Digital Image Processing Lab Report 3
CSE 4734

Abdullah Al Jubaer Gem
ID: 210041226
Section: 2B

December 21, 2025

1 Task 1: Linear Filter

1.1 Problem Statement

Implement Smoothing Operation with Average Filter (Box & Weighted Average filters). The filters should have user-defined parameters for different levels of blurring.

1.2 Solution Approach

Linear smoothing filters reduce noise and blur images by averaging pixel values in a neighborhood. The box filter uses a uniform kernel where all neighboring pixels contribute equally to the average, resulting in simple but effective blurring. The weighted average filter employs a Gaussian kernel, which gives more weight to central pixels and less to distant ones, producing smoother blurring that mimics natural diffusion. Both filters handle RGB images by applying the operation to each color channel independently, ensuring color information is preserved while smoothing intensity variations. Padding is used to handle image boundaries, maintaining the output size.

1.3 Implementation

```
def boxFilter(image, size):
    image = np.array(image, dtype=float)
    if len(image.shape) == 3:
        result = np.zeros_like(image)
        for c in range(3):
            channel = image[:, :, c]
            pad = size // 2
            channel_padded = np.pad(channel, pad, mode='constant')
            blur_channel = np.zeros_like(channel, dtype=float)
            kernel = np.ones((size, size)) / (size * size)
            for i in range(channel.shape[0]):
                for j in range(channel.shape[1]):
                    region = channel_padded[i:i+size, j:j+size]
                    blur_channel[i, j] = np.sum(region * kernel)
            result[:, :, c] = blur_channel
        return result
    else:
        pad = size // 2
        image_padded = np.pad(image, pad, mode='constant')
        blur_image = np.zeros_like(image, dtype=float)
        kernel = np.ones((size, size)) / (size * size)
        for i in range(image.shape[0]):
            for j in range(image.shape[1]):
                region = image_padded[i:i+size, j:j+size]
                blur_image[i, j] = np.sum(region * kernel)
        return blur_image

def weightedFilter(image, size):
    image = np.array(image, dtype=float)
    if len(image.shape) == 3:
```

```
        result = np.zeros_like(image)
        for c in range(3):
            channel = image[:, :, c]
            pad = size // 2
            channel_padded = np.pad(channel, pad, mode='constant')
            blur_channel = np.zeros_like(channel, dtype=float)
            x = np.linspace(-1, 1, size)
            xx, yy = np.meshgrid(x, x)
            kernel = np.exp(-(xx**2 + yy**2))
            kernel /= np.sum(kernel)
            for i in range(channel.shape[0]):
                for j in range(channel.shape[1]):
                    region = channel_padded[i:i+size, j:j+size]
                    blur_channel[i, j] = np.sum(region * kernel)
            result[:, :, c] = blur_channel
        return result
    else:
        pad = size // 2
        image_padded = np.pad(image, pad, mode='constant')
        blur = np.zeros_like(image, dtype=float)
        x = np.linspace(-1, 1, size)
        xx, yy = np.meshgrid(x, x)
        kernel = np.exp(-(xx**2 + yy**2))
        kernel /= np.sum(kernel)
        for i in range(image.shape[0]):
            for j in range(image.shape[1]):
                region = image_padded[i:i+size, j:j+size]
                blur[i, j] = np.sum(region * kernel)
        return blur
```

1.4 Output

The following figure shows the original image and the results of box and weighted filters in RGB and HSV color spaces.

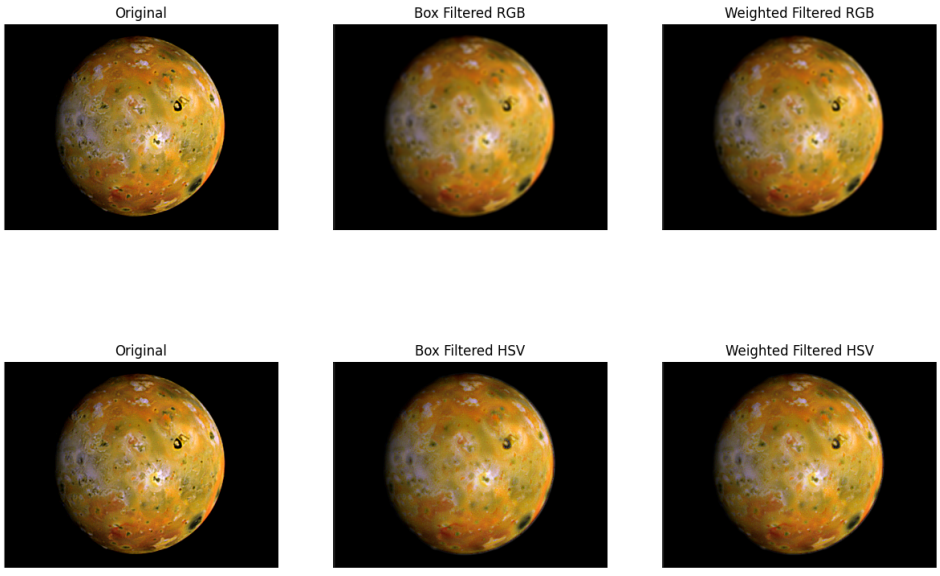

Figure 1: Task 1: Box and Weighted Filters

1.5 Analysis

1. **How the parameters affect the blurring effect:** The primary parameter is the filter size, which determines the kernel dimensions. A larger size increases the number of pixels averaged for each output pixel, leading to more pronounced blurring as high-frequency details are suppressed. For example, a 3x3 kernel provides mild smoothing, while a 15x15 kernel can significantly reduce texture and noise but may cause loss of important features like edges.

2. **Intensity scaling after averaging filters:** Intensity scaling is not required because averaging operations naturally produce values within the valid range. Since the kernel weights sum to 1, the output is a convex combination of input intensities, ensuring values stay between 0 and 255 for 8-bit images without overflow or underflow.

3. **Differences between RGB and HSV:** In RGB color space, each channel (R, G, B) is filtered independently, which can lead to color artifacts or shifts if the blurring affects channels differently due to their correlation. HSV color space decouples intensity (V) from color information (H, S), so filtering only the V channel preserves hue and saturation, resulting in more natural-looking blurred images that maintain color fidelity.

2 Task 2: Noise Addition and Filtering

2.1 Problem Statement

Implement Salt & Pepper noise addition and Median, Min, Max filters for noise removal.

2.2 Solution Approach

Salt and pepper noise corrupts random pixels by setting them to extreme values (0 or 255). The noise level parameter controls the percentage of affected pixels. For filtering, the median filter replaces each pixel with the median of its neighborhood, effectively removing impulse noise while preserving edges. Min and max filters use the minimum or maximum values, which can be useful for morphological operations but may not be ideal for general noise removal. All filters use edge padding to handle boundaries and process color images channel-wise.

2.3 Implementation

```python
def salt_and_pepper(image, noise_level):
    noisy_image = image.copy()
    total_pixels = image.size
    num_noisy = int(noise_level * total_pixels)

    coords = np.random.randint(0, total_pixels, num_noisy)

    flat = noisy_image.flatten()

    for idx in coords:
        flat[idx] = 255 if np.random.rand() < 0.5 else 0

    return flat.reshape(image.shape)

def medianFilter(noisy_image, size):
    noisy_image = np.array(noisy_image, dtype=float)
    if len(noisy_image.shape) == 3:
        result = np.zeros_like(noisy_image)
        for c in range(3):
            channel = noisy_image[:, :, c]
            pad = size // 2
            padded = np.pad(channel, pad, mode='edge')
            corrected = np.zeros_like(channel)
            for i in range(channel.shape[0]):
                for j in range(channel.shape[1]):
                    region = padded[i:i+size, j:j+size]
                    corrected[i, j] = np.median(region)
            result[:, :, c] = corrected
        return result
    else:
        pad = size // 2
```

```
        padded = np.pad(noisy_image, pad, mode='edge')
        corrected = np.zeros_like(noisy_image)
        for i in range(noisy_image.shape[0]):
            for j in range(noisy_image.shape[1]):
                region = padded[i:i+size, j:j+size]
                corrected[i, j] = np.median(region)
        return corrected

def minFilter(noisy_image, size):
    # Similar implementation for min
    ...

def maxFilter(noisy_image, size):
    # Similar implementation for max
    ...
```

2.4 Output

The following figure shows the noisy image and the results of median, min, and max filters in RGB and HSV color spaces.

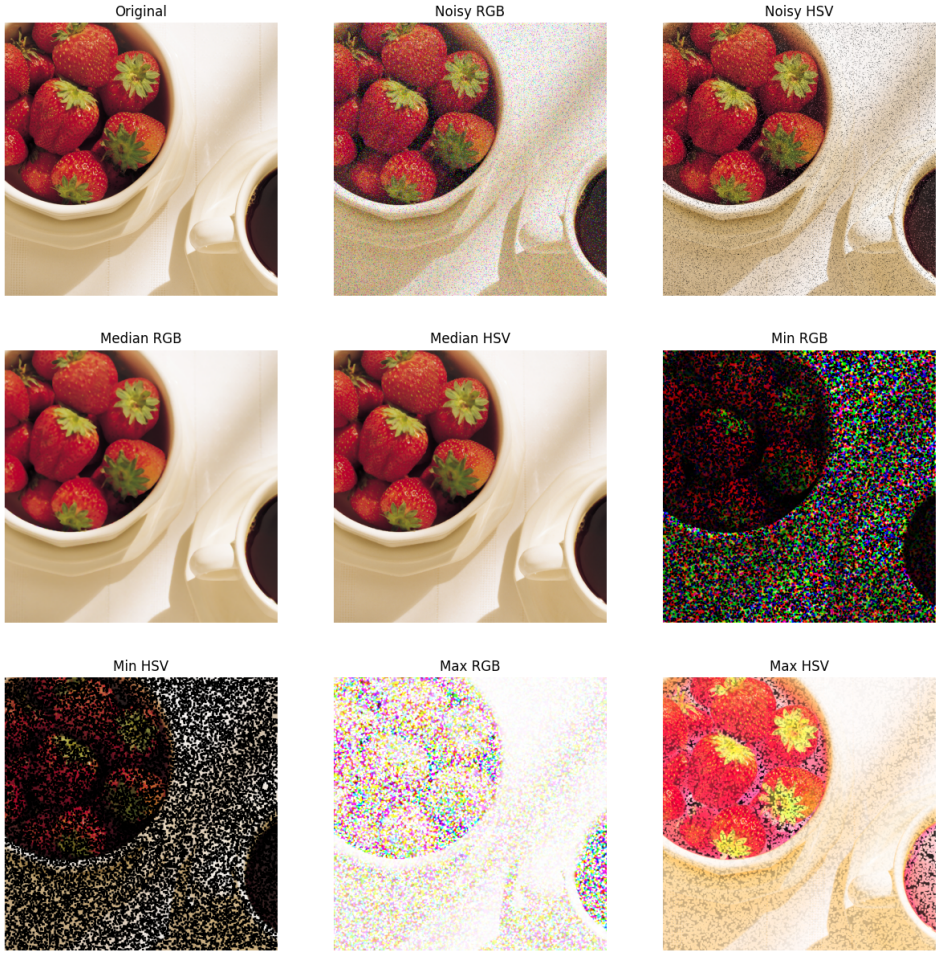

Figure 2: Task 2: Noise Addition and Filtering

2.5 Analysis

1. **Effect of noise level increase:** As the noise level increases, more pixels are corrupted, making complete noise removal challenging. With higher noise density, even larger filters may leave residual noise, and the effectiveness of median filtering decreases if the noise exceeds 50% in local neighborhoods, potentially leading to distorted outputs.

2. **Filter size effect on noise reduction:** Larger filter sizes consider more neighboring pixels, improving noise reduction by increasing the likelihood of including uncorrupted values in the median calculation. However, this comes at the cost of detail preservation, as edges and fine structures may be blurred or lost with oversized kernels.

3. **Differences between RGB and HSV:** RGB processing applies filters to each color channel separately, which can alter color relationships if noise affects channels unevenly. HSV processing focuses on the intensity channel (V), leaving color components untouched, thus maintaining color accuracy and avoiding artifacts that might arise from independent channel filtering.

3 Task 3: Laplacian Sharpening

3.1 Problem Statement

Implement Laplacian filter for edge detection and a sharpen function for image sharpening.

3.2 Solution Approach

The Laplacian operator computes the second derivative of the image intensity, highlighting regions of rapid intensity change (edges). The standard 3x3 Laplacian kernel used here emphasizes the center pixel negatively against its neighbors. The sharpen function enhances image details by adding a scaled version of the Laplacian response back to the original image, amplifying edges. The sharpen_level parameter controls the enhancement strength, and clipping ensures output values remain in the valid range. For color images, the process is applied per channel.

3.3 Implementation

```
def laplacian(image):
    image = np.array(image, dtype=float)
    if len(image.shape) == 3:
        result = np.zeros_like(image)
        for c in range(3):
            channel = image[:, :, c]
            kernel = np.array([[0, 1, 0],
                               [1, -4, 1],
                               [0, 1, 0]], dtype=float)
            edge_response = cv2.filter2D(channel, -1, kernel)
            result[:, :, c] = edge_response
        return result
    else:
        kernel = np.array([[0, 1, 0],
                           [1, -4, 1],
                           [0, 1, 0]], dtype=float)
        edge_response = cv2.filter2D(image, -1, kernel)
        return edge_response

def sharpen(image, sharpen_level=1.0):
    img = np.array(image, dtype=float)
    if len(img.shape) == 3:
        result = np.zeros_like(img)
        for c in range(3):
            channel = img[:, :, c]
            edges = laplacian(channel)
            sharp_channel = channel + sharpen_level * edges
            result[:, :, c] = np.clip(sharp_channel, 0, 255)
        return Image.fromarray(result.astype(np.uint8))
    else:
        edges = laplacian(img)
        sharp_image = img + sharpen_level * edges
        sharp_image = np.clip(sharp_image, 0, 255).astype(np.uint8)
```

```
        return Image.fromarray(sharp_image)
```

3.4 Output

The following figure shows the original image and the sharpened results in RGB and HSV color spaces.

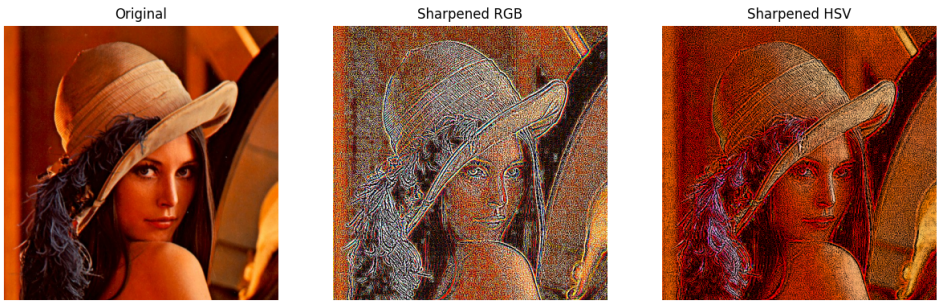

Figure 3: Task 3: Laplacian Sharpening

3.5 Analysis

1. **Why intensity scaling after sharpening:** Sharpening involves adding edge information to the original image, which can result in pixel values exceeding 255 or dropping below 0 due to the Laplacian's response. Clipping (or scaling) is necessary to constrain values to the displayable range, preventing overflow and maintaining image integrity.

2. **Effect of sharpen_level:** The sharpen_level parameter multiplies the Laplacian output before addition. Higher values amplify edge enhancement, making details crisper but risking oversharpening artifacts like ringing or noise amplification. Lower values provide subtle enhancement, preserving the natural look while still improving perceived sharpness.

3. **Differences between RGB and HSV:** Direct sharpening in RGB can introduce color halos or shifts because edges in different channels may not align perfectly. Sharpening only the value (V) channel in HSV enhances intensity details without affecting hue or saturation, leading to cleaner, more color-accurate results.

4 Task 4: Unsharp Masking and High-boost Filtering

4.1 Problem Statement

Implement a function for Unsharp Masking (k=1) and High-boost Filtering (k¿1).

4.2 Solution Approach

Unsharp masking enhances image sharpness by subtracting a blurred version from the original and adding the result back, effectively amplifying high-frequency components. For $k = 1$, this performs standard unsharp masking. Higher k values (high-boost filtering) further amplify the details, emphasizing edges more strongly. The process uses a weighted average filter for blurring, and the mask represents the difference between original and blurred images. Color images are handled by processing each channel, with clipping to maintain valid intensity ranges.

4.3 Implementation

```
def high_boost_filter(image, k, filter_size=5):
  original_image = image.copy().astype(np.float32)

  if len(original_image.shape) == 3:
      sharpened_channels = []
```

```
        for i in range(3):
            channel = original_image[:, :, i]

            blurred_channel = weightedFilter(channel, filter_size)

            mask = channel - blurred_channel

            sharpened_channel = channel + k * mask
            sharpened_channels.append(sharpened_channel)

        filtered_image = np.stack(sharpened_channels, axis=2)

    else:
        blurred_image = weightedFilter(original_image, filter_size)
        mask = original_image - blurred_image

        filtered_image = original_image + k * mask

    filtered_image = np.clip(filtered_image, 0, 255).astype(np.uint8)

    return filtered_image
```

4.4 Output

The following figure shows the results of unsharp masking and high-boost filtering in RGB and HSV color spaces.

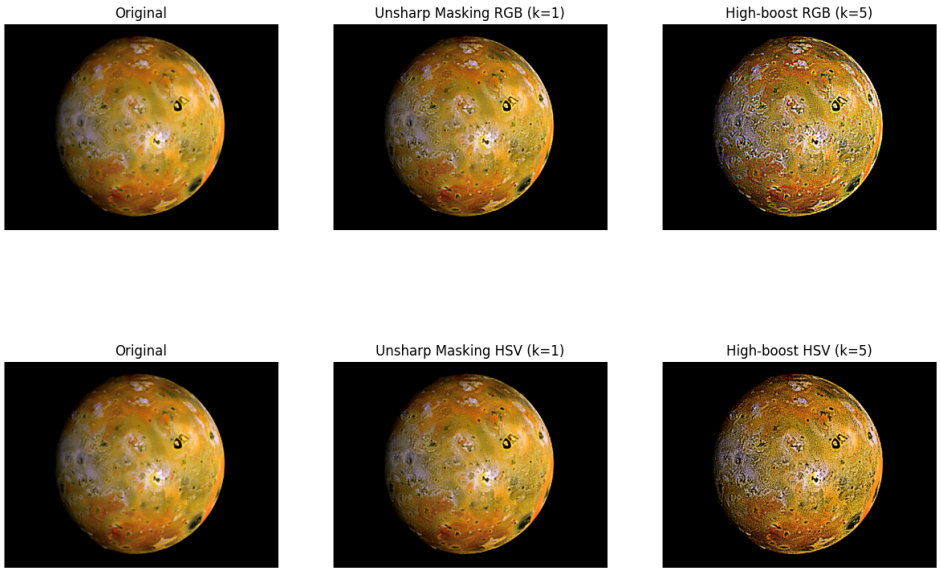

Figure 4: Task 4: Unsharp Masking and High-boost Filtering

4.5 Analysis

1. **Differences in RGB and HSV:** In RGB, the sharpening operation is applied independently to each color channel, which can cause color imbalances or artifacts if the high-frequency components differ across channels. In HSV, only the value (intensity) channel is sharpened, preserving the color information (hue and saturation) intact, resulting in more visually pleasing and color-consistent outputs.

2. **What happens if $k < 1$:** When $0 < k < 1$, the sharpening effect is reduced compared to unsharp masking ($k = 1$), providing mild enhancement that emphasizes details less aggressively. At $k = 0$, the image remains unchanged. For $k < 0$, the process inverts, effectively blurring the image by subtracting amplified details, which can be used for smoothing but is not typical for sharpening applications.