PROPOSED HYBRID DENOISING METHODS

1. INTRODUCTION:

Noise reduction in images is a fundamental task in image processing. While various classical and advanced denoising techniques exist, many struggle with the inherent trade-off between effective noise suppression and the preservation of fine image details, particularly edges. Edges carry crucial structural information, and their blurring or loss significantly degrades image quality. This task focuses on developing and evaluating two novel hybrid denoising approaches designed to mitigate this trade-off by intelligently adapting their denoising strength based on local image characteristics.

2. PROPOSED HYBRID DENOISING METHODS:

We developed two distinct hybrid denoising strategies:

2.1 Hybrid Edge-Aware Denoising for Enhanced Image Quality:

1. **Objective :**

The aim of this task is to develop a novel hybrid image denoising method that effectively reduces noise while preserving edges and fine image details. Our method combines edge detection and local variance analysis to perform spatially adaptive denoising - applying stronger smoothing in uniform areas and preserving structure in edge-rich regions.

2. **Proposed Methodology:**

We implement an edge-aware denoising algorithm using:

- Edge detection via the Canny Edge Detector.
- Local image variance to assess noise and flatness.
- Spatially adaptive Gaussian filtering guided by both edge and variance maps.
- Weighted merging of filtered and original image content.

3. **Algorithm Steps:**

1. Load Grayscale Image using OpenCV.
2. Apply Canny edge detection to create an edge mask.
3. Estimate local variance using a sliding window approach.
4. Normalize the variance map to [0,1].

5. Perform spatially adaptive Gaussian filtering based on local variance and edges.
6. Merge the denoised image with the original using adaptive weights.

4. **Implementation Tools:**

Language: Python

Libraries: OpenCV (cv2), NumPy

```python
import cv2
import numpy as np

# Load grayscale image
image = cv2.imread('noisy_image.jpg', cv2.IMREAD_GRAYSCALE)

# Edge Detection
edges = cv2.Canny(image, 50, 150)

# Local Variance Calculation
def local_variance(img, window_size=7):
    mean = cv2.blur(img.astype(np.float32), (window_size, window_size))
    sqr_mean = cv2.blur((img.astype(np.float32))**2, (window_size, window_size))
    return sqr_mean - mean**2

var_map = local_variance(image)
var_map_norm = cv2.normalize(var_map, None, 0, 1, cv2.NORM_MINMAX)

# Adaptive Filtering
def adaptive_filtering(image, var_map_norm, edge_mask, min_ksize=3, max_ksize=15):
    output = np.zeros_like(image, dtype=np.float32)
    h, w = image.shape

    for y in range(h):
        for x in range(w):
            if edge_mask[y, x] > 0:
                output[y, x] = image[y, x]  # Preserve edges
            else:
                weight = 1 - var_map_norm[y, x]
                ksize = int(min_ksize + weight * (max_ksize - min_ksize))
                ksize = ksize if ksize % 2 == 1 else ksize + 1
                half = ksize // 2
                ymin = max(y - half, 0)
                ymax = min(y + half + 1, h)
                xmin = max(x - half, 0)
                xmax = min(x + half + 1, w)
                patch = image[ymin:ymax, xmin:xmax]
                if patch.shape[0] > 1 and patch.shape[1] > 1:
                    smoothed = cv2.GaussianBlur(patch, (ksize, ksize), 0)
                    output[y, x] = smoothed[patch.shape[0]//2, patch.shape[1]//2]
                else:
                    output[y, x] = image[y, x]
    return output.astype(np.uint8)

adaptive_denoised = adaptive_filtering(image, var_map_norm, edges)

# Merge (optional)
output = cv2.addWeighted(image, 0.4, adaptive_denoised, 0.6, 0)

# Save result
cv2.imwrite('denoised_output.jpg', output)

print("✅ Denoising complete. Check 'denoised_output.jpg'")
```

5. **Results:**

- Input: Noisy grayscale image (noisy_image.jpg)

- Output: Edge map, variance map, and adaptive denoised image (denoised_output.jpg)

6. **Outcome**: Smoothed flat areas; edges and textures preserved

7. **Error Handling:** The script checks whether the image file is successfully loaded and exits with a clear message if not. This prevents runtime errors due to missing or invalid input files.

8. **Limitations:**

- Slower processing due to pixel-wise operations.
- Currently limited to grayscale images

9. **Conclusion:**

This hybrid denoising approach offers a practical, intelligent alternative to global smoothing techniques. By combining edge awareness and local statistics, it preserves critical visual features while mitigating noise, making it suitable for applications in medical imaging, remote sensing, and computer vision pipelines.

2.2 **Iterative Residual Denoising (IRD)**

Concept

Iterative Residual Denoising (IRD) is a multi-pass denoising technique designed to progressively enhance image quality by refining the residual noise components left after each denoising step. Unlike single-pass methods that may over-smooth or lose details, IRD leverages repeated residual cleaning to recover fine textures and reduce noise gradually.

Algorithm Steps

1. **Initial Denoising:**
   Apply a base denoising algorithm such as median filtering, wavelet denoising, or Non-Local Means (NLM) to the noisy image.
2. **Compute Residual**:
   Calculate the residual noise component as:

   $$\text{residual} = \text{noisy\_image} - \text{denoised\_image}$$

3. **Secondary Denoising on Residual**:
   Apply a denoising filter on the residual image to clean it further.
4. **Residual Addition**:
   Add the cleaned residual back to the denoised image:

   $$\text{updated\_image} = \text{denoised\_image} + \text{cleaned\_residual}$$

5. **Iterate**:
   Repeat steps 2–4 for NNN iterations or until the improvement between iterations converges below a threshold.

```python
import cv2
import numpy as np

def iterative_residual_denoising(noisy_img, iterations=3, median_ksize=3, residual_blur_ksize=3):
    """
    Perform Iterative Residual Denoising (IRD).

    Parameters:
    - noisy_img: input noisy grayscale image (np.ndarray)
    - iterations: number of IRD iterations
    - median_ksize: kernel size for median filter (initial denoising)
    - residual_blur_ksize: kernel size for Gaussian blur on residual

    Returns:
    - denoised_img: final denoised image after IRD
    """
    # Initial denoising with median filter
    denoised_img = cv2.medianBlur(noisy_img, median_ksize)

    for i in range(iterations):
        # Compute residual noise
        residual = cv2.subtract(noisy_img, denoised_img)

        # Clean residual with Gaussian blur
        cleaned_residual = cv2.GaussianBlur(residual, (residual_blur_ksize, residual_blur_ksize), 0)

        # Add cleaned residual back to the denoised image
        denoised_img = cv2.add(denoised_img, cleaned_residual)
```

```python
        # Clip values to valid range [0, 255]
        denoised_img = np.clip(denoised_img, 0, 255).astype(np.uint8)

    return denoised_img

# Example usage
if __name__ == "__main__":
    # Load noisy grayscale image
    noisy_img = cv2.imread("noisy_image.jpg", cv2.IMREAD_GRAYSCALE)
    if noisy_img is None:
        raise FileNotFoundError("Input image not found!")

    # Apply Iterative Residual Denoising
    denoised_img = iterative_residual_denoising(noisy_img, iterations=5)

    # Save or display results
    cv2.imwrite("ird_denoised_output.jpg", denoised_img)
    cv2.imshow("Noisy Image", noisy_img)
    cv2.imshow("IRD Denoised Image", denoised_img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

3. **Advantages:**

- **Detail Recovery**: Recovers fine image details and textures that may be smoothed out in the first denoising pass.
- **Progressive Noise Reduction**: Gradually removes both structured and random noise components.
- **Flexible**: Can be combined with different denoising methods to form hybrid models (e.g., wavelet + Non-Local Means).

- **Reduced Over-Smoothing**: Prevents excessive blurring common in single-pass filters by focusing on residual noise.

4. Evaluation Metrics

- **Peak Signal-to-Noise Ratio (PSNR)**: Measures the ratio between the maximum possible power of a signal and the power of corrupting noise. Higher PSNR indicates better denoising quality.
- **Structural Similarity Index Measure (SSIM)**: Evaluates the visual impact of three characteristics—luminance, contrast, and structure—between original and denoised images. Higher SSIM reflects better perceptual quality and edge preservation.

5. **Results:**

- Input: Noisy grayscale image (noisy_image.jpg)

- Output: