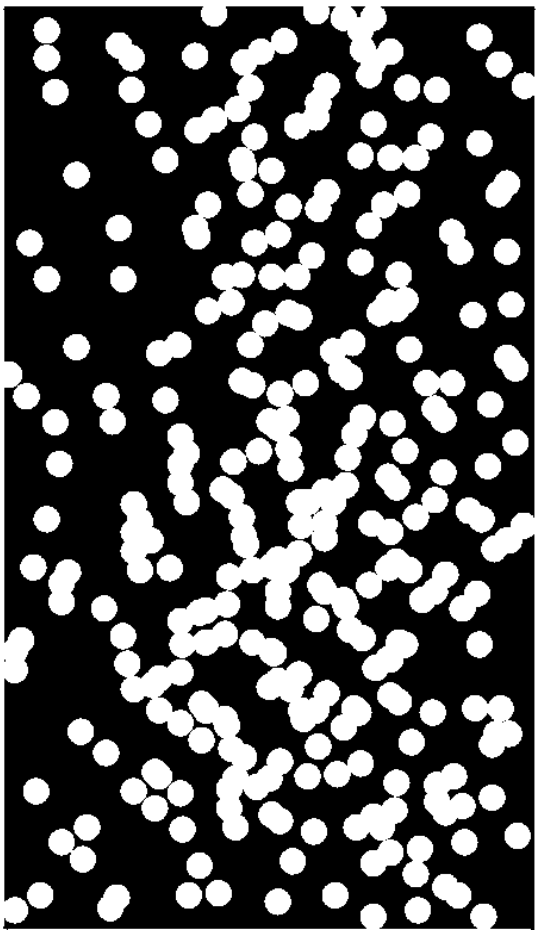

Figure 15: Binary Image

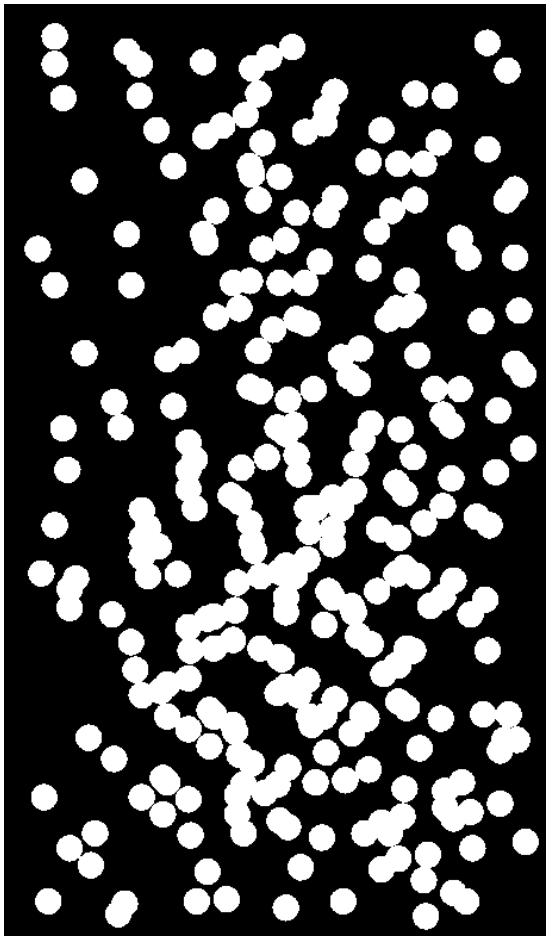

Figure 16: Cleaned Binary Image

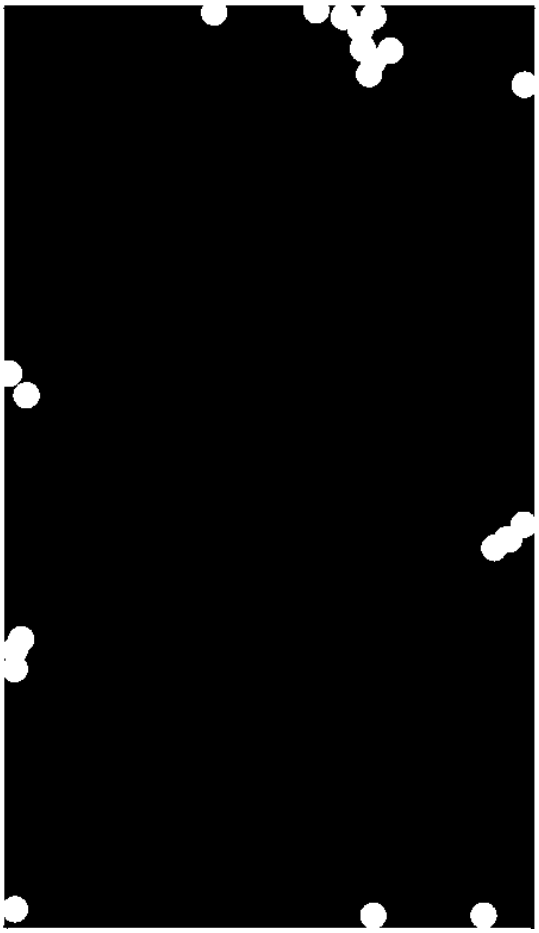

Figure 17: Boundary Particles

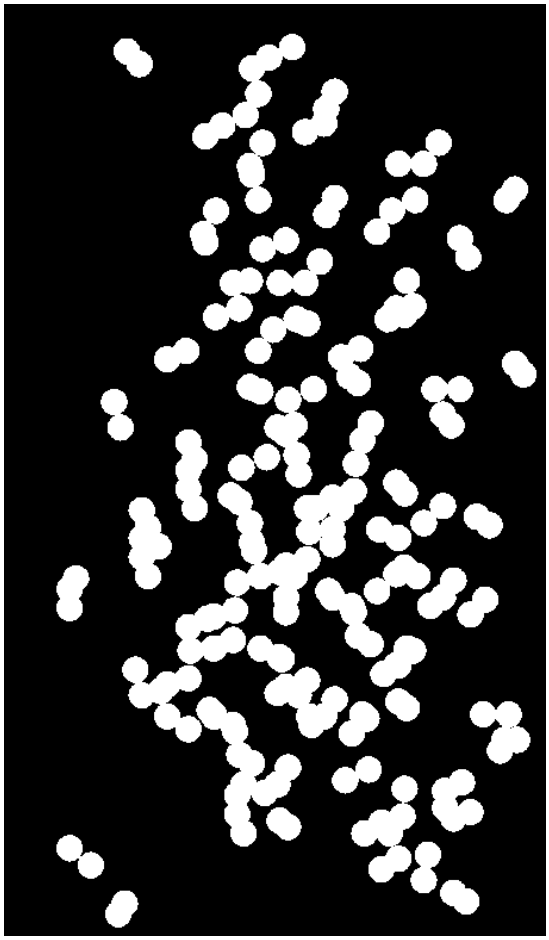

Figure 18: Overlapping Particles

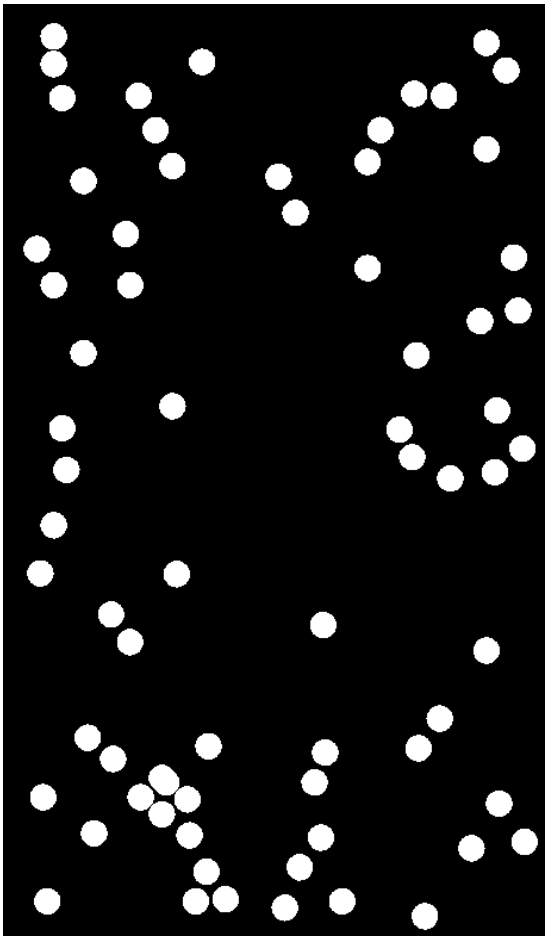

Figure 19: Non-overlapping Particles

4.5 Analysis

The microscopy preprocessing algorithm successfully separates particles based on boundary contact and clustering. Particles touching the image boundary are identified and removed using iterative morphological operations, preventing edge artifacts. The remaining particles are classified by area: larger areas indicate overlapping particles, while smaller ones are non-overlapping. This method provides clean segmentation for further analysis, such as counting individual particles or measuring sizes, which is crucial for automated microscopy applications.