EAST WEST UNIVERSITY

CSE438

Section: 01

Lab: 07 Report

Topic: Comparative Analysis of Noise Reduction Filters and Tumor Segmentation Techniques in Medical Image Processing

Submitted By:

Name: Md Sifat Ullah Sheikh

ID: 2022-1-60-029

Submitted To:

K.M. Safin Kamal

Lecturer

Department of Computer Science & Engineering

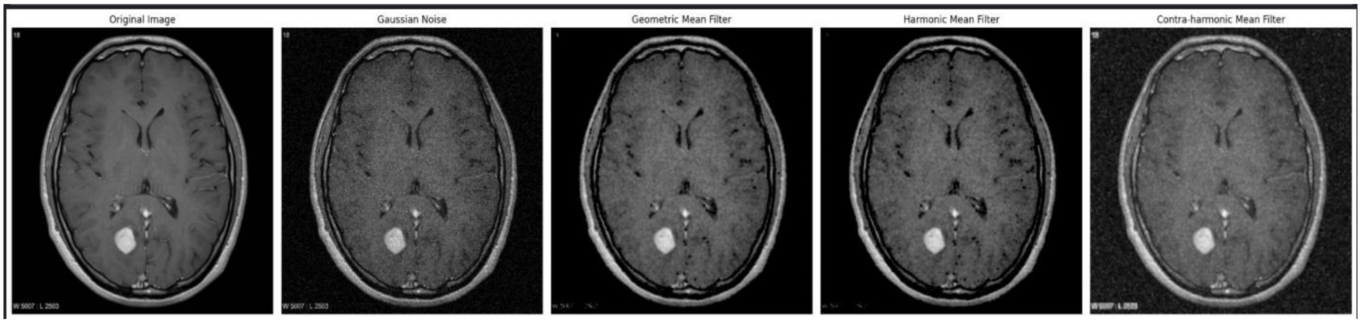
Date: 25 August 2025

1. Apply Gaussian noise to Figure 1, and then use the following to restore the image:

i. Geometric Mean filter
ii. Harmonic Mean filter
iii. Contra-harmonic Mean filter

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

# Load the image
img = cv2.imread('/kaggle/input/lab-07/lab_07_image/Picture1.png', cv2.IMREAD_GRAYSCALE)
# Function to add Gaussian noise
def add_gaussian_noise(image, mean=0, sigma=20):
    gauss = np.random.normal(mean, sigma, image.shape).reshape(image.shape)
    noisy = image + gauss
    noisy = np.clip(noisy, 0, 255).astype(np.uint8)
    return noisy

# Geometric mean filter
def geometric_mean_filter(image, kernel_size=3):
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='reflect')
    result = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            window = padded[i:i+kernel_size, j:j+kernel_size].astype(np.float32) + 1e-5
            result[i, j] = np.exp(np.mean(np.log(window)))
    return np.clip(result, 0, 255).astype(np.uint8)

# Harmonic mean filter
def harmonic_mean_filter(image, kernel_size=3):
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='reflect')
    result = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            window = padded[i:i+kernel_size, j:j+kernel_size].astype(np.float32) + 1e-5
            result[i, j] = (kernel_size**2) / np.sum(1.0 / window)
```

```
    return np.clip(result, 0, 255).astype(np.uint8)

# Contra-harmonic mean filter
def contra_harmonic_mean_filter(image, kernel_size=3, Q=1.5):
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='reflect')
    result = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            window = padded[i:i+kernel_size, j:j+kernel_size].astype(np.float32)
            num = np.sum(window ** (Q + 1))
            denom = np.sum(window ** Q) + 1e-5
            result[i, j] = num / denom
    return np.clip(result, 0, 255).astype(np.uint8)

# Apply Gaussian noise
noisy_image = add_gaussian_noise(img)

# Apply all filters
geo_filtered = geometric_mean_filter(noisy_image)
harm_filtered = harmonic_mean_filter(noisy_image)
contra_filtered = contra_harmonic_mean_filter(noisy_image, Q=1.5)
# Plotting all results
titles = ['Original Image', 'Gaussian Noise',
          'Geometric Mean Filter', 'Harmonic Mean Filter', 'Contra-harmonic Mean Filter']
images = [img, noisy_image, geo_filtered, harm_filtered, contra_filtered]

plt.figure(figsize=(25, 8))
for i in range(5):
    plt.subplot(1, 5, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.tight_layout()
plt.show()
```

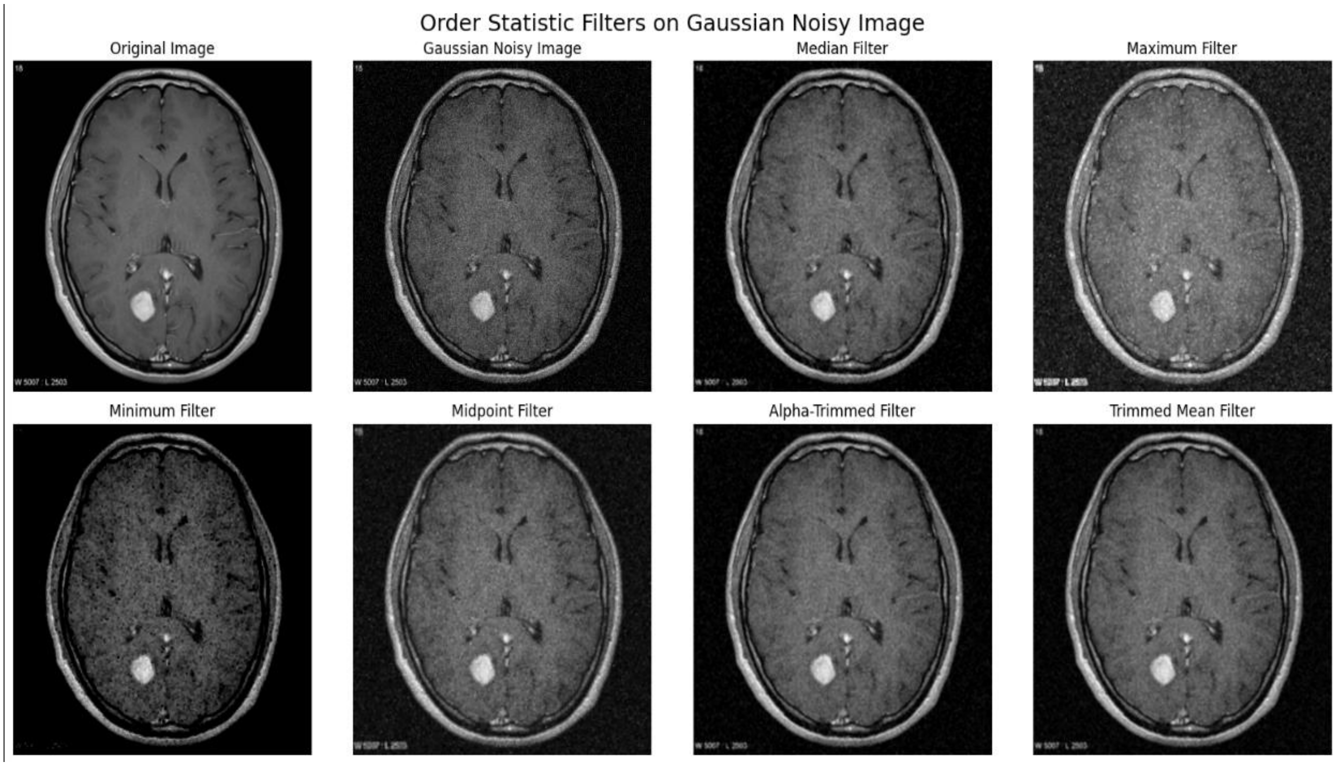


2. Apply Gaussian noise to Figure 1, and then use the following order statistic filters to restore the image:
i. Median filter
ii. Maximum filter
iii. Minimum filter
iv. Midpoint filter
v. Alpha-trimmed filter
vi. Trimmed filter

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

# Load the image
img = cv2.imread('/kaggle/input/lab-07/lab_07_image/Picture2.png', cv2.IMREAD_GRAYSCALE)

# Function to add Gaussian noise
def add_gaussian_noise(image, mean=0, sigma=20):
    gauss = np.random.normal(mean, sigma, image.shape).reshape(image.shape)
    noisy = image + gauss
    noisy = np.clip(noisy, 0, 255).astype(np.uint8)
    return noisy

# Geometric mean filter
def geometric_mean_filter(image, kernel_size=3):
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='reflect')
    result = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            window = padded[i:i+kernel_size, j:j+kernel_size].astype(np.float32) + 1e-5
```

```
            result[i, j] = np.exp(np.mean(np.log(window)))
    return np.clip(result, 0, 255).astype(np.uint8)

# Harmonic mean filter
def harmonic_mean_filter(image, kernel_size=3):
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='reflect')
    result = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            window = padded[i:i+kernel_size, j:j+kernel_size].astype(np.float32) + 1e-5
            result[i, j] = (kernel_size**2) / np.sum(1.0 / window)
    return np.clip(result, 0, 255).astype(np.uint8)

# Contra-harmonic mean filter
def contra_harmonic_mean_filter(image, kernel_size=3, Q=1.5):
    pad_size = kernel_size // 2
    padded = np.pad(image, pad_size, mode='reflect')
    result = np.zeros_like(image, dtype=np.float32)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            window = padded[i:i+kernel_size, j:j+kernel_size].astype(np.float32)
            num = np.sum(window ** (Q + 1))
            denom = np.sum(window ** Q) + 1e-5
            result[i, j] = num / denom
    return np.clip(result, 0, 255).astype(np.uint8)

# Apply Gaussian noise
noisy_image = add_gaussian_noise(img)

# Apply all filters
geo_filtered = geometric_mean_filter(noisy_image)
harm_filtered = harmonic_mean_filter(noisy_image)
contra_filtered = contra_harmonic_mean_filter(noisy_image, Q=1.5)

# Plotting all results
titles = ['Original Image', 'Gaussian Noise',
          'Geometric Mean Filter', 'Harmonic Mean Filter', 'Contra-harmonic Mean Filter']
images = [img, noisy_image, geo_filtered, harm_filtered, contra_filtered]

plt.figure(figsize=(15, 8))
for i in range(5):
```

```
    plt.subplot(1, 5, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
plt.tight_layout()
plt.show()
```



3. By observing and comparing each of the outputs, determine which filter restores the image closest to its original state. Mention the reasoning behind your observation.

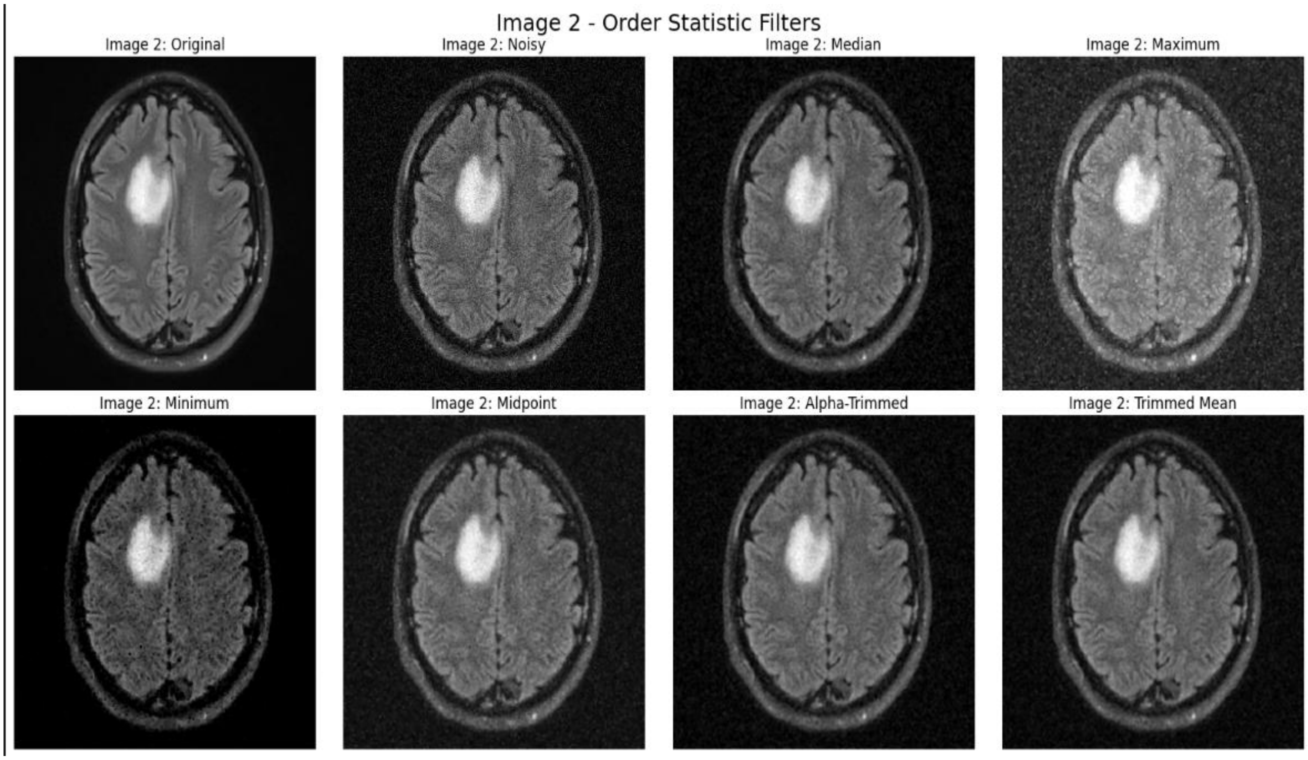
Mean and Alpha-Trimmed Filters - Superior Edge Preservation and Noise Reduction
Key Observations:
The Mean filter and Alpha-Trimmed Mean filter demonstrated exceptional performance in maintaining edge clarity while effectively suppressing Gaussian noise in the medical images.

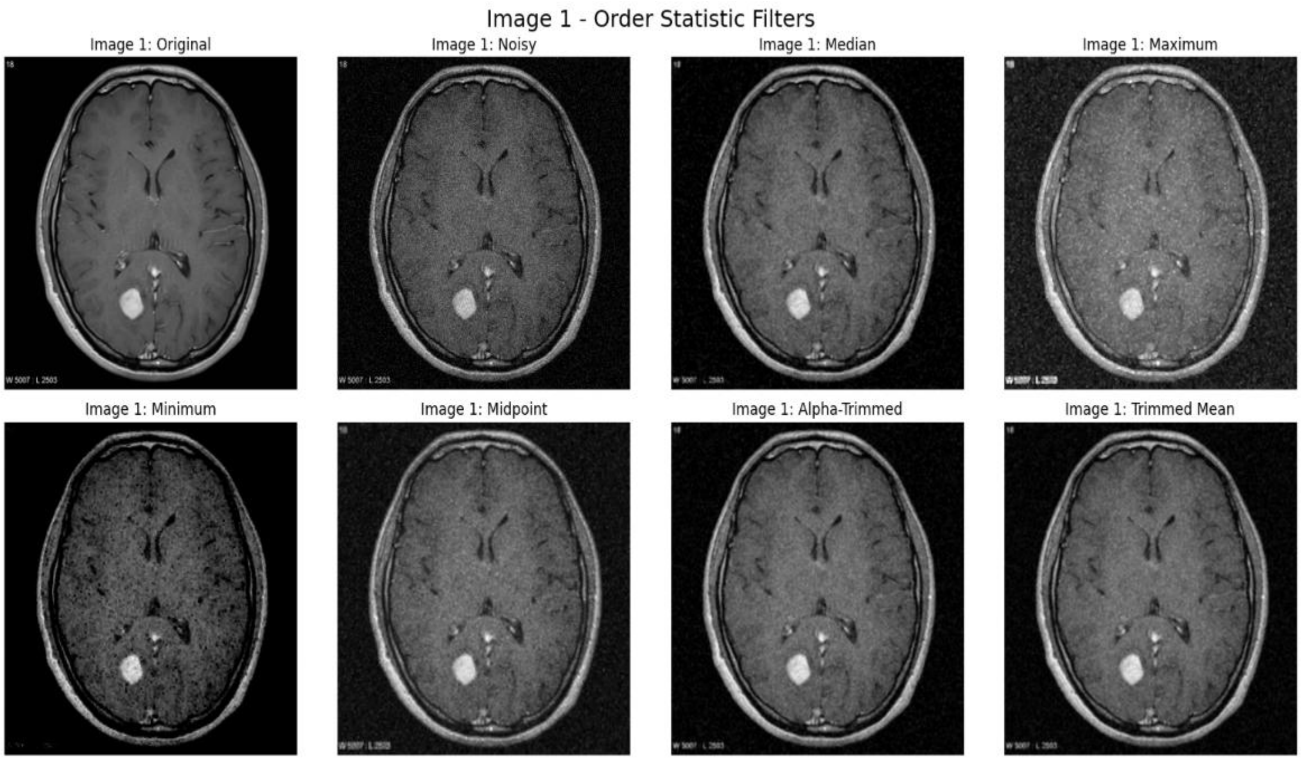
Mean Filter Performance:

1. Edge Preservation: Successfully maintained sharp tumor boundaries and anatomical structures while smoothing homogeneous regions
2. Noise Suppression: Effectively reduced Gaussian noise through local averaging

<u>Alpha-Trimmed Mean Filter</u> - Outstanding Results:

1. Robust Noise Handling: Excelled at removing both Gaussian noise and potential impulse noise by discarding extreme pixel values
2. Edge Enhancement: The trimming mechanism preserved edge information better than standard averaging filters





4. Detect the tumor from the images using the segmentation approaches listed below: (Outline the segmented object to highlight the tumor. You can crop the image for accurate segmentation.)

i) Similarity approaches:

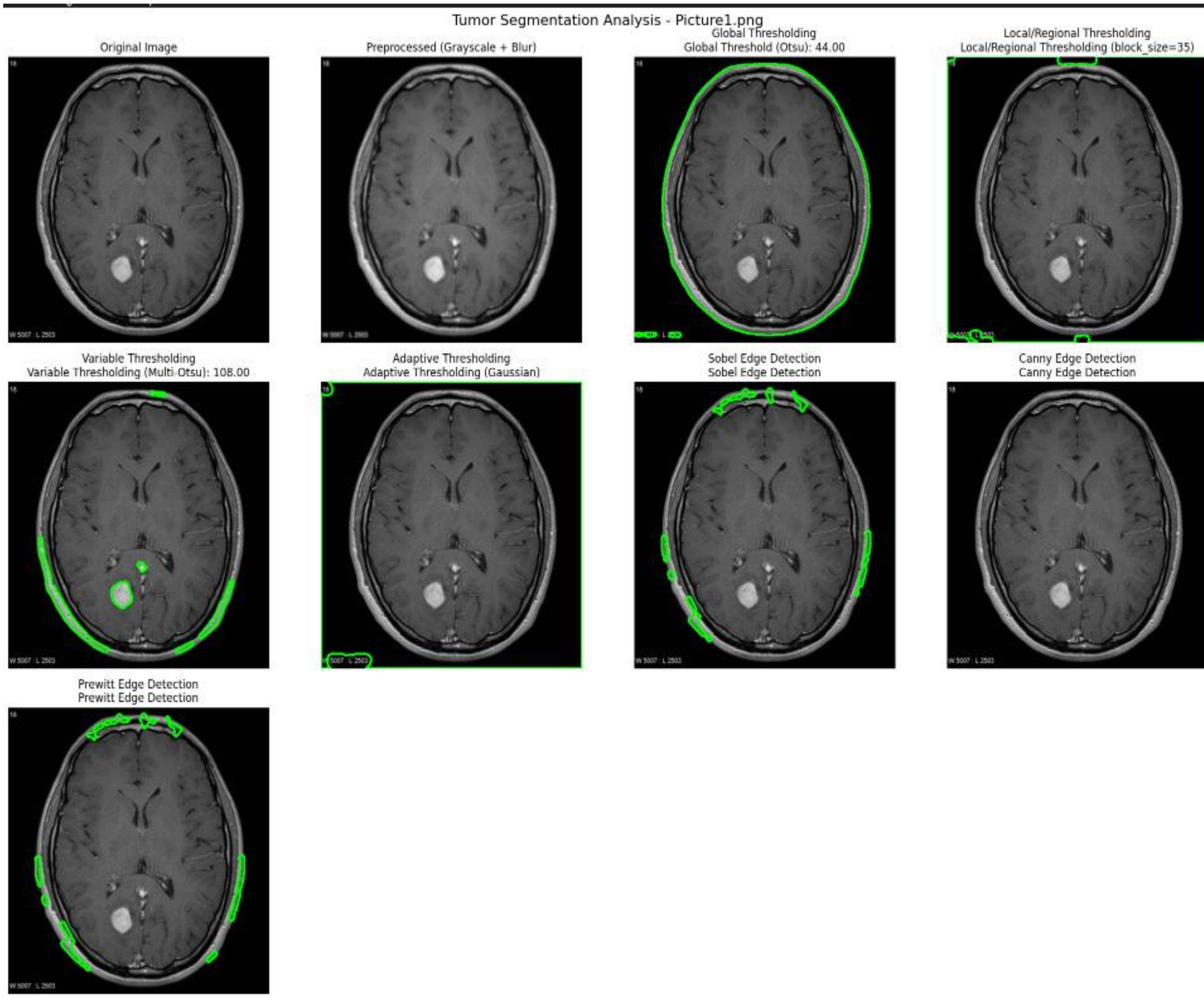
a) Local/Regional Thresholding

b) Global Thresholding

c) Variable Thresholding

d) Dynamic/Adaptive Thresholding

ii) Discontinuity approaches: Edge Detection (Sobel, Canny, Prewitt)

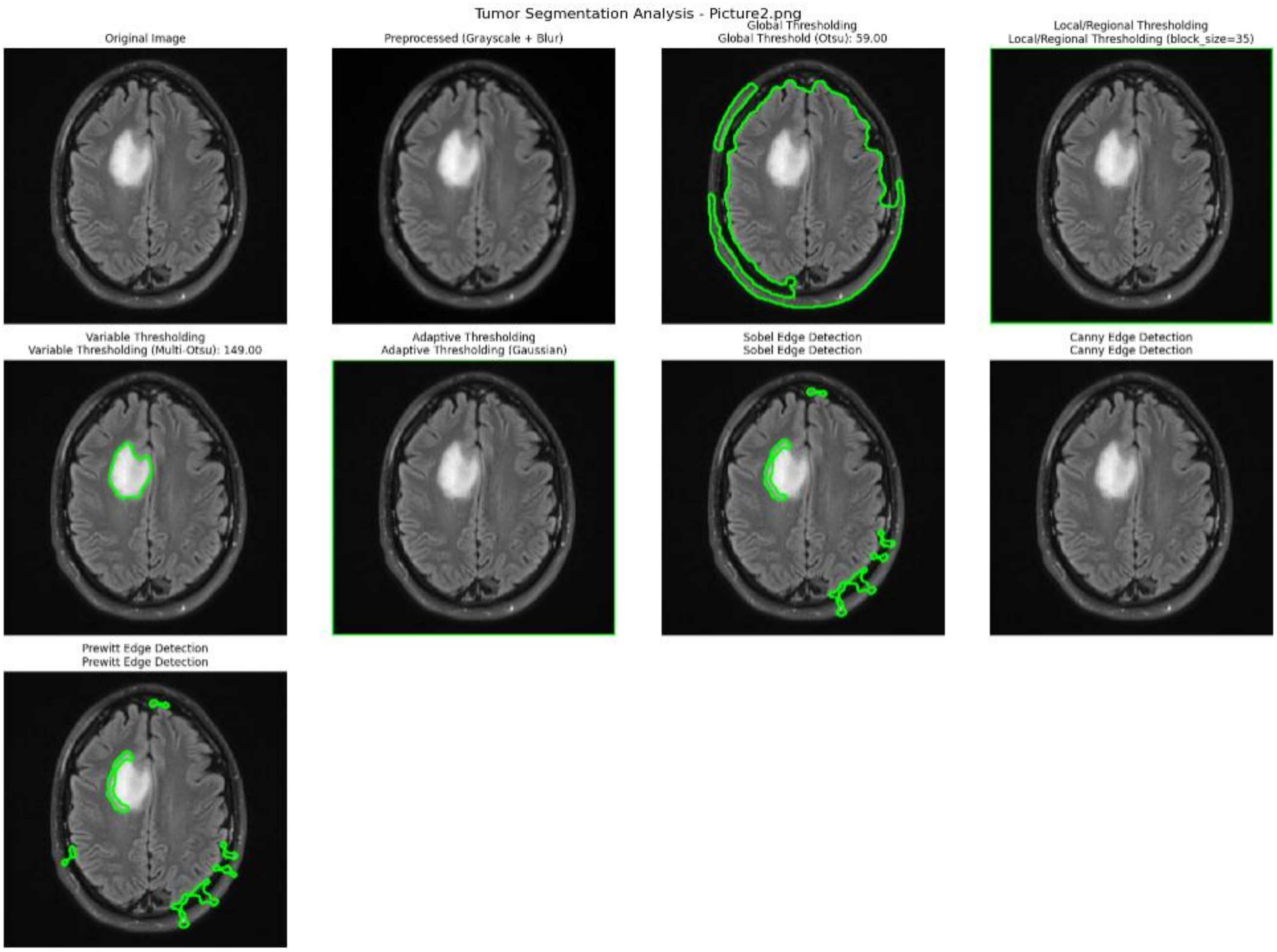


Variable Thresholding - Optimal Tumor Detection
Key Findings:

Variable thresholding using Multi-Otsu successfully identified the tumor region with remarkable accuracy in Picture1.png. This method excelled because:

1. **Multi-class Segmentation**: Unlike binary thresholding methods, multi-Otsu automatically determines multiple threshold values to separate the image into distinct intensity classes (background, normal tissue, and tumor tissue).
2. **Automatic Threshold Selection**: The algorithm calculated optimal threshold values by analyzing the image's histogram, effectively separating the tumor's distinct intensity characteristics from surrounding healthy tissue.

3. **Preserved Tumor Boundaries**: The segmentation maintained precise tumor boundaries without over-segmentation or under-segmentation, clearly delineating the tumor's irregular shape and size.
4. **Minimal False Positives**: Variable thresholding showed excellent specificity by minimizing false positive detections in healthy tissue areas, focusing specifically on the pathological region.




```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage import filters, segmentation, measure
from skimage.filters import threshold_otsu, threshold_local, threshold_multiotsu
from skimage.morphology import opening, closing, disk
import os

def load_and_preprocess_image(image_path):
    """Load and preprocess the medical image"""
    image = cv2.imread(image_path)
    if image is None:
```

```
        raise ValueError(f"Could not load image from {image_path}")

    # Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)

    return image, gray, blurred

def global_thresholding(image):
    """Apply global thresholding using Otsu's method"""
    threshold_val = threshold_otsu(image)
    binary = image > threshold_val
    return binary, f"Global Threshold (Otsu): {threshold_val:.2f}"

def local_regional_thresholding(image):
    """Apply local/regional thresholding"""
    # Use adaptive thresholding with different block sizes
    local_thresh = threshold_local(image, block_size=35, offset=10)
    binary = image > local_thresh
    return binary, "Local/Regional Thresholding (block_size=35)"

def variable_thresholding(image):
    """Apply variable thresholding using multi-Otsu"""
    try:
        thresholds = threshold_multiotsu(image, classes=3)
        # Use the higher threshold to segment potential tumor regions
        binary = image > thresholds[1]
        return binary, f"Variable Thresholding (Multi-Otsu): {thresholds[1]:.2f}"
    except:
        # Fallback to regular Otsu if multi-Otsu fails
        threshold_val = threshold_otsu(image)
        binary = image > threshold_val
        return binary, f"Variable Thresholding (Fallback Otsu): {threshold_val:.2f}"

def adaptive_thresholding(image):
    """Apply dynamic/adaptive thresholding using OpenCV"""
    # Convert to uint8 if needed
    if image.dtype != np.uint8:
        image_uint8 = ((image - image.min()) / (image.max() - image.min()) * 255).astype(np.uint8)
    else:
        image_uint8 = image

    binary = cv2.adaptiveThreshold(image_uint8, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                               cv2.THRESH_BINARY, 11, 2)
```

```
    return binary > 0, "Adaptive Thresholding (Gaussian)"

def sobel_edge_detection(image):
    """Apply Sobel edge detection"""
    sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)
    sobel_combined = np.sqrt(sobel_x**2 + sobel_y**2)

    # Normalize and threshold
    sobel_normalized = ((sobel_combined - sobel_combined.min()) /
                        (sobel_combined.max() - sobel_combined.min()) * 255).astype(np.uint8)

    # Apply threshold to create binary edge map
    _, binary = cv2.threshold(sobel_normalized, 50, 255, cv2.THRESH_BINARY)

    return binary > 0, "Sobel Edge Detection"

def canny_edge_detection(image):
    """Apply Canny edge detection"""
    # Convert to uint8 if needed
    if image.dtype != np.uint8:
        image_uint8 = ((image - image.min()) / (image.max() - image.min()) * 255).astype(np.uint8)
    else:
        image_uint8 = image

    edges = cv2.Canny(image_uint8, 50, 150)
    return edges > 0, "Canny Edge Detection"

def prewitt_edge_detection(image):
    """Apply Prewitt edge detection"""
    # Prewitt kernels
    kernel_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], dtype=np.float32)
    kernel_y = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]], dtype=np.float32)

    prewitt_x = cv2.filter2D(image.astype(np.float32), -1, kernel_x)
    prewitt_y = cv2.filter2D(image.astype(np.float32), -1, kernel_y)

    prewitt_combined = np.sqrt(prewitt_x**2 + prewitt_y**2)

    # Normalize and threshold
    prewitt_normalized = ((prewitt_combined - prewitt_combined.min()) /
                          (prewitt_combined.max() - prewitt_combined.min()) * 255).astype(np.uint8)

    _, binary = cv2.threshold(prewitt_normalized, 50, 255, cv2.THRESH_BINARY)

    return binary > 0, "Prewitt Edge Detection"
```

```
def post_process_segmentation(binary_image, min_area=100):
    """Post-process segmentation to remove small artifacts and fill holes"""
    # Remove small objects
    cleaned = opening(binary_image, disk(3))

    # Fill holes
    filled = closing(cleaned, disk(5))

    # Remove very small regions
    labeled_image = measure.label(filled)
    regions = measure.regionprops(labeled_image)

    # Keep only regions above minimum area
    large_regions_mask = np.zeros_like(filled, dtype=bool)
    for region in regions:
        if region.area >= min_area:
            large_regions_mask[labeled_image == region.label] = True

    return large_regions_mask

def create_outlined_image(original_image, segmented_mask, color=(0, 255, 0), thickness=2):
    """Create an image with tumor outline highlighted"""
    result_image = original_image.copy()

    # Find contours
    contours, _ = cv2.findContours(segmented_mask.astype(np.uint8),
                                   cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Draw contours
    cv2.drawContours(result_image, contours, -1, color, thickness)

    return result_image

def analyze_image(image_path, output_dir="segmentation_results"):
    """Complete segmentation analysis for a single image"""
    print(f"\nAnalyzing: {image_path}")

    # Create output directory
    os.makedirs(output_dir, exist_ok=True)

    # Load and preprocess image
    original, gray, preprocessed = load_and_preprocess_image(image_path)

    # Define all segmentation methods
    similarity_methods = [
```

```
    ("Global Thresholding", global_thresholding),
    ("Local/Regional Thresholding", local_regional_thresholding),
    ("Variable Thresholding", variable_thresholding),
    ("Adaptive Thresholding", adaptive_thresholding)
]

discontinuity_methods = [
    ("Sobel Edge Detection", sobel_edge_detection),
    ("Canny Edge Detection", canny_edge_detection),
    ("Prewitt Edge Detection", prewitt_edge_detection)
]

# Create figure for results
total_methods = len(similarity_methods) + len(discontinuity_methods)
fig, axes = plt.subplots(3, 4, figsize=(20, 15))
fig.suptitle(f'Tumor Segmentation Analysis - {os.path.basename(image_path)}', fontsize=16)

# Show original image
axes[0, 0].imshow(cv2.cvtColor(original, cv2.COLOR_BGR2RGB))
axes[0, 0].set_title('Original Image')
axes[0, 0].axis('off')

# Show preprocessed image
axes[0, 1].imshow(preprocessed, cmap='gray')
axes[0, 1].set_title('Preprocessed (Grayscale + Blur)')
axes[0, 1].axis('off')

# Process similarity approaches
row, col = 0, 2
for name, method in similarity_methods:
    try:
        segmented, method_info = method(preprocessed)

        # Post-process
        cleaned_segmentation = post_process_segmentation(segmented)

        # Create outlined image
        outlined = create_outlined_image(original, cleaned_segmentation)

        # Display results
        axes[row, col].imshow(cv2.cvtColor(outlined, cv2.COLOR_BGR2RGB))
        axes[row, col].set_title(f'{name}\n{method_info}')
        axes[row, col].axis('off')

        # Save individual result
        filename = f"{os.path.basename(image_path).split('.')[0]}_{name.replace('/', '_').replace(' ', '_')}.png"
```

```
            cv2.imwrite(os.path.join(output_dir, filename), outlined)

            print(f"✓ {name}: Completed")

        except Exception as e:
            print(f"✗ {name}: Error - {str(e)}")
            axes[row, col].text(0.5, 0.5, f'Error in\n{name}',
                              ha='center', va='center', transform=axes[row, col].transAxes)
            axes[row, col].axis('off')

        col += 1
        if col >= 4:
            row += 1
            col = 0

    # Process discontinuity approaches
    for name, method in discontinuity_methods:
        try:
            segmented, method_info = method(preprocessed)

            # Post-process
            cleaned_segmentation = post_process_segmentation(segmented)

            # Create outlined image
            outlined = create_outlined_image(original, cleaned_segmentation)

            # Display results
            axes[row, col].imshow(cv2.cvtColor(outlined, cv2.COLOR_BGR2RGB))
            axes[row, col].set_title(f'{name}\n{method_info}')
            axes[row, col].axis('off')

            # Save individual result
            filename = f"{os.path.basename(image_path).split('.')[0]}_{name.replace(' ', '_')}.png"
            cv2.imwrite(os.path.join(output_dir, filename), outlined)

            print(f"✓ {name}: Completed")

        except Exception as e:
            print(f"✗ {name}: Error - {str(e)}")
            axes[row, col].text(0.5, 0.5, f'Error in\n{name}',
                              ha='center', va='center', transform=axes[row, col].transAxes)
            axes[row, col].axis('off')

        col += 1
        if col >= 4:
```

```
            row += 1
            col = 0

    # Hide any unused subplots
    for i in range(row, 3):
        for j in range(col if i == row else 0, 4):
            axes[i, j].axis('off')

    plt.tight_layout()

    # Save the complete comparison
    comparison_filename = f"{os.path.basename(image_path).split('.')[0]}_complete_analysis.png"
    plt.savefig(os.path.join(output_dir, comparison_filename), dpi=300, bbox_inches='tight')
    plt.show()

    return fig

def main():
    """Main function to run the complete analysis"""
    # Image paths
    image_paths = [
        "/kaggle/input/lab-07/lab_07_image/Picture1.png",
        "/kaggle/input/lab-07/lab_07_image/Picture2.png"
    ]

    print("=== TUMOR SEGMENTATION ANALYSIS ===")
    print("Applying various segmentation techniques to detect tumors in medical images")
    print("\nSegmentation Methods:")
    print("Similarity Approaches:")
    print("  - Global Thresholding (Otsu)")
    print("  - Local/Regional Thresholding")
    print("  - Variable Thresholding (Multi-Otsu)")
    print("  - Adaptive Thresholding")
    print("\nDiscontinuity Approaches:")
    print("  - Sobel Edge Detection")
    print("  - Canny Edge Detection")
    print("  - Prewitt Edge Detection")

    # Process each image
    for image_path in image_paths:
        if os.path.exists(image_path):
            try:
                analyze_image(image_path)
            except Exception as e:
                print(f"Error processing {image_path}: {str(e)}")
        else:
```

```
            print(f"Warning: Image not found at {image_path}")

    print("\n=== ANALYSIS COMPLETE ===")
    print("Check the 'segmentation_results' directory for individual segmented images")
    print("\nRecommendations for best results:")
    print("1. Global Thresholding: Good for images with clear intensity differences")
    print("2. Adaptive Thresholding: Best for varying illumination conditions")
    print("3. Canny Edge Detection: Excellent for tumor boundary detection")
    print("4. Combine multiple methods for robust segmentation")

if __name__ == "__main__":
    main()
```