

# 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

```
1 import numpy as np
2
3 def dilation(image, SE):
4     img_h, img_w = image.shape
5     se_h, se_w = SE.shape
6
7     pad_h = se_h // 2
8     pad_w = se_w // 2
9
10    padded = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
11    dilated_image = np.zeros_like(image)
12
13    for i in range(img_h):
14        for j in range(img_w):
15            region = padded[i:i+se_h, j:j+se_w]
16            dilated_image[i, j] = np.max(region[SE == 1])
17
18    return dilated_image
19
20 def erosion(image, SE):
21     img_h, img_w = image.shape
22     se_h, se_w = SE.shape
23
24     pad_h = se_h // 2
25     pad_w = se_w // 2
26
27     padded = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
28     eroded_image = np.zeros_like(image)
29
30     for i in range(img_h):
31         for j in range(img_w):
32             region = padded[i:i+se_h, j:j+se_w]
33             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.

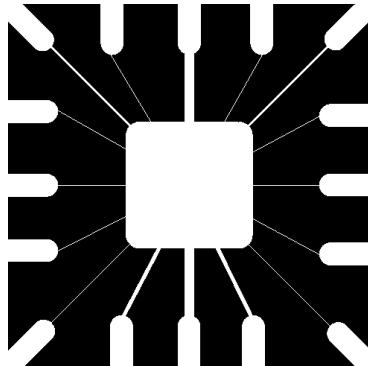


Figure 1: Original Image

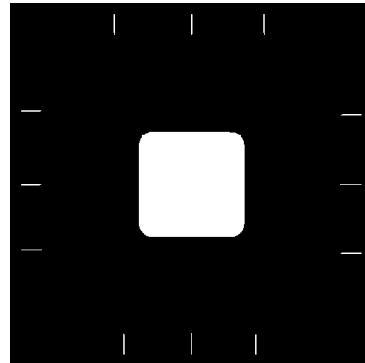


Figure 2: Eroded Image (1 time)

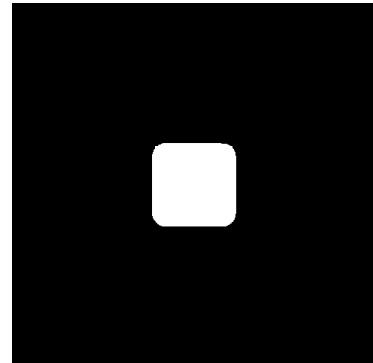


Figure 3: Eroded Image (2 times)

## 1.5 Analysis

- What happens when you apply dilation multiple times:** Applying dilation multiple times progressively expands the bright regions in the image. Each iteration adds more pixels to the boundaries, increasing the size of objects and filling small gaps. This can lead to merging of nearby objects if applied excessively.
- What happens when you apply erosion multiple times:** Erosion shrinks bright regions by removing boundary pixels with each application. Multiple erosions can eliminate small objects entirely and thin larger ones, potentially disconnecting connected components or creating holes.

## 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
10    return mask.astype(np.uint8)

```

```

11
12 def granulometry(image):
13     radii = []
14     diff_intensity = []
15     avg_intensity = []
16     prev_avg = None
17
18     for r in range(5, 50, 5):
19         SE = circular_se(r)
20         open_image = opening(image, SE)
21
22         if r in [5, 25, 45]:
23             cv2.imwrite(f'output_images/task2_opened_r{r}.png', open_image)
24
25         avg = np.mean(open_image)
26         radii.append(r)
27         avg_intensity.append(avg)
28
29         if prev_avg is not None:
30             diff_intensity.append(prev_avg - avg)
31         prev_avg = avg
32
33     plt.figure(figsize=(6,4))
34     plt.plot(radii, avg_intensity, marker='o')
35     plt.xlabel("Structuring Element Radius")
36     plt.ylabel("Average Intensity")
37     plt.title("Granulometry Curve")
38     plt.grid(True)
39     plt.savefig('output_images/task2_granulometry_curve.png')
40     plt.close()
41
42     plt.figure(figsize=(6,4))
43     plt.plot(radii[1:], diff_intensity, marker='o')
44     plt.xlabel("Structuring Element Radius")
45     plt.ylabel("Average Intensity difference")
46     plt.title("Granulometry intensity difference Curve")
47     plt.grid(True)
48     plt.savefig('output_images/task2_intensity_difference_curve.png')
49     plt.close()
50
51     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 ( $r=5$ )

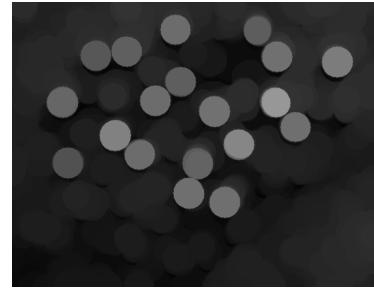


Figure 6: Opened ( $r=25$ )



Figure 7: Opened ( $r=45$ )

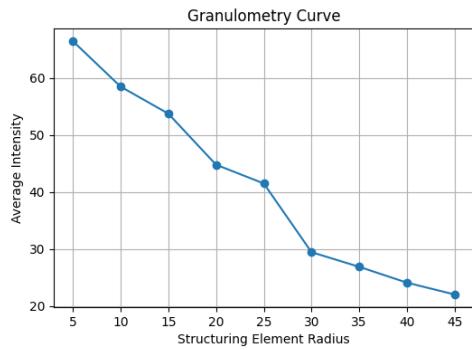


Figure 8: Granulometry Curve

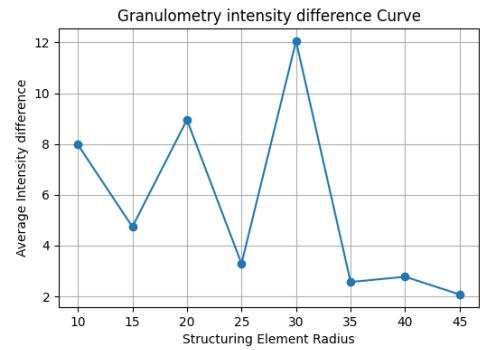


Figure 9: Intensity Difference Curve

## 2.5 Analysis

1. **Why is it recommended to apply smoothing first:** Smoothing reduces noise and small variations in the image before morphological operations. This prevents small noisy regions from being interpreted as particles, leading to more accurate size distribution measurements in granulometry.

## 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

```

1 def locateBoundary(image):
2     binary_img = (image < 128).astype(np.uint8)
3     radii = list(range(14, 26, 2))
4
5     plt.figure(figsize=(12, 8))
6     for idx, r in enumerate(radii):
7         SE = circular_se(r)
8         opened_img = opening(binary_img, SE)
9         plt.subplot(2, 3, idx + 1)
10        plt.imshow(opened_img, cmap='gray')
11        plt.title(f"Opening (r = {r})")
12        plt.axis('off')
13    plt.tight_layout()

```

```

14     plt.savefig('output_images/task3_openings.png')
15     plt.close()
16
17     SE = circular_se(23)
18     opened_img = opening(binary_img, SE)
19     radii_outer = list(range(32, 44, 2))
20
21     plt.figure(figsize=(12, 8))
22     for idx, r in enumerate(radii_outer):
23         SE = circular_se(r)
24         dilated = dilation(opened_img, SE)
25         closed = erosion(dilated, SE)
26         plt.subplot(2, 3, idx + 1)
27         plt.imshow(closed, cmap='gray')
28         plt.title(f"Closing (r = {r})")
29         plt.axis('off')
30     plt.tight_layout()
31     plt.savefig('output_images/task3_closings.png')
32     plt.close()
33
34     inner_radius = 23
35     outer_radius = 38
36     required_radius = outer_radius
37
38     SE = circular_se(inner_radius)
39     opened_img = opening(binary_img, SE)
40
41     SE = circular_se(required_radius)
42     closed = closing(opened_img, SE)
43
44     SE = circular_se(5)
45     final_close = erosion(closed, SE)
46     boundary = final_close - closed
47
48     plt.imshow(boundary, cmap='gray')
49     plt.title('Boundary')
50     plt.axis('off')
51     plt.savefig('output_images/task3_boundary.png')
52     plt.close()
53
54     return boundary

```

### 3.4 Output

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

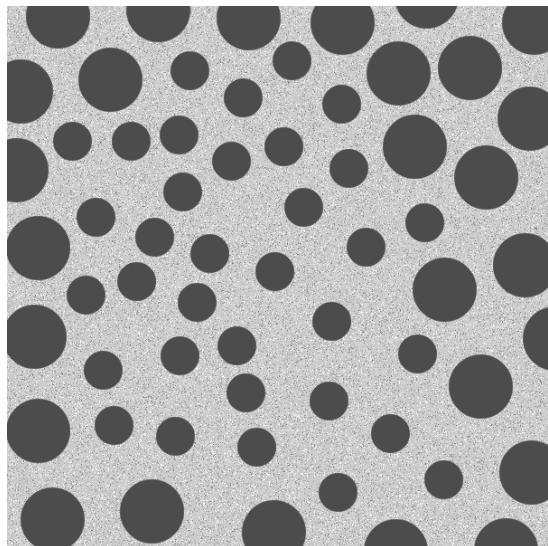


Figure 10: Original Image

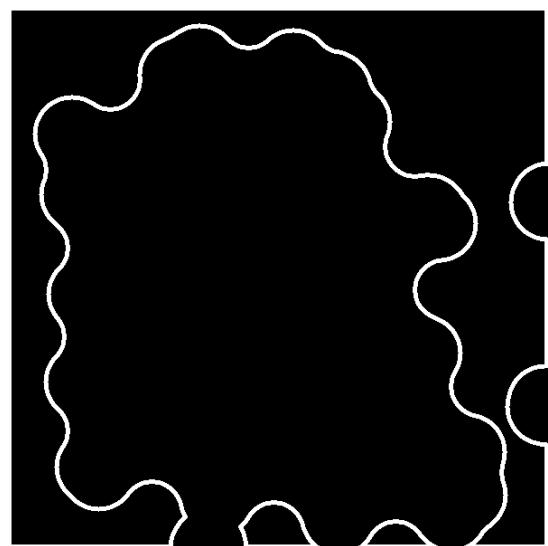


Figure 11: Detected Boundary

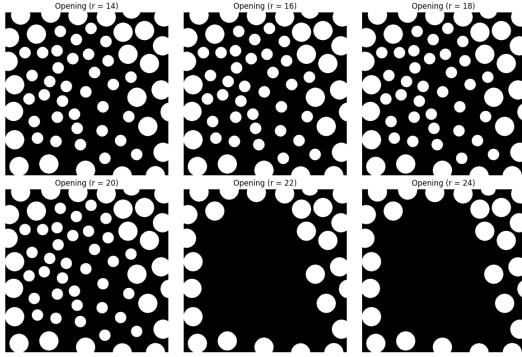


Figure 12: Intermediate Openings

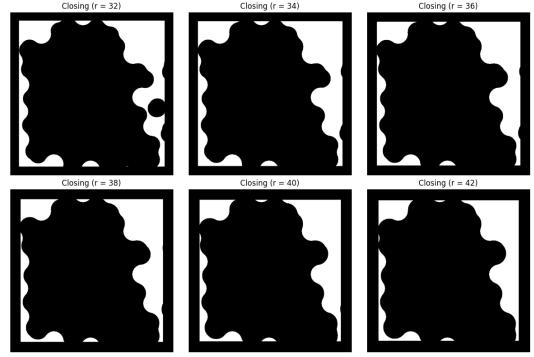


Figure 13: Intermediate Closings

### 3.5 Analysis

The boundary detection successfully isolates the region between the small inner circles and the larger outer circles by using morphological operations at appropriate scales.

## 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

```

1  from scipy.ndimage import label
2
3  def particles_touching_boundary(binary_img):
4      h, w = binary_img.shape
5      boundary = np.zeros_like(binary_img)
6      boundary[0, :] = 1
7      boundary[-1, :] = 1
8      boundary[:, 0] = 1
9      boundary[:, -1] = 1
10     touching = binary_img * boundary
11     SE = circular_se(3)
12     prev = np.zeros_like(binary_img)
13     curr = touching.copy()
14     while not np.array_equal(prev, curr):
15         prev = curr.copy()
16         curr = dilation(curr, SE) * binary_img
17     return curr
18
19 def microscopy(image):
20     binary_img = (image > 128).astype(np.uint8)
21     cv2.imwrite('output_images/task4_binary.png', binary_img * 255)
22     boundaryParticles = particles_touching_boundary(binary_img)
23     binary_img_clean = binary_img - boundaryParticles
24     cv2.imwrite('output_images/task4_clean_binary.png', binary_img_clean * 255)
25     labeled, num = label(binary_img_clean)
26     overlappingParticle = np.zeros_like(binary_img)
27     nonoverlappingParticles = np.zeros_like(binary_img)
28     AREA_THRESHOLD = 530
29     for lab in range(1, num + 1):
30         component = (labeled == lab)

```

```
31     area = np.sum(component)
32     if area > AREA_THRESHOLD:
33         overlappingParticle[component] = 255
34     else:
35         nonoverlappingParticles[component] = 255
36 boundaryParticles = boundaryParticles * 255
37 return boundaryParticles, overlappingParticle, nonoverlappingParticles
```

#### 4.4 Output

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

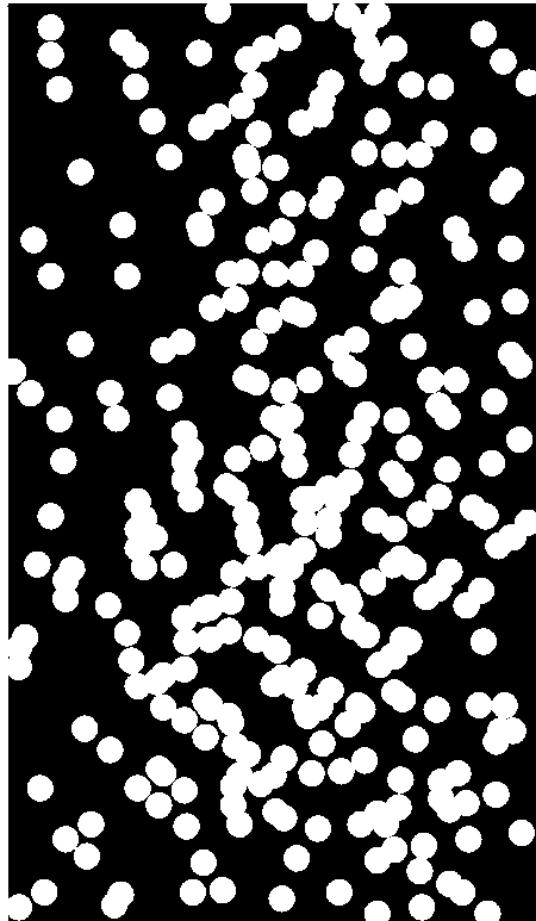


Figure 14: Original Image

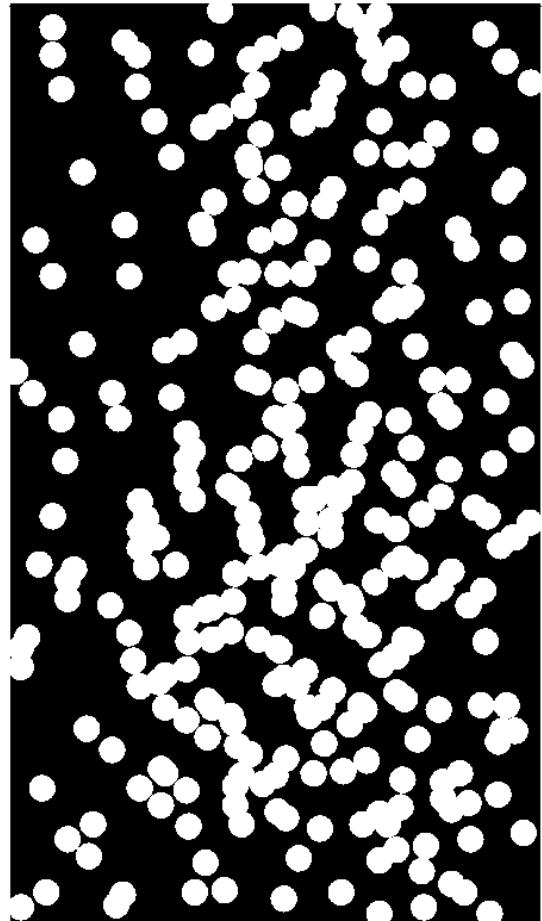


Figure 15: Binary Image

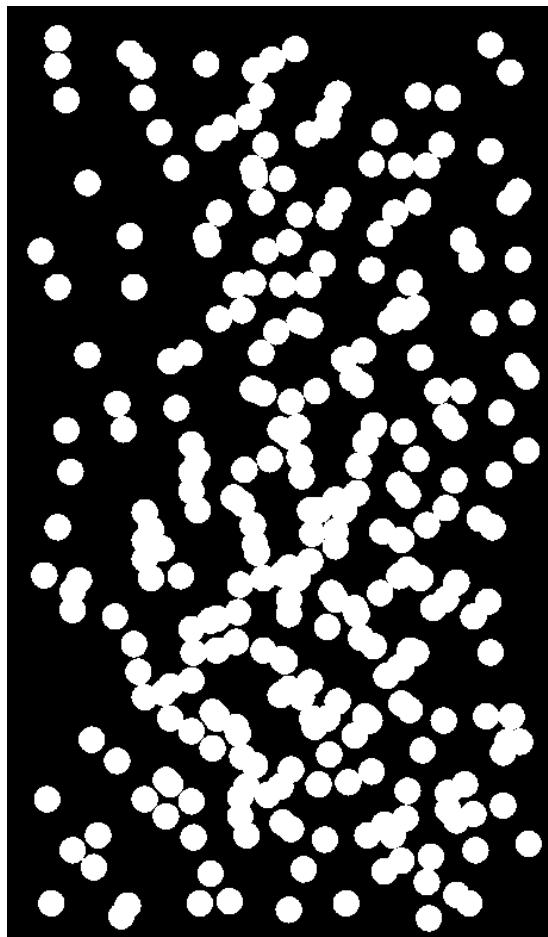


Figure 16: Cleaned Binary Image

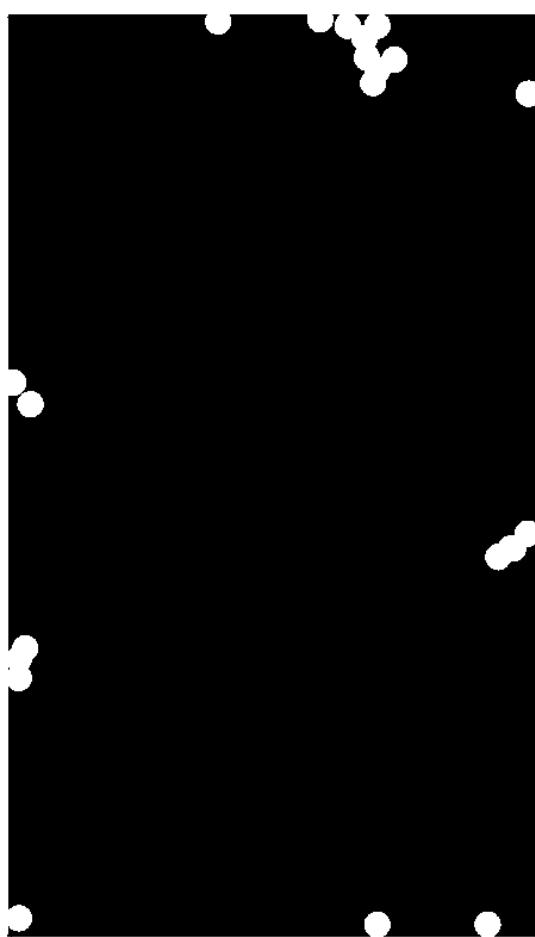


Figure 17: Boundary Particles

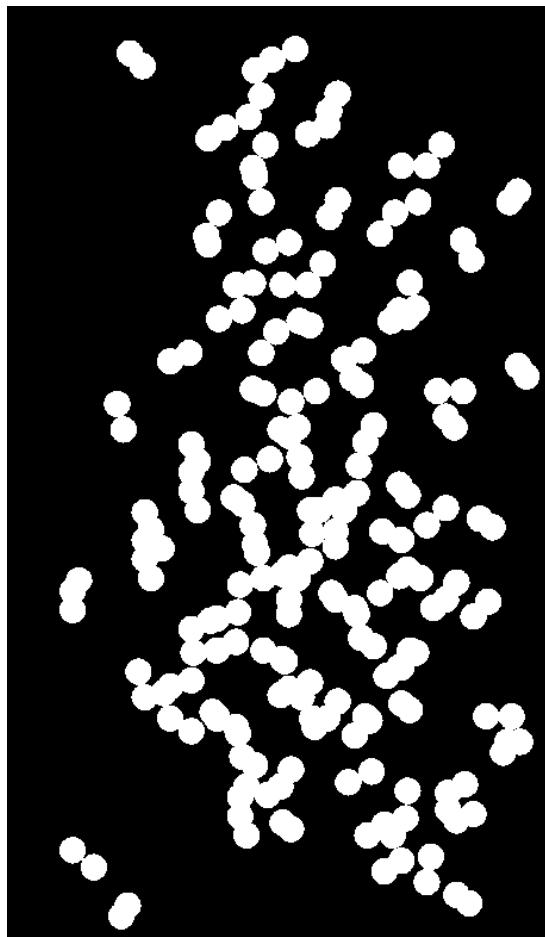


Figure 18: Overlapping Particles

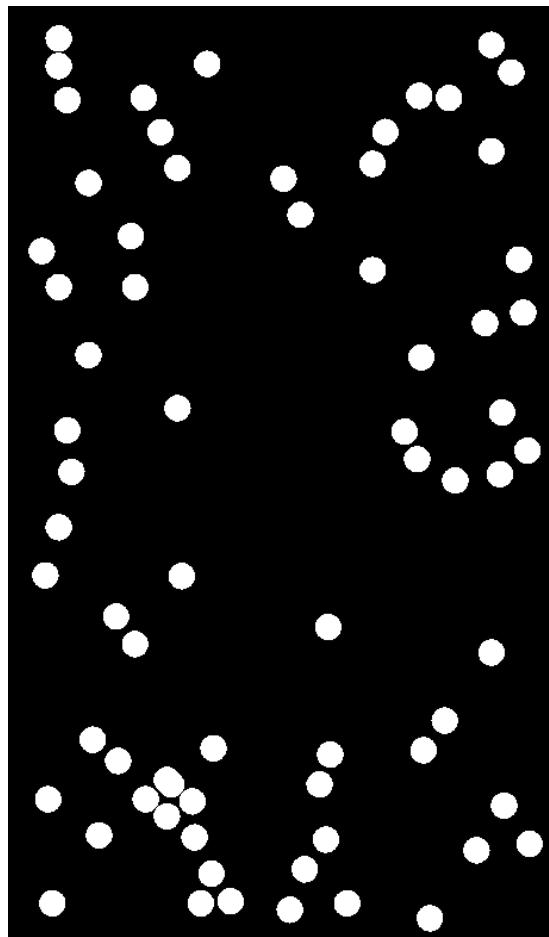


Figure 19: Non-overlapping Particles

#### 4.5 Analysis

The algorithm effectively separates particles based on boundary contact and size, allowing for further analysis of individual vs. clustered particles in microscopy images.