```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

def joint_bilt_filter(D, C, w=3, sigma=(3, 0.1)):
    """
    2-D Joint bilateral filtering for grayscale images.

    Args:
        D (numpy.ndarray): Input grayscale image.
        C (numpy.ndarray): Guiding grayscale image.
        w (int): Half-size of the Gaussian bilateral filter window.
        sigma (tuple): Standard deviations for spatial and intensity
domains.

    Returns:
        numpy.ndarray: Filtered image.
    """
    # Validate input image D
    if D is None or not isinstance(D, np.ndarray) or D.dtype !=
np.float64 or D.min() < 0 or D.max() > 1:
        raise ValueError("Input image D must be a double precision
matrix of size NxM on the closed interval [0,1].")

    # Validate guiding image C
    if C is None or not isinstance(C, np.ndarray) or C.dtype !=
np.float64 or C.min() < 0 or C.max() > 1:
        raise ValueError("Input image C must be a double precision
matrix of size NxM on the closed interval [0,1].")

    w = int(w)  # Ensure window size is an integer
    sigma_d, sigma_r = sigma  # Spatial and range standard deviations

    # Initialize output image
    final = np.zeros_like(D)

    # Create a spatial Gaussian filter
    X, Y = np.meshgrid(np.arange(-w, w + 1), np.arange(-w, w + 1))
    G = np.exp(-(X**2 + Y**2) / (2 * sigma_d**2))

    dim = D.shape
    a = 0

    # Apply bilateral filter to each pixel
    while a < dim[0]:
        b = 0
        while b < dim[1]:
            # Define the local window
            iMin = max(a - w, 0)
```

```
            iMax = min(a + w, dim[0] - 1)
            jMin = max(b - w, 0)
            jMax = min(b + w, dim[1] - 1)

            # Extract local regions from D and C
            I = D[iMin:iMax + 1, jMin:jMax + 1]
            J = C[iMin:iMax + 1, jMin:jMax + 1]

            # Compute range kernel based on intensity differences
            H = np.exp(-((J - C[a, b])**2) / (2 * sigma_r**2))

            # Combine spatial and range kernels
            F = H * G[iMin - a + w:iMax - a + w + 1, jMin - b + w:jMax
- b + w + 1]

            # Compute the filtered pixel value
            final[a, b] = np.sum(F * I) / np.sum(F)
            b += 1
        a += 1

    return final


def joint_bil_2_color(N, F, w=3, sigma=(3, 0.1)):
    """
    Apply joint bilateral filter to each color channel of the image.

    Args:
        N (numpy.ndarray): Input color image.
        F (numpy.ndarray): Guiding color image.
        w (int): Half-size of the Gaussian bilateral filter window.
        sigma (tuple): Standard deviations for spatial and intensity
domains.

    Returns:
        numpy.ndarray: Filtered color image.
    """
    # Initialize output image
    B = np.zeros_like(N)

    # Apply the joint bilateral filter to each color channel
    for channel in range(N.shape[2]):
        B[:, :, channel] = joint_bilt_filter(N[:, :, channel], F[:, :,
channel], w, sigma)

    return B


def solution(image_path_a, image_path_b):
    """
```

```
    Process images with joint bilateral filtering to reduce noise and
enhance details.

    Args:
        image_path_a (str): Path to the no-flash image.
        image_path_b (str): Path to the flash image.

    Returns:
        numpy.ndarray: Processed image with enhanced details.
    """
    # Load images and normalize to [0, 1]
    path1 = image_path_a
    path2 = image_path_b

    input_image1 = cv2.imread(path1) / 255.0  # No-flash image
    input_image2 = cv2.imread(path2) / 255.0  # Flash image

    image_N = np.copy(input_image1)
    image_F = np.copy(input_image2)

    # Apply joint bilateral filtering
    current_time1 = datetime.now()
    print("Calculating A_Base Image....")
    A_base = joint_bil_2_color(image_N, image_N, w=11, sigma=(3, 0.2))


    current_time2 = datetime.now()
    print('Implimented A_Base calculation in:',current_time2-
current_time1)

    print("Calculating Image_NR (Noise reduced version of Image by
BLF)....")
    A_nr = joint_bil_2_color(image_N, image_F, w=11, sigma=(3, 0.1))

    current_time3 = datetime.now()
    print('Implimented A_nr calculation in:',current_time3-
current_time2)

    print("Calculating Flash Image Base.....")
    F_base = joint_bil_2_color(image_F, image_F, w=11, sigma=(3, 0.1))


    current_time4 = datetime.now()
    print('Implimented F_base calculation in:',current_time4-
current_time3)

    eps = 0.02
    print("Almost Done....!!")
    F_detail = (image_F.astype(np.float64) + eps) /
(F_base.astype(np.float64) + eps)
```

```
    print('Finished!')
    print('Total Time Taken:',current_time4-current_time1)

    # Mask creation
    eps = 0.02
    T = -50  # Set your threshold value here

    # Load images for mask creation
    i1 = cv2.imread(path1)
    i2 = cv2.imread(path2)

    # Convert images to grayscale if necessary
    if len(i1.shape) > 2:
        gA = cv2.cvtColor(i1, cv2.COLOR_BGR2GRAY).astype(np.float64)
        gF = cv2.cvtColor(i2, cv2.COLOR_BGR2GRAY).astype(np.float64)
    else:
        gA = i1.astype(np.float64)
        gF = i2.astype(np.float64)

    # Compute the difference between flash and no-flash images
    diff = gF - gA
    mf = np.zeros_like(diff)  # Initialize mask for shadow detection
    ms = np.zeros_like(diff)  # Initialize mask for specularities

    # Detect shadows based on intensity difference
    mf[diff <= T] = 1

    # Detect specularities based on intensity threshold
    ms[gF / np.max(gF) > 0.95] = 1

    # Initialize the final mask
    M = np.zeros_like(i1)
    se = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (4, 4))  #
Structuring element for dilation

    # Combine shadow and specularities masks and dilate
    for i in range(i1.shape[2]):  # Build the flash mask
        m = np.logical_or(mf, ms).astype(np.uint8)  # Merge two masks
        m = m * 255
        M[:, :, i] = cv2.dilate(m, se)

    M = M / 255  # Normalize mask

    # Combine the filtered images using the mask
    out = ((1 - M) * A_nr * F_detail + M * A_base)

    return out

# Example usage:
```

```python
path_to_no_flash_image = '4_a.jpg'
path_to_flash_image = '4_b.jpg'

Clear_image = solution(path_to_no_flash_image, path_to_flash_image)

# Convert Clear_image to uint8 format (range [0, 255])
Clear_image_uint8 = (Clear_image * 255).astype(np.uint8)

# Read the original images
original_flash_image = cv2.imread(path_to_no_flash_image)
original_noflash_image = cv2.imread(path_to_flash_image)

# Convert the images from BGR to RGB format
image1_rgb = cv2.cvtColor(Clear_image_uint8, cv2.COLOR_BGR2RGB)
image2_rgb = cv2.cvtColor(original_flash_image, cv2.COLOR_BGR2RGB)
image3_rgb = cv2.cvtColor(original_noflash_image, cv2.COLOR_BGR2RGB)

# Create a figure with custom GridSpec
fig = plt.figure(figsize=(12, 6))
gs = GridSpec(2, 3, figure=fig)

# Display the first image with a larger area
ax1 = fig.add_subplot(gs[:, :2])  # Span 2 rows and 2 columns
ax1.imshow(image1_rgb)
ax1.set_title('BL Filtered Image')
ax1.axis('off')  # Hide the axis

# Display the second and third images
ax2 = fig.add_subplot(gs[0, 2])  # Occupy top-right space
ax2.imshow(image2_rgb)
ax2.set_title('Original No Flash Image')
ax2.axis('off')  # Hide the axis

ax3 = fig.add_subplot(gs[1, 2])  # Occupy bottom-right space
ax3.imshow(image3_rgb)
ax3.set_title('Original Flash Image')
ax3.axis('off')  # Hide the axis

# Display the images
plt.show()
```

```
Calculating A_Base Image....
Implimented A_Base calculation in: 0:00:42.860440
Calculating Image_NR (Noise reduced version of Image by BLF)....
Implimented A_nr calculation in: 0:00:39.161237
Calculating Flash Image Base.....
Implimented F_base calculation in: 0:00:38.428852
Almost Done....!!
Finished!
Total Time Taken: 0:02:00.450529
```

BL Filtered Image

Original No Flash Image

Original Flash Image