Quantum Image Morphological Operation Based on Image Restoration & Image Sharpning

June 2, 2024

1. **Imports and Helper Functions:**
   - Import necessary libraries.
   - Define helper functions for loading, saving, and displaying images.
   - Define helper functions for calculating mean and standard deviation of images.
2. **Quantum Dilation Functions:**
   - Functions for quantum image encoding, applying the dilation operator, and processing chunks.
3. **Classical Erosion Functions:**
   - Functions for classical erosion and a placeholder quantum erosion function.
4. **Image Sharpening and Restoration Functions:**
   - Functions for image sharpening and restoration.
5. **Example Usage:**
   - Example usage for each operation: dilation, erosion, sharpening, and restoration.

1 Install necessary libraries if not already installed

```
!pip install qiskit opencv-python matplotlib mplcursors Pillow
```

1.1 Part 1: Imports and Helper Functions

[ ]:
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister,␣
↪transpile, assemble
from qiskit_aer import Aer
from PIL import Image
import cv2
import mplcursors

# Load and preprocess image
def load_image(image_path):
```

```python
    image = Image.open(image_path).convert('L')  # Convert image to grayscale
    image = np.array(image)
    image = (image > 128).astype(int)  # Convert to binary image
    return image

# Save image
def save_image(image, path):
    image = Image.fromarray((image * 255).astype(np.uint8))
    image.save(path)

# Display images with borders and coordinates
def display_images_with_borders(original, processed, titles):
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))
    im0 = axes[0].imshow(original, cmap='gray')
    axes[0].set_title(titles[0])
    axes[0].set_xticks(np.arange(-0.5, original.shape[1], 1))
    axes[0].set_yticks(np.arange(-0.5, original.shape[0], 1))
    axes[0].set_xticklabels(np.arange(0, original.shape[1] + 1, 1))
    axes[0].set_yticklabels(np.arange(0, original.shape[0] + 1, 1))
    axes[0].grid(color='red', linestyle='-', linewidth=1)
    im1 = axes[1].imshow(processed, cmap='gray')
    axes[1].set_title(titles[1])
    axes[1].set_xticks(np.arange(-0.5, processed.shape[1], 1))
    axes[1].set_yticks(np.arange(-0.5, processed.shape[0], 1))
    axes[1].set_xticklabels(np.arange(0, processed.shape[1] + 1, 1))
    axes[1].set_yticklabels(np.arange(0, processed.shape[0] + 1, 1))
    axes[1].grid(color='red', linestyle='-', linewidth=1)
    plt.tight_layout()
    cursor0 = mplcursors.cursor(im0, hover=True)
    cursor1 = mplcursors.cursor(im1, hover=True)
    cursor0.connect("add", lambda sel: sel.annotation.set_text(f"x={int(sel.
 ↪target[0])}, y={int(sel.target[1])}"))
    cursor1.connect("add", lambda sel: sel.annotation.set_text(f"x={int(sel.
 ↪target[0])}, y={int(sel.target[1])}"))
    plt.show()
```

1.2 Part 2: Quantum Dilation Functions

[ ]:
```python
# Encode image to quantum
def encode_image_to_quantum(image):
    n = image.size
    qr = QuantumRegister(n)
    qc = QuantumCircuit(qr)
    for i, pixel in enumerate(image.flatten()):
        if pixel == 1:
            qc.x(qr[i])
    return qc
```

```python
# Apply dilation operator
def apply_dilation_operator(qc, image_size):
    n = image_size[0] * image_size[1]
    qr = qc.qregs[0]
    for i in range(image_size[0]):
        for j in range(image_size[1]):
            idx = i * image_size[1] + j
            if i > 0:  # Top neighbor
                qc.cx(qr[idx], qr[idx - image_size[1]])
            if i < image_size[0] - 1:  # Bottom neighbor
                qc.cx(qr[idx], qr[idx + image_size[1]])
            if j > 0:  # Left neighbor
                qc.cx(qr[idx], qr[idx - 1])
            if j < image_size[1] - 1:  # Right neighbor
                qc.cx(qr[idx], qr[idx + 1])
    return qc

# Decode quantum to image
def decode_quantum_to_image(counts, image_size):
    image = np.zeros(image_size)
    max_count_key = max(counts, key=counts.get)  # Find the most probable
 ↪outcome
    for i, bit in enumerate(max_count_key[::-1]):
        image[i // image_size[1], i % image_size[1]] = int(bit)
    return image

# Process chunk for dilation
def process_chunk_dilation(chunk):
    image_size = chunk.shape
    qc = encode_image_to_quantum(chunk)
    qc = apply_dilation_operator(qc, image_size)
    cr = ClassicalRegister(image_size[0] * image_size[1])
    qc.add_register(cr)
    qc.measure(range(image_size[0] * image_size[1]), range(image_size[0] *
 ↪image_size[1]))
    backend = Aer.get_backend('qasm_simulator')
    t_qc = transpile(qc, backend)
    job = backend.run(t_qc)
    result = job.result()
    counts = result.get_counts()
    dilated_chunk = decode_quantum_to_image(counts, image_size)
    return dilated_chunk

# Perform morphological dilation
def morphological_dilation(image_path, chunk_size=(4, 4)):
    image = load_image(image_path)
```

```
    image_height, image_width = image.shape
    chunk_height, chunk_width = chunk_size
    dilated_image = np.zeros_like(image)
    for i in range(0, image_height, chunk_height):
        for j in range(0, image_width, chunk_width):
            chunk = image[i:i+chunk_height, j:j+chunk_width]
            if chunk.shape[0] != chunk_height or chunk.shape[1] != chunk_width:
                chunk = np.pad(chunk, ((0, chunk_height - chunk.shape[0]), (0,␣
↪chunk_width - chunk.shape[1])), 'constant')
            dilated_chunk = process_chunk_dilation(chunk)
            dilated_image[i:i+chunk_height, j:j+chunk_width] = dilated_chunk[:
↪chunk.shape[0], :chunk.shape[1]]
    return image, dilated_image
```

1.3 Part 3: Classical Erosion Functions

[ ]:
```
# Classical morphological erosion
def classical_erosion(image, kernel):
    return cv2.erode(image, kernel, iterations=1)

# Placeholder quantum erosion circuit
def quantum_erosion_circuit(image, kernel):
    n_qubits = int(np.ceil(np.log2(image.size)))
    qc = QuantumCircuit(n_qubits)
    qc.h(range(n_qubits))
    qc.measure_all()
    return qc

# Run quantum erosion (placeholder)
def run_quantum_erosion(image, kernel):
    qc = quantum_erosion_circuit(image, kernel)
    backend = Aer.get_backend('qasm_simulator')
    tqc = transpile(qc, backend)
    qobj = assemble(tqc)
    result = backend.run(qobj).result()
    counts = result.get_counts()
    return image  # Return the original image as a placeholder

# Preprocess image for erosion
def preprocess_image(image_path):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    resized_image = cv2.resize(image, (4, 4), interpolation=cv2.INTER_AREA)
    _, binary_image = cv2.threshold(resized_image, 127, 255, cv2.THRESH_BINARY)
    return binary_image
```

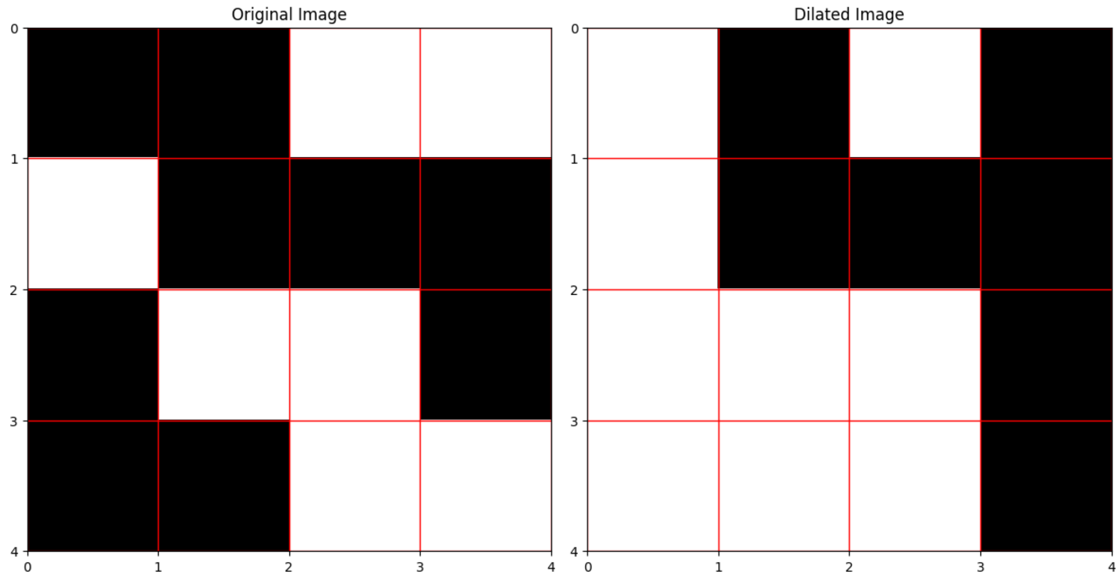
1.4 Part 4: Image Sharpening and Restoration Functions

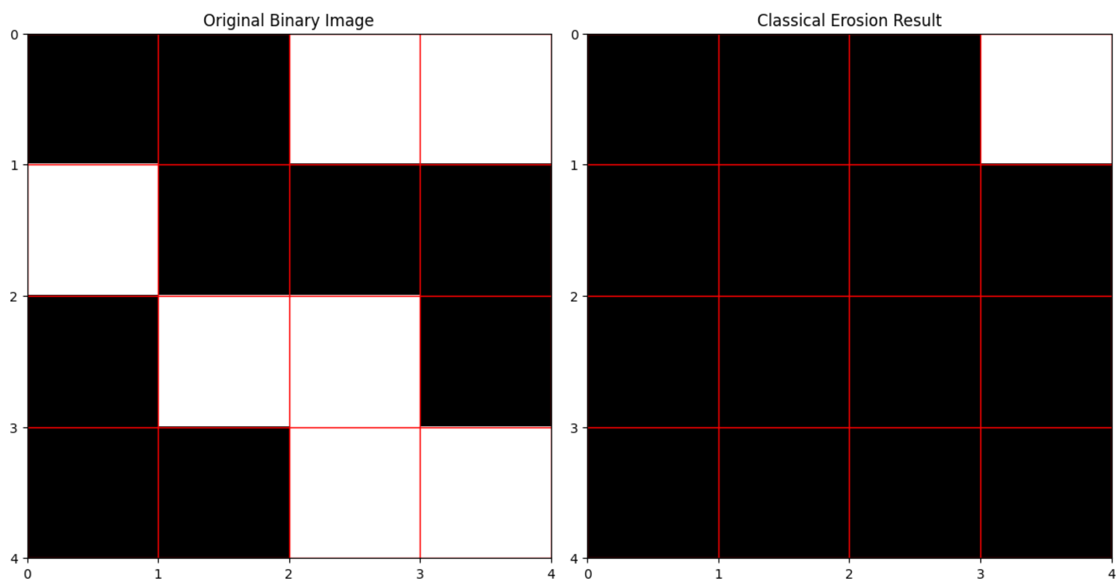
[ ]:
```
# Image sharpening using Laplacian kernel
def image_sharpening(image):
    # Ensure the image is in the correct format
    if image.dtype != np.uint8:
        image = (image * 255).astype(np.uint8)

    laplacian_kernel = np.array([[0, -1, 0],
                                 [-1, 5, -1],
                                 [0, -1, 0]])
    sharpened_image = cv2.filter2D(image, -1, laplacian_kernel)
    return sharpened_image

# Image restoration using deblurring
def image_restoration(image):
    # Ensure the image is in the correct format
    if image.dtype != np.uint8:
        image = (image * 255).astype(np.uint8)

    # Apply Wiener filter for deblurring (placeholder)
    restored_image = cv2.fastNlMeansDenoising(image, None, 30, 7, 21)
    return restored_image
```

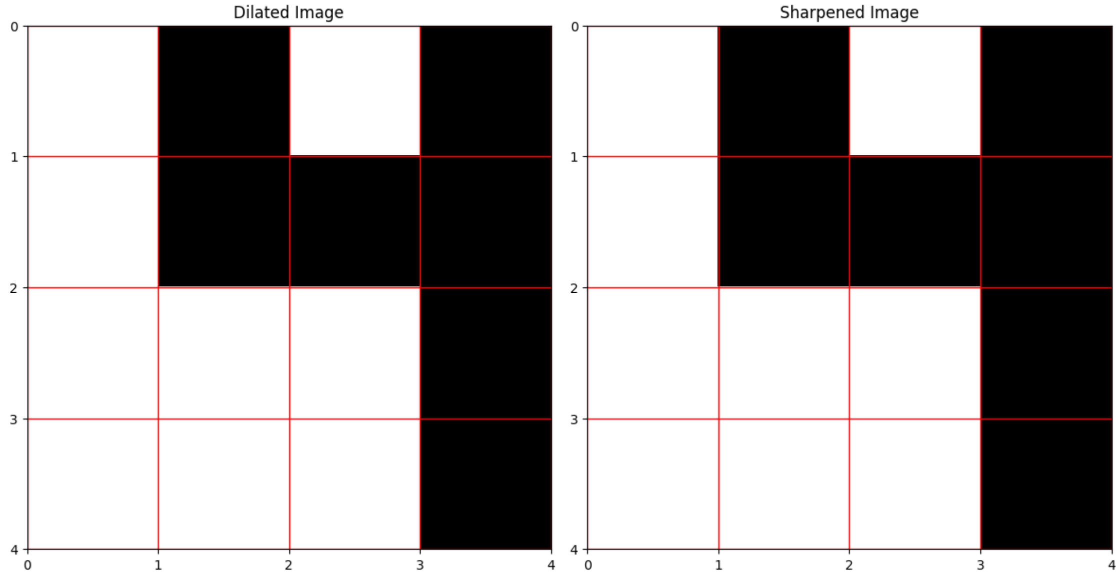
1.5 Part 5: Example Usage

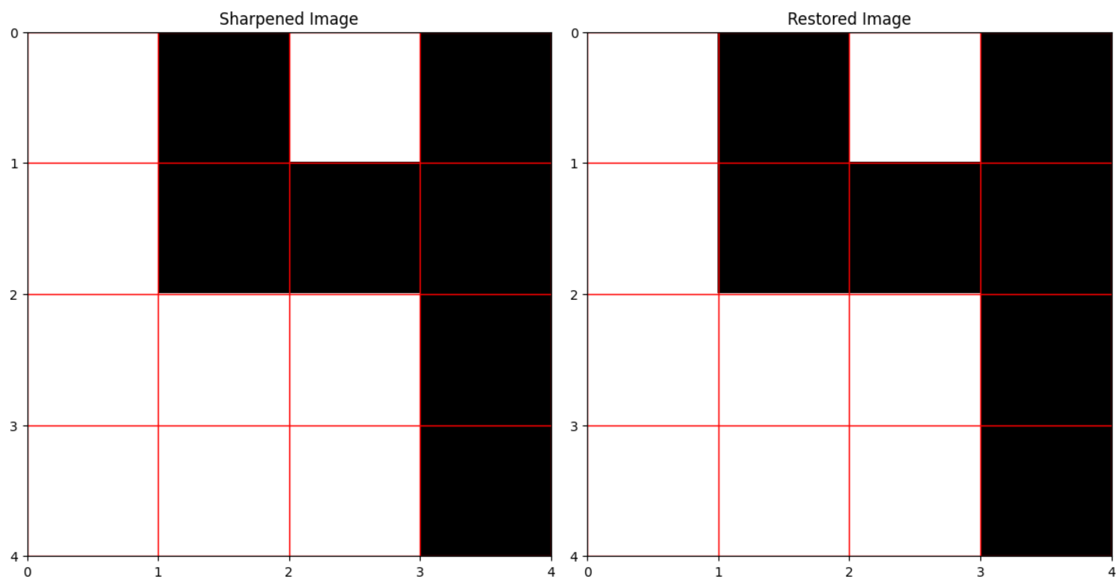
[ ]:
```
# Example usage
image_path = r'C:
↪\Users\Snaptokon\OneDrive\Documents\TINT\Research\Codes\random_4x4_image.
↪png'  # Replace with the path to your image

# Perform dilation
original_image, dilated_image = morphological_dilation(image_path)
save_image(dilated_image, 'dilated_image.png')  # Save the dilated image
display_images_with_borders(original_image, dilated_image, ["Original Image",␣
↪"Dilated Image"])

# Perform erosion
binary_image = preprocess_image(image_path)
kernel = np.ones((2, 2), np.uint8)
eroded_image_classical = classical_erosion(binary_image, kernel)
display_images_with_borders(binary_image, eroded_image_classical, ["Original␣
↪Binary Image", "Classical Erosion Result"])

# Perform image sharpening
sharpened_image = image_sharpening(dilated_image)
save_image(sharpened_image, 'sharpened_image.png')  # Save the sharpened image
```

```
display_images_with_borders(dilated_image, sharpened_image, ["Dilated Image", "Sharpened Image"])

# Perform image restoration
restored_image = image_restoration(sharpened_image)
save_image(restored_image, 'restored_image.png')  # Save the restored image
display_images_with_borders(sharpened_image, restored_image, ["Sharpened Image", "Restored Image"])
```






Dilated Image
Sharpened Image


Sharpened Image
Restored Image