Project Presentation on Python Programming Lab(DA514)

Segmentation of Brain MRI Image Using Watershed Transform Algorithm

Presented By

Sri Gautam Talukdar

Research Scholar

Department of EEE

Roll No.246102026

Introduction/Motivation

1. **Medical Importance of Early Diagnosis:**

Brain tumors can be life-threatening, and early, accurate detection is essential for effective treatment planning.

Automated segmentation can quickly highlight areas of concern, helping radiologists focus on specific regions that may need further examination.

The Watershed algorithm, known for its precision in edge-based segmentation, can help identify tumor boundaries in complex MRI images.

2. **Complexity of MRI Images**:

MRI images often contain noise, complex textures, and subtle contrast variations, especially in brain scans.

The Watershed algorithm's gradient-based approach allows it to differentiate structures effectively, even in complex images, aiding radiologists in accurately identifying tumor regions without extensive manual intervention.

Introduction/Motivation

3. **Advantages of Marker-Based Segmentation**:

The Watershed algorithm benefits from marker-based segmentation, where foreground and background markers can be defined beforehand.

4. **Consistency and Reproducibility**:

Manual segmentation can vary from person to person, introducing subjectivity.

Automated methods like the Watershed algorithm help in achieving consistent and reproducible results, which is critical in clinical diagnostics, where accuracy and consistency are paramount.

5. **Foundation for Advanced Segmentation Methods**:

The Watershed algorithm serves as a foundational method in image segmentation, often used as a pre-processing or initial segmentation step in more advanced methods.

Pseuso-code view of code

1. Load the image

- Read the brain MRI image.
- If it is color, convert it to gray scale image to simplify processing.

2. Compute gradient magnitude

- Use Sobel operators to calculate gradients in the x and y directions
- Combine these gradients to get edge strength (gradient magnitude) image.

  Input : Blurred Image

  Output: Gradient Magnitude Image

3. Define markers

- Apply thresholding to identify regions in the image as "foreground" (possible tumor regions) and "background.“
- Assign initial marker labels:

  Background Marker: Label pixels below a low-intensity threshold.

  Foreground Marker: Label pixels above a high-intensity threshold.

  Input: Gradient Magnitude Image
  Output: Marker Image (binary regions for foreground and background)

Pseudocode view of code

4. Apply Watershed Algorithm

- Use the defined markers as the starting points for the Watershed algorithm.
- Perform region growing from markers, assigning labels to pixels based on gradient values.
- Flood-fill regions, with the algorithm expanding each region from the marker points until boundaries meet.

  Input : Marker Image and Gradient Magnitude Image

  Output: Labeled Regions Image (Tumor regions highlighted)

5. Post-process and Visualize Results

- Map the watershed-labeled regions onto the original image for visualization.
- Display or save the segmented image, showing tumor boundaries clearly.

Pythonics features used in the Program

1. List Comprehensions:

pixel_list = [(i, j) for i in range(height) for j in range(width) if segmented[i, j] > 0]

List comprehensions provide a concise and readable way to generate lists, here capturing all the initial marker pixels with values above zero.

2. functions like define_markers

def define_markers(img, threshold=100):

3. Numpy for Efficient Array Processing

- gradient_magnitude = np.sqrt(sobel_x ** 2 + sobel_y ** 2)

Pythonic features used

4. Docstrings for Function documentation

```
def compute_gradient(img):
```

5. Matplotlib for visualization.

The matplotlib library provides simple, high-level commands to visualize images and graphs.

```
plt.imshow(gradient, cmap='gray')
```

6. Efficient Sorting with Lambda Functions:

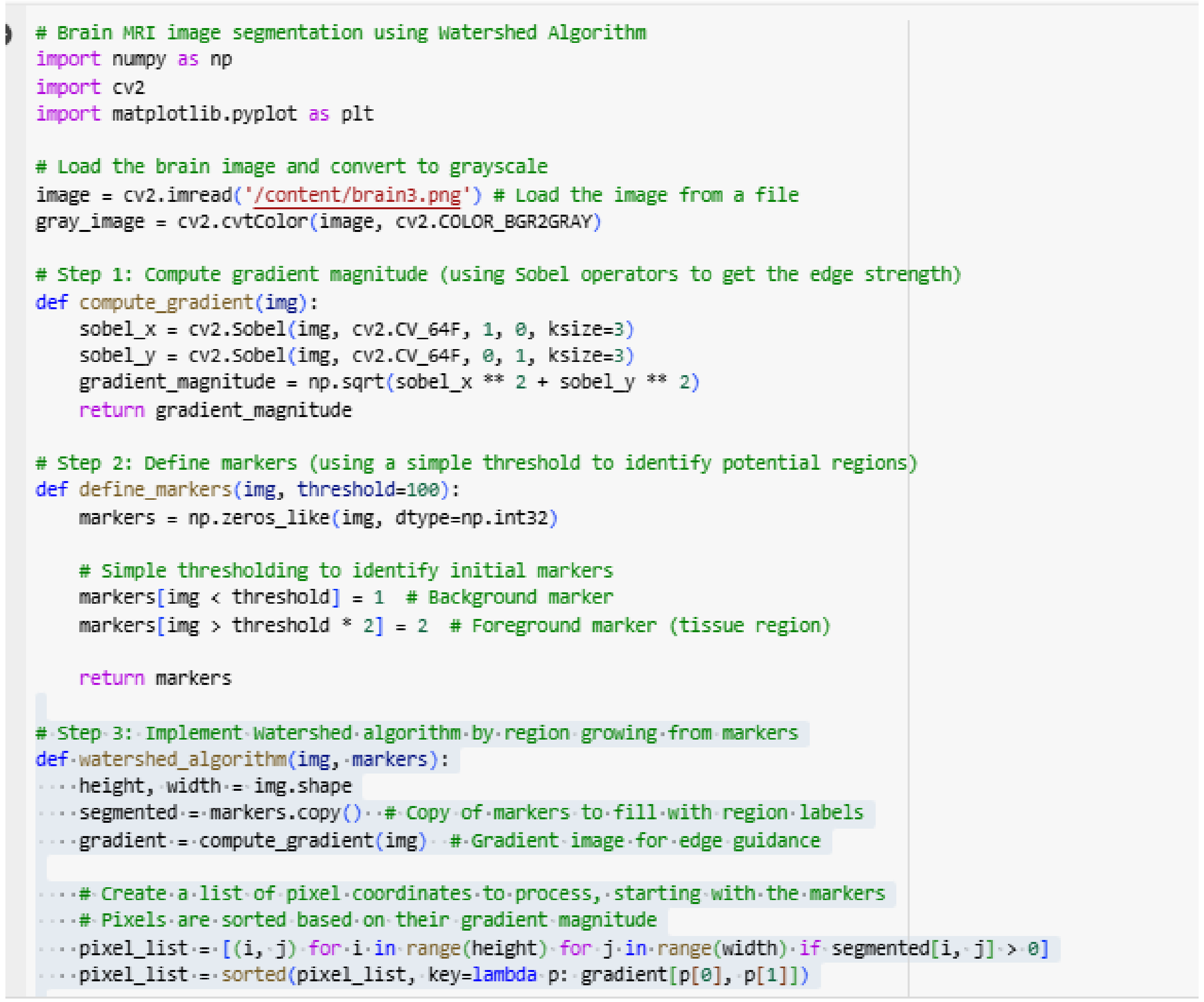
```
pixel_list = sorted(pixel_list, key=lambda p: gradient[p[0], p[1]])
```

7. Dynamic List updates in while loop

The code dynamically grows pixel_list by appending pixels to it as it iterates through each pixel’s neighbors.

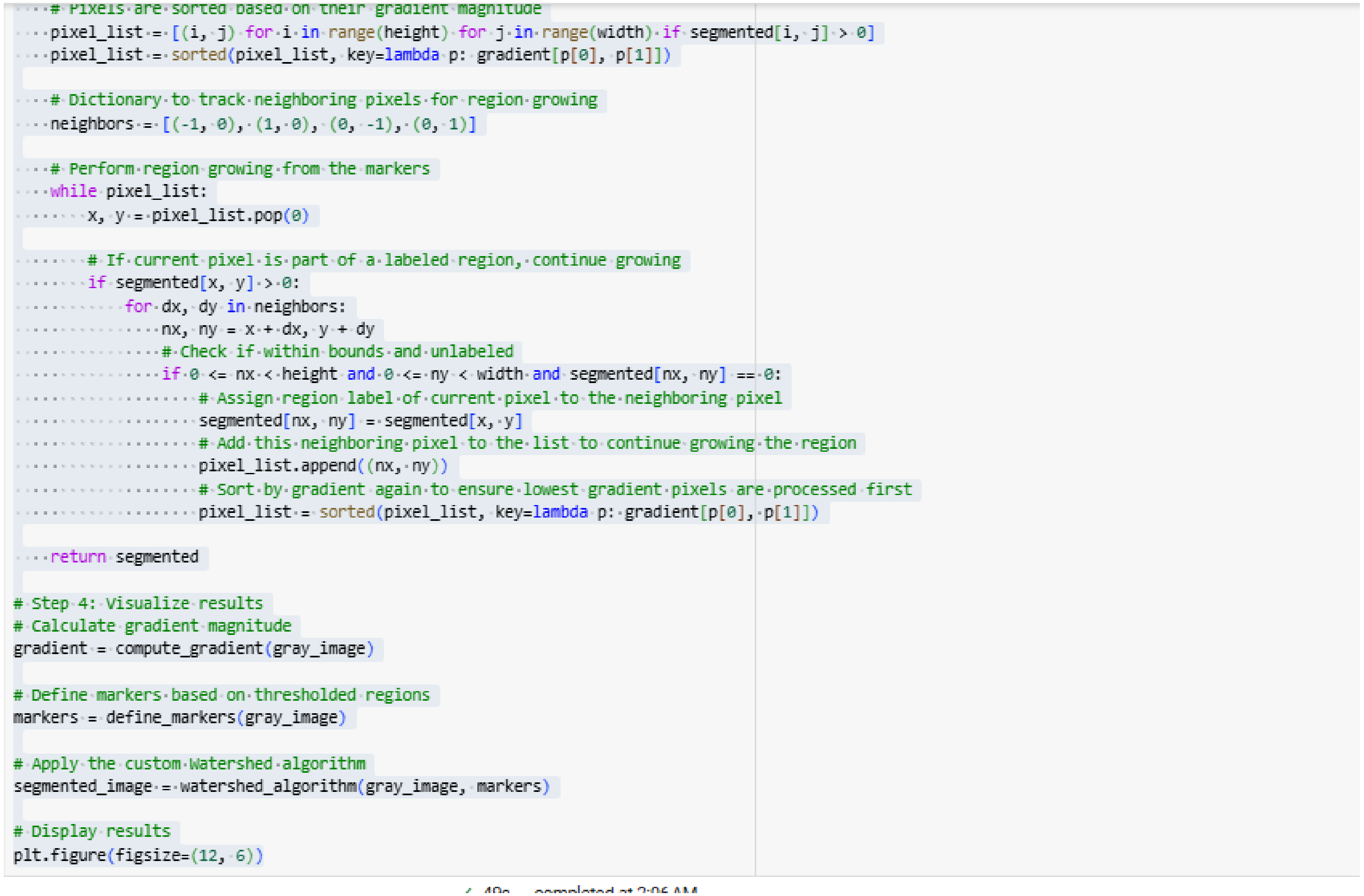
It is used to handle unknown sizes during region – growing segmentation.

Snapshots of Python features used



```python
# Brain MRI image segmentation using Watershed Algorithm
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Load the brain image and convert to grayscale
image = cv2.imread('/content/brain3.png') # Load the image from a file
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Step 1: Compute gradient magnitude (using Sobel operators to get the edge strength)
def compute_gradient(img):
    sobel_x = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)
    gradient_magnitude = np.sqrt(sobel_x ** 2 + sobel_y ** 2)
    return gradient_magnitude

# Step 2: Define markers (using a simple threshold to identify potential regions)
def define_markers(img, threshold=100):
    markers = np.zeros_like(img, dtype=np.int32)

    # Simple thresholding to identify initial markers
    markers[img < threshold] = 1  # Background marker
    markers[img > threshold * 2] = 2  # Foreground marker (tissue region)

    return markers

# Step 3: Implement Watershed algorithm by region growing from markers
def watershed_algorithm(img, markers):
    height, width = img.shape
    segmented = markers.copy()  # Copy of markers to fill with region labels
    gradient = compute_gradient(img)  # Gradient image for edge guidance

    # Create a list of pixel coordinates to process, starting with the markers
    # Pixels are sorted based on their gradient magnitude
    pixel_list = [(i, j) for i in range(height) for j in range(width) if segmented[i, j] > 0]
    pixel_list = sorted(pixel_list, key=lambda p: gradient[p[0], p[1]])
```

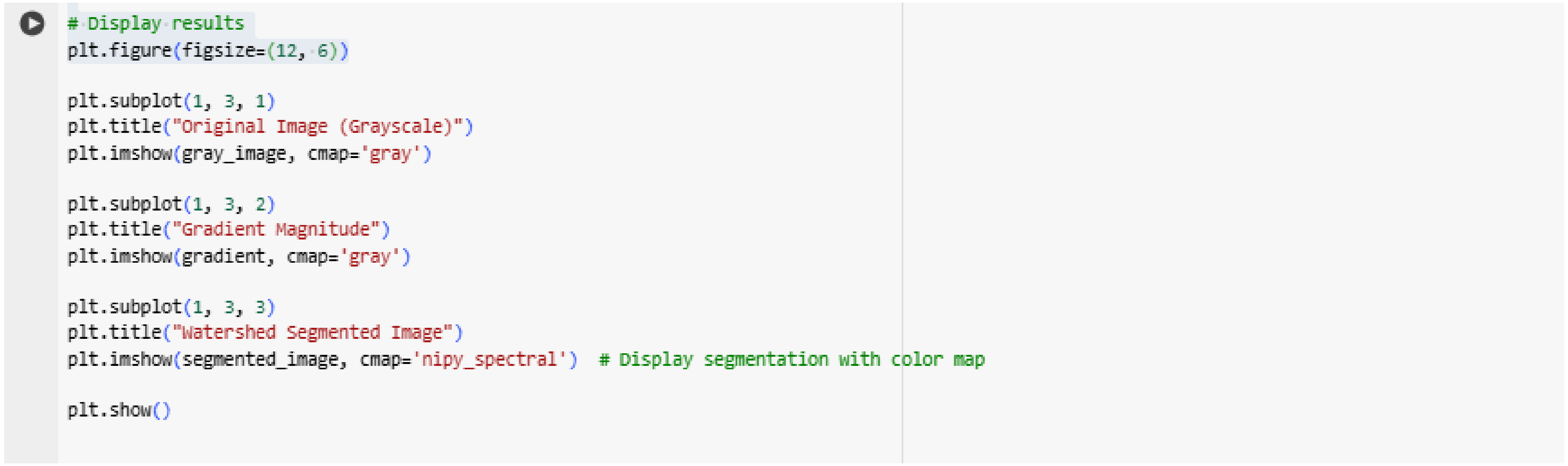
Snapshots of python features used



```
    # Pixels are sorted based on their gradient magnitude
    pixel_list = [(i, j) for i in range(height) for j in range(width) if segmented[i, j] > 0]
    pixel_list = sorted(pixel_list, key=lambda p: gradient[p[0], p[1]])

    # Dictionary to track neighboring pixels for region growing
    neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Perform region growing from the markers
    while pixel_list:
        x, y = pixel_list.pop(0)

        # If current pixel is part of a labeled region, continue growing
        if segmented[x, y] > 0:
            for dx, dy in neighbors:
                nx, ny = x + dx, y + dy
                # Check if within bounds and unlabeled
                if 0 <= nx < height and 0 <= ny < width and segmented[nx, ny] == 0:
                    # Assign region label of current pixel to the neighboring pixel
                    segmented[nx, ny] = segmented[x, y]
                    # Add this neighboring pixel to the list to continue growing the region
                    pixel_list.append((nx, ny))
                    # Sort by gradient again to ensure lowest gradient pixels are processed first
                    pixel_list = sorted(pixel_list, key=lambda p: gradient[p[0], p[1]])

    return segmented

# Step 4: Visualize results
# Calculate gradient magnitude
gradient = compute_gradient(gray_image)

# Define markers based on thresholded regions
markers = define_markers(gray_image)

# Apply the custom Watershed algorithm
segmented_image = watershed_algorithm(gray_image, markers)

# Display results
plt.figure(figsize=(12, 6))
```

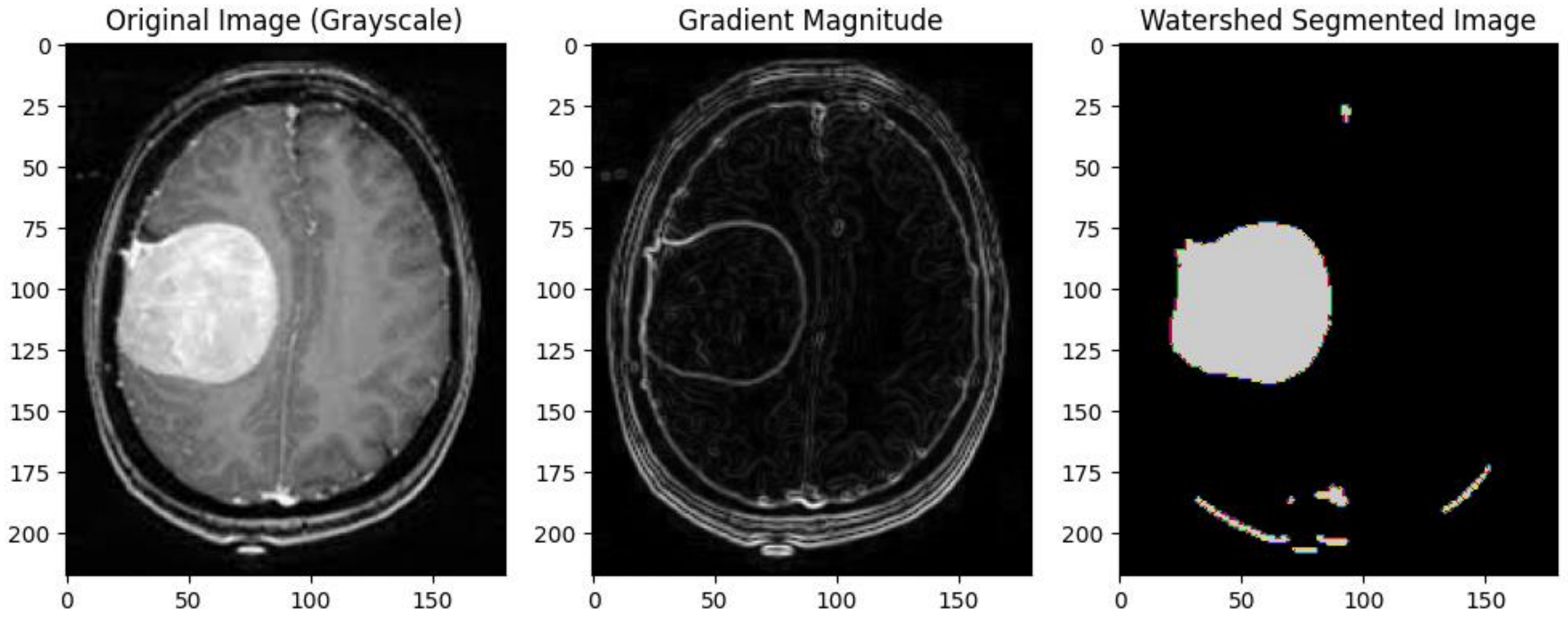
✓ 49s completed at 2:06 AM

Screenshot of python features used



```
# Display results
plt.figure(figsize=(12, 6))

plt.subplot(1, 3, 1)
plt.title("Original Image (Grayscale)")
plt.imshow(gray_image, cmap='gray')

plt.subplot(1, 3, 2)
plt.title("Gradient Magnitude")
plt.imshow(gradient, cmap='gray')

plt.subplot(1, 3, 3)
plt.title("Watershed Segmented Image")
plt.imshow(segmented_image, cmap='nipy_spectral')  # Display segmentation with color map

plt.show()
```

Results


Original Image (Grayscale)
Gradient Magnitude
Watershed Segmented Image
0
25
50
75
100
125
150
175
200
0
50
100
150

Observations

1. **Edge Detection Using Sobel Operator**:

In the compute_gradient function, the code calculates edge strenghth using Sobel operators along the x and y axes.

This approach helps to identify high-intensity transitions in the image, useful for detecting boundaries in MRI scans.

2. **Marker based Segmentation:**

The define_markers function creates markers based on intensity thresholds.

3. Watershed Implementation:

The watershed_algorithm function performs a manual, region-growing watershed algorithm.

4. Gradient-Based Pixel Processing Order:

The pixel processing order in watershed_algorithm is determined by sorting pixels based on their gradient magnitude. This ensures that regions expand from lower gradient areas first, giving smoother region boundaries and helping to prevent inaccurate segment growth in high gradient areas(edges).

5. **Dynamic Region Growing with Neighbor Tracking**:

The algorithm dynamically grows regions by tracking each pixel’s neighbors.

Observation and learning

6. **Image Visualization with Matplotlib**:

Visualization using matplotlib in the final part of the code provides a way to inspect the results step-by-step , displaying the original grayscale image, the gradient magnitude, and the segmented output.

7. **Computational Intensity:**

The code sorts pixel_list at each step within a loop. This watershed implementation may be slower on large images.

8. **Thresholding Simplicity**:

The define_markers function uses a fixed threshold to define regions, which may be overly simplistic for complex MRI images where intensity ranges can vary greatly across differents cans or subjects. More advanced methods such as adaptive thresholding or clustering could improve initial marker placement.

Thank You