DEPARTMENT OF COMPUTER ENGINEERING

FACULTY OF ENGINEERING

UNIVERSITY OF RUHUNA

EC7212 – Computer Vision and Image Processing

Take Home Assignment 2

DISSANAYAKE D.M.M.I.T

EG/2020/3912

Image-Processing-Experiments-Take Home 2

GitHub Repository with source code**:** https://github.com/ThisaruDissanayake/Image-Segmentation-Tasks-Take-Home-2.git

1 Introduction

Image processing is an essential technique in computer vision for analyzing and understanding visual data. In this assignment, two main techniques were implemented:

- Otsu's Thresholding: An automatic method to separate objects from the background.
- Region Growing: A segmentation technique to group pixels based on intensity similarity.

I used a synthetic image with two objects (a circle and a rectangle) and a background. Then applied Gaussian noise to simulate real-world conditions before using Otsu's method and region-growing segmentation.

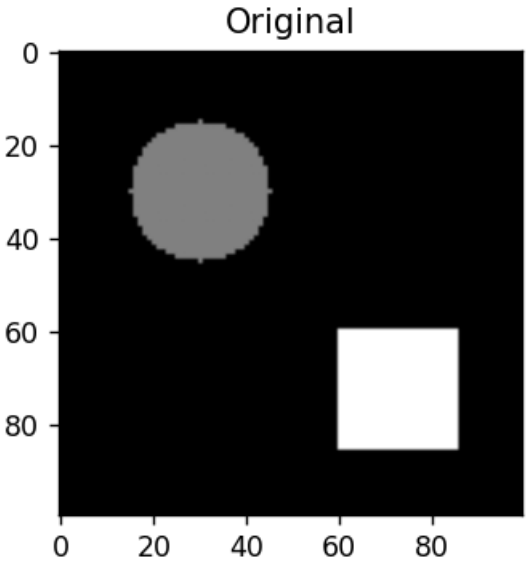

Figure 1-1 Original Image

2 Methodology

2.1 Image Creation

We used either a synthetic image (a black background with a gray circle and a brighter square) or a local image for testing.

2.2 Gaussian Noise

Gaussian noise was added to simulate real-world imperfections in image acquisition. This helps test the robustness of our segmentation techniques.

2.3 Otsu's Thresholding

Otsu's method automatically determines the best threshold value to separate foreground and background based on pixel intensity histograms.

Steps:

- Blur the image using Gaussian filter to reduce noise.
- Use cv2.threshold() with cv2.THRESH_OTSU.

2.4 Region Growing

Region growing starts from a given seed point and includes neighboring pixels that are similar in intensity. This method is useful when the object's intensity is known or roughly consistent.

Steps:

- Pick a seed point inside the object.
- Check if neighboring pixels are within a threshold range from the seed.
- Add them recursively if they are similar.

3 Code Implementation

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# IMAGE SOURCE SELECTION
def load_image(use_local=False, local_path='test.jpg'):
    if use_local:
        image = cv2.imread(local_path, cv2.IMREAD_GRAYSCALE)
        if image is None:
            raise FileNotFoundError(f"Image not found at: {local_path}")
        return image
    else:
        image = np.zeros((100, 100), dtype=np.uint8)
        cv2.circle(image, (30, 30), 15, 85, -1)
        cv2.rectangle(image, (60, 60), (85, 85), 170, -1)
        return image
```

```python
#  ADD GAUSSIAN NOISE
def add_gaussian_noise(image, mean=0, std=50):
    noise = np.random.normal(mean, std, image.shape).astype(np.int16)
    noisy_img = np.clip(image.astype(np.int16) + noise, 0,
255).astype(np.uint8)
    return noisy_img

#  OTSU'S THRESHOLDING
def apply_otsu(image):
    blur = cv2.GaussianBlur(image, (5, 5), 0)

    otsu_val, thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)
    print(f"Otsu's threshold value: {otsu_val}")
    return thresh

#  REGION GROWING
def region_growing(image, seeds, threshold=10):
    visited = np.zeros_like(image, dtype=bool)
    output = np.zeros_like(image, dtype=np.uint8)
    h, w = image.shape
    seed_value = image[seeds[0][1], seeds[0][0]]
    stack = list(seeds)

    while stack:
        x, y = stack.pop()
        if visited[y, x]:
            continue
        visited[y, x] = True
        current_val = image[y, x]
        if abs(int(current_val) - int(seed_value)) <= threshold:
            output[y, x] = 255
            for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < w and 0 <= ny < h and not visited[ny, nx]:
                    stack.append((nx, ny))
    return output

# Toggle between synthetic and local image
use_local_image = False  # Set to True to load your own image
local_image_path = 'D:/7 Semester/computer vision/Take Home 2/test1.png'

image = load_image(use_local=use_local_image, local_path=local_image_path)
noisy_image = add_gaussian_noise(image)
otsu_result = apply_otsu(noisy_image)
seed_points = [(30, 30)] if not use_local_image else [(70, 70)]  # adjust seed
region_result = region_growing(image, seed_points, threshold=20)
```

```
plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1), plt.imshow(image, cmap='gray'), plt.title("Original")
plt.subplot(2, 2, 2), plt.imshow(noisy_image, cmap='gray'), plt.title("With
Gaussian Noise")
plt.subplot(2, 2, 3), plt.imshow(otsu_result, cmap='gray'), plt.title("Otsu
Threshold")
plt.subplot(2, 2, 4), plt.imshow(region_result, cmap='gray'),
plt.title("Region Grown")
plt.tight_layout()
plt.show()
```

4 Results

The figure below shows the results of each step in the image processing:

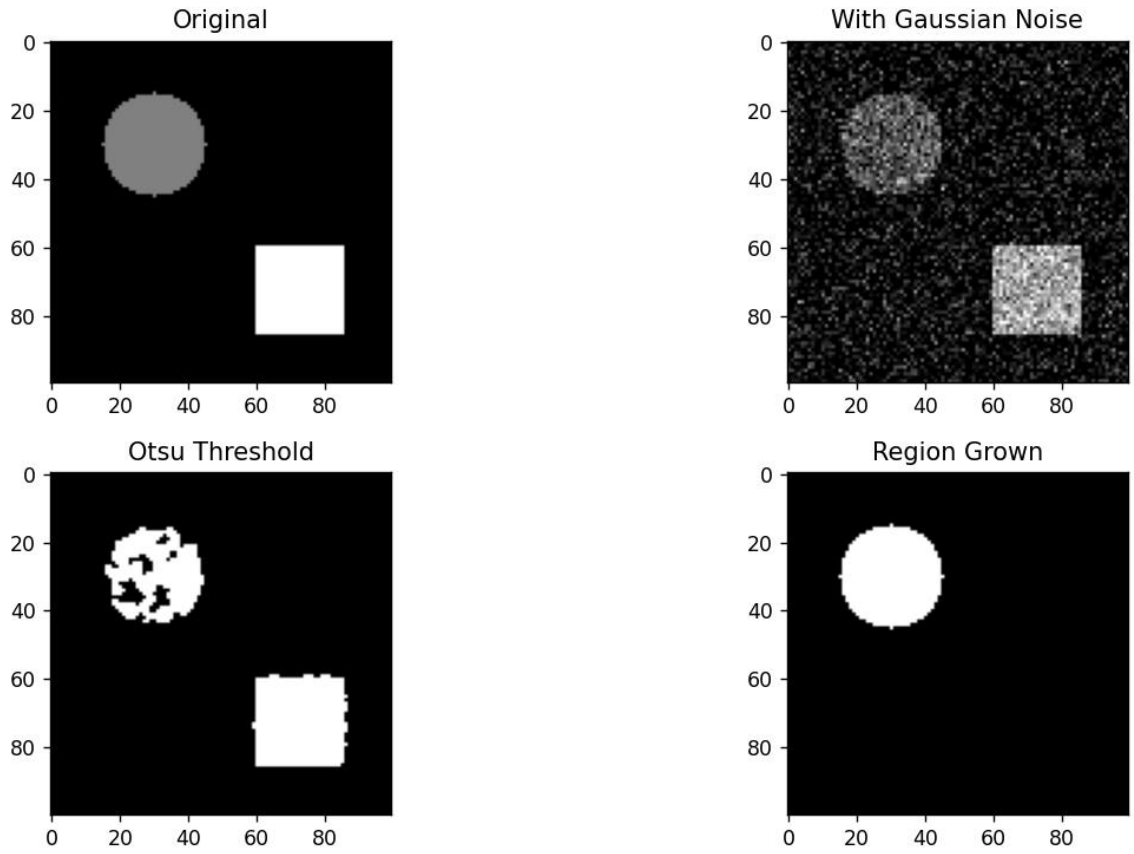

Figure 4-1 Output Results Images

1. Original Image:

The image contains two clear objects a **gray circle** and a **white square** placed on a black background. These represent two distinct pixel intensity levels (gray = 85, white = 170), along with the background (black = 0).

2. With Gaussian Noise:

Gaussian noise was added to simulate a real-world scenario where the image is not perfectly clean. As seen in the image, the objects now appear with random intensity variations, especially around their boundaries. However, their general shapes are still visible.

3. Otsu Threshold:

Otsu's algorithm automatically calculated the optimal threshold to separate the objects from the background. In the output, the square is detected quite accurately, but the noisy circle appears broken and patchy. This shows that Otsu's method can be affected by noise, especially on objects with lower intensity contrast (like the gray circle).

Gaussian noise was added to simulate real-world image imperfections. The noise was controlled using the standard deviation (**`std`**) value in the noise function. A higher `std` value adds stronger noise and distorts the image, while a lower `std` value keeps the noise minimal, preserving the original image quality.

Table 4-1 Variation of std value

Standard Deviation (std)	Image Appearance	Otsu's Thresholding Result
std = 10	Slightly noisy, still very clear	Very accurate segmentation
std = 50 (used here)	Moderate noise, objects visible	Acceptable but some noise inside objects
std = 80 or 100	Strong noise, very grainy image	Poor segmentation, boundaries unclear

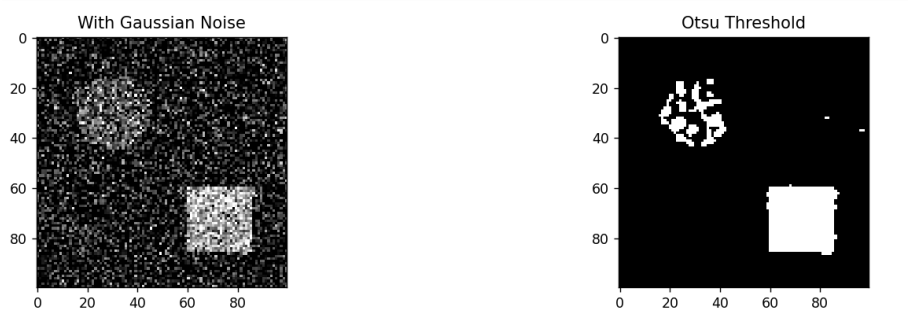

Figure 4-2 Standard deviation value 80,Strong noise and Poor segmentation image

Table 4-2Standard deviation value 10,Low noise and good accurate segmentation image

4. Region Grown:

Starting from a seed inside the circle, the region-growing algorithm successfully segmented the entire circular object. It did this by checking neighboring pixels with similar intensity values. This method was very effective even in the presence of noise, as it focused on local pixel similarity rather than a global threshold.

If you change the seed point to inside the square (70, 70), the algorithm will grow the region to segment the white square instead of the circle. This is because the algorithm will match pixel values close to the square's brightness.

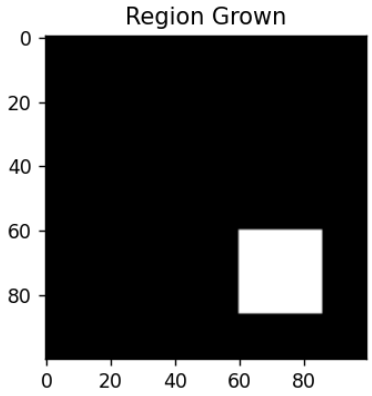

Figure 4-3 Region Grown Image in Seed Point 70,70

If you select a point in the background, such as (5, 5), the region-growing algorithm will only highlight parts of the black background, because it will only include dark pixels near the seed.

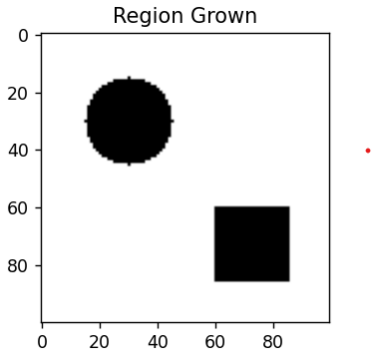

Figure 4-4 Region Grown Image in Seed Point 5,5