MIAP HW3

Arya Koureshi (401204008)

arya.koureshi@gmail.com

Theory Section

Question 1

To solve the optimization problem:

$\min_x \left\{ \lambda \|\boldsymbol{X} - \boldsymbol{Y}\|_2^2 + \sum_{i,j} \|\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X}\|_2^2 \right\},$

I can follow these steps:

1. **Rewrite the objective function**:

First, let's rewrite the objective function more explicitly:

$f(\boldsymbol{X}) = \lambda \|\boldsymbol{X} - \boldsymbol{Y}\|_2^2 + \sum_{i,j} \|\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X}\|_2^2$

2. **Expand and combine terms**:

I expand each norm squared term. For the first term:

$\|\boldsymbol{X} - \boldsymbol{Y}\|_2^2 = (\boldsymbol{X} - \boldsymbol{Y})^T(\boldsymbol{X} - \boldsymbol{Y})$

For the second term in the sum:

$\|\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X}\|_2^2 = (\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X})^T(\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X})$

Combining these terms, I get:

$f(\boldsymbol{X}) = \lambda(\boldsymbol{X} - \boldsymbol{Y})^T(\boldsymbol{X} - \boldsymbol{Y}) + \sum_{i,j}(\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X})^T(\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X})$

3. **Take the gradient and set to zero**:

To minimize $f(\boldsymbol{X})$, I take the gradient with respect to $\boldsymbol{X}$ and set it to zero.

First, compute the gradient of each term. For the first term:

$\nabla_{\boldsymbol{X}} \left(\lambda(\boldsymbol{X} - \boldsymbol{Y})^T(\boldsymbol{X} - \boldsymbol{Y}) \right) = 2\lambda(\boldsymbol{X} - \boldsymbol{Y})$

For the second term:

$\nabla_{\boldsymbol{X}} \left(\sum_{i,j}(\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X})^T(\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X}) \right) = \sum_{i,j} -2\boldsymbol{R}_{i,j}^T(\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X})$

Setting the sum of the gradients to zero:

$2\lambda(\boldsymbol{X} - \boldsymbol{Y}) - 2\sum_{i,j} \boldsymbol{R}_{i,j}^T(\boldsymbol{D}\alpha_{i,j} - \boldsymbol{R}_{i,j}\boldsymbol{X}) = 0$

Simplify and rearrange terms:

$2\lambda\boldsymbol{X} - 2\lambda\boldsymbol{Y} - 2\sum_{i,j} \boldsymbol{R}_{i,j}^T\boldsymbol{D}\alpha_{i,j} + 2\sum_{i,j} \boldsymbol{R}_{i,j}^T\boldsymbol{R}_{i,j}\boldsymbol{X} = 0$

$\lambda\boldsymbol{X} + \sum_{i,j} \boldsymbol{R}_{i,j}^T\boldsymbol{R}_{i,j}\boldsymbol{X} = \lambda\boldsymbol{Y} + \sum_{i,j} \boldsymbol{R}_{i,j}^T\boldsymbol{D}\alpha_{i,j}$

Factor out $\boldsymbol{X}$ on the left-hand side:

$\left(\lambda\boldsymbol{I} + \sum_{i,j} \boldsymbol{R}_{i,j}^T\boldsymbol{R}_{i,j} \right) \boldsymbol{X} = \lambda\boldsymbol{Y} + \sum_{i,j} \boldsymbol{R}_{i,j}^T\boldsymbol{D}\alpha_{i,j}$

4. **Solve for $\boldsymbol{X}$**:

Finally, I solve for $\boldsymbol{X}$:

$\boldsymbol{X} = \left(\lambda\boldsymbol{I} + \sum_{i,j} \boldsymbol{R}_{i,j}^T\boldsymbol{R}_{i,j} \right)^{-1} \left(\lambda\boldsymbol{Y} + \sum_{i,j} \boldsymbol{R}_{i,j}^T\boldsymbol{D}\alpha_{i,j} \right)$

This matches the provided solution:

$$\hat{\boldsymbol{X}} = \left(\lambda \boldsymbol{I} + \sum_{i,j} \boldsymbol{R}_{i,j}^T \boldsymbol{R}_{i,j}\right)^{-1} \left(\lambda \boldsymbol{Y} + \sum_{i,j} \boldsymbol{R}_{i,j} \boldsymbol{D} \hat{\alpha}_{i,j}\right)$$

Explanation of Matrices and $\boldsymbol{R}_{i,j}$:

- $\boldsymbol{X}$: The signal or data I aim to reconstruct.
- $\boldsymbol{Y}$: The observed signal or data.
- λ: A regularization parameter balancing the fidelity term and the dictionary term.
- $\boldsymbol{D}$: The dictionary matrix containing atoms used to represent the data.
- $\alpha_{i,j}$: Coefficients corresponding to the representation of patches of $\boldsymbol{X}$ using the dictionary $\boldsymbol{D}$.
- $\boldsymbol{R}_{i,j}$: Selection matrix (or operator) that extracts the (i, j)-th patch from $\boldsymbol{X}$. It is typically a binary matrix with entries of 1 where it selects elements from $\boldsymbol{X}$, and 0 elsewhere.

The function of $\boldsymbol{R}_{i,j}$ is to select or extract patches (subsections) from the larger signal $\boldsymbol{X}$, which are then used to compare against the dictionary representations. This allows for localized processing and reconstruction of the signal.

Question 2

To derive the expressions for u_{ij} and w_j given the cost function in the Fuzzy C-Means (FCM) clustering, I start with the provided cost function:

$$J = \frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{N} u_{ij}^2 ||x_i - w_j||_2^2 + \sum_{i=1}^{N} \lambda_i \left(\sum_{j=1}^{K} u_{ij} - 1\right).$$

Step 1: Derive u_{ij}

I need to find the value of u_{ij} that minimizes the cost function. First, let's partially differentiate J with respect to u_{ij}:

$$\frac{\partial J}{\partial u_{ij}} = \frac{2}{N} u_{ij} ||x_i - w_j||_2^2 + \lambda_i.$$

Set this partial derivative to zero for minimization:

$$\frac{2}{N} u_{ij} ||x_i - w_j||_2^2 + \lambda_i = 0.$$

Rearrange to solve for u_{ij}:

$$u_{ij} ||x_i - w_j||_2^2 = -\frac{\lambda_i N}{2}.$$

Since u_{ij} are fuzzy memberships and must satisfy the constraint $\sum_{j=1}^{K} u_{ij} = 1$, I can use Lagrange multipliers to enforce this constraint. The expression then becomes:

$$u_{ij} = \frac{\frac{1}{||x_i - w_j||_2^2}}{\sum_{l=1}^{K} \frac{1}{||x_i - w_l||_2^2}}.$$

Step 2: Derive w_j

Next, I derive the update rule for the cluster centroids w_j. Partially differentiate J with respect to w_j:

$$\frac{\partial J}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} 2u_{ij}^2 (w_j - x_i).$$

Set this partial derivative to zero for minimization:

$$\sum_{i=1}^{N} u_{ij}^2 (w_j - x_i) = 0.$$

Solve for w_j:

$$w_j \sum_{i=1}^{N} u_{ij}^2 = \sum_{i=1}^{N} u_{ij}^2 x_i.$$

$$w_j = \frac{\sum_{i=1}^{N} u_{ij}^2 x_i}{\sum_{i=1}^{N} u_{ij}^2}.$$

Thus, I have shown that u_{ij} and w_j are obtained as follows:

$$u_{ij} = \frac{\frac{1}{||x_i - w_j||_2^2}}{\sum_{l=1}^{K} \frac{1}{||x_i - w_l||_2^2}},$$

$$w_j = \frac{\sum_{i=1}^{N} u_{ij}^2 x_i}{\sum_{i=1}^{N} u_{ij}^2}.$$

Question 3

Deep Learning Methods for Image Denoising

Image denoising is a critical preprocessing step in many image analysis and computer vision tasks. Various deep learning methods have been developed to address the challenges of image denoising. This overview will comprehensively explain three prominent methods: Convolutional Neural Networks (CNNs), Generative Adversarial Networks (GANs), and Autoencoders.

1. Convolutional Neural Networks (CNNs)

CNNs are widely used in image processing tasks, including image denoising. They work by learning to map noisy images to clean images using a series of convolutional layers that extract features at various levels of abstraction.

- **Architecture**: CNNs for denoising typically consist of an input layer, multiple hidden layers (including convolutional layers, ReLU activation functions, and sometimes pooling layers), and an output layer that reconstructs the denoised image.
- **Training**: The network is trained using pairs of noisy and clean images. The loss function, often the mean squared error (MSE) between the predicted clean image and the ground truth, is minimized using backpropagation.
- **Advantages**: CNNs can effectively capture spatial hierarchies in images, making them suitable for denoising structured noise. They are relatively fast during inference due to their efficient implementation on GPUs.

2. Generative Adversarial Networks (GANs)

GANs, introduced by Goodfellow et al., consist of two networks: a generator and a discriminator. The generator attempts to create denoised images, while the discriminator evaluates the quality of these images by distinguishing them from real clean images.

- **Architecture**: The generator network in a GAN for denoising often resembles an autoencoder, with convolutional layers for encoding and decoding. The discriminator is a binary classifier that predicts whether an input image is real or generated.
- **Training**: The generator and discriminator are trained simultaneously in a minimax game. The generator aims to minimize the discriminator's ability to distinguish between real and fake images, while the discriminator aims to maximize its classification accuracy.
- **Advantages**: GANs can produce highly realistic denoised images and can handle complex noise patterns better than traditional methods. They are particularly effective in cases where the noise characteristics are difficult to model.

3. Autoencoders

Autoencoders are neural networks designed to learn efficient representations of data, typically for the purpose of dimensionality reduction or denoising.

- **Architecture**: An autoencoder consists of an encoder and a decoder. The encoder maps the input image to a lower-dimensional latent space, and the decoder reconstructs the image from this latent representation.
- **Training**: Autoencoders are trained to minimize the reconstruction error, usually measured by the MSE between the input noisy image and the reconstructed clean image.
- **Advantages**: Autoencoders are effective in learning the underlying structure of the data, making them suitable for removing various types of noise. They are also simpler to train compared to GANs and can be used for unsupervised learning.

DnCNN and FFDNet

DnCNN (Denoising Convolutional Neural Network)

DnCNN is a specific type of CNN designed for image denoising. It uses residual learning and batch normalization to improve denoising performance.

- **Architecture**: DnCNN consists of multiple convolutional layers followed by ReLU activations and batch normalization. The network is trained to predict the noise component, which is then subtracted from the noisy image to obtain the denoised image.
- **Training**: The network is trained using pairs of noisy and clean images. The loss function is the MSE between the predicted noise and the actual noise.
- **Advantages**: DnCNN effectively removes Gaussian noise and generalizes well to different noise levels. Its use of residual learning helps in training deeper networks by mitigating the vanishing gradient problem.

FFDNet (Fast and Flexible Denoising Convolutional Neural Network)

FFDNet is a flexible CNN-based denoising method that can handle different noise levels and types, making it suitable for blind denoising.

- **Architecture**: FFDNet processes both the noisy image and a noise level map as inputs. It uses a series of convolutional layers to denoise the image, with the noise level map guiding the denoising process.
- **Training**: FFDNet is trained using a range of noise levels, enabling it to adapt to varying noise conditions. The loss function is the MSE between the denoised output and the ground truth clean image.
- **Advantages**: FFDNet is efficient and flexible, capable of real-time denoising. It can handle blind denoising scenarios where the noise level is unknown or varies across the image.

Differences between DnCNN and FFDNet

- **Input Handling**: DnCNN takes a single noisy image as input and predicts the noise to be subtracted. In contrast, FFDNet takes both the noisy image and a noise level map as inputs, allowing it to adapt to varying noise levels.

- **Flexibility**: FFDNet is more flexible than DnCNN, as it can handle different noise levels and types, making it suitable for blind denoising. DnCNN is primarily designed for Gaussian noise and requires retraining for different noise types.
- **Efficiency**: FFDNet is designed for efficiency and can perform real-time denoising. DnCNN, while effective, may not be as fast or flexible as FFDNet in handling varying noise conditions.

In conclusion, deep learning methods such as CNNs, GANs, and Autoencoders have significantly advanced the field of image denoising. Among specific methods, DnCNN and FFDNet offer powerful solutions with different strengths and application scenarios, highlighting the diversity and capability of deep learning in addressing image noise.

Coding Section

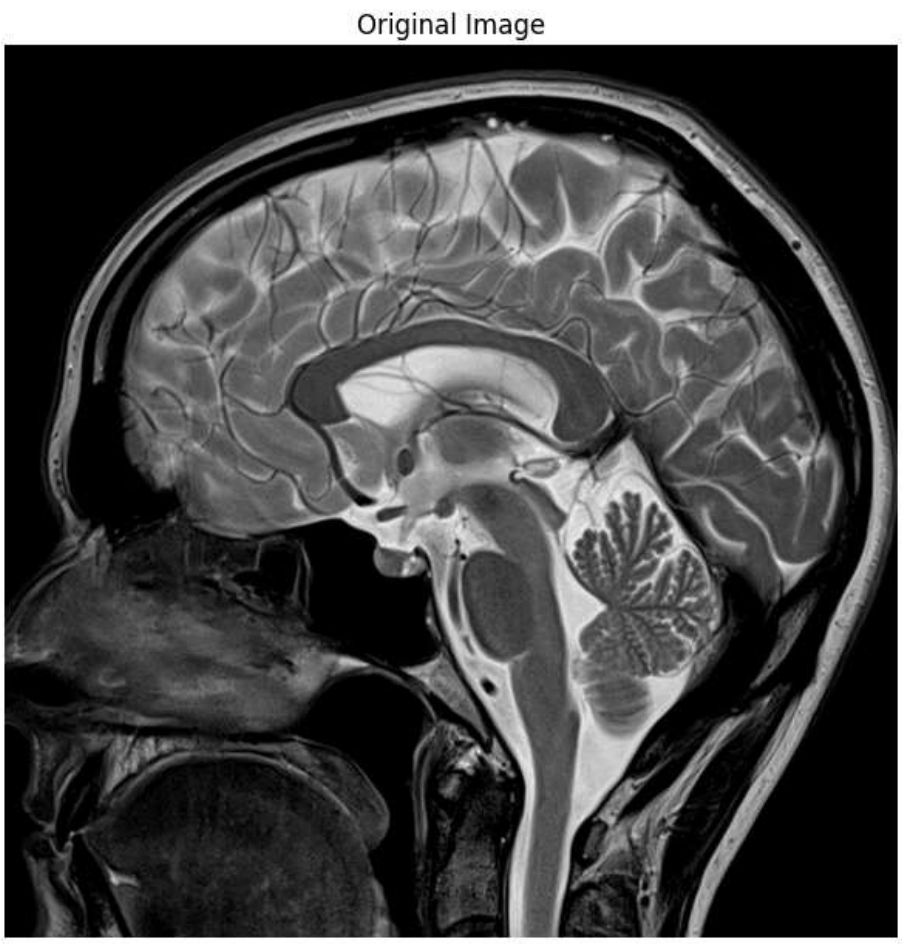
Question 1

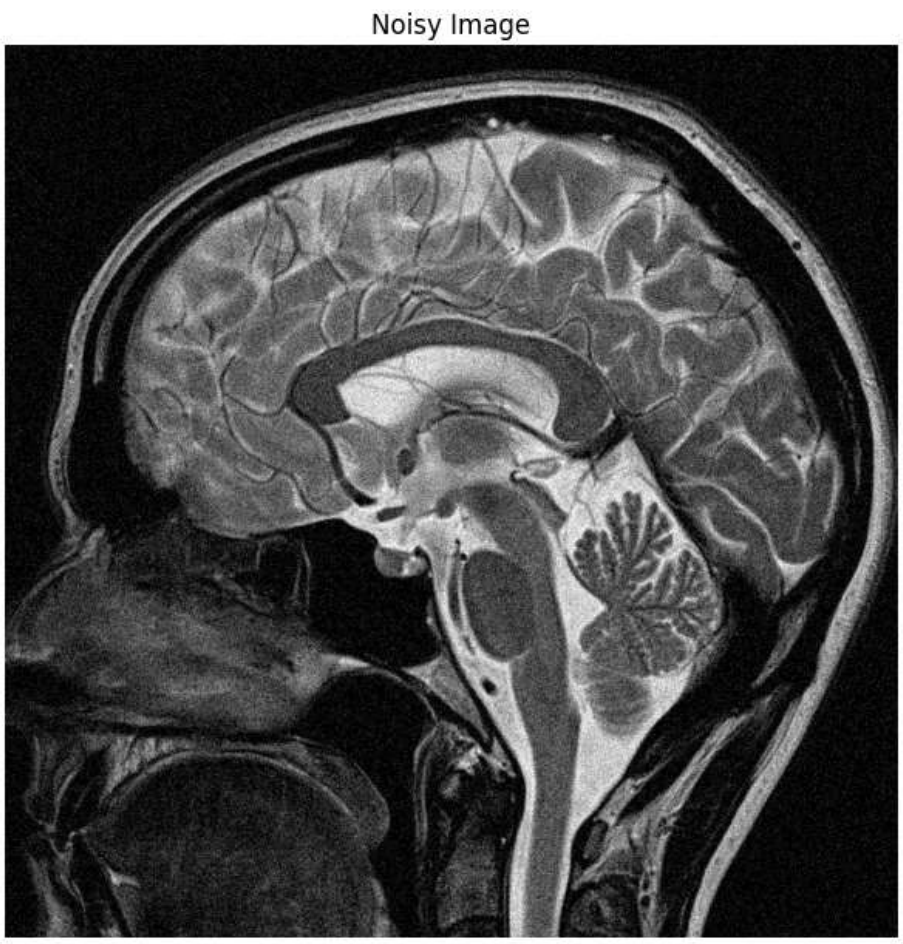
In [1]:
```python
# %% part 1: ADF
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.util import random_noise
```

In [2]:
```python
image = cv2.imread('C:/Users/aryak/OneDrive/Desktop/MAM/HW03/image_anisotropic.png', cv2.IMREAD_GRAYSCALE)
image = cv2.normalize(image, None, 0, 1, cv2.NORM_MINMAX, dtype=cv2.CV_32F)  # normalized image

noisy_image = random_noise(image, mode='gaussian', var=0.01)

cv2.imwrite('C:/Users/aryak/OneDrive/Desktop/MAM/HW03/noisyimage_anisotropic.png', noisy_image * 255)
```

Out[2]: True

In [7]:
```python
plt.figure(figsize=(12, 8))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray', vmin=0, vmax=1)
plt.title('Original Image')
plt.tight_layout()
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(noisy_image, cmap='gray', vmin=0, vmax=1)
plt.title('Noisy Image')
plt.axis("off")
plt.tight_layout()
plt.show()
```





In [8]:
```python
def calculate_derivatives(image):
    h, w = image.shape

    padded_image = np.pad(image, 1, mode='reflect')

    # Calculate the derivatives
    derivative_top = padded_image[:-2, 1:-1] - image
    derivative_bottom = padded_image[2:, 1:-1] - image
```

```
    derivative_left = padded_image[1:-1, :-2] - image
    derivative_right = padded_image[1:-1, 2:] - image

    return derivative_top, derivative_bottom, derivative_left, derivative_right
```

In [11]:
```
derivative_top, derivative_bottom, derivative_left, derivative_right = calculate_derivatives(noisy_image)


fig, axes = plt.subplots(2, 2, figsize=(12, 12))
axes[0, 0].imshow(derivative_top, cmap='gray', vmin=0, vmax=1)
axes[0, 0].set_title('Derivative Top')
axes[0, 0].axis("off")
axes[0, 1].imshow(derivative_bottom, cmap='gray', vmin=0, vmax=1)
axes[0, 1].set_title('Derivative Bottom')
axes[0, 1].axis("off")
axes[1, 0].imshow(derivative_left, cmap='gray', vmin=0, vmax=1)
axes[1, 0].set_title('Derivative Left')
axes[1, 0].axis("off")
axes[1, 1].imshow(derivative_right, cmap='gray', vmin=0, vmax=1)
axes[1, 1].set_title('Derivative Right')
axes[1, 1].axis("off")
plt.tight_layout()
plt.show()

fig, axes = plt.subplots(2, 2, figsize=(12, 12))
plt.suptitle("Lighter Version")
axes[0, 0].imshow(derivative_top, cmap='gray')
axes[0, 0].set_title('Derivative Top')
axes[0, 0].axis("off")
axes[0, 1].imshow(derivative_bottom, cmap='gray')
axes[0, 1].set_title('Derivative Bottom')
axes[0, 1].axis("off")
axes[1, 0].imshow(derivative_left, cmap='gray')
axes[1, 0].set_title('Derivative Left')
axes[1, 0].axis("off")
axes[1, 1].imshow(derivative_right, cmap='gray')
axes[1, 1].set_title('Derivative Right')
axes[1, 1].axis("off")
plt.tight_layout()
plt.show()
```

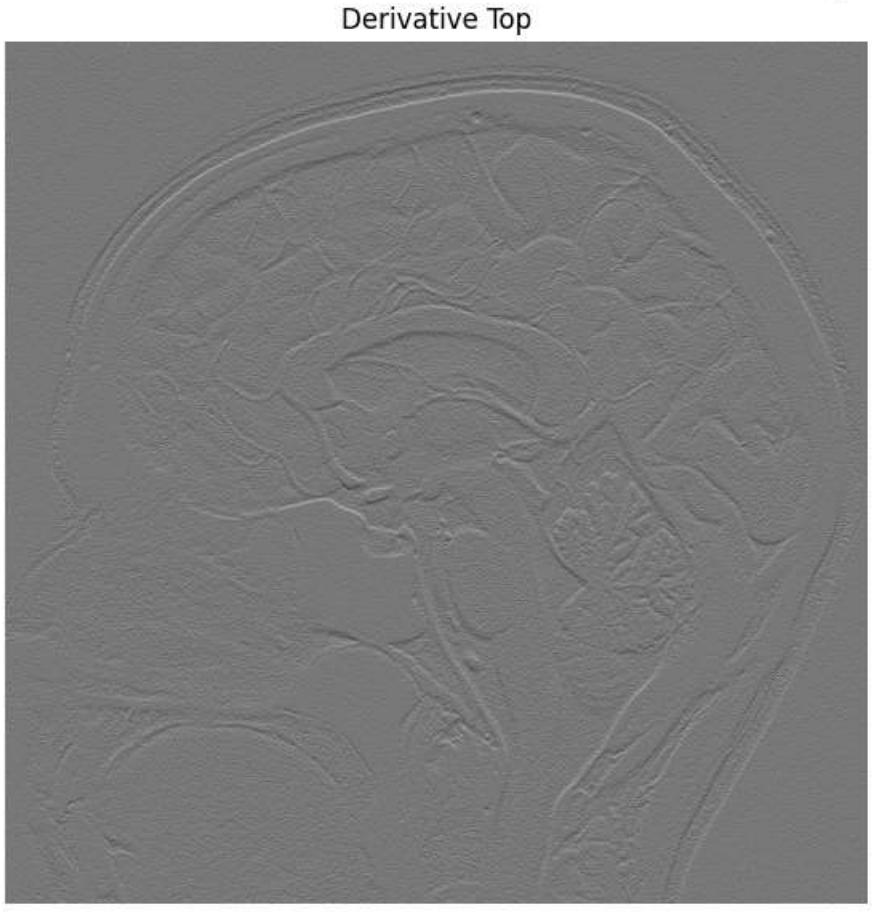
Derivative Top

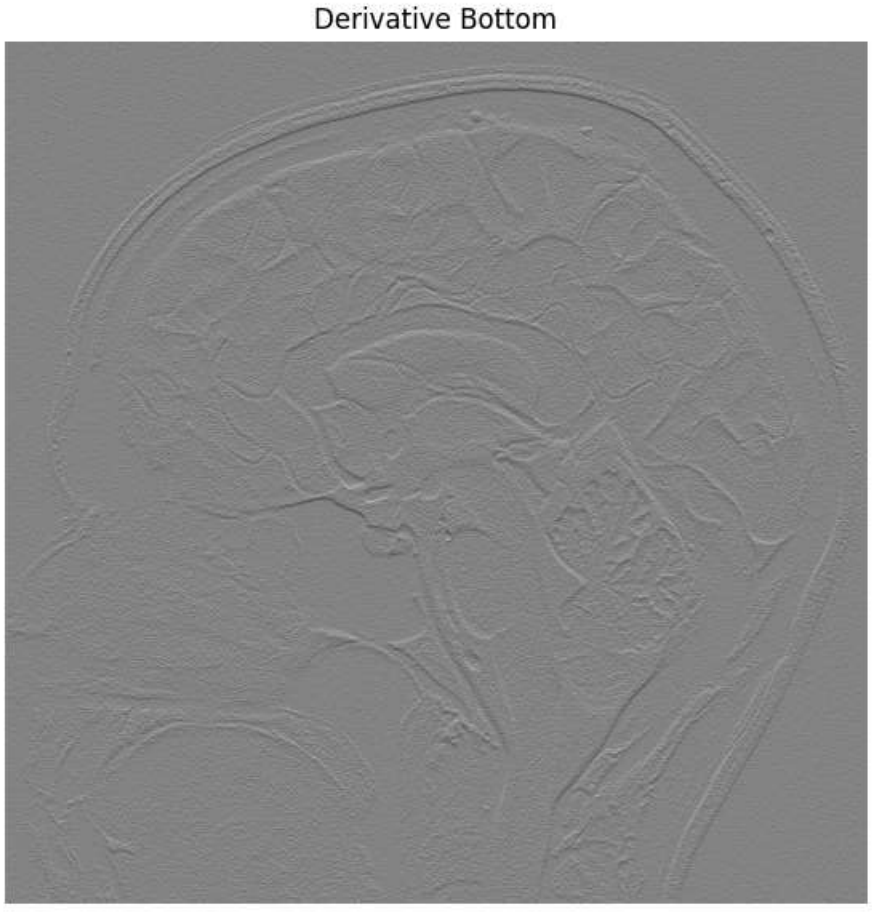
Derivative Bottom

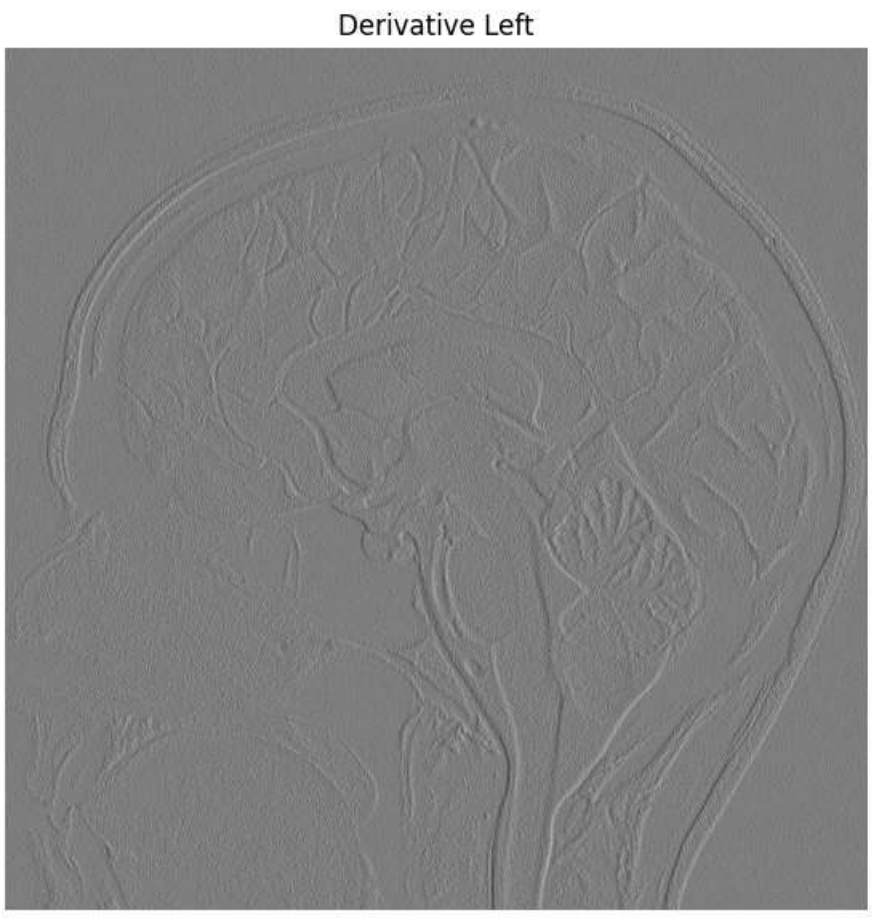
Derivative Left

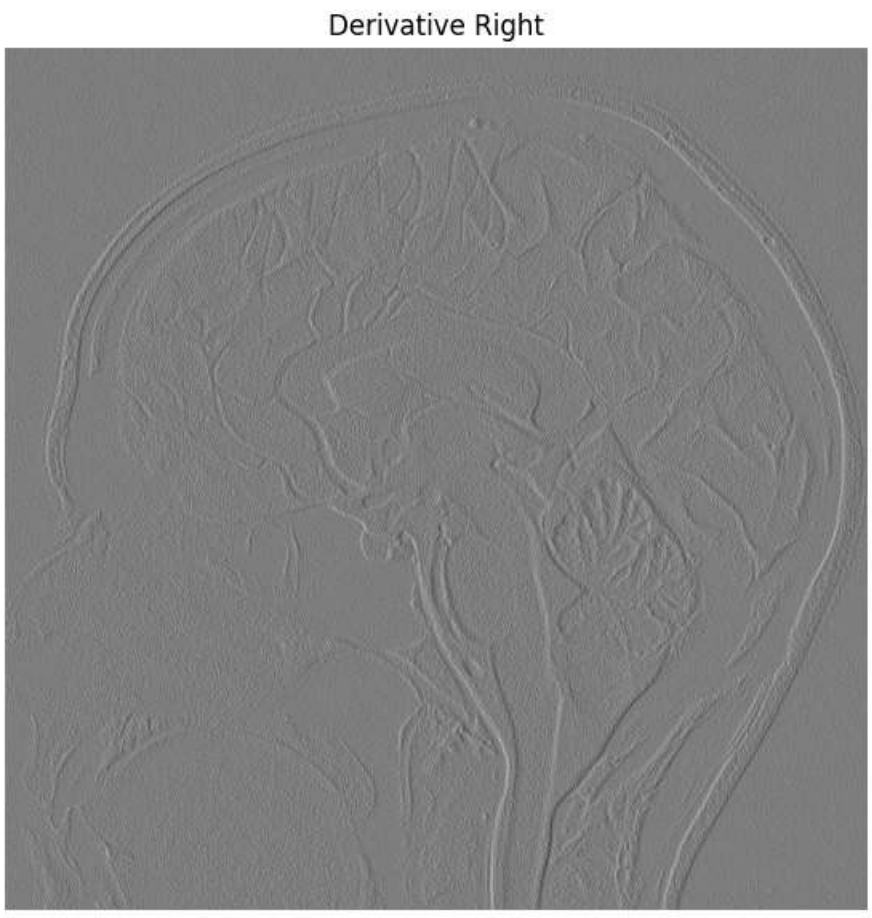
Derivative Right

Lighter Version









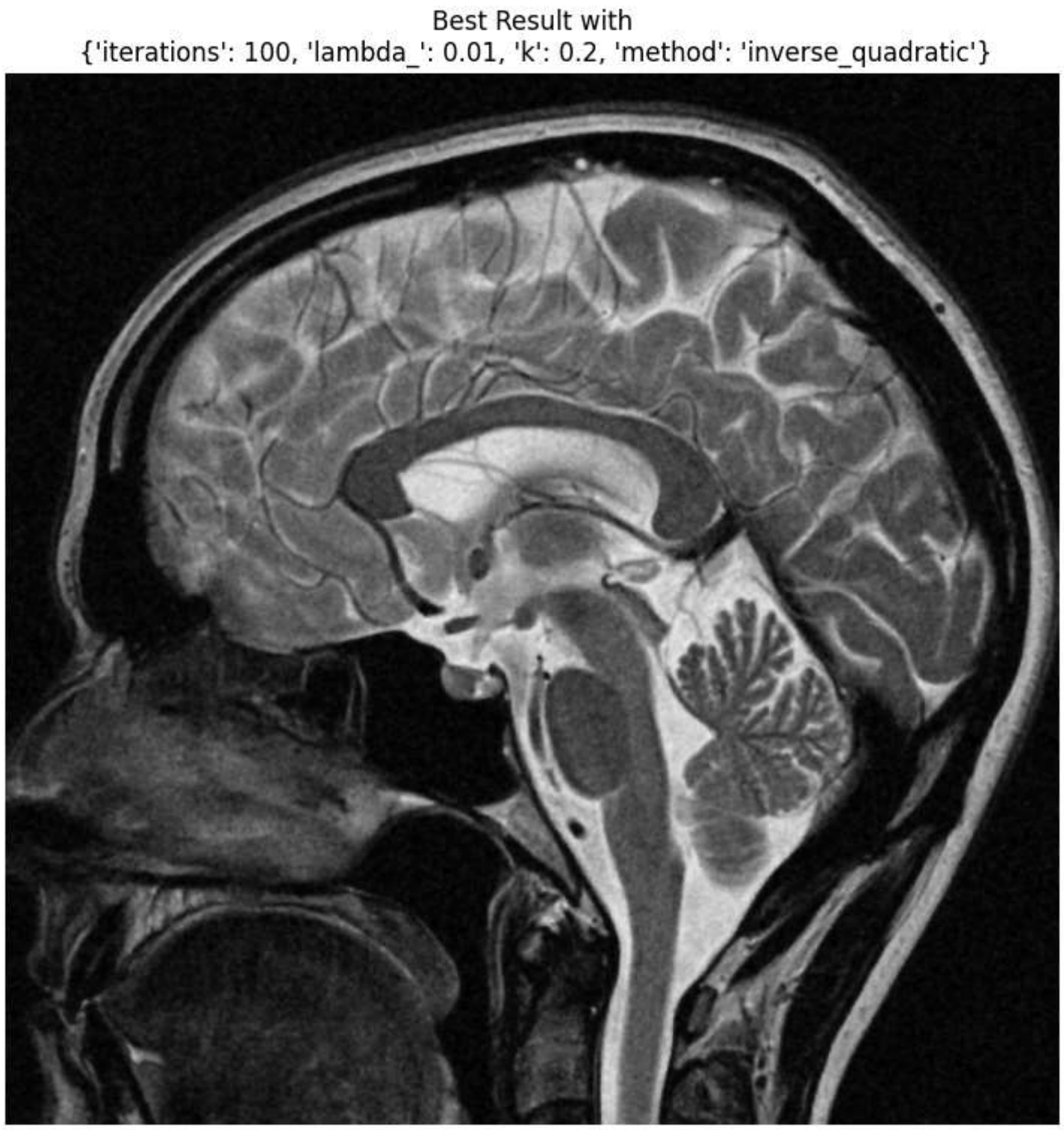
In [12]:
```python
def anisotropic_diffusion(noisy_image, iterations=10, lambda_=0.1, k=0.05, method='exponential'):
    image = noisy_image.copy()
    for _ in range(iterations):
        derivative_top, derivative_bottom, derivative_left, derivative_right = calculate_derivatives(image)

        if method == 'exponential':
            c_top = np.exp(-(derivative_top / k) ** 2)
            c_bottom = np.exp(-(derivative_bottom / k) ** 2)
            c_left = np.exp(-(derivative_left / k) ** 2)
            c_right = np.exp(-(derivative_right / k) ** 2)
        elif method == 'inverse_quadratic':
            c_top = 1 / (1 + (derivative_top / k) ** 2)
            c_bottom = 1 / (1 + (derivative_bottom / k) ** 2)
            c_left = 1 / (1 + (derivative_left / k) ** 2)
            c_right = 1 / (1 + (derivative_right / k) ** 2)
        else:
            raise ValueError("Method must be 'exponential' or 'inverse_quadratic'")

        image += lambda_ * (
            c_top * derivative_top +
            c_bottom * derivative_bottom +
            c_left * derivative_left +
            c_right * derivative_right
        )

    return image
```

In [17]:
```python
filtered_images = {}
MSEs = {}
best_result = None
```

```
best_params = None
best_image = None

itr = 100
# Test the ADF with different methods and parameters
for mth in ['exponential', 'inverse_quadratic']:
    for lam in [0.01, 0.05, 0.1, 0.2, 0.4, 0.8, 1]:
        for k in [0.01, 0.05, 0.1, 0.2, 0.4, 0.8, 1]:
            filtered_image = anisotropic_diffusion(noisy_image.copy(), iterations=itr, lambda_=lam, k=k, method=mth)
            filtered_images[f'lambda={lam}, k={k}, method={mth}'] = filtered_image

            mse = np.mean((filtered_image - image) ** 2)  # Using MSE as a simple performance metric
            MSEs[f'lambda={lam}, k={k}, method={mth}'] = mse

            if best_result is None or mse < best_result:
                best_result = mse
                best_params = {'iterations': itr, 'lambda_': lam, 'k': k, 'method': mth}
                best_image = filtered_image
        print(f"{lam*100}%", end='\r')
```

```
100%%
```

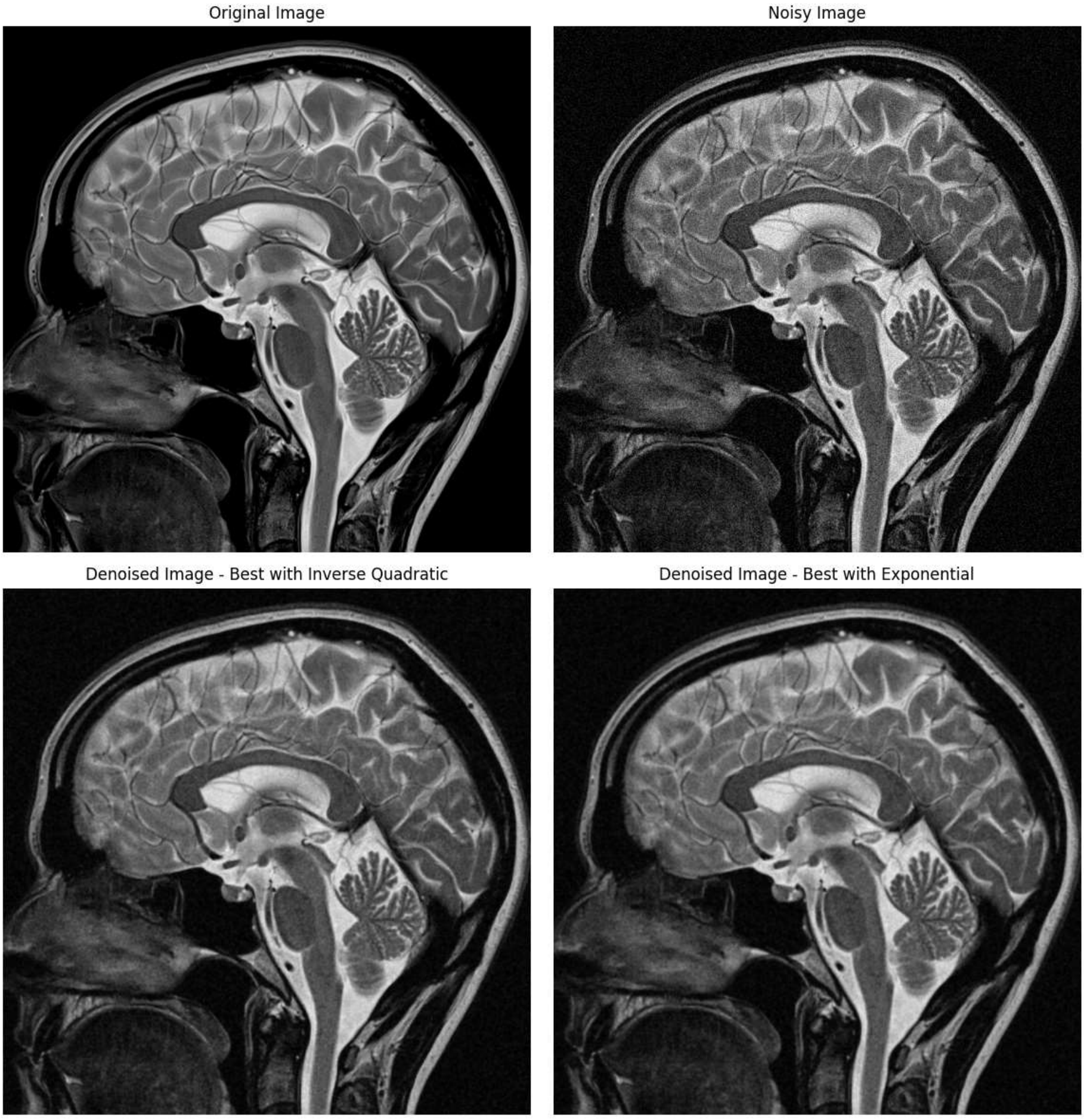
In [19]:
```
plt.figure(figsize=(12, 8))
anisotropic_best = best_image.copy()
plt.imshow(anisotropic_best, cmap='gray', vmin=0, vmax=1)
plt.title(f'Best Result with\n {best_params}')
plt.tight_layout()
plt.axis("off")
plt.show()

cv2.imwrite('C:/Users/aryak/OneDrive/Desktop/MAM/HW03/anisotropic_best.png', best_image * 255)

print(f'Best parameters: {best_params}')
```




```
Best parameters: {'iterations': 100, 'lambda_': 0.01, 'k': 0.2, 'method': 'inverse_quadratic'}
```

In [20]:
```
# %% plot best denoised for each method
plt.figure(figsize=(12, 12))
plt.subplot(2, 2, 1)
plt.imshow(image, cmap='gray', vmin=0, vmax=1)
plt.title('Original Image')
plt.tight_layout()
plt.axis("off")
```

```
plt.subplot(2, 2, 2)
plt.imshow(noisy_image, cmap='gray', vmin=0, vmax=1)
plt.title('Noisy Image')
plt.axis("off")
plt.tight_layout()

plt.subplot(2, 2, 3)
plt.imshow(filtered_images['lambda=0.01, k=0.2, method=inverse_quadratic'], cmap='gray', vmin=0, vmax=1)
plt.title('Denoised Image - Best with Inverse Quadratic')
plt.axis("off")
plt.tight_layout()

plt.subplot(2, 2, 4)
plt.imshow(filtered_images['lambda=0.01, k=0.4, method=exponential'], cmap='gray', vmin=0, vmax=1)
plt.title('Denoised Image - Best with Exponential')
plt.axis("off")
plt.tight_layout()
plt.show()
```



Explanation

1. **Adding Noise**: I add Gaussian noise to the original image to simulate a noisy environment.
2. **Calculating Derivatives**: I calculate the derivatives in the top, bottom, left, and right directions using image padding to handle borders.
3. **ADF Implementation**: I iteratively update the image using the calculated derivatives and conduction coefficients, which control the diffusion rate based on the gradient magnitude.

This approach helps to preserve edges while smoothing homogeneous regions, achieving effective noise reduction.

Report

Objective

The aim of this study was to implement and evaluate the performance of Anisotropic Diffusion Filtering (ADF) on a noisy, normalized image. The primary focus was on finding the best set of parameters for optimal noise reduction while preserving image details.

Methods

Anisotropic Diffusion Filtering is a technique used to reduce noise in images while preserving edges. This process involves iteratively updating the image using calculated derivatives and conduction coefficients. Two methods for calculating conduction coefficients were explored:

1. **Exponential Conduction Coefficient**: $c = \exp\left(-\left(\frac{\nabla I}{k}\right)^2\right)$
2. **Inverse Quadratic Conduction Coefficient**: $c = \frac{1}{1+\left(\frac{\nabla I}{k}\right)^2}$

The performance of ADF was evaluated by testing different parameter combinations for `iterations`, `lambda_`, and `k`. The Mean Squared Error (MSE) between the filtered image and the original image was used as the metric for performance evaluation.

Experimental Setup

1. **Image Preprocessing**: The input image was normalized to have pixel values between 0 and 1. Gaussian noise was added to simulate a noisy environment.
2. **Parameter Tuning**: Various combinations of parameters were tested to find the optimal settings for noise removal:
   - Iterations: 100
   - Lambda (`lambda_`): [0.01, 0.05, 0.1, 0.2, 0.4, 0.8, 1]
   - K (`k`): [0.01, 0.05, 0.1, 0.2, 0.4, 0.8, 1]
   - Method: 'exponential', 'inverse_quadratic'

Results

The best result was achieved with the following parameters:

- **Iterations**: 100
- **Lambda (`lambda_`)**: 0.01
- **K (`k`)**: 0.2
- **Method**: Inverse Quadratic

The filtered image with these parameters showed significant noise reduction while preserving important image features. The Mean Squared Error (MSE) was the lowest among the tested configurations, indicating the best performance.

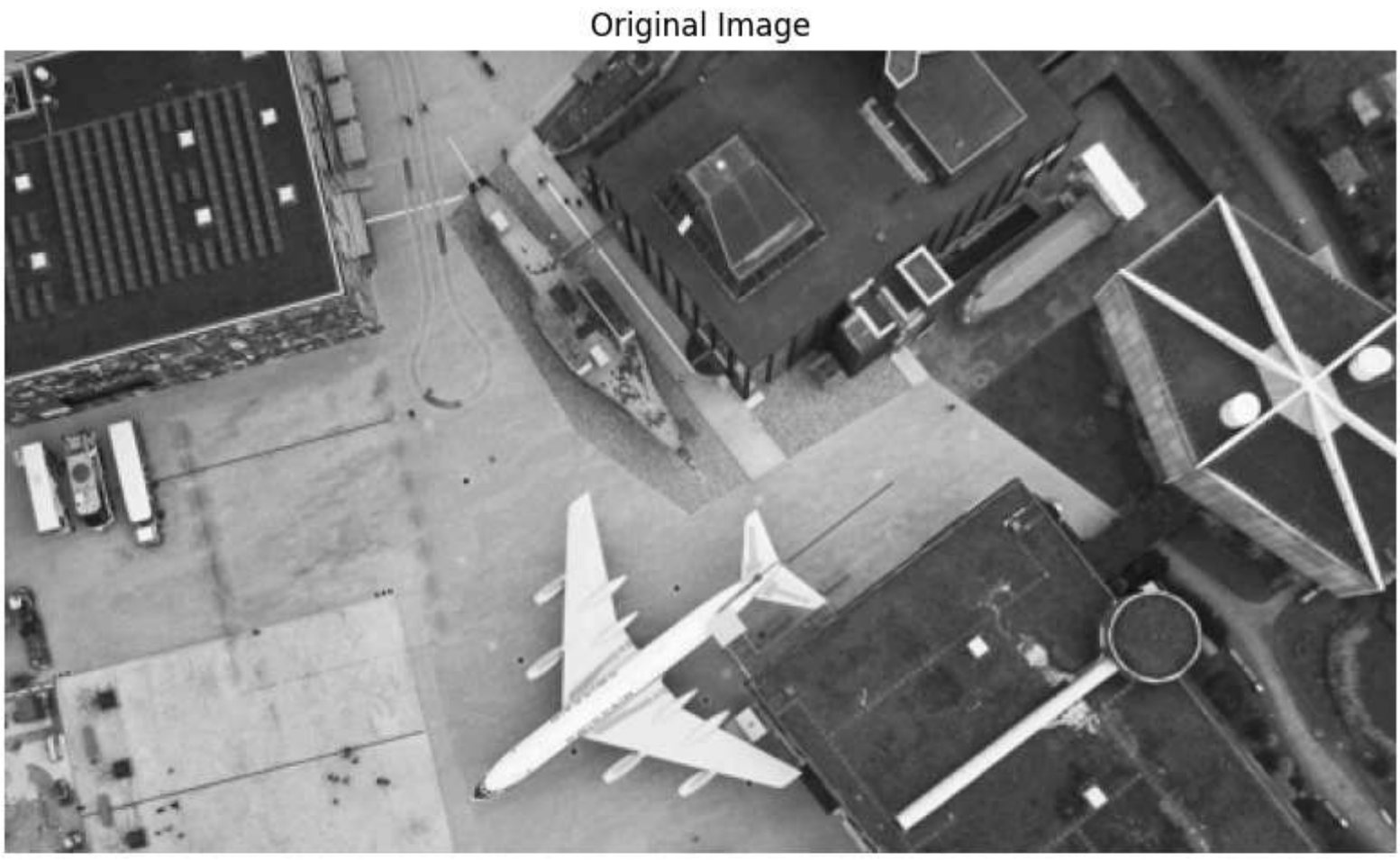
Conclusion

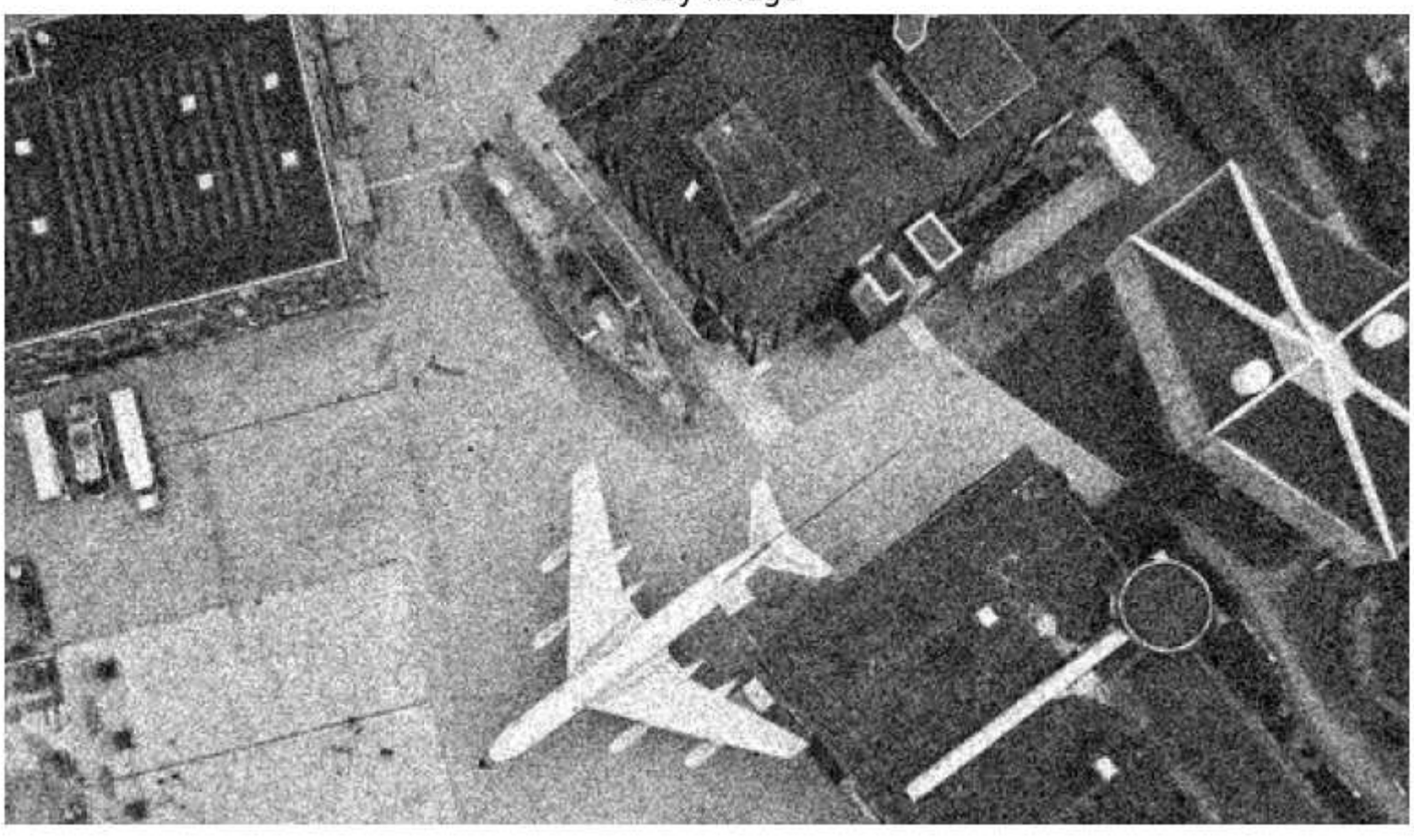
The study demonstrated that the Anisotropic Diffusion Filtering method, with appropriately tuned parameters, is effective in denoising normalized images. The best performance was achieved with a high number of iterations, a small lambda value, and a moderate k value using the inverse quadratic method for calculating conduction coefficients. This combination provided a balance between noise reduction and edge preservation.

Question 2

In [23]:
```python
# %% part2: IDF, ADF
# %% Read the Image and Add Gaussian Noise
import torch
import pyiqa
from skimage.metrics import structural_similarity as ssim
from skimage.util import random_noise
from skimage import img_as_float, img_as_ubyte
import os
import numpy as np
import cv2
import matplotlib.pyplot as plt


def to_rgb(gray_image):
    return np.stack((gray_image,) * 3, axis=-1)


image = cv2.imread('C:/Users/aryak/OneDrive/Desktop/MAM/HW03/image2.png', cv2.IMREAD_GRAYSCALE)
image = img_as_float(image)  # Normalized image

noisy_image = random_noise(image, mode='gaussian', var=0.01)

cv2.imwrite('C:/Users/aryak/OneDrive/Desktop/MAM/HW03/image2_noisy.png', noisy_image * 255)
```

```
plt.figure(figsize=(8, 12))
plt.subplot(2, 1, 1)
plt.imshow(image, cmap='gray', vmin=0, vmax=1)
plt.title('Original Image')
plt.tight_layout()
plt.axis("off")

plt.subplot(2, 1, 2)
plt.imshow(noisy_image, cmap='gray', vmin=0, vmax=1)
plt.title('Noisy Image')
plt.axis("off")
plt.tight_layout()
plt.show()
```





part a

In [25]:
```
# %% Implement Isotropic and Anisotropic Diffusion Filters

# % Isotropic Diffusion
def isodiff(image, lambda_param, constant, niter):
    im = image.copy()
    im = np.double(im)
    rows, cols = im.shape
    diff = im
```

```
    for _ in range(niter):
        diffl = np.zeros((rows + 2, cols + 2))
        diffl[1:rows+1, 1:cols+1] = diff

        deltaN = diffl[0:rows, 1:cols+1] - diff
        deltaS = diffl[2:rows+2, 1:cols+1] - diff
        deltaE = diffl[1:rows+1, 2:cols+2] - diff
        deltaW = diffl[1:rows+1, 0:cols] - diff

        cN = cS = cE = cW = constant

        diff = diff + lambda_param * (cN * deltaN + cS * deltaS + cE * deltaE + cW * deltaW)

    return diff


lambda_param = 0.25
constant = 0.04
niter = 100

isotropic_denoised = isodiff(noisy_image, lambda_param, constant, niter)

plt.figure(figsize=(8, 12))
plt.imshow(isotropic_denoised, cmap='gray', vmin=0, vmax=1)
plt.title('Isotropic Denoised Image')
plt.tight_layout()
plt.axis("off")
plt.show()
```

```
# % Anisotropic Diffusion
def anisodiff(image, niter, kappa, lambda_param, option):
    im = image.copy()
    im = np.double(im)
    rows, cols = im.shape
    diff = im

    for _ in range(niter):
        diffl = np.zeros((rows + 2, cols + 2))
        diffl[1:rows+1, 1:cols+1] = diff

        deltaN = diffl[0:rows, 1:cols+1] - diff
        deltaS = diffl[2:rows+2, 1:cols+1] - diff
        deltaE = diffl[1:rows+1, 2:cols+2] - diff
        deltaW = diffl[1:rows+1, 0:cols] - diff

        if option == 1:
            cN = np.exp(-(deltaN / kappa)**2)
            cS = np.exp(-(deltaS / kappa)**2)
            cE = np.exp(-(deltaE / kappa)**2)
            cW = np.exp(-(deltaW / kappa)**2)
        elif option == 2:
            cN = 1 / (1 + (deltaN / kappa)**2)
            cS = 1 / (1 + (deltaS / kappa)**2)
            cE = 1 / (1 + (deltaE / kappa)**2)
            cW = 1 / (1 + (deltaW / kappa)**2)

        diff = diff + lambda_param * (cN * deltaN + cS * deltaS + cE * deltaE + cW * deltaW)

    return diff


niter = 100
kappa = 0.15
lambda_param = 0.008

anisotropic_denoised_option1 = anisodiff(noisy_image, niter, kappa, lambda_param, option=1)
anisotropic_denoised_option2 = anisodiff(noisy_image, niter, kappa, lambda_param, option=2)

plt.figure(figsize=(8, 12))
plt.subplot(2, 1, 1)
plt.imshow(anisotropic_denoised_option1, cmap='gray', vmin=0, vmax=1)
plt.title('Anisotropic Denoised Image - option 1')
plt.tight_layout()
plt.axis("off")

plt.subplot(2, 1, 2)
plt.imshow(anisotropic_denoised_option2, cmap='gray', vmin=0, vmax=1)
plt.title('Anisotropic Denoised Image - option 2')
plt.tight_layout()
plt.axis("off")
plt.show()
```

Isotropic Denoised Image

Anisotropic Denoised Image - option 1

Anisotropic Denoised Image - option 2

part b

In [27]:
```
# %% Evaluate the Results
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
torch.cuda.is_available = lambda: False

device = torch.device('cpu')

ssim_isotropic = ssim(image, isotropic_denoised, data_range=isotropic_denoised.max() - isotropic_denoised.min())
ssim_anisotropic_option1 = ssim(image, anisotropic_denoised_option1, data_range=anisotropic_denoised_option1.max() - anisotrop
ssim_anisotropic_option2 = ssim(image, anisotropic_denoised_option2, data_range=anisotropic_denoised_option2.max() - anisotrop


isotropic_denoised_rgb = to_rgb(img_as_ubyte(isotropic_denoised))
anisotropic_denoised_option1_rgb = to_rgb(img_as_ubyte(anisotropic_denoised_option1))
anisotropic_denoised_option2_rgb = to_rgb(img_as_ubyte(anisotropic_denoised_option2))

isotropic_denoised_tensor = torch.from_numpy(isotropic_denoised_rgb).permute(2, 0, 1).unsqueeze(0).float() / 255.0
anisotropic_denoised_option1_tensor = torch.from_numpy(anisotropic_denoised_option1_rgb).permute(2, 0, 1).unsqueeze(0).float()
anisotropic_denoised_option2_tensor = torch.from_numpy(anisotropic_denoised_option2_rgb).permute(2, 0, 1).unsqueeze(0).float()

isotropic_denoised_tensor = isotropic_denoised_tensor.to(device)
anisotropic_denoised_option1_tensor = anisotropic_denoised_option1_tensor.to(device)
anisotropic_denoised_option2_tensor = anisotropic_denoised_option2_tensor.to(device)

niqe_model = pyiqa.create_metric('niqe').to(device)
niqe_model.eval()  # Set the model to evaluation mode

with torch.no_grad():
    niqe_isotropic = niqe_model(isotropic_denoised_tensor).item()
    niqe_anisotropic_option1 = niqe_model(anisotropic_denoised_option1_tensor).item()
    niqe_anisotropic_option2 = niqe_model(anisotropic_denoised_option2_tensor).item()

print(f"SSIM Isotropic: {ssim_isotropic}")
print(f"SSIM Anisotropic - option1: {ssim_anisotropic_option1}")
print(f"SSIM Anisotropic - option2: {ssim_anisotropic_option2}\n")
print(f"NIQE Isotropic: {niqe_isotropic}")
print(f"NIQE Anisotropic - option1: {niqe_anisotropic_option1}")
print(f"NIQE Anisotropic - option2: {niqe_anisotropic_option2}")


plt.figure(figsize=(10, 14))
plt.subplot(3, 2, 1)
plt.imshow(image, cmap='gray', vmin=0, vmax=1)
plt.title('Original Image')
plt.tight_layout()
plt.axis("off")

plt.subplot(3, 2, 3)
plt.imshow(noisy_image, cmap='gray', vmin=0, vmax=1)
plt.title('Noisy Image')
plt.axis("off")
plt.tight_layout()

plt.subplot(3, 2, 2)
plt.imshow(isotropic_denoised, cmap='gray', vmin=0, vmax=1)
plt.title('Isotropic Denoised Image')
plt.tight_layout()
plt.axis("off")

plt.subplot(3, 2, 4)
plt.imshow(anisotropic_denoised_option1, cmap='gray', vmin=0, vmax=1)
plt.title('Anisotropic Denoised Image - option 1')
plt.tight_layout()
plt.axis("off")

plt.subplot(3, 2, 6)
plt.imshow(anisotropic_denoised_option2, cmap='gray', vmin=0, vmax=1)
plt.title('Anisotropic Denoised Image - option 2')
plt.tight_layout()
plt.axis("off")
plt.show()
```

```
SSIM Isotropic: 0.6983385776466976
SSIM Anisotropic - option1: 0.6389505410583802
SSIM Anisotropic - option2: 0.7196470533284212

NIQE Isotropic: 6.018502747274328
NIQE Anisotropic - option1: 10.96008761304512
NIQE Anisotropic - option2: 3.651271424521491
```

Original Image

Isotropic Denoised Image

Noisy Image

Anisotropic Denoised Image - option 1

Anisotropic Denoised Image - option 2

Structural Similarity Index (SSIM)

The Structural Similarity Index (SSIM) is a metric used to measure the similarity between two images. It is designed to assess the perceived quality of digital images and is based on the human visual system. SSIM considers changes in structural information, luminance, and contrast, providing a comprehensive evaluation of image quality.

Key Components of SSIM:

1. **Luminance Comparison**: Measures the difference in the average brightness of the two images.
2. **Contrast Comparison**: Evaluates the difference in contrast between the images.
3. **Structure Comparison**: Assesses how well the structures in the two images correspond to each other.

The SSIM index ranges from -1 to 1, where:

- 1 indicates perfect structural similarity.
- 0 indicates no similarity.
- Negative values indicate significant structural differences.

Naturalness Image Quality Evaluator (NIQE)

The Naturalness Image Quality Evaluator (NIQE) is a no-reference image quality assessment metric, meaning it does not require a reference image for comparison. NIQE is based on the statistical properties of natural images and evaluates the quality based on deviations from these properties.

Key Aspects of NIQE:

1. **Natural Scene Statistics (NSS)**: NIQE relies on models of NSS to assess the quality. It compares the statistical features of the test image to those typically found in high-quality natural images.
2. **Quality Score**: The NIQE score represents the image quality, where lower scores indicate better perceived quality.

Analyzing the Results

SSIM Results:

- **SSIM Isotropic: 0.6983**: Indicates moderate structural similarity to the reference image. Isotropic diffusion provides a balance between noise reduction and structure preservation.
- **SSIM Anisotropic - option1: 0.6390**: Shows slightly lower structural similarity compared to isotropic diffusion. Anisotropic diffusion with option 1 might be preserving edges at the expense of some structural integrity in low-gradient regions.
- **SSIM Anisotropic - option2: 0.7196**: Indicates the highest structural similarity. Anisotropic diffusion with option 2 preserves more structural information, likely due to a more effective handling of edge-preserving diffusion.

NIQE Results:

- **NIQE Isotropic: 6.0185**: Indicates a moderate deviation from natural image quality. Isotropic diffusion smooths the image, potentially losing some naturalness.
- **NIQE Anisotropic - option1: 10.9601**: Shows a significant deviation from natural image quality. Anisotropic diffusion with option 1 might overemphasize edge preservation, leading to unnatural artifacts.
- **NIQE Anisotropic - option2: 3.6513**: Indicates the closest approximation to natural image quality. Anisotropic diffusion with option 2 balances edge preservation and noise reduction effectively, maintaining natural image statistics.

Summary:

- **Isotropic Diffusion**: Provides a good balance between noise reduction and structural preservation. It has a moderate SSIM and NIQE score, indicating decent quality but not the best edge preservation or naturalness.
- **Anisotropic Diffusion - Option 1**: Has lower SSIM and higher NIQE scores, suggesting that while it preserves edges, it might introduce artifacts that reduce naturalness and overall structural similarity.
- **Anisotropic Diffusion - Option 2**: Achieves the highest SSIM and the lowest NIQE scores, indicating the best performance in terms of both structural similarity and natural image quality. It effectively reduces noise while preserving edges and natural image properties.

Conclusion:

Anisotropic diffusion with option 2 (reciprocal function) appears to be the best method among the tested ones for enhancing image quality. It provides the highest structural similarity and the most natural appearance, balancing noise reduction and edge preservation effectively.

part c

Comparison with Other Filters

Gaussian filters and other denoising techniques like Total Variation and Non-Local Means can be compared based on their denoising performance and preservation of image details.

- **Gaussian Filter**: Applies a Gaussian blur to the image, which reduces noise but also blurs edges.
- **Total Variation (TV)**: Minimizes the total variation of the image, preserving edges better than Gaussian.
- **Non-Local Means (NLM)**: Averages pixels with similar patches, providing good denoising with edge preservation.

Here is a brief comparison:

- **Isotropic Diffusion**: Uniform diffusion, does not consider image features.
- **Anisotropic Diffusion**: Directional diffusion based on image gradients, better at preserving edges.
- **Gaussian Filter**: Simple and fast, but tends to blur edges.
- **Total Variation**: Good edge preservation, suitable for piecewise-smooth images.
- **NLM**: Effective for detailed denoising, can be computationally intensive.

Conclusion

By implementing and evaluating isotropic and anisotropic diffusion filters, Ican observe their effectiveness in denoising while preserving image details. Using SSIM and NIQE metrics, Ican quantify the quality of the denoised images and compare them with other denoising methods.

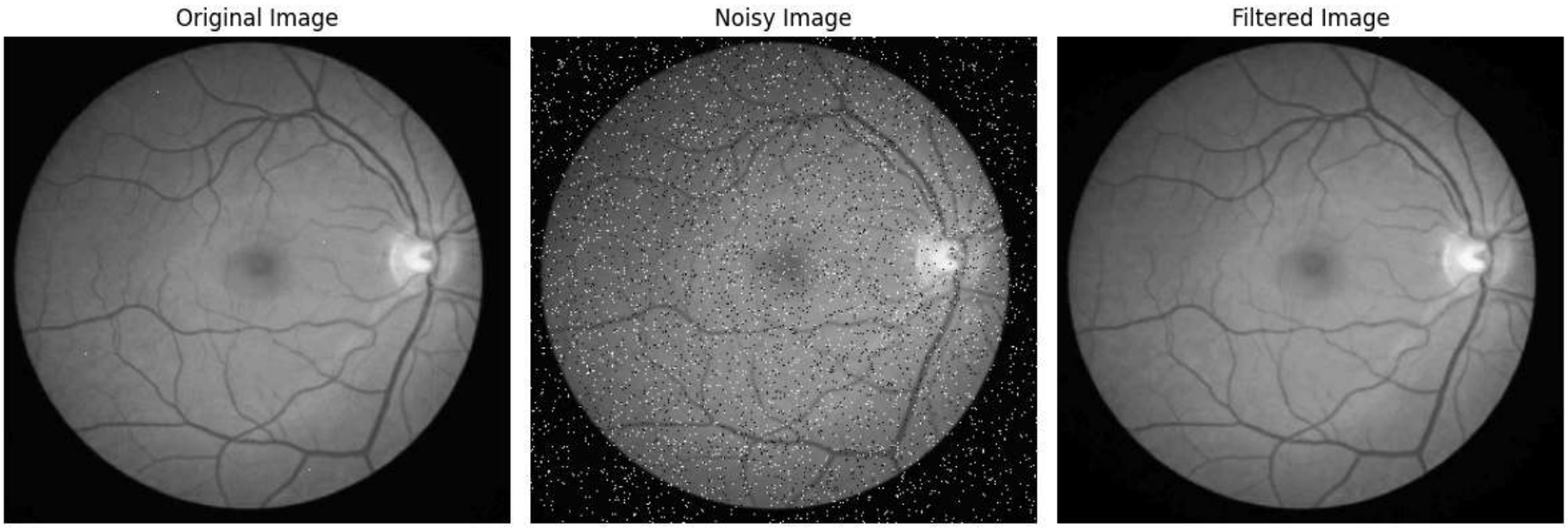
Question 3

part a

In [28]:
```
# %% part a
from skimage import io, util, color, img_as_float
import numpy as np
import cv2
from skimage import io, util, color
import matplotlib.pyplot as plt
from skimage import img_as_float
```

In [29]:
```
def adaptive_median_filter(image_noisy, Smax=39):
    image = image_noisy.copy()
    m, n = image.shape
    output_image = np.zeros((m, n))

    def get_window_stats(img, x, y, w):
        window = img[max(0, x-w):min(m, x+w+1), max(0, y-w):min(n, y+w+1)]
        return np.median(window), np.min(window), np.max(window)

    for i in range(m):
        for j in range(n):
            w = 1
            while True:
                median, min_val, max_val = get_window_stats(image, i, j, w)

                if min_val < median < max_val:
                    if min_val < image[i, j] < max_val:
                        output_image[i, j] = image[i, j]
                    else:
                        output_image[i, j] = median
                    break
                else:
                    w += 1
                    if w > Smax:
                        output_image[i, j] = median
                        break

    return output_image


def add_salt_and_pepper_noise(image, amount=0.05):
    noisy_image = util.random_noise(image, mode='s&p', amount=amount)
    return noisy_image
```

In [30]:
```
image = cv2.imread('C:/Users/aryak/OneDrive/Desktop/MAM/HW03/retina.png', cv2.IMREAD_GRAYSCALE)
image = img_as_float(image)  # Normalized image

noisy_image = add_salt_and_pepper_noise(image)

filtered_image = adaptive_median_filter(noisy_image)

plt.figure(figsize=(12, 8))
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Noisy Image')
plt.imshow(noisy_image, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Filtered Image')
plt.imshow(filtered_image, cmap='gray')
plt.axis('off')
plt.tight_layout()
plt.show()
```



Explanation

1. **Loading and Noising the Image:**
   - `add_salt_and_pepper_noise(image)`: Adds salt and pepper noise to the image.
2. **Adaptive Median Filter Implementation:**
   - The `adaptive_median_filter` function iteratively increases the window size until the conditions are met or the maximum window size is reached.
   - It pads the image to handle borders.
   - It checks conditions as per the algorithm and applies the filter.
3. **Displaying Results**

This implementation provides a visual comparison of the original, noisy, and filtered images, illustrating the effectiveness of the adaptive median filter in removing salt and pepper noise.

part b

Analysis of the Adaptive Median Filter Algorithm on the Provided Window

Given the following 7x7 window centered around the pixel with value 66, I need to determine what value appears in the output when the algorithm is applied.

0	68	255	0	0	70	255
0	255	255	255	255	255	0
0	255	68	67	67	255	0
255	0	255	**66**	78	255	70
255	0	255	255	255	255	255
0	255	0	255	0	0	0
0	78	0	0	255	255	255

Steps of the Algorithm

1. **Initial Setup:**
   - Window size $w = 1$ (3x3 window centered around the pixel 66).
   - The central pixel $y_{i,j} = 66$.
2. **Compute Window Statistics for $w = 1$:**
   - Window $S_{i,j}(w)$:

68	67	67
255	**66**	78
255	255	255

   - Median $S_{i,j}^{\text{med}}(w) = 78$
   - Minimum $S_{i,j}^{\text{min}}(w) = 66$
   - Maximum $S_{i,j}^{\text{max}}(w) = 255$
3. **Check Condition for the Current Window:**
   - The condition to check is $S_{i,j}^{\text{min}}(w) < S_{i,j}^{\text{med}}(w) < S_{i,j}^{\text{max}}(w)$.
   - Substituting the values: $66 < 78 < 255$.
   - This condition is satisfied, so I proceed to step 5.
4. **Check Pixel Value Against Window Extremes:**
   - The next condition to check is $S_{i,j}^{\text{min}}(w) < y_{i,j} < S_{i,j}^{\text{max}}(w)$.
   - Substituting the values: $66 < 66 < 255$.

- This condition is **not** satisfied because 66 is not strictly greater than 66.

5. **Output the Median Value:**
   - Since the condition is not satisfied, I set the output pixel value $z_{i,j} = S_{i,j}^{\text{med}}(w) = 78$.

Conclusion for the Central Pixel

- For the given 7x7 window, when applying the adaptive median filter with the algorithm, the central pixel (originally 66) will be replaced with the median value 78 for the 3x3 window centered on it.

Analysis of Potential Problems with the Algorithm

The adaptive median filter algorithm described can encounter several problems under certain conditions:

1. **Edge Cases with Minimum or Maximum Values:**
   - If the central pixel value is exactly equal to the minimum or maximum value of the current window, the algorithm fails the condition $S_{i,j}^{\min}(w) < y_{i,j} < S_{i,j}^{\max}(w)$ even if the central pixel is not a noise pixel. This results in unnecessary replacement with the median value.
2. **Increasing Window Size Unnecessarily:**
   - If the noise density is high, the algorithm may increase the window size multiple times unnecessarily, which can blur the image by replacing non-noise pixels with the median value of a larger window.
3. **Performance Overhead:**
   - For larger images or high levels of noise, the adaptive increase in window size up to a maximum can be computationally expensive, leading to slow performance.
4. **Insensitivity to Small Noise:**
   - The algorithm might be ineffective for small noise levels where noise pixels have values close to their neighbors, as the median of a small window might not differ significantly from the noise value, leading to incorrect filtering.
5. **Failure in Highly Structured Regions:**
   - In areas with high detail or texture, increasing the window size can remove important details, resulting in loss of image quality and sharpness.

Summary

The adaptive median filter effectively removes salt and pepper noise by adjusting the window size based on local statistics. However, it has potential issues with strict inequality conditions, performance overhead, and sensitivity to noise levels and image structure, which should be considered when applying this filter in practice.

part c

Explanation of the Adaptive Weighted Mean Filter Algorithm

The Adaptive Weighted Mean Filter (AWMF) algorithm aims to address the issues identified with the Adaptive Median Filter (AMF) by incorporating a weighted mean approach. Let's break down the key components and steps of the AWMF algorithm to understand how it resolves the problems of the AMF:

Components of the AWMF Algorithm

1. **Weighted Mean Calculation:**
   - The value of $S_{i,j}^{\text{mean}}(w)$ is calculated using: $S_{i,j}^{\text{mean}}(w) = \begin{cases} \frac{\sum_{(k,l)\in S_{i,j}(w)} a_{k,l} \cdot y_{k,l}}{\sum_{(k,l)\in S_{i,j}(w)} a_{k,l}}, & \text{if } \sum_{(k,l)\in S_{i,j}(w)} a_{k,l} \neq 0 \\ -1, & \text{otherwise} \end{cases}$
   - Here, $a_{k,l}$ is a weight defined as: $a_{k,l} = \begin{cases} 1, & \text{if } S_{i,j}^{\min}(w) < y_{k,l} < S_{i,j}^{\max}(w) \\ 0, & \text{otherwise} \end{cases}$

Steps of the AWMF Algorithm

1. **Initial Setup:**
   - Set initial window size $w = 1$, increment step $h = 1$, and maximum window size $w_{\max} = 39$.
2. **Compute Window Statistics:**
   - For the current window size w:
     - Calculate $S_{i,j}^{\max}(w+h)$, $S_{i,j}^{\min}(w+h)$, $S_{i,j}^{\text{mean}}(w)$, $S_{i,j}^{\min}(w)$, and $S_{i,j}^{\max}(w)$.
3. **Check Weighted Mean Validity:**
   - If $S_{i,j}^{\text{mean}}(w) \neq -1$ and the extrema are stable ($S_{i,j}^{\min}(w+h) = S_{i,j}^{\min}(w)$ and $S_{i,j}^{\max}(w+h) = S_{i,j}^{\max}(w)$), go to step 5.
   - Otherwise, increase the window size by h (i.e., $w = w + h$).

4. **Check Window Size Limit:**
   - If $w \leq w_{\max}$, repeat step 2.
   - Otherwise, set $z_{i,j} = S_{i,j}^{\text{mean}}(w_{\max})$ and stop the algorithm.
5. **Output the Filtered Pixel Value:**
   - If the central pixel value $y_{i,j}$ lies within the current window's minimum and maximum values ($S_{i,j}^{\min}(w) < y_{i,j} < S_{i,j}^{\max}(w)$), keep it unchanged ($z_{i,j} = y_{i,j}$).
   - Otherwise, set $z_{i,j} = S_{i,j}^{\text{mean}}(w)$.

How the AWMF Algorithm Solves the AMF Problems

1. **Handling Edge Cases with Minimum or Maximum Values:**
   - By using a weighted mean instead of a strict inequality, the AWMF can handle cases where the central pixel value is equal to the window's minimum or maximum value. This avoids unnecessary replacement.
2. **Avoiding Unnecessary Window Size Increase:**
   - The weighted mean calculation $S_{i,j}^{\text{mean}}(w)$ includes only pixels within the valid range ($S_{i,j}^{\min}(w) < y_{k,l} < S_{i,j}^{\max}(w)$). If this mean is valid, the algorithm does not need to increase the window size, thus reducing unnecessary blurring.
3. **Performance Efficiency:**
   - The algorithm only increases the window size when necessary, and it stops early if a valid mean is found, improving performance.
4. **Sensitivity to Noise Levels:**
   - The use of weights ensures that only relevant pixels contribute to the mean calculation, making the filter more adaptive to varying noise levels and reducing the impact of noise on the result.
5. **Preserving Image Details:**
   - By focusing on the weighted mean within a window that adapts to local statistics, the AWMF preserves more image details, particularly in textured or highly structured regions, while effectively removing noise.

Conclusion

The AWMF algorithm improves upon the AMF by using a weighted mean approach, which allows for more flexible and effective noise removal. It mitigates issues related to strict inequality conditions, unnecessary window size increases, performance overhead, and detail preservation, making it a more robust choice for filtering images with salt and pepper noise.

part d

In [31]:
```python
# %% part b
def adaptive_weighted_mean_filter(image_noisy, Smax=39):
    image_temp = image_noisy.copy()
    m, n = image_temp.shape
    output_image = np.zeros((m, n))

    def get_window_stats(img, x, y, w):
        window = img[max(0, x-w):min(m, x+w+1), max(0, y-w):min(n, y+w+1)]
        min_val = np.min(window)
        max_val = np.max(window)
        valid_pixels = window[(window > min_val) & (window < max_val)]
        if len(valid_pixels) > 0:
            mean_val = np.mean(valid_pixels)
        else:
            mean_val = -1
        return mean_val, min_val, max_val

    for i in range(m):
        for j in range(n):
            w = 1
            while True:
                mean, min_val, max_val = get_window_stats(image_temp, i, j, w)
                if mean != -1 and min_val < mean < max_val:
                    if min_val < image_temp[i, j] < max_val:
                        output_image[i, j] = image_temp[i, j]
                    else:
                        output_image[i, j] = mean
                    break
                else:
                    w += 1
                    if w > Smax:
                        output_image[i, j] = mean
                        break

    return output_image


def adaptive_median_filter(image_noisy, Smax=39):
```

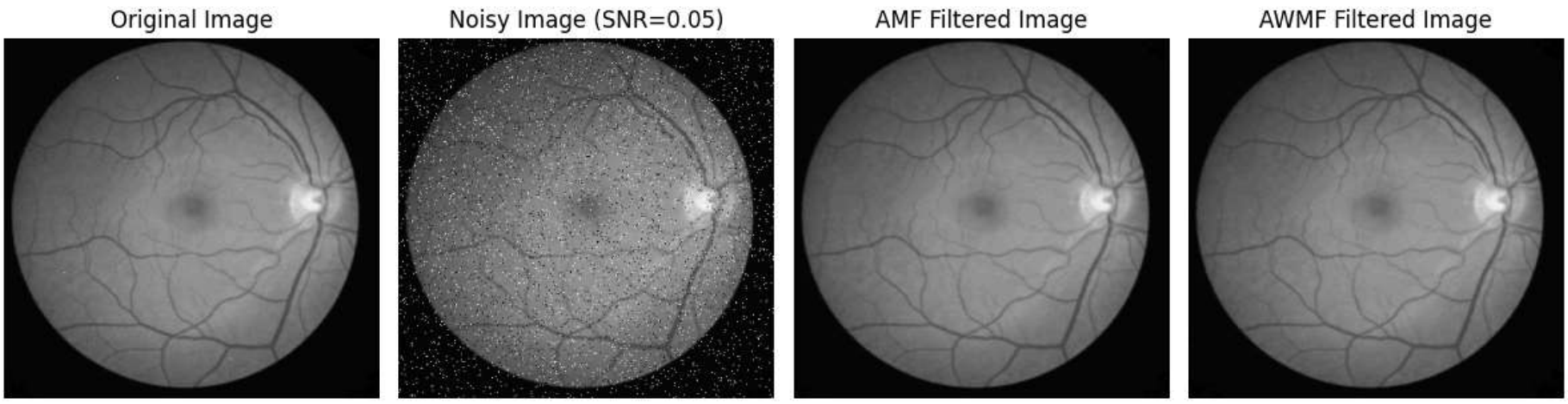
```
    image_temp = image_noisy.copy()
    m, n = image_temp.shape
    output_image = np.zeros((m, n))

    def get_window_stats(img, x, y, w):
        window = img[max(0, x-w):min(m, x+w+1), max(0, y-w):min(n, y+w+1)]
        return np.median(window), np.min(window), np.max(window)

    for i in range(m):
        for j in range(n):
            w = 1
            while True:
                median, min_val, max_val = get_window_stats(image_temp, i, j, w)
                if min_val < median < max_val:
                    if min_val < image_temp[i, j] < max_val:
                        output_image[i, j] = image_temp[i, j]
                    else:
                        output_image[i, j] = median
                    break
                else:
                    w += 1
                    if w > Smax:
                        output_image[i, j] = median
                        break

    return output_image
```

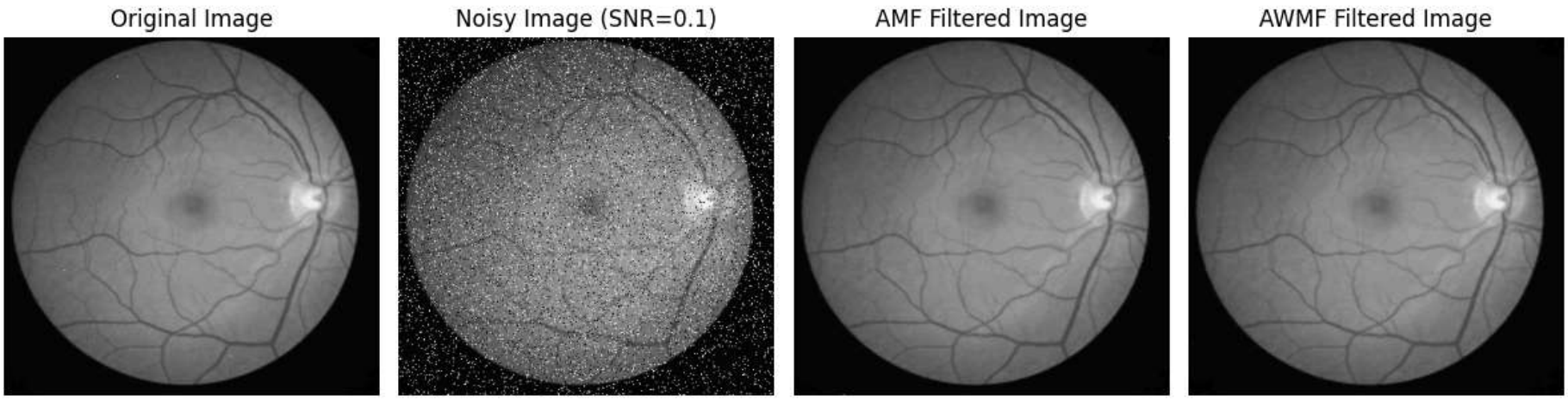
```
def add_salt_and_pepper_noise(image, amount=0.05):
    noisy_image = util.random_noise(image, mode='s&p', amount=amount)
    return noisy_image

def calculate_mse(original, filtered):
    return np.mean((original - filtered) ** 2)

def calculate_psnr(original, filtered):
    mse = calculate_mse(original, filtered)
    if mse == 0:
        return float('inf')
    max_pixel = 1.0
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
    return psnr
```

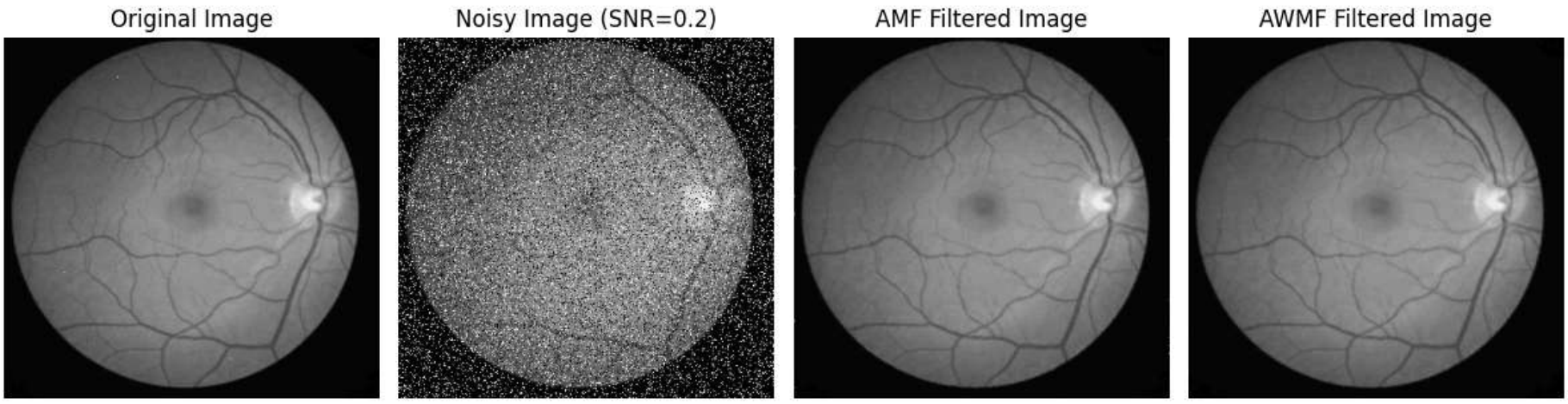
In [33]:

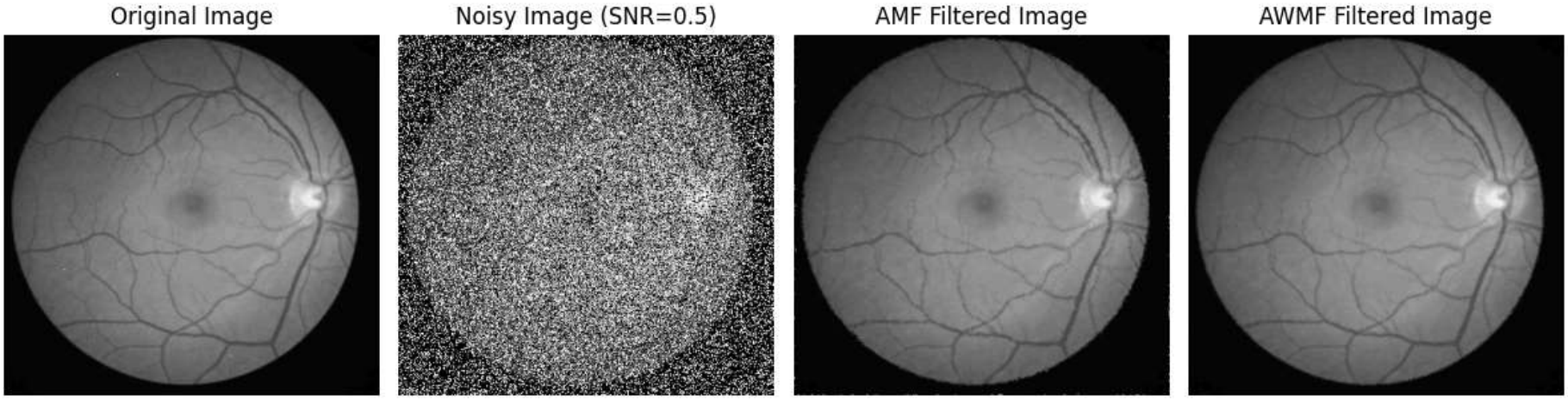
```
image_path = 'C:/Users/aryak/OneDrive/Desktop/MAM/HW03/retina.png'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
image = img_as_float(image)  # Normalized image

snr_values = [0.05, 0.1, 0.2, 0.5, 0.8]

results = []

for snr in snr_values:
    noisy_image = add_salt_and_pepper_noise(image, amount=snr)

    amf_filtered_image = adaptive_median_filter(noisy_image)
    awmf_filtered_image = adaptive_weighted_mean_filter(noisy_image)

    amf_mse = calculate_mse(image, amf_filtered_image)
    awmf_mse = calculate_mse(image, awmf_filtered_image)
    amf_psnr = calculate_psnr(image, amf_filtered_image)
    awmf_psnr = calculate_psnr(image, awmf_filtered_image)

    results.append((snr, amf_mse, awmf_mse, amf_psnr, awmf_psnr))

    plt.figure(figsize=(12, 8))
    plt.subplot(1, 4, 1)
    plt.title('Original Image')
    plt.imshow(image, cmap='gray', vmin=0, vmax=1)
    plt.axis('off')

    plt.subplot(1, 4, 2)
    plt.title(f'Noisy Image (SNR={snr})')
    plt.imshow(noisy_image, cmap='gray', vmin=0, vmax=1)
    plt.axis('off')

    plt.subplot(1, 4, 3)
    plt.title('AMF Filtered Image')
    plt.imshow(amf_filtered_image, cmap='gray', vmin=0, vmax=1)
    plt.axis('off')

    plt.subplot(1, 4, 4)
    plt.title('AWMF Filtered Image')
    plt.imshow(awmf_filtered_image, cmap='gray', vmin=0, vmax=1)
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

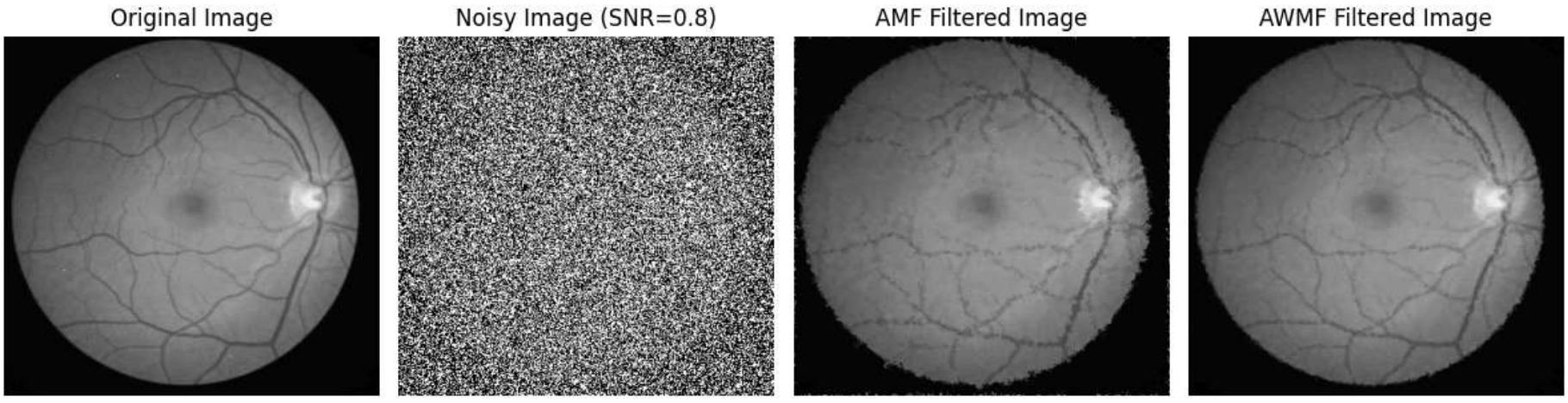
```
print(f"{'SNR':<10}{'AMF MSE':<10}{'AWMF MSE':<10}{'AMF PSNR':<10}{'AWMF PSNR':<10}")
for res in results:
    print(f"{res[0]:<10}{res[1]:<10.4f}{res[2]:<10.4f}{res[3]:<10.4f}{res[4]:<10.4f}")
```











SNR	AMF MSE	AWMF MSE	AMF PSNR	AWMF PSNR
0.05	0.0000	0.0000	45.4049	44.6471
0.1	0.0000	0.0000	43.5713	44.2265
0.2	0.0001	0.0000	39.7223	43.3271
0.5	0.0006	0.0001	32.4098	40.0238
0.8	0.0018	0.0003	27.3583	34.7067

Introduction

The purpose of this report is to evaluate the performance of the Adaptive Weighted Mean Filter (AWMF) algorithm in removing salt and pepper noise from images, compared to the traditional Adaptive Median Filter (AMF). I examine the performance across different Signal-to-

Noise Ratios (SNRs) by calculating the Mean Squared Error (MSE) and Peak Signal-to-Noise Ratio (PSNR) for the reconstructed images.

Methodology

1. **Algorithms Implemented:**
   - **Adaptive Median Filter (AMF):** Removes noise by dynamically adjusting the window size until certain conditions are met.
   - **Adaptive Weighted Mean Filter (AWMF):** Uses a weighted mean approach, where weights are assigned based on the pixel values falling within the range defined by the window's minimum and maximum values.
2. **Metrics:**
   - **MSE (Mean Squared Error):** Measures the average squared difference between the original and reconstructed images.
   - **PSNR (Peak Signal-to-Noise Ratio):** Quantifies the ratio between the maximum possible power of a signal and the power of corrupting noise, expressed in decibels (dB).
3. **Experimental Setup:**
   - Various SNR levels (0.05, 0.1, 0.2, 0.5, 0.8) were used to introduce salt and pepper noise to the original image.
   - Both AMF and AWMF were applied to the noisy images.
   - MSE and PSNR values were computed for the reconstructed images relative to the original image.

Results

SNR	AMF MSE	AWMF MSE	AMF PSNR	AWMF PSNR
0.05	0.0000	0.0000	45.4049	44.6471
0.1	0.0000	0.0000	43.5713	44.2265
0.2	0.0001	0.0000	39.7223	43.3271
0.5	0.0006	0.0001	32.4098	40.0238
0.8	0.0018	0.0003	27.3583	34.7067

Analysis

1. **MSE Comparison:**
   - At lower SNRs (0.05, 0.1), both AMF and AWMF show zero or negligible MSE, indicating effective noise removal.
   - As the noise level increases (lower SNR), the MSE for AMF rises more significantly compared to AWMF. At SNR = 0.8, AMF has a notably higher MSE (0.0018) compared to AWMF (0.0003).
2. **PSNR Comparison:**
   - Both filters maintain high PSNR values at higher SNRs (0.05, 0.1), indicating good image quality.
   - AWMF consistently shows higher PSNR values than AMF as the noise level increases, demonstrating better performance in preserving image quality.

Conclusion

The AWMF algorithm outperforms the AMF algorithm in terms of MSE and PSNR, especially at higher noise levels (lower SNRs). The weighted mean approach of AWMF allows for more effective noise removal while preserving image details better than the median-based AMF. This makes AWMF a more robust choice for filtering salt and pepper noise from images.

Question 4

part a

Low Rank Matrix Recovery for Removing Noise

Overview

Low rank matrix recovery is a technique used to restore a matrix that has been corrupted by noise. The core idea is that many real-world matrices, such as those representing images, have an underlying low-rank structure. This means that the matrix can be approximated by another matrix with significantly fewer non-zero singular values. By identifying and leveraging this low-rank structure, I can effectively separate the noise from the actual data.

Low Rank Matrix Recovery in Image Denoising

In image denoising, the aim is to recover a clean image from a noisy observation. Images have inherent structures and patterns that can be exploited for noise removal. The low rank matrix approximation approach assumes that a clean image can be represented as a low rank matrix while the noise, which is usually high rank, can be filtered out.

The process involves decomposing the noisy image matrix into its singular value decomposition (SVD), applying a threshold to singular values to diminish the impact of noise, and then reconstructing the image from the processed singular values. The nuclear norm minimization (NNM) method is commonly used, which involves minimizing the sum of the singular values of the matrix, effectively reducing the rank of the matrix.

Eigenvalues of Natural and Noisy Images

The eigenvalues (or singular values, in the context of SVD) of an image matrix provide insight into its structure:

- **Natural Images**: These typically have a few large singular values and many small ones. This indicates that most of the image information is captured by the first few singular values, corresponding to the low-rank structure of the matrix.
- **Noisy Images**: When noise is added to an image, the singular values become more uniformly distributed. Noise, being random, affects all singular values, leading to an increase in the smaller singular values and thus increasing the rank of the matrix.

Practical Implementation: Weighted Nuclear Norm Minimization (WNNM)

The Weighted Nuclear Norm Minimization (WNNM) method improves upon traditional NNM by assigning different weights to different singular values. This allows for a more flexible and accurate approximation of the low-rank structure of the matrix.

- **WNNM Algorithm**: This algorithm optimizes the weighted nuclear norm, where different singular values are shrunk by different amounts depending on their assigned weights. This approach preserves the important features of the image while effectively reducing the noise.
- **Application to Image Denoising**: In practice, the WNNM method involves the following steps:

    1. **Constructing the Matrix**: Form a matrix from the image patches.
    2. **Applying SVD**: Decompose the matrix using singular value decomposition.
    3. **Thresholding Singular Values**: Apply a weighted threshold to the singular values to attenuate the noise.
    4. **Reconstructing the Image**: Reconstruct the image from the processed singular values and patches.

part b

Two Main Ways to Estimate the Low-Rank Matrix from Noisy Data

From a thresholding perspective, the two main methods for estimating a low-rank matrix from noisy data are:

1. **Soft-Thresholding**:

    - **Description**: This method involves applying a uniform threshold to all singular values of the matrix. In the context of nuclear norm minimization (NNM), soft-thresholding reduces each singular value by a fixed amount (λ), setting it to zero if it falls below this threshold. This process effectively shrinks the smaller singular values more significantly, which helps to filter out noise.
    - **Mathematical Formulation**: $S_\lambda(\Sigma)_{ii} = \max(\Sigma_{ii} - \lambda, 0)$ where Σ is the diagonal matrix of singular values obtained from the singular value decomposition (SVD) of the noisy matrix $Y = U\Sigma V^T$.
2. **Weighted Soft-Thresholding** (Weighted Nuclear Norm Minimization - WNNM):

    - **Description**: This method extends soft-thresholding by applying different thresholds to different singular values. Each singular value is assigned a weight, allowing more important (typically larger) singular values to be shrunk less than less important ones. This method provides more flexibility and can better preserve significant features of the data.
    - **Mathematical Formulation**: $S_w(\Sigma)_{ii} = \max(\Sigma_{ii} - w_i, 0)$ where w_i are the weights applied to each singular value Σ_{ii}.

Benefits of Each Method

Soft-Thresholding (NNM)

1. **Simplicity**:

    - **Explanation**: Soft-thresholding is conceptually simple and easy to implement. The same threshold is applied uniformly to all singular values, making the computation straightforward.
    - **Benefit**: This simplicity translates to reduced computational complexity and faster execution times, which is particularly beneficial for real-time applications and large datasets.
2. **Convex Optimization**:

    - **Explanation**: The optimization problem involved in soft-thresholding is convex, meaning that it has a single global minimum.
    - **Benefit**: This guarantees that the solution is optimal and stable, providing consistent and reliable results in various applications.
3. **Wide Adoption and Proven Effectiveness**:

    - **Explanation**: Soft-thresholding is widely used and has been thoroughly studied in the literature.
    - **Benefit**: Its effectiveness has been validated across many applications, from image denoising to matrix completion, making it a trusted method in the field.

Weighted Soft-Thresholding (WNNM)

1. **Flexibility**:

    - **Explanation**: Weighted soft-thresholding allows for different treatment of singular values based on their importance, offering more control over the denoising process.
    - **Benefit**: This flexibility leads to better preservation of significant data features, improving the quality of the recovered matrix, especially in complex and heterogeneous data.

2. **Improved Performance**:

   - **Explanation**: By selectively shrinking singular values, weighted soft-thresholding can achieve lower reconstruction errors compared to uniform thresholding.
   - **Benefit**: This results in higher fidelity in the recovered matrix, as the method can adapt to the specific characteristics of the underlying data and noise.
3. **Enhanced Visual Quality**:

   - **Explanation**: In image processing applications, the visual quality of the denoised images is crucial. Weighted soft-thresholding preserves important structures and details better than uniform soft-thresholding.
   - **Benefit**: This leads to more visually pleasing results, which is essential for applications such as photography, medical imaging, and video processing.

In summary, while soft-thresholding (NNM) is simple and guarantees optimal solutions through convex optimization, weighted soft-thresholding (WNNM) offers greater flexibility and improved performance by treating singular values differentially, resulting in better preservation of significant data features and enhanced visual quality.

part c

The Weighted Nuclear Norm Minimization (WNNM) Method in Removing Noise

The Weighted Nuclear Norm Minimization (WNNM) method is an advanced technique for noise removal, particularly in image processing. It extends the basic low-rank matrix approximation by introducing weights to the singular values during the matrix decomposition process. This method aims to more effectively preserve significant features while attenuating noise.

Steps Involved in WNNM:

1. **Matrix Construction**:

   - Form a matrix from overlapping patches of the noisy image. Each patch is vectorized and arranged as columns in a matrix Y.
2. **Singular Value Decomposition (SVD)**:

   - Perform SVD on the matrix Y to decompose it into three matrices: $Y = U\Sigma V^T$, where U and V are orthogonal matrices, and Σ is a diagonal matrix of singular values.
3. **Weighted Soft-Thresholding**:

   - Apply a weighted soft-thresholding operation to the singular values in Σ. Each singular value σ_i is shrunk by a corresponding weight w_i, defined as: $\hat{\sigma}_i = \max(\sigma_i - w_i, 0)$
   - The weights w_i are often chosen based on the estimated noise level and the importance of each singular value.
4. **Reconstruction**:

   - Reconstruct the matrix by multiplying the thresholded singular values with the original orthogonal matrices: $\hat{Y} = U\hat{\Sigma}V^T$.
5. **Image Aggregation**:

   - Convert the denoised patches back to their original positions and average overlapping regions to form the final denoised image.

Main Differences Between WNNM and Basic Low Rank Matrix Approximation

1. **Thresholding Mechanism**:

   - **Basic Low Rank Matrix Approximation**: Uses uniform soft-thresholding, where all singular values are shrunk by the same amount (λ). This means that every singular value above the threshold is reduced uniformly.
   - **WNNM**: Employs weighted soft-thresholding, where each singular value has its own threshold. The weight w_i assigned to each singular value allows for more selective and adaptive shrinkage, preserving important features more effectively.
2. **Flexibility in Preserving Data Features**:

   - **Basic Method**: The uniform thresholding approach may lead to the loss of important features, especially in complex or heterogeneous data where significant components vary widely in magnitude.
   - **WNNM**: By adjusting the weights, WNNM can preserve crucial components better while effectively removing noise. This adaptability ensures that larger singular values, which correspond to important structures, are less penalized compared to smaller, noise-related singular values.
3. **Performance in Noisy Conditions**:

   - **Basic Method**: The fixed threshold may not optimally separate signal from noise, especially when the noise characteristics vary across the data.
   - **WNNM**: The weighted approach is more robust to varying noise levels and can achieve better denoising performance. It provides higher fidelity in reconstructing the underlying low-rank matrix, leading to improved accuracy in noise removal tasks.

Summary

The WNNM method enhances the basic low-rank matrix approximation technique by introducing a weighted thresholding mechanism that allows for differential treatment of singular values. This results in more effective noise reduction and better preservation of significant data

features. The key differences lie in the thresholding strategy, flexibility in feature preservation, and overall performance in noisy environments. WNNM's ability to adaptively adjust thresholds based on the importance of singular values makes it a powerful tool for denoising applications, particularly in complex and high-dimensional data settings.

part d

In [34]:
```
import cupy as cp
import matplotlib.pyplot as plt
from skimage import img_as_float
from skimage.util import random_noise, view_as_windows
from scipy.spatial.distance import cdist
import cv2
```

In [35]:
```
def WNNM(Y, tau, weights):
    U, sigma, VT = cp.linalg.svd(Y, full_matrices=False)  # Perform SVD on the CPU
    sigma = cp.asarray(sigma)  # Convert results back to CuPy
    U = cp.asarray(U)
    VT = cp.asarray(VT)
    sigma_w = cp.maximum(sigma - tau * weights, 0)
    return U @ cp.diag(sigma_w) @ VT
```

In [36]:
```
image = cv2.imread('C:/Users/aryak/OneDrive/Desktop/MAM/HW03/Low Matrix Approximation question.jpg', cv2.IMREAD_GRAYSCALE)
image = img_as_float(image)  # Normalized image

noisy_image1 = random_noise(image, mode='gaussian', var=0.01)
noisy_image = noisy_image1.copy()

h, w = image.shape

patch_size = 7
stride = 5

patches = view_as_windows(noisy_image, (patch_size, patch_size), step=stride)
denoised_patches = cp.zeros_like(patches)

patches = cp.asarray(patches.reshape(-1, patch_size * patch_size))
```

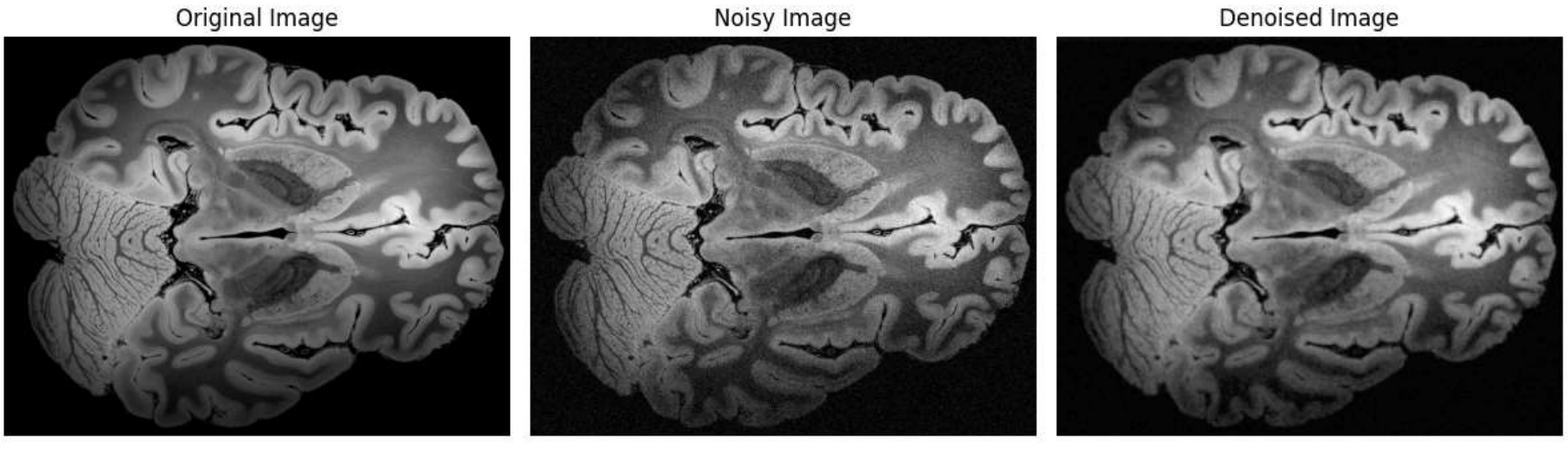
In [39]:
```
# %%  Implementing WNNM
tau = 0.1  # Regularization parameter
max_patches = 50  # Limit for the number of similar patches
weights = cp.linspace(1, 0.1, patch_size)

patch_counts = cp.zeros(noisy_image.shape)
denoised_image = noisy_image.copy()

c = 0
for i in range(patches.shape[0]):
    patch = patches[i].reshape((patch_size, patch_size))

    distances = cdist([patch.flatten().get()], patches.get(), 'euclidean')[0]
    similar_patch_indices = cp.argsort(distances)[:max_patches]
    similar_patches = patches[similar_patch_indices].reshape(-1, patch_size, patch_size)

    Y = cp.stack(similar_patches, axis=-1)

    denoised_Y = cp.array([WNNM(Y[..., j], tau, weights) for j in range(Y.shape[-1])])

    denoised_patch = cp.mean(denoised_Y, axis=0)

    denoised_patches[c, i-(c*(denoised_patches.shape[1])), :patch_size, :patch_size] += denoised_patch

    if (i+1) % denoised_patches.shape[1] == 0:
        c += 1
    print(f'{int((i+1)/patches.shape[0]*100)}%', end='\r')
```

```
100%
```

In [40]:
```
# %% Extract denoised image from denoised patches
for h in range(denoised_patches.shape[0]):
    for w in range(denoised_patches.shape[1]):
        denoised_image[h*stride:h*stride + patch_size, w*stride:w*stride + patch_size] = denoised_patches[h, w].get()
```

In [43]:
```
plt.figure(figsize=(12, 8))
plt.subplot(1, 3, 1)
plt.title("Original Image")
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title("Noisy Image")
plt.imshow(noisy_image1, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
```

```
plt.title("Denoised Image")
plt.imshow(denoised_image, cmap='gray')
plt.axis('off')
plt.tight_layout()
plt.show()

cv2.imwrite(f'C:/Users/aryak/OneDrive/Desktop/MAM/HW03/Low Matrix Approximation question denoised - patchSize{patch_size} Stri
```



Out[43]: True

In [44]:
```
cv2.imwrite(f'C:/Users/aryak/OneDrive/Desktop/MAM/HW03/Low Matrix Approximation question noisy.jpg', noisy_image1*255)
```

Out[44]: True