Digital Image Processing Lab Report 3

Name: Nazmus Sadiq
ID: 210041139
Date: December 13, 2025

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
grayscale_image = cv2.imread("/content/Fig0115(b)(100-dollars).tif")
color_image = cv2.imread("/content/gow.jpg")
color_image = color_image[..., ::-1].copy()   #conversion to rgb
color_image2 = cv2.imread("/content/expedition33.png")
color_image2 = color_image2[..., ::-1].copy()   #conversion to rgb
```

Task 1: Linear Filter

You are required to implement the Smoothing Operation with Average Filter (Box & Weighted Average filters). Both filters can contain user defined parameters to obtain different levels of blurring effects, which is the size of the filter in this case. Complete the following two functions

```
def boxFilter(image, size):
    # need odd sized image for single center
    if size % 2 == 0:
        raise ValueError("Filter size must be odd")
    img = np.array(image, dtype=np.float32)
    kernel = np.ones((size, size), np.float32) / (size * size)
    #slides the kernel and computes the sum of elementwise products of
each kernel, -1 means same depth
    blur_image = cv2.filter2D(img, -1, kernel)

    return blur_image.astype(np.uint8)

def weightedAverageFilter(image, size):
    if size % 2 == 0:
        raise ValueError("Filter size must be odd")
    img = np.array(image, dtype=np.float32)
    sigma = size / 6.0    #as gaussian spans +- 3*sigma
    kernel_1D = cv2.getGaussianKernel(size, sigma)
    kernel = kernel_1D @ kernel_1D.T

    kernel = kernel / np.sum(kernel)    # normalizing so that sum =1
    blur_image = cv2.filter2D(img, -1, kernel)

    return blur_image.astype(np.uint8)
```

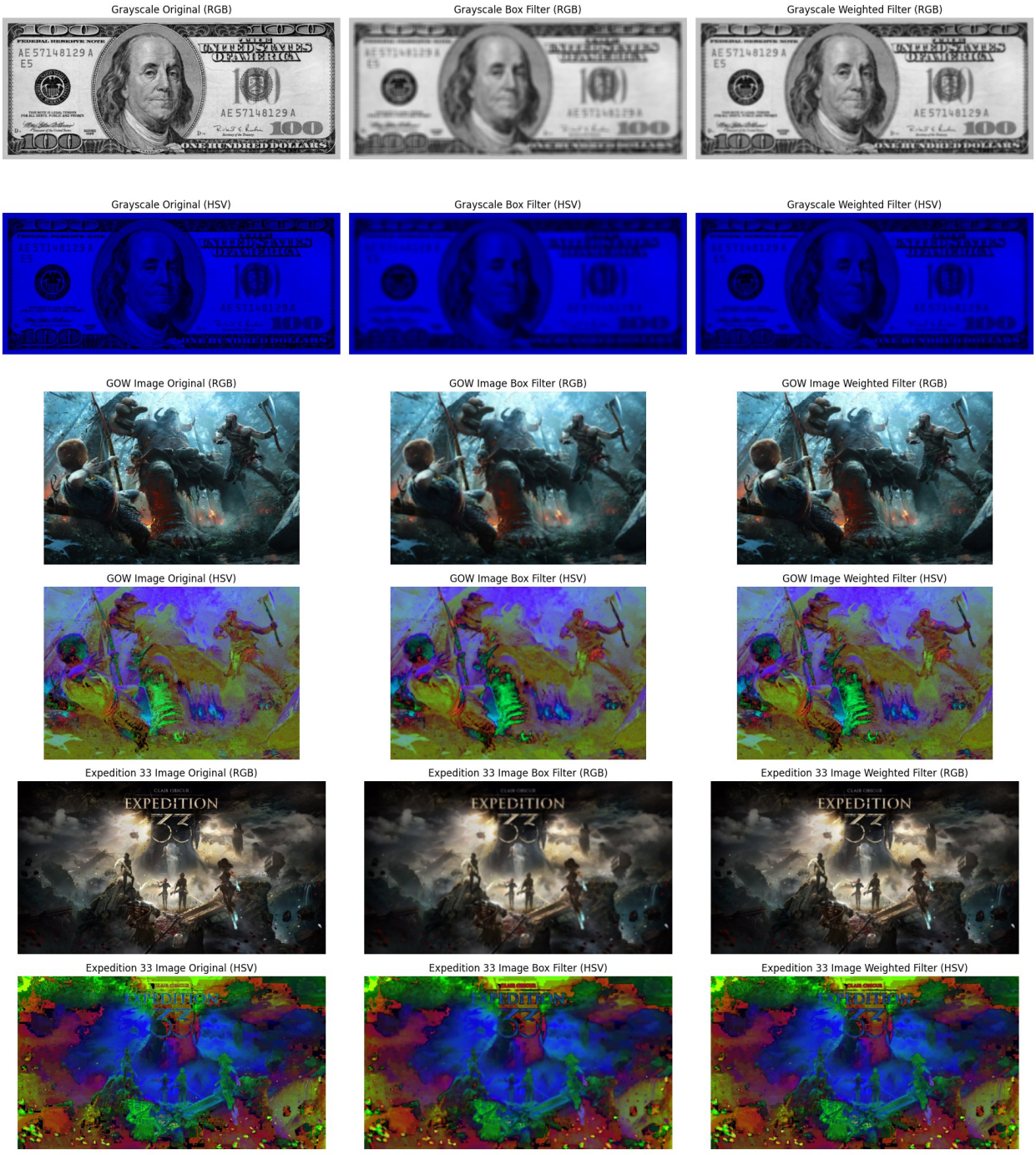
Task 1: Output and Analysis

For this task, use an appropriate color image from Chapter 06.zip. Use your previously implemented functions to show the outputs of the two averaging filters on the sample image in both RGB and HSV color spaces. Provide your analysis on the following topics: Explain how the parameters affect the increase or decrease of the blurring effect.

```
# Generate outputs for Task 1 here

# --- Grayscale Image Processing ---
box_blur_gray = boxFilter(grayscale_image, 15)
weighted_blur_gray = weightedAverageFilter(grayscale_image, 15)
hsv_original_gray = cv2.cvtColor(grayscale_image, cv2.COLOR_RGB2HSV)
hsv_box_gray = cv2.cvtColor(box_blur_gray, cv2.COLOR_RGB2HSV)
hsv_weighted_gray = cv2.cvtColor(weighted_blur_gray,
cv2.COLOR_RGB2HSV)

# --- Color Image 1 Processing ---
box_blur_color1 = boxFilter(color_image, 15)
weighted_blur_color1 = weightedAverageFilter(color_image, 15)
hsv_original_color1 = cv2.cvtColor(color_image, cv2.COLOR_RGB2HSV)
hsv_box_color1 = cv2.cvtColor(box_blur_color1, cv2.COLOR_RGB2HSV)
hsv_weighted_color1 = cv2.cvtColor(weighted_blur_color1,
cv2.COLOR_RGB2HSV)

# --- Color Image 2 Processing ---
# Check if color_image2 is loaded before processing
if color_image2 is not None:
    box_blur_color2 = boxFilter(color_image2, 15)
    weighted_blur_color2 = weightedAverageFilter(color_image2, 15)
    hsv_original_color2 = cv2.cvtColor(color_image2,
cv2.COLOR_RGB2HSV)
    hsv_box_color2 = cv2.cvtColor(box_blur_color2, cv2.COLOR_RGB2HSV)
    hsv_weighted_color2 = cv2.cvtColor(weighted_blur_color2,
cv2.COLOR_RGB2HSV)
else:
    # Create placeholder arrays or handle the error appropriately
    # For plotting, we can use blank images or simply skip this row if
color_image2 is None
    box_blur_color2 = np.zeros_like(grayscale_image)
    weighted_blur_color2 = np.zeros_like(grayscale_image)
    hsv_original_color2 = np.zeros_like(grayscale_image)
    hsv_box_color2 = np.zeros_like(grayscale_image)
    hsv_weighted_color2 = np.zeros_like(grayscale_image)

# ---- PLOT BOTH RGB + HSV for all images----
fig, axes = plt.subplots(6, 3, figsize=(18, 20))

# Grayscale Image - RGB row
```

```
axes[0, 0].imshow(grayscale_image)
axes[0, 0].set_title("Grayscale Original (RGB)")
axes[0, 1].imshow(box_blur_gray)
axes[0, 1].set_title("Grayscale Box Filter (RGB)")
axes[0, 2].imshow(weighted_blur_gray)
axes[0, 2].set_title("Grayscale Weighted Filter (RGB)")

# Grayscale Image - HSV row
axes[1, 0].imshow(hsv_original_gray)
axes[1, 0].set_title("Grayscale Original (HSV)")
axes[1, 1].imshow(hsv_box_gray)
axes[1, 1].set_title("Grayscale Box Filter (HSV)")
axes[1, 2].imshow(hsv_weighted_gray)
axes[1, 2].set_title("Grayscale Weighted Filter (HSV)")

# Color Image 1 - RGB row
axes[2, 0].imshow(color_image)
axes[2, 0].set_title("GOW Image Original (RGB)")
axes[2, 1].imshow(box_blur_color1)
axes[2, 1].set_title("GOW Image Box Filter (RGB)")
axes[2, 2].imshow(weighted_blur_color1)
axes[2, 2].set_title("GOW Image Weighted Filter (RGB)")

# Color Image 1 - HSV row
axes[3, 0].imshow(hsv_original_color1)
axes[3, 0].set_title("GOW Image Original (HSV)")
axes[3, 1].imshow(hsv_box_color1)
axes[3, 1].set_title("GOW Image Box Filter (HSV)")
axes[3, 2].imshow(hsv_weighted_color1)
axes[3, 2].set_title("GOW Image Weighted Filter (HSV)")

# Color Image 2 - RGB row
axes[4, 0].imshow(color_image2)
axes[4, 0].set_title("Expedition 33 Image Original (RGB)")
axes[4, 1].imshow(box_blur_color2)
axes[4, 1].set_title("Expedition 33 Image Box Filter (RGB)")
axes[4, 2].imshow(weighted_blur_color2)
axes[4, 2].set_title("Expedition 33 Image Weighted Filter (RGB)")

# Color Image 2 - HSV row
axes[5, 0].imshow(hsv_original_color2)
axes[5, 0].set_title("Expedition 33 Image Original (HSV)")
axes[5, 1].imshow(hsv_box_color2)
axes[5, 1].set_title("Expedition 33 Image Box Filter (HSV)")
axes[5, 2].imshow(hsv_weighted_color2)
axes[5, 2].set_title("Expedition 33 Image Weighted Filter (HSV)")

# Remove axes
for ax in axes.flat:
    ax.axis('off')
```

```
plt.tight_layout()
plt.show()
```



Analysis

1. Effect of the `size` Parameter on Blurring

The `size` parameter in both the `boxFilter` and `weightedAverageFilter` functions determines the dimensions of the convolution kernel (filter mask). As the value of `size` increases, the kernel covers a larger neighborhood around each pixel, incorporating more surrounding pixel intensities into the averaging process. This results in stronger smoothing, where fine details and high-frequency components (such as edges and texture) are increasingly suppressed, producing a more pronounced blurring effect.

Conversely, a smaller `size` corresponds to a smaller neighborhood and fewer pixels being averaged. In this case, less smoothing occurs, more local details are preserved, and the resulting blur is relatively mild. Therefore, the `size` parameter directly controls the trade-off between noise reduction/smoothing and detail preservation.

---

2. Requirement of Intensity Scaling After Averaging Filters

In general, explicit intensity scaling is not required after applying averaging-based filters. Both box filters and weighted average filters typically use normalized kernels, meaning the sum of all kernel weights is equal to 1 (e.g., `size × size` normalization for a box filter or division by `np.sum(kernel)` for a weighted average filter). This normalization ensures that the overall brightness of the image is preserved, preventing systematic brightening or darkening.

However, due to floating-point computations and boundary handling, some pixel values may slightly exceed the valid display range ([0, 255]). In such cases, clipping (e.g., using `np.clip`) is necessary before converting the result to `uint8` for proper visualization. This clipping step should not be confused with intensity scaling, as it merely enforces valid intensity bounds rather than adjusting the image's contrast or brightness.

---

3. Differences Between RGB and HSV Color Spaces

Grayscale Images:
For grayscale images, filtering the single intensity channel directly or converting the image to RGB (where all three channels are identical) yields equivalent results. When a grayscale image is converted to HSV, the Hue and Saturation channels are typically zero or undefined, while the Value (V) channel contains the original intensity information. Consequently, filtering in HSV space effectively reduces to filtering the V channel, producing results equivalent to grayscale filtering.

Color Images:
For color images, the choice of color space has a noticeable impact on filtering behavior:

- **RGB Color Space:**
  Filtering in RGB space applies the blur independently to the Red, Green, and Blue channels. This approach smooths intensity variations within each channel and

blends colors locally. While effective, it can sometimes lead to slight color mixing across edges, potentially affecting color fidelity.

- **HSV Color Space:**
  In HSV space, a common and perceptually motivated approach is to apply filtering only to the Value (V) channel, while keeping Hue (H) and Saturation (S) unchanged. This smooths brightness variations without altering the underlying color information, often resulting in a more visually natural blur.
  If filtering is applied indiscriminately to all HSV channels (which is generally not recommended), it may introduce hue shifts or reduce saturation, leading to perceptual distortions.

In the current results, the filters were applied in RGB space and the filtered images were then converted to HSV for visualization. As such, the HSV images represent the HSV interpretation of RGB-filtered results, rather than a true comparison between filtering performed directly in RGB versus HSV space. A proper comparison would require applying the filter directly to the V channel in HSV space and then converting the result back to RGB.

Task 2

For this task, you are required to write a function designed to artificially add Salt & Pepper noise to a given image. The function should contain a parameter called noise_level (value between 0 and 1) that determines the percentage of pixels that will be affected by Salt & Pepper noise.

Also, write three separate functions to implement the Median Filter, Min Filter, and Max Filter.

```
import numpy as np

def salt_and_pepper(image, noise_level):
    noisy_image = image.copy()

    total_pixels = image.shape[0] * image.shape[1]
    num_noise = int(noise_level * total_pixels)

    # Randomly choose pixel positions
    coords = [
        np.random.randint(0, image.shape[0], num_noise),
        np.random.randint(0, image.shape[1], num_noise)
    ]

    # Half salt (white), half pepper (black)
    half = num_noise // 2

    # Pepper (0) for first half
    noisy_image[coords[0][:half], coords[1][:half]] = 0

    # Salt (255) for second half
    noisy_image[coords[0][half:], coords[1][half:]] = 255
```

```
    return noisy_image

# eliminates salt and pepper noise(both 255 and 0)
def medianFilter(noisy_image, size):
    return cv2.medianBlur(noisy_image, ksize=size)

# eliminates salt noise(255)
def minFilter(noisy_image, size):
    kernel = np.ones((size, size), np.uint8)
    return cv2.erode(noisy_image, kernel)

# eliminates pepper noise(0)
def maxFilter(noisy_image, size):
    kernel = np.ones((size, size), np.uint8)
    return cv2.dilate(noisy_image, kernel)
```

Task 2: Output and Analysis

Use your previously implemented functions to show the addition of salt & pepper noise to a sample image, as well as how your implemented filters can remove the noise.

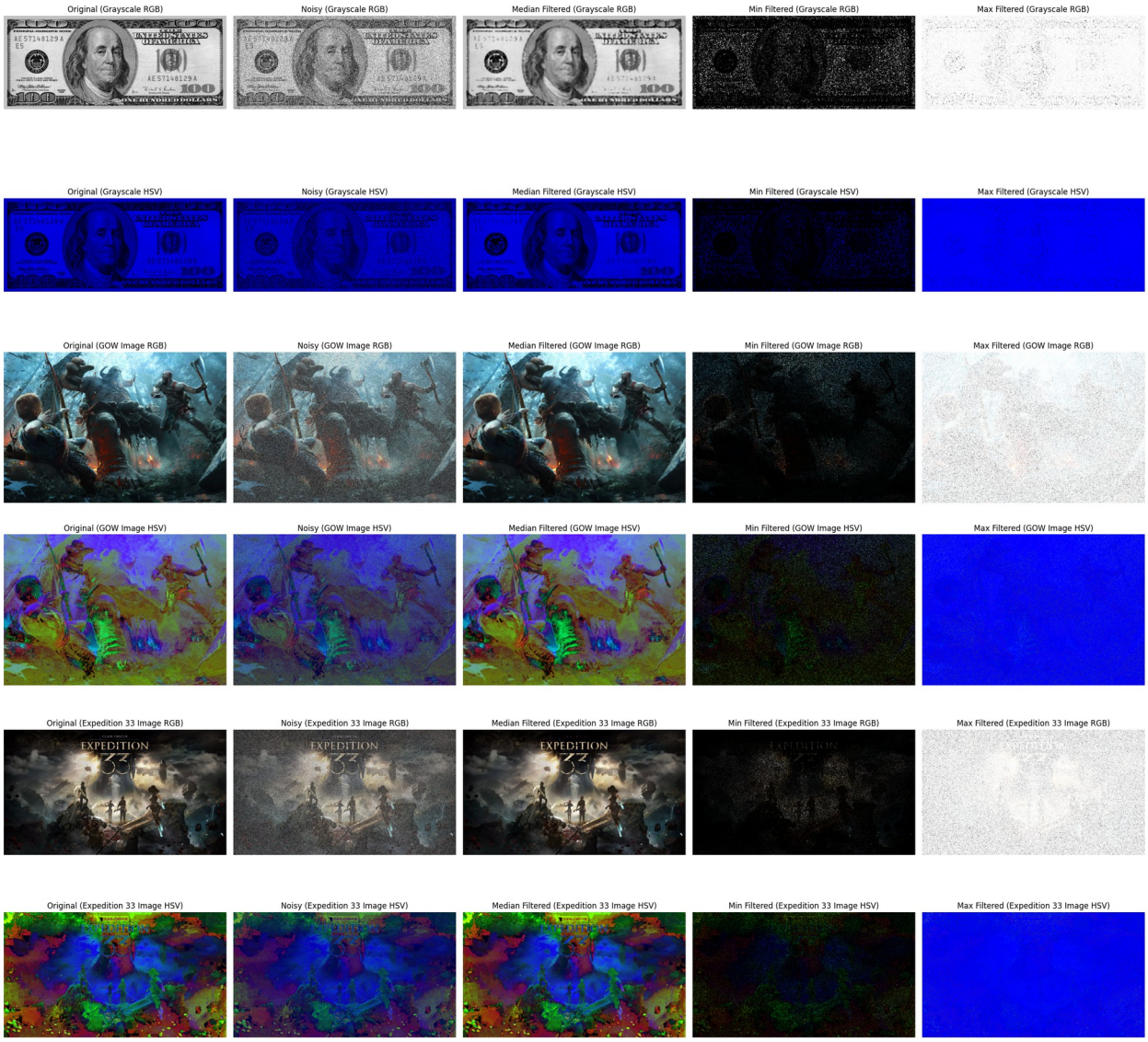
Show the results for both RGB and HSV color spaces.

Provide your analysis on the following topic: What happens when the noise level increases for a particular filter size?

```
#Generate output for Task 2 here

# --- Grayscale Image Processing ---
noisy_gray = salt_and_pepper(grayscale_image, 0.5)
median_corrected_gray = medianFilter(noisy_gray, 9)
min_corrected_gray = minFilter(noisy_gray, 3)
max_corrected_gray = maxFilter(noisy_gray, 3)
hsv_original_gray = cv2.cvtColor(grayscale_image, cv2.COLOR_RGB2HSV)
hsv_noisy_gray = cv2.cvtColor(noisy_gray, cv2.COLOR_RGB2HSV)
hsv_median_gray = cv2.cvtColor(median_corrected_gray,
cv2.COLOR_RGB2HSV)
hsv_min_gray = cv2.cvtColor(min_corrected_gray, cv2.COLOR_RGB2HSV)
hsv_max_gray = cv2.cvtColor(max_corrected_gray, cv2.COLOR_RGB2HSV)

# --- Color Image 1 Processing ---
noisy_color1 = salt_and_pepper(color_image, 0.5)
median_corrected_color1 = medianFilter(noisy_color1, 9)
min_corrected_color1 = minFilter(noisy_color1, 3)
max_corrected_color1 = maxFilter(noisy_color1, 3)
hsv_original_color1 = cv2.cvtColor(color_image, cv2.COLOR_RGB2HSV)
hsv_noisy_color1 = cv2.cvtColor(noisy_color1, cv2.COLOR_RGB2HSV)
hsv_median_color1 = cv2.cvtColor(median_corrected_color1,
```

```
cv2.COLOR_RGB2HSV)
hsv_min_color1 = cv2.cvtColor(min_corrected_color1, cv2.COLOR_RGB2HSV)
hsv_max_color1 = cv2.cvtColor(max_corrected_color1, cv2.COLOR_RGB2HSV)

# --- Color Image 2 Processing ---
# Check if color_image2 is loaded before processing
if color_image2 is not None:
    noisy_color2 = salt_and_pepper(color_image2, 0.5)
    median_corrected_color2 = medianFilter(noisy_color2, 9)
    min_corrected_color2 = minFilter(noisy_color2, 3)
    max_corrected_color2 = maxFilter(noisy_color2, 3)
    hsv_original_color2 = cv2.cvtColor(color_image2,
cv2.COLOR_RGB2HSV)
    hsv_noisy_color2 = cv2.cvtColor(noisy_color2, cv2.COLOR_RGB2HSV)
    hsv_median_color2 = cv2.cvtColor(median_corrected_color2,
cv2.COLOR_RGB2HSV)
    hsv_min_color2 = cv2.cvtColor(min_corrected_color2,
cv2.COLOR_RGB2HSV)
    hsv_max_color2 = cv2.cvtColor(max_corrected_color2,
cv2.COLOR_RGB2HSV)
else:
    # Create placeholder arrays or handle the error appropriately
    noisy_color2 = np.zeros_like(grayscale_image)
    median_corrected_color2 = np.zeros_like(grayscale_image)
    min_corrected_color2 = np.zeros_like(grayscale_image)
    max_corrected_color2 = np.zeros_like(grayscale_image)
    hsv_original_color2 = np.zeros_like(grayscale_image)
    hsv_noisy_color2 = np.zeros_like(grayscale_image)
    hsv_median_color2 = np.zeros_like(grayscale_image)
    hsv_min_color2 = np.zeros_like(grayscale_image)
    hsv_max_color2 = np.zeros_like(grayscale_image)

# Plot RGB and HSV versions for all images
fig, axes = plt.subplots(6, 5, figsize=(25, 24))

titles = ['Original', 'Noisy', 'Median Filtered', 'Min Filtered', 'Max
Filtered']

# Grayscale Image
images_rgb_gray = [grayscale_image, noisy_gray, median_corrected_gray,
min_corrected_gray, max_corrected_gray]
images_hsv_gray = [hsv_original_gray, hsv_noisy_gray, hsv_median_gray,
hsv_min_gray, hsv_max_gray]

for i, (img, title) in enumerate(zip(images_rgb_gray, titles)):
    axes[0, i].imshow(img)
    axes[0, i].set_title(title + ' (Grayscale RGB)')
    axes[0, i].axis('off')
for i, (img, title) in enumerate(zip(images_hsv_gray, titles)):
    axes[1, i].imshow(img)
```

```
    axes[1, i].set_title(title + ' (Grayscale HSV)')
    axes[1, i].axis('off')

# Color Image 1
images_rgb_color1 = [color_image, noisy_color1,
median_corrected_color1, min_corrected_color1, max_corrected_color1]
images_hsv_color1 = [hsv_original_color1, hsv_noisy_color1,
hsv_median_color1, hsv_min_color1, hsv_max_color1]

for i, (img, title) in enumerate(zip(images_rgb_color1, titles)):
    axes[2, i].imshow(img)
    axes[2, i].set_title(title + ' (GOW Image RGB)')
    axes[2, i].axis('off')
for i, (img, title) in enumerate(zip(images_hsv_color1, titles)):
    axes[3, i].imshow(img)
    axes[3, i].set_title(title + ' (GOW Image HSV)')
    axes[3, i].axis('off')

# Color Image 2
images_rgb_color2 = [color_image2, noisy_color2,
median_corrected_color2, min_corrected_color2, max_corrected_color2]
images_hsv_color2 = [hsv_original_color2, hsv_noisy_color2,
hsv_median_color2, hsv_min_color2, hsv_max_color2]

for i, (img, title) in enumerate(zip(images_rgb_color2, titles)):
    axes[4, i].imshow(img)
    axes[4, i].set_title(title + ' (Expedition 33 Image RGB)')
    axes[4, i].axis('off')
for i, (img, title) in enumerate(zip(images_hsv_color2, titles)):
    axes[5, i].imshow(img)
    axes[5, i].set_title(title + ' (Expedition 33 Image HSV)')
    axes[5, i].axis('off')

plt.tight_layout()
plt.show()
```



Analysis

1. Effect of Increasing Noise Level for a Fixed Filter Size

As the noise level increases for a given filter size, a larger proportion of image pixels are corrupted by salt (high-intensity) and pepper (low-intensity) noise. This directly impacts the effectiveness of spatial filters, as the probability of noisy pixels dominating the filter window increases.

- **Median Filter:**
  The median filter is highly effective for low to moderate levels of salt-and-pepper noise, as it replaces each pixel with the median value of its neighborhood, which is robust to outliers. However, at high noise densities (e.g., 50% or more), the likelihood that the majority of pixels within the window are noisy increases. In such cases, the median itself may correspond to a noisy value, leading to incomplete

noise removal, residual artifacts, and a patchy or blurred appearance. While it generally outperforms min and max filters under heavy noise, its effectiveness still degrades as noise density increases.

- **Min Filter:**
  The min filter replaces each pixel with the minimum intensity value in its neighborhood. It is particularly effective at removing salt noise (bright outliers). As noise density increases—especially pepper noise—the min filter tends to excessively darken the image. Bright regions are eroded, and pepper-like artifacts may spread, resulting in a loss of contrast and overall darkening.

- **Max Filter:**
  The max filter replaces each pixel with the maximum intensity value in its neighborhood and is therefore effective at removing pepper noise (dark outliers). With increasing noise levels—especially salt noise—the max filter causes excessive brightening. Dark regions are dilated, salt-like artifacts may propagate, and the image can appear overly bright and washed out.

---

2. Effect of Filter Size on Noise Reduction

Filter size (kernel dimensions such as 3×3, 5×5, or 9×9) plays a critical role in balancing noise reduction and detail preservation.

- **Small Filter Size (e.g., 3×3):**
  - **Noise Reduction:** Limited effectiveness for dense noise, as the small neighborhood may still contain a high proportion of corrupted pixels.
  - **Detail Preservation:** Better preservation of edges, textures, and fine details due to localized processing and minimal spatial smoothing.
- **Large Filter Size (e.g., 9×9 or 15×15):**
  - **Noise Reduction:** More effective at suppressing high-density noise, as the larger window increases the likelihood of including sufficient uncorrupted pixels.
  - **Detail Preservation:** Increased blurring and loss of fine details. Edges may become smeared, and the image can appear overly smooth or washed out due to excessive spatial aggregation.
- **Optimal Filter Size:**
  The ideal filter size depends on the noise density and the acceptable level of detail loss. Sparse noise typically requires only small kernels, while dense noise necessitates larger kernels. For median filtering, odd-sized kernels (e.g., 3, 5, 7, 9) are preferred to ensure a well-defined median value.

---

3. Differences Between RGB and HSV Color Spaces

Grayscale Images

For grayscale images, there is effectively no difference between RGB and HSV color spaces in the context of filtering. In RGB representation, all three channels (R, G, B) contain identical intensity values. In HSV representation, the Hue (H) and Saturation (S) channels are zero or undefined, and all intensity information resides in the Value (V) channel. Consequently, filtering the grayscale image directly or filtering the V channel in HSV yields equivalent results.

Color Images

For color images, the choice of color space significantly affects filtering behavior, especially for non-linear filters such as Median, Min, and Max.

- **RGB Color Space:**
  Filtering is applied independently to each of the R, G, and B channels. For example, a median filter computes separate medians for each channel. While this approach is straightforward, it can introduce color artifacts or shifts, particularly when noise affects channels unevenly. Because the filters operate directly on color components, perceived color consistency may be compromised.

- **HSV Color Space:**
  HSV (or HSI) offers a more perceptually meaningful separation of color and intensity. A common practice is to apply filtering primarily to the **Value (V)** or **Intensity (I)** channel, while leaving Hue (H) and Saturation (S) unchanged. This approach smooths brightness variations while preserving chromatic information, often producing more visually natural denoising results. However, filtering H or S channels—especially with non-linear filters—can lead to severe color distortions such as hue shifts or saturation loss, which are generally undesirable.

Task 3

For this task, you are required to design a Laplacian filter that will be used to compute edge responses. The function will return the edge response as an image, which will be called inside the sharpen function to generate a sharpened version of an image. The sharpen function contains a user defined parameter that can be used to control the level of sharpening.

```
def laplacian(image):
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    # Standard 4-neighbor Laplacian
    kernel = np.array([[0, -1, 0],
                       [-1, 4, -1],
                       [0, -1, 0]], dtype=np.float32)

    # Apply kernel using filter2D
    edge_response = cv2.filter2D(gray, -1, kernel)
```

```
    # Convert to float32 for further operations if needed
    edge_response = edge_response.astype(np.float32)

    return edge_response

def sharpen(image, sharpen_level):
    edges = laplacian(image)
    image_f = image.astype(np.float32)

    # Add edges scaled by sharpen_level to original image
    sharp_image = image_f + sharpen_level * edges[..., np.newaxis]  #
expand dims to apply to all channels

    # Clip and convert back to uint8
    sharp_image = np.clip(sharp_image, 0, 255).astype(np.uint8)

    return sharp_image
```

Task 3: Output and Analysis

Use your previously implemented laplacian function to show the edge response of a sample image. use an appropriate color image from Chapter 06.zip for this task. Next, show the sharpened image after using the sharpen function.

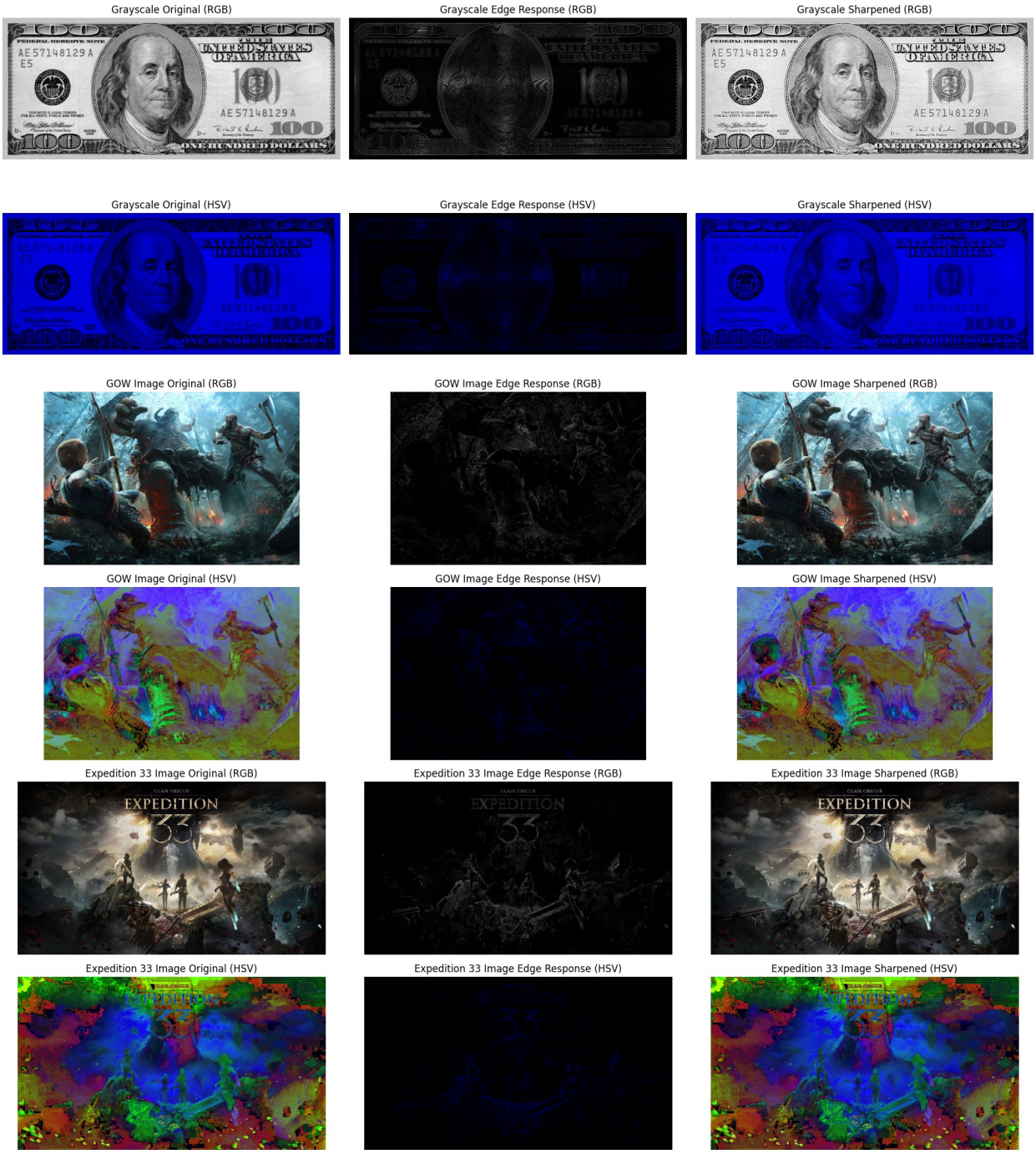
Show the results for both RGB and HSV color spaces.

Provide your analysis on the following topic: Why do we require intensity scaling after computing the sharpening? How does the value of sharpen_level affect the sharpening? Is there any differences observed between the results of RGB and HSV color spaces?

```
#Generate output for Task 3 here

# --- Grayscale Image Processing ---
edge_response_gray = laplacian(grayscale_image)
edge_uint8_gray = np.clip(edge_response_gray, 0, 255).astype(np.uint8)
sharp_img_gray = sharpen(grayscale_image, sharpen_level=1.5)

hsv_original_gray = cv2.cvtColor(grayscale_image, cv2.COLOR_RGB2HSV)
edge_hsv_gray = cv2.merge([edge_uint8_gray, edge_uint8_gray,
edge_uint8_gray])
edge_hsv_gray = cv2.cvtColor(edge_hsv_gray, cv2.COLOR_RGB2HSV)
sharp_hsv_gray = cv2.cvtColor(sharp_img_gray, cv2.COLOR_RGB2HSV)

# --- Color Image 1 Processing ---
edge_response_color1 = laplacian(color_image)
edge_uint8_color1 = np.clip(edge_response_color1, 0,
255).astype(np.uint8)
```

```
sharp_img_color1 = sharpen(color_image, sharpen_level=1.5)

hsv_original_color1 = cv2.cvtColor(color_image, cv2.COLOR_RGB2HSV)
edge_hsv_color1 = cv2.merge([edge_uint8_color1, edge_uint8_color1,
edge_uint8_color1])
edge_hsv_color1 = cv2.cvtColor(edge_hsv_color1, cv2.COLOR_RGB2HSV)
sharp_hsv_color1 = cv2.cvtColor(sharp_img_color1, cv2.COLOR_RGB2HSV)

# --- Color Image 2 Processing ---
# Check if color_image2 is loaded before processing
if color_image2 is not None:
    edge_response_color2 = laplacian(color_image2)
    edge_uint8_color2 = np.clip(edge_response_color2, 0,
255).astype(np.uint8)
    sharp_img_color2 = sharpen(color_image2, sharpen_level=1.5)

    hsv_original_color2 = cv2.cvtColor(color_image2,
cv2.COLOR_RGB2HSV)
    edge_hsv_color2 = cv2.merge([edge_uint8_color2, edge_uint8_color2,
edge_uint8_color2])
    edge_hsv_color2 = cv2.cvtColor(edge_hsv_color2, cv2.COLOR_RGB2HSV)
    sharp_hsv_color2 = cv2.cvtColor(sharp_img_color2,
cv2.COLOR_RGB2HSV)
else:
    edge_response_color2 = np.zeros_like(grayscale_image)
    edge_uint8_color2 = np.zeros_like(grayscale_image)
    sharp_img_color2 = np.zeros_like(grayscale_image)

    hsv_original_color2 = np.zeros_like(grayscale_image)
    edge_hsv_color2 = np.zeros_like(grayscale_image)
    sharp_hsv_color2 = np.zeros_like(grayscale_image)

# Plot RGB and HSV versions for all images
fig, axes = plt.subplots(6, 3, figsize=(18, 20))

# Grayscale Image - RGB row
axes[0,0].imshow(grayscale_image)
axes[0,0].set_title('Grayscale Original (RGB)')
axes[0,1].imshow(edge_uint8_gray, cmap='gray')
axes[0,1].set_title('Grayscale Edge Response (RGB)')
axes[0,2].imshow(sharp_img_gray)
axes[0,2].set_title('Grayscale Sharpened (RGB)')

# Grayscale Image - HSV row
axes[1,0].imshow(hsv_original_gray)
axes[1,0].set_title('Grayscale Original (HSV)')
axes[1,1].imshow(edge_hsv_gray)
axes[1,1].set_title('Grayscale Edge Response (HSV)')
axes[1,2].imshow(sharp_hsv_gray)
axes[1,2].set_title('Grayscale Sharpened (HSV)')
```

```
# Color Image 1 - RGB row
axes[2,0].imshow(color_image)
axes[2,0].set_title('GOW Image Original (RGB)')
axes[2,1].imshow(edge_uint8_color1, cmap='gray')
axes[2,1].set_title('GOW Image Edge Response (RGB)')
axes[2,2].imshow(sharp_img_color1)
axes[2,2].set_title('GOW Image Sharpened (RGB)')

# Color Image 1 - HSV row
axes[3,0].imshow(hsv_original_color1)
axes[3,0].set_title('GOW Image Original (HSV)')
axes[3,1].imshow(edge_hsv_color1)
axes[3,1].set_title('GOW Image Edge Response (HSV)')
axes[3,2].imshow(sharp_hsv_color1)
axes[3,2].set_title('GOW Image Sharpened (HSV)')

# Color Image 2 - RGB row
axes[4,0].imshow(color_image2)
axes[4,0].set_title('Expedition 33 Image Original (RGB)')
axes[4,1].imshow(edge_uint8_color2, cmap='gray')
axes[4,1].set_title('Expedition 33 Image Edge Response (RGB)')
axes[4,2].imshow(sharp_img_color2)
axes[4,2].set_title('Expedition 33 Image Sharpened (RGB)')

# Color Image 2 - HSV row
axes[5,0].imshow(hsv_original_color2)
axes[5,0].set_title('Expedition 33 Image Original (HSV)')
axes[5,1].imshow(edge_hsv_color2)
axes[5,1].set_title('Expedition 33 Image Edge Response (HSV)')
axes[5,2].imshow(sharp_hsv_color2)
axes[5,2].set_title('Expedition 33 Image Sharpened (HSV)')

for ax in axes.flatten():
    ax.axis('off')

plt.tight_layout()
plt.show()
```



Analysis

1. Why Intensity Scaling Is Required After Sharpening

Sharpening operations—particularly those based on Laplacian filtering—enhance high-frequency components by emphasizing intensity differences between neighboring pixels. The Laplacian computes a second-order derivative, which naturally produces both positive and negative values around edges and sharp transitions.

When this edge response is added back to the original image, the resulting pixel intensities may fall outside the valid range for an 8-bit image ([0, 255]).

- **Negative values:** Strong negative edge responses can reduce pixel intensities below 0.
- **Values above 255:** Strong positive edge responses can push intensities above the maximum representable value.

As a result, **intensity scaling via clipping** (e.g., `np.clip(..., 0, 255)`) is essential to ensure numerical validity and proper visualization. Without clipping, out-of-range values would either wrap around, be truncated, or produce display artifacts when cast to `uint8`. Thus, intensity scaling is not a cosmetic step but a necessary safeguard to maintain visual correctness after sharpening.

---

2. Effect of `sharpen_level` on the Sharpening Strength

The `sharpen_level` parameter directly controls the contribution of the edge information to the final sharpened image. It acts as a gain factor applied to the Laplacian response before it is added to the original image.

- **`sharpen_level = 0`:**
  No sharpening occurs. The edge term is nullified, and the output is identical to the original image.
- **`sharpen_level = 1`:**
  A standard level of sharpening, where the full edge response is added back. This typically enhances edges and fine details without introducing severe artifacts.
- **`sharpen_level > 1`:**
  Produces stronger sharpening by amplifying edge magnitudes. While this enhances details more aggressively, it can also introduce undesirable artifacts such as halos around edges, exaggerated contrast, or amplified noise.
- **`0 < sharpen_level < 1`:**
  Results in a milder sharpening effect, offering subtle edge enhancement while minimizing the risk of artifacts.

In essence, `sharpen_level` functions as a control knob for edge enhancement strength: higher values increase sharpness but also the likelihood of artifacts, while lower values provide more conservative sharpening.

---

3. Differences Between RGB and HSV Color Spaces

Grayscale Images

For grayscale images, the sharpening process is inherently intensity-based. In the provided implementation, the Laplacian is computed after converting the image to grayscale, meaning

edge detection operates on a single luminance channel. When the sharpened result is later visualized in HSV, the Hue (H) and Saturation (S) channels remain zero or undefined, and the Value (V) channel represents the intensity.

Therefore, there is no fundamental difference in the sharpening outcome when comparing RGB and HSV *representations* of grayscale images—the operation itself is purely intensity-driven.

Color Images

For color images, the current implementation follows a perceptually motivated approach:

- The input RGB image is first converted to grayscale to compute the Laplacian (edge map).
- This single-channel edge map is then expanded and added uniformly to all three RGB channels.

This strategy effectively performs **luminance-based sharpening**, ensuring that edges are enhanced according to overall brightness variations rather than per-channel color differences.

If sharpening were instead applied independently to the R, G, and B channels, mismatches between channel-specific edges could lead to color shifts, unnatural highlights, or chromatic artifacts. By basing sharpening on grayscale luminance, such issues are largely avoided.

When the RGB-sharpened image is converted to HSV for visualization, the sharpening effect primarily manifests in the **Value (V)** channel, which corresponds to perceived brightness. The Hue (H) and Saturation (S) channels typically remain largely unchanged, aside from minor indirect effects caused by changes in intensity. This behavior aligns well with human visual perception and results in more natural-looking sharpened color images.

Task 4

For this task, you are required to design a function that takes two parameters: an image and a value k. The function should perform Unsharp masking ($k=1$) and High-boost filtering ($k>1$) and return a filtered image.

```
def high_boost_filter(image, k):
    img_f = image.astype(np.float32)

    # Apply Gaussian blur to get smooth version
    blurred = cv2.GaussianBlur(img_f, (5,5), sigmaX=1.0)

    # Compute the mask (difference between original and blurred)
    unsharp_mask = img_f - blurred

    # Add scaled mask to original
    filtered_image = img_f + k * unsharp_mask

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

```
    return filtered_image
```

Task 4: Output and Analysis

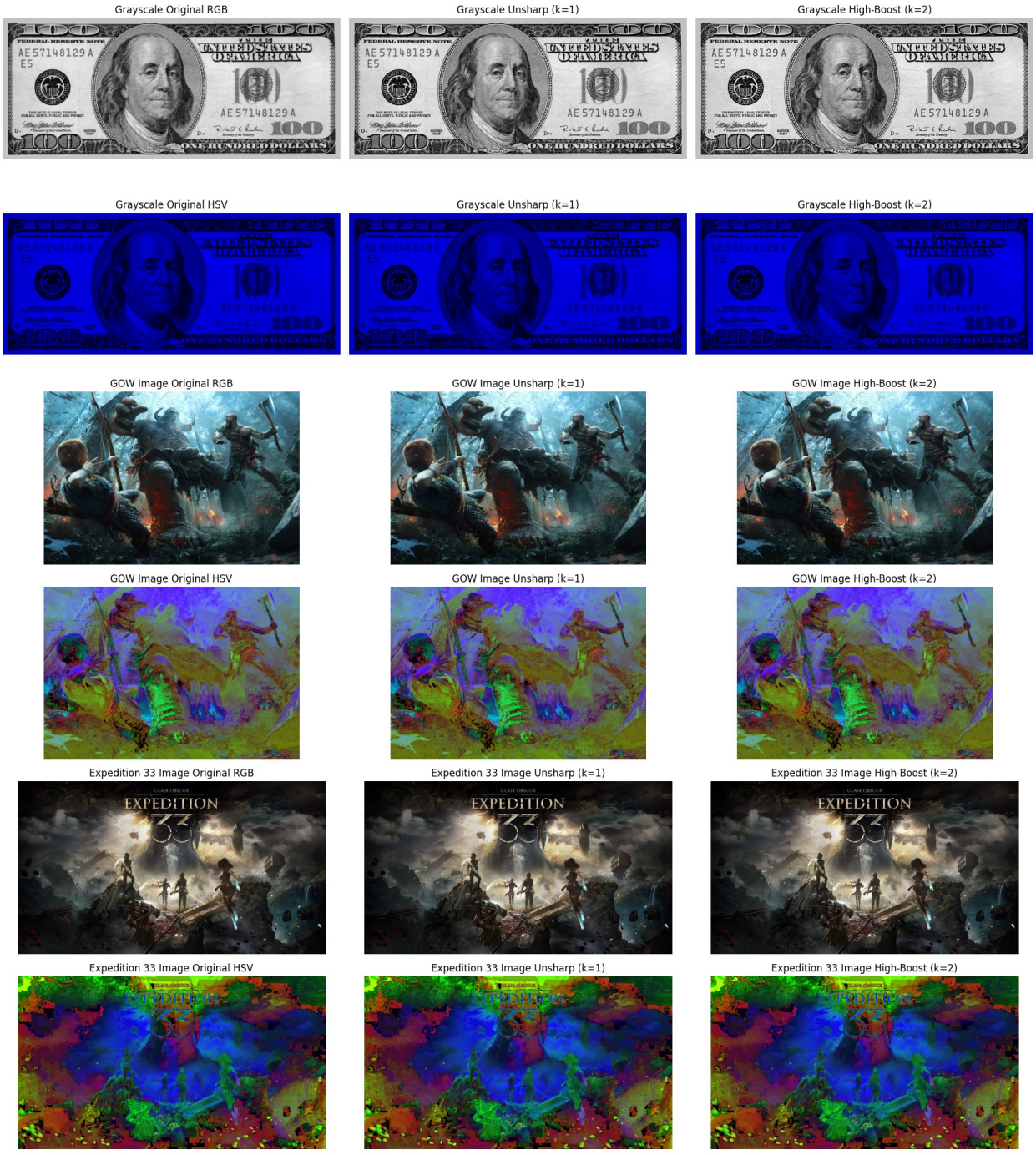
Use your previously implemented high_boost_filter function to show the output of Unsharp Masking and High-boost filtering on an RGB image. Next, show both outputs for the HSI color space.

Provide your analysis on the following topic: Is there any difference in applying unsharp masking or high-boost filtering for RGB and HSI color space? What happens if $k < 1$?

```
# --- Grayscale Image Processing ---
rgb_unsharp_gray = high_boost_filter(grayscale_image, k=1)
rgb_highboost_gray = high_boost_filter(grayscale_image, k=2)
hsv_image_gray = cv2.cvtColor(grayscale_image, cv2.COLOR_RGB2HSV)
h_gray, s_gray, v_gray = cv2.split(hsv_image_gray)
v_unsharp_gray = high_boost_filter(v_gray, k=1)
v_highboost_gray = high_boost_filter(v_gray, k=2)
hsv_unsharp_gray = cv2.merge([h_gray, s_gray, v_unsharp_gray])
hsv_highboost_gray = cv2.merge([h_gray, s_gray, v_highboost_gray])

# --- Color Image 1 Processing ---
rgb_unsharp_color1 = high_boost_filter(color_image, k=1)
rgb_highboost_color1 = high_boost_filter(color_image, k=2)
hsv_image_color1 = cv2.cvtColor(color_image, cv2.COLOR_RGB2HSV)
h_color1, s_color1, v_color1 = cv2.split(hsv_image_color1)
v_unsharp_color1 = high_boost_filter(v_color1, k=1)
v_highboost_color1 = high_boost_filter(v_color1, k=2)
hsv_unsharp_color1 = cv2.merge([h_color1, s_color1, v_unsharp_color1])
hsv_highboost_color1 = cv2.merge([h_color1, s_color1,
v_highboost_color1])

# --- Color Image 2 Processing ---
# Check if color_image2 is loaded before processing
if color_image2 is not None:
    rgb_unsharp_color2 = high_boost_filter(color_image2, k=1)
    rgb_highboost_color2 = high_boost_filter(color_image2, k=2)
    hsv_image_color2 = cv2.cvtColor(color_image2, cv2.COLOR_RGB2HSV)
    h_color2, s_color2, v_color2 = cv2.split(hsv_image_color2)
    v_unsharp_color2 = high_boost_filter(v_color2, k=1)
    v_highboost_color2 = high_boost_filter(v_color2, k=2)
    hsv_unsharp_color2 = cv2.merge([h_color2, s_color2,
v_unsharp_color2])
    hsv_highboost_color2 = cv2.merge([h_color2, s_color2,
v_highboost_color2])
else:
    rgb_unsharp_color2 = np.zeros_like(grayscale_image)
```

```
    rgb_highboost_color2 = np.zeros_like(grayscale_image)
    hsv_image_color2 = np.zeros_like(grayscale_image)
    h_color2, s_color2, v_color2 =
cv2.split(np.zeros_like(grayscale_image))
    v_unsharp_color2 = np.zeros_like(v_gray)
    v_highboost_color2 = np.zeros_like(v_gray)
    hsv_unsharp_color2 = np.zeros_like(grayscale_image)
    hsv_highboost_color2 = np.zeros_like(grayscale_image)

# ---- Plotting ----
fig, axes = plt.subplots(6, 3, figsize=(18,20))

# Grayscale Image - RGB row
axes[0,0].imshow(grayscale_image)
axes[0,0].set_title("Grayscale Original RGB")
axes[0,1].imshow(rgb_unsharp_gray)
axes[0,1].set_title("Grayscale Unsharp (k=1)")
axes[0,2].imshow(rgb_highboost_gray)
axes[0,2].set_title("Grayscale High-Boost (k=2)")

# Grayscale Image - HSV row
axes[1,0].imshow(hsv_image_gray)
axes[1,0].set_title("Grayscale Original HSV")
axes[1,1].imshow(hsv_unsharp_gray)
axes[1,1].set_title("Grayscale Unsharp (k=1)")
axes[1,2].imshow(hsv_highboost_gray)
axes[1,2].set_title("Grayscale High-Boost (k=2)")

# Color Image 1 - RGB row
axes[2,0].imshow(color_image)
axes[2,0].set_title("GOW Image Original RGB")
axes[2,1].imshow(rgb_unsharp_color1)
axes[2,1].set_title("GOW Image Unsharp (k=1)")
axes[2,2].imshow(rgb_highboost_color1)
axes[2,2].set_title("GOW Image High-Boost (k=2)")

# Color Image 1 - HSV row
axes[3,0].imshow(hsv_image_color1)
axes[3,0].set_title("GOW Image Original HSV")
axes[3,1].imshow(hsv_unsharp_color1)
axes[3,1].set_title("GOW Image Unsharp (k=1)")
axes[3,2].imshow(hsv_highboost_color1)
axes[3,2].set_title("GOW Image High-Boost (k=2)")

# Color Image 2 - RGB row
axes[4,0].imshow(color_image2)
axes[4,0].set_title("Expedition 33 Image Original RGB")
axes[4,1].imshow(rgb_unsharp_color2)
axes[4,1].set_title("Expedition 33 Image Unsharp (k=1)")
axes[4,2].imshow(rgb_highboost_color2)
```

```
axes[4,2].set_title("Expedition 33 Image High-Boost (k=2)")

# Color Image 2 - HSV row
axes[5,0].imshow(hsv_image_color2)
axes[5,0].set_title("Expedition 33 Image Original HSV")
axes[5,1].imshow(hsv_unsharp_color2)
axes[5,1].set_title("Expedition 33 Image Unsharp (k=1)")
axes[5,2].imshow(hsv_highboost_color2)
axes[5,2].set_title("Expedition 33 Image High-Boost (k=2)")

for ax in axes.flatten():
    ax.axis('off')

plt.tight_layout()
plt.show()
```



Analysis

1. Differences in Applying Unsharp Masking or High-Boost Filtering in RGB and HSV Color Spaces

When applying unsharp masking or high-boost filtering, the choice of color space has a significant impact on the visual outcome, especially for color images.

RGB Color Space

In the RGB color space, unsharp masking or high-boost filtering is typically applied independently to each of the Red, Green, and Blue channels. While this approach is simple to implement, it has several important implications:

- **Potential Color Shifts:**
  Since each channel is sharpened separately, edges that differ across channels may be enhanced unevenly. This can introduce color fringing or subtle hue shifts, particularly around high-contrast edges.

- **Luminance–Chrominance Coupling:**
  In RGB, brightness and color information are intertwined. Sharpening all channels equally does not distinguish between luminance and chrominance variations, even though the human visual system is far more sensitive to changes in luminance. As a result, RGB-based sharpening may enhance color noise or produce less perceptually natural results.

HSV Color Space

The HSV color space explicitly separates intensity information from color information, making it well-suited for perceptually guided sharpening.

- **Targeted Sharpening on the V Channel:**
  A common and preferred strategy is to apply unsharp masking or high-boost filtering **only to the Value (V)** channel, while keeping Hue (H) and Saturation (S) unchanged. This approach offers several advantages:
  - **Preservation of Color Integrity:** Enhancing only the V channel increases local brightness contrast without altering hue or saturation, avoiding unwanted color shifts.
  - **Perceptual Alignment:** Human vision primarily relies on luminance contrast for edge perception. Sharpening the V channel therefore enhances edges in a way that aligns well with visual perception.
- **Filtering H or S Channels:**
  Applying sharpening to the Hue or Saturation channels is generally avoided, as it can lead to abrupt hue transitions, exaggerated colors, or desaturation artifacts. Such operations are usually reserved for specialized or artistic effects rather than standard image enhancement.

Observation from the Results:
The results demonstrate that direct RGB sharpening leads to edge enhancement across all color components, which may subtly alter color appearance. In contrast, sharpening applied only to the V channel in HSV space primarily increases local contrast and detail while preserving the original color palette. This typically yields a more natural and visually pleasing sharpened image.

---

2. Effect of the High-Boost Parameter (k)

In high-boost filtering, the output image is computed as:

[filtered_image = img_f + k \cdot (img_f - blurred)]

which can be rewritten as:

[filtered_image = (1 + k),img_f - k,blurred.]

The parameter (k) controls the strength and nature of the filtering:

- **(k = 1) (Unsharp Masking):**
  The original image is enhanced by adding the full unsharp mask. This produces a standard sharpening effect with noticeable edge enhancement.

- **(k > 1) (High-Boost Filtering):**
  The unsharp mask is amplified before being added back. This results in stronger edge enhancement and increased detail visibility, but also raises the risk of artifacts such as halos and amplified noise.

- **(0 < k < 1) (Mild Sharpening):**
  The contribution of the unsharp mask is reduced, leading to a subtle sharpening effect. Edges are enhanced gently, making this range suitable when only minor detail enhancement is desired with minimal artifacts.

- **(k = 0):**
  No sharpening occurs. The output image is identical to the original input.

- **(k < 0) (Smoothing Effect):**
  Negative values of (k) invert the role of the unsharp mask, effectively blending the original image with its blurred version. This produces a smoothing or blurring effect rather than sharpening and is generally not used for enhancement purposes.