Student Name: Anumanchi Agastya Sai Ram Likhit
Student ID: 20048

Computer Vision
Assignment 2

1 Question 1

Compute (i) forward, (ii) backward, and (iii) central gradient image. Using the gradient compute the gradient magnitude image and gradient direction image. Analyze the difference you observed in the magnitude and direction images.

Answer 1

For the forward difference, the gradient in the x and y directions is computed as follows:

$$G_x(i,j) = I(i,j+1) - I(i,j)$$
$$G_y(i,j) = I(i+1,j) - I(i,j)$$

For the backward difference, the gradient in the x and y directions is computed as:

$$G_x(i,j) = I(i,j) - I(i,j-1)$$
$$G_y(i,j) = I(i,j) - I(i-1,j)$$

The central difference method averages the forward and backward differences:

$$G_x(i,j) = \frac{I(i,j+1) - I(i,j-1)}{2}$$
$$G_y(i,j) = \frac{I(i+1,j) - I(i-1,j)}{2}$$

1.1 Forward Difference

As mentioned above we now develop the python code to compute the forward difference

```python
# Convert image_array to a data type that can handle negative numbers
image_array = image_array.astype('int16')

# Initialize gradient images with zeros for Forward Gradient
rows, cols = image_array.shape
forward_gradient_x = np.zeros_like(image_array, dtype='int16')
forward_gradient_y = np.zeros_like(image_array, dtype='int16')

# Compute Forward Gradient
for i in range(rows):
    for j in range(cols - 1):
```

```
        forward_gradient_x[i, j] = image_array[i, j + 1] - image_array[i, j]
for i in range(rows - 1):
    for j in range(cols):
        forward_gradient_y[i, j] = image_array[i + 1, j] - image_array[i, j]
```

1.2 Backward Difference

As mentioned above we now develop the python code to compute the backward difference

```
# Initialize gradient images with zeros for Backward Gradient
backward_gradient_x = np.zeros_like(image_array, dtype='int16')
backward_gradient_y = np.zeros_like(image_array, dtype='int16')

# Compute Backward Gradient
for i in range(rows):
    for j in range(1, cols):
        backward_gradient_x[i, j] = image_array[i, j] - image_array[i, j - 1]
for i in range(1, rows):
    for j in range(cols):
        backward_gradient_y[i, j] = image_array[i, j] - image_array[i - 1, j]
```

1.3 Central Difference

As mentioned above we now develop the python code to compute the central difference

```
# Initialize gradient images with zeros for Central Gradient
central_gradient_x = np.zeros_like(image_array, dtype='int16')
central_gradient_y = np.zeros_like(image_array, dtype='int16')

# Compute Central Gradient
for i in range(1, rows - 1):
    for j in range(1, cols - 1):
        central_gradient_x[i, j] = (image_array[i, j + 1] - image_array[i, j - 1]) // 2
        central_gradient_y[i, j] = (image_array[i + 1, j] - image_array[i - 1, j]) // 2
```

You can see that we have used $dtype =' int16'$ that is to convert the data type of the image array to signed 16-bit integers. This is done to handle negative values that can occur when calculating gradients using differences between pixel intensities.

In the original image array, the data type is usually uint8, which supports unsigned integers ranging from 0 to 255. When calculating gradients, the difference between adjacent pixel values can be negative, and uint8 cannot represent negative numbers. This leads to overflow issues. Switching to int16 allows the array to store negative numbers, thus avoiding any overflow and accurately representing the gradients.

1.4 Results

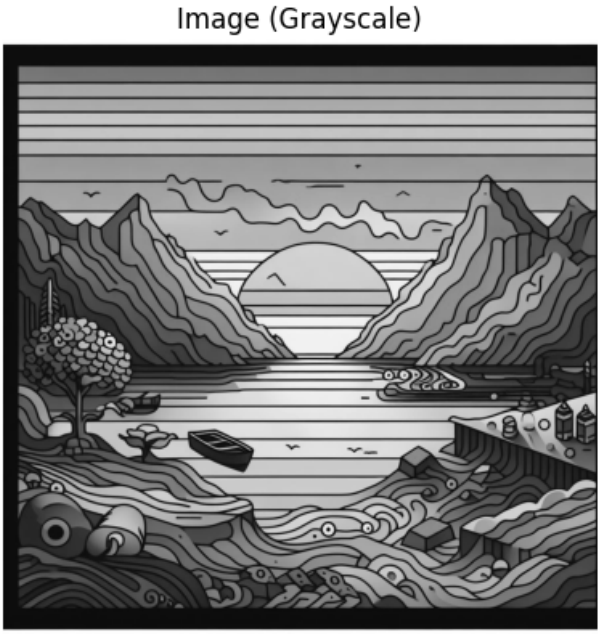

Figure 1: Orginal Image

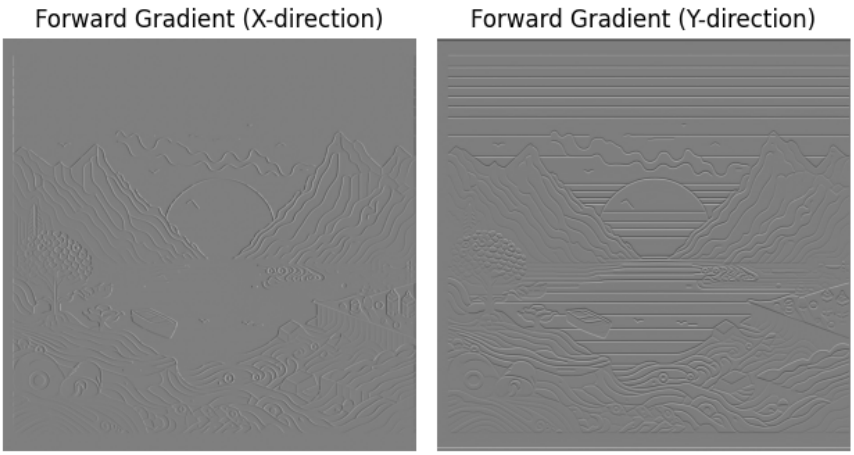

Figure 2: Forward Gradient in X and Y Direction

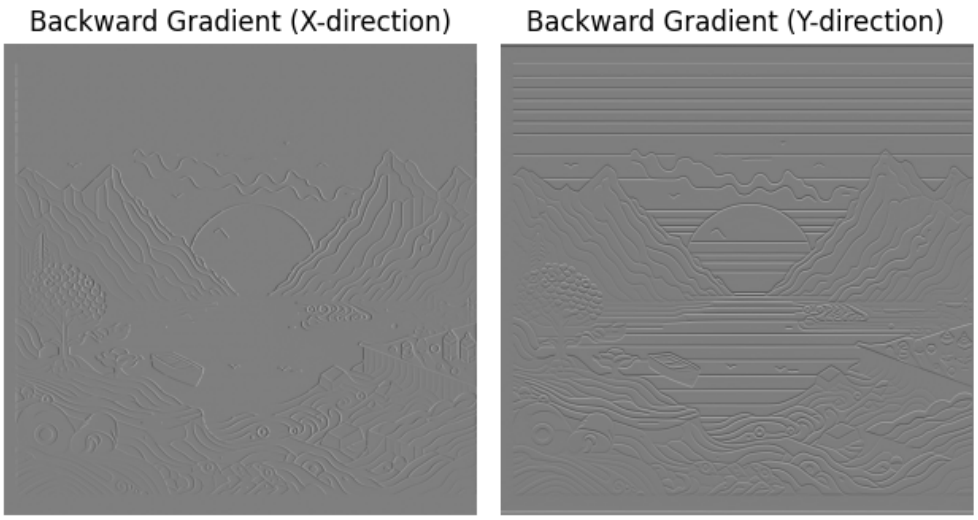

Figure 3: Backward Gradient in X and Y Direction

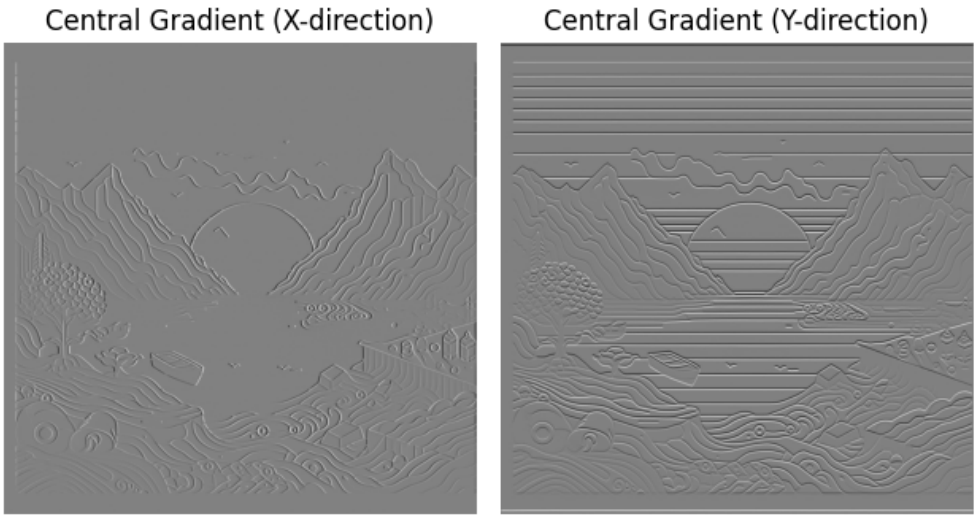

Figure 4: Central Gradient in X and Y Direction

The gradient magnitude M and direction θ are calculated as:

$$M(i,j) = \sqrt{G_x^2(i,j) + G_y^2(i,j)}$$
$$\theta(i,j) = \arctan 2(G_y(i,j), G_x(i,j))$$

Normally taking the $M(i,j) = \sqrt{G_x^2(i,j) + G_y^2(i,j)}$ is showing an warning that is due to overflow in the data type being used. The int16 data type can handle values from -32,768 to 32,767. When you square the values, they can easily exceed this range.
To avoid this, we can convert the gradient arrays to a floating-point data type (like float32 or float64) before performing the squaring and square root operations.This will prevent overflow issues.

Farward

```
# Convert the gradients to float32 to prevent overflow
forward_gradient_x_float = forward_gradient_x.astype('float32')
forward_gradient_y_float = forward_gradient_y.astype('float32')

# Compute the gradient magnitude
gradient_magnitude_forward = np.sqrt(forward_gradient_x_float**2 +
    forward_gradient_y_float**2)

# Compute the gradient direction
gradient_direction_forward = np.arctan2(forward_gradient_y_float,
    forward_gradient_x_float) * 180 / np.pi

gradient_magnitude_forward, gradient_direction_forward
```

Backward

```
# Convert the gradients to float32 to prevent overflow
backward_gradient_x_float = backward_gradient_x.astype('float32')
backward_gradient_y_float = backward_gradient_y.astype('float32')

# Compute the gradient magnitude
gradient_magnitude_backward = np.sqrt(backward_gradient_x_float**2 +
    backward_gradient_y_float**2)

# Compute the gradient direction
gradient_direction_backward = np.arctan2(backward_gradient_y_float,
    backward_gradient_x_float) * 180 / np.pi

gradient_magnitude_backward, gradient_direction_forward
gradient_magnitude_central, gradient_direction_central
```

Central

```
# Convert the gradients to float32 to prevent overflow
central_gradient_x_float = central_gradient_x.astype('float32')
central_gradient_y_float = central_gradient_y.astype('float32')

# Compute the gradient magnitude
gradient_magnitude_central = np.sqrt(central_gradient_x_float**2 +
    central_gradient_y_float**2)

# Compute the gradient direction
gradient_direction_central = np.arctan2(central_gradient_y_float,
    central_gradient_x_float) * 180 / np.pi

gradient_magnitude_central, gradient_direction_central
```

Figure 5: Gradient Magnitude and Gradient Direction in Forward Difference

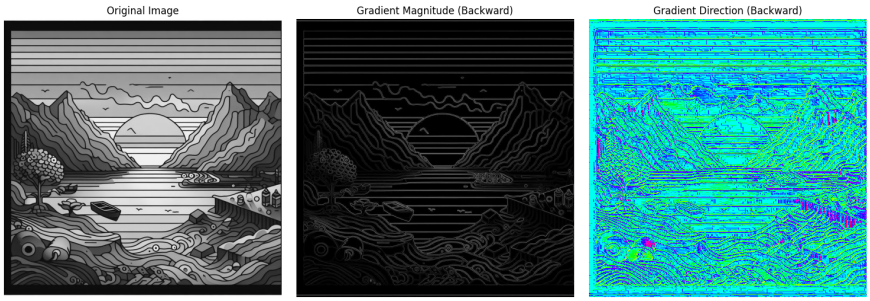

Figure 6: Gradient Magnitude and Gradient Direction in Backward Difference

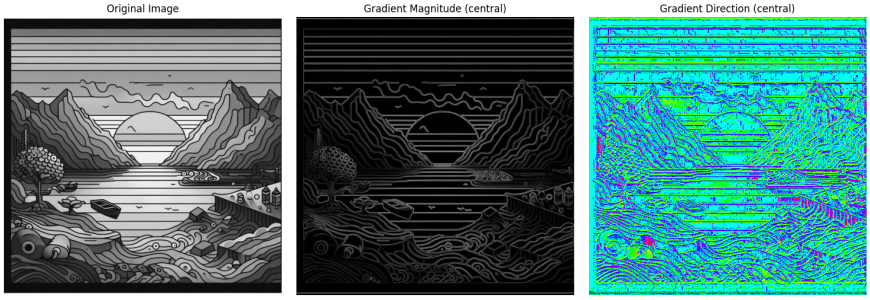

Figure 7: Gradient Magnitude and Gradient Direction in Central Difference

1.5 Anaylsis

Gradient Magnitude Image : In the gradient magnitude image we can see the edges or transition regions,the brighter regions correspond to higher magnitude which is directly proportional to the intensity of edges.where as the darker regions have indicate little to no change in intensity values, meaning uniform regions in the original image.The magnitude image is useful for identifying areas of interest in the original image, such as boundaries, outlines, or significant features.

Gradient Direction Image : We can see that the colors in the gradient direction image represent the orientation of the gradient at each pixel. This is an indication of the local orientation of edges or any features.some areas might appear to have random or mixed hues, indicating regions where the edge direction varies rapidly or where there is noise above in the images we can see such random hues,and the direction image is valuable for understanding the structure and orientation of the features within the image.

2 Question 2

Compute the LoG, DoG, and Gaussian pyramid scale space of the image

Answer 2

Here we will be using the same image which we used for the above question to compute the Laplacian of Gaussian , Difference of Gaussian and Gaussian pyramid scale space of the image.

2.1 Laplacian of Gaussian (LoG) :

The Laplacian of Gaussian (LoG) is a combination of two separate processes:

1. Gaussian Smoothing: This involves convolving the image with a Gaussian filter.
2. Laplacian Filtering: This is the application of a Laplacian filter to the image, which highlights regions of rapid intensity change.

The equation for the LoG for a 2D function is given by:

$$LoG(x, y, \sigma) = -\frac{1}{\sqrt{2\pi}\sigma^3}\left(2 - \frac{x^2+y^2}{\sigma^2}\right)e^{-\frac{x^2+y^2}{2\sigma^2}}$$

`The liberty is given to use the builtin functions for LOG and DOG with the understanding of the function.`

```
import cv2
# Gaussian smoothing
sigma = 1.0
gaussian_blur = cv2.GaussianBlur(image_array, (0, 0), sigma)
# Laplacian filtering
laplacian = cv2.Laplacian(gaussian_blur, cv2.CV_64F)
```

1. sigma = 1.0 : This line defines the standard deviation (σ) of the Gaussian filter. The value of σ determines the extent of smoothing. A higher σ will lead to more blurring, while a lower σ will produce a sharper image.
2. $cv2.GaussianBlur(image\_array, (0, 0), sigma)$: This is the main function that applies the Gaussian blur.the second argument, (0, 0), defines the kernel size. By setting it to (0, 0), you're allowing OpenCV to automatically determine the size based on the provided σ value.
3. The second argument in $cv2.Laplacian(gaussian_blur, cv2.CV_64F)$, $cv2.CV\_64F$, specifies the depth of the resulting image. it indicates that the result should be a floating-point image. This is crucial because the Laplacian filter can produce both positive and negative variations, indicating dark-to-light and light-to-dark transitions, respectively

Figure 8: LoG

2.2 Difference of Gaussians (DoG)

The Difference of Gaussians (DoG) is closely related to the Laplacian of Gaussian (as we discussed above) we take Difference of 2 Gaussian Blurred, and is primarily used for blob detection.

1. Blur the image with two different Gaussian filters, where one has a slightly larger standard deviation σ than the other.
2. Subtract one blurred image from the other.

```
# Define two different sigmas
sigma1 = 1.0
sigma2 = 1.5

# Apply Gaussian blurring with the two sigmas
gaussian_blur1 = cv2.GaussianBlur(image_array, (0, 0), sigma1)
gaussian_blur2 = cv2.GaussianBlur(image_array, (0, 0), sigma2)

# Compute the DoG
dog = gaussian_blur2 - gaussian_blur1
```

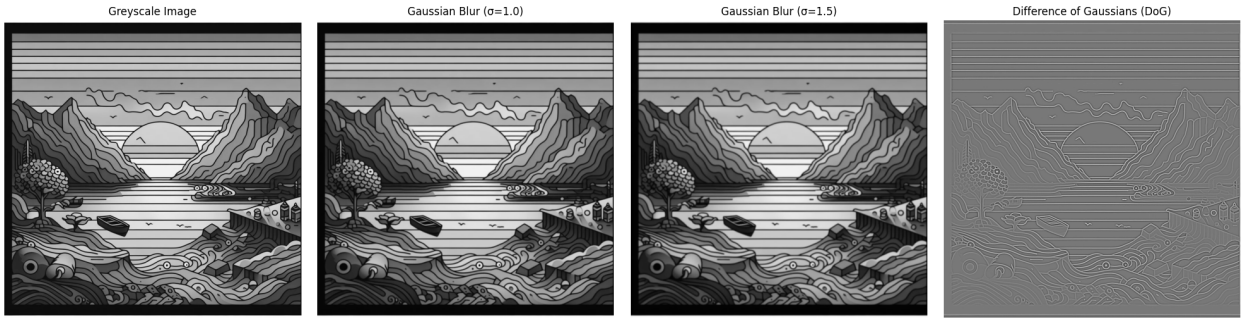

Figure 9: DoG

2.3 Gaussian Pyramid

The Gaussian pyramid is a multi-scale representation of an image. It is constructed by repeatedly reducing the size of the image and smoothing it with a Gaussian filter at each level. Each level of the pyramid is called a "scale".

1. Start with the original image.
2. Smooth the image using a Gaussian filter.
3. Reduce the size of the smoothed image by half in each dimension (typically, by sub-sampling every alternate pixel in both x and y directions).
4. Repeat the above two steps for the reduced image to get the next level of the pyramid.

We define the needed functions such as a function to convolve the gaussian filter to the image , a function to define the generate the guassian filter according to the size and the σ value and a function to genrate the different levels of guassian pyramid by using above 2 functions and down-sampling the image.

1.

```python
def generate_gaussian_filter(size, sigma):
    # Initialize the kernel with zeros
    kernel = np.zeros((size, size))

    # Calculate the center of the kernel
    center = size // 2

    # fill the kernel with values from the Gaussian function
    for i in range(size):
        for j in range(size):
            x = i - center
            y = j - center
            kernel[i, j] = (1/(2*np.pi*sigma**2)) * np.exp(-(x**2 + y**2) / (2*sigma**2))

    # Normalize the kernel
    kernel /= kernel.sum()

    return kernel
```

2.

```python
def pad_image(image, pad_size):
    return np.pad(image, ((pad_size, pad_size), (pad_size, pad_size)), mode='constant')

def convolve2D(image, kernel):
    m, n = kernel.shape
    pad_size = m // 2  # Assuming kernel is always odd-sized
    padded_image = pad_image(image, pad_size)

    y, x = padded_image.shape
    y = y - m + 1
    x = x - n + 1
    new_image = np.zeros((y,x))
```

```
    for i in range(y):
        for j in range(x):
            new_image[i][j] = np.sum(padded_image[i:i+m, j:j+n] * kernel)
    return new_image
```

3.

```
def gaussian_pyramid(image, num_levels, sigma):
    pyramid = [image]

    for i in range(num_levels - 1):
        # Create a Gaussian filter
        gaussian_filter = generate_gaussian_filter(5, sigma)

        # Convolve the image using the Gaussian filter
        smoothed_image = convolve2D(pyramid[-1], gaussian_filter)

        # Downsample the smoothed image
        reduced_image = downsample(smoothed_image)

        # Append the reduced image to the pyramid
        pyramid.append(reduced_image)

    return pyramid
```

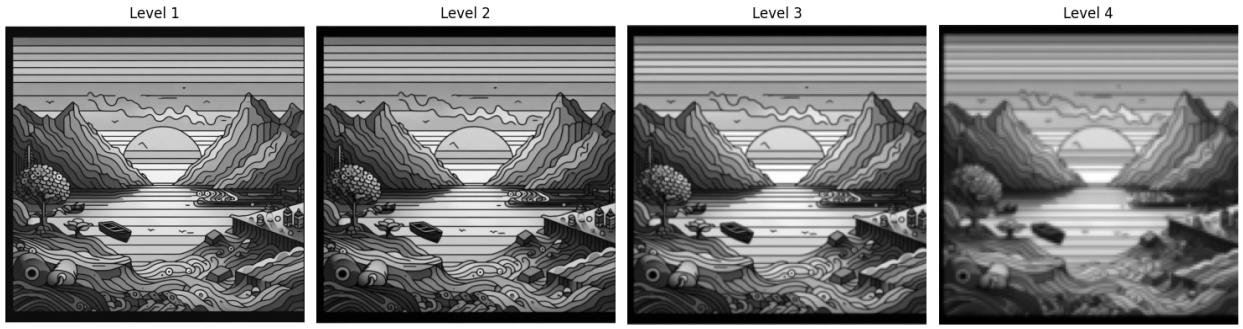

Figure 10: Gaussian Pyramid

3 Question 3

Develop the algorithm to merge the apple and orange using the pyramids as discussed in the class

Answer 3

Algorithm to merge apple and ornage using pyramids

1. Generate Laplacian pyramid of orange image.
2. Generate Laplacian pyramid of apple image.

3. Generate Combined Laplacian pyramid by copying left half of nodes at each level from apple and right half of nodes from orange pyramids.

4. Reconstruct combined image from Combined Laplacian pyramid.

Already we have the code for getting the Gaussian pyramid of an image as discussed in previous question now we will build upon that to get the Laplacian pyramid of an image.

1. We get the Gaussian pyramid of the image and then proceed like this to get the Laplacian pyramid,we need to up sample the Gaussian pyramid level and subtract it with the before one.

```
# upsampling the nighbouring level
def upsample2x(image):
    return image.repeat(2, axis=0).repeat(2, axis=1)

# Laplacian Pyramid
def laplacian_pyramid(gaussian_pyramid):
    laplacian_pyramid = []
    for i in range(len(gaussian_pyramid) - 1):
        upsampled = upsample2x(gaussian_pyramid[i + 1])
        if upsampled.shape != gaussian_pyramid[i].shape:
            upsampled = upsampled[:gaussian_pyramid[i].shape[0], :gaussian_pyramid[i].
                shape[1]]
        laplacian = gaussian_pyramid[i] - upsampled
        laplacian_pyramid.append(laplacian)
    laplacian_pyramid.append(gaussian_pyramid[-1])
    return laplacian_pyramid
```

2. Now after getting the Laplacian pyramid we do blending and reconstruct the image as mentioned above.

```
def blend_laplacian_pyramids(laplacian1, laplacian2):
    blended_pyramid = []
    for i in range(len(laplacian1)):
        # Take the left half from the apple image and the right half from the orange
            image
        rows, cols = laplacian1[i].shape
        blended = np.hstack((laplacian1[i][:, :cols//2], laplacian2[i][:, cols//2:]))
        blended_pyramid.append(blended)
    return blended_pyramid
# Function to reconstruct an image from a Laplacian pyramid
def reconstruct_from_laplacian(laplacian_pyramid):
    reconstructed_image = laplacian_pyramid[-1]  # Start with the smallest image
    for i in range(len(laplacian_pyramid) - 2, -1, -1):
        # Upsample and add to the next level
        upsampled = upsample2x(reconstructed_image)
        if upsampled.shape != laplacian_pyramid[i].shape:
            upsampled = upsampled[:laplacian_pyramid[i].shape[0], :laplacian_pyramid[i].
                shape[1]]
        reconstructed_image = upsampled + laplacian_pyramid[i]
    return reconstructed_image
```

Blending Laplacian Pyramids Function : The *blend_laplacian_pyramids* function combines two Laplacian pyramids by horizontally stacking the left half of each image from the first pyramid with the right half from the second pyramid. This results in a new pyramid where each level has the left side sourced from the first image and the right side from the second image, effectively blending the two original images.

Reconstructing from Laplacian Pyramid Function : The $reconstruct\_from\_laplacian$ function rebuilds an image from its Laplacian pyramid representation. Starting with the smallest image in the pyramid, it successively upsamples and adds the current image to the next pyramid level. This process reconstructs the original (or in the case of our blend, the combined) image from its multi-resolution representation in the Laplacian pyramid.

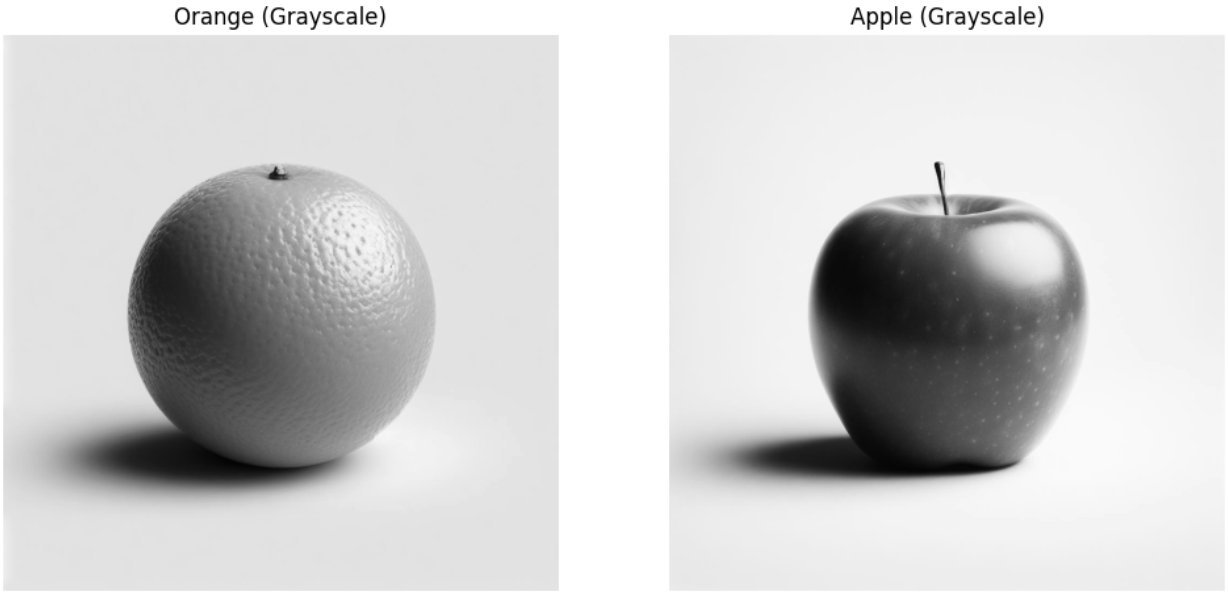

Figure 11: Orange and Apple Images

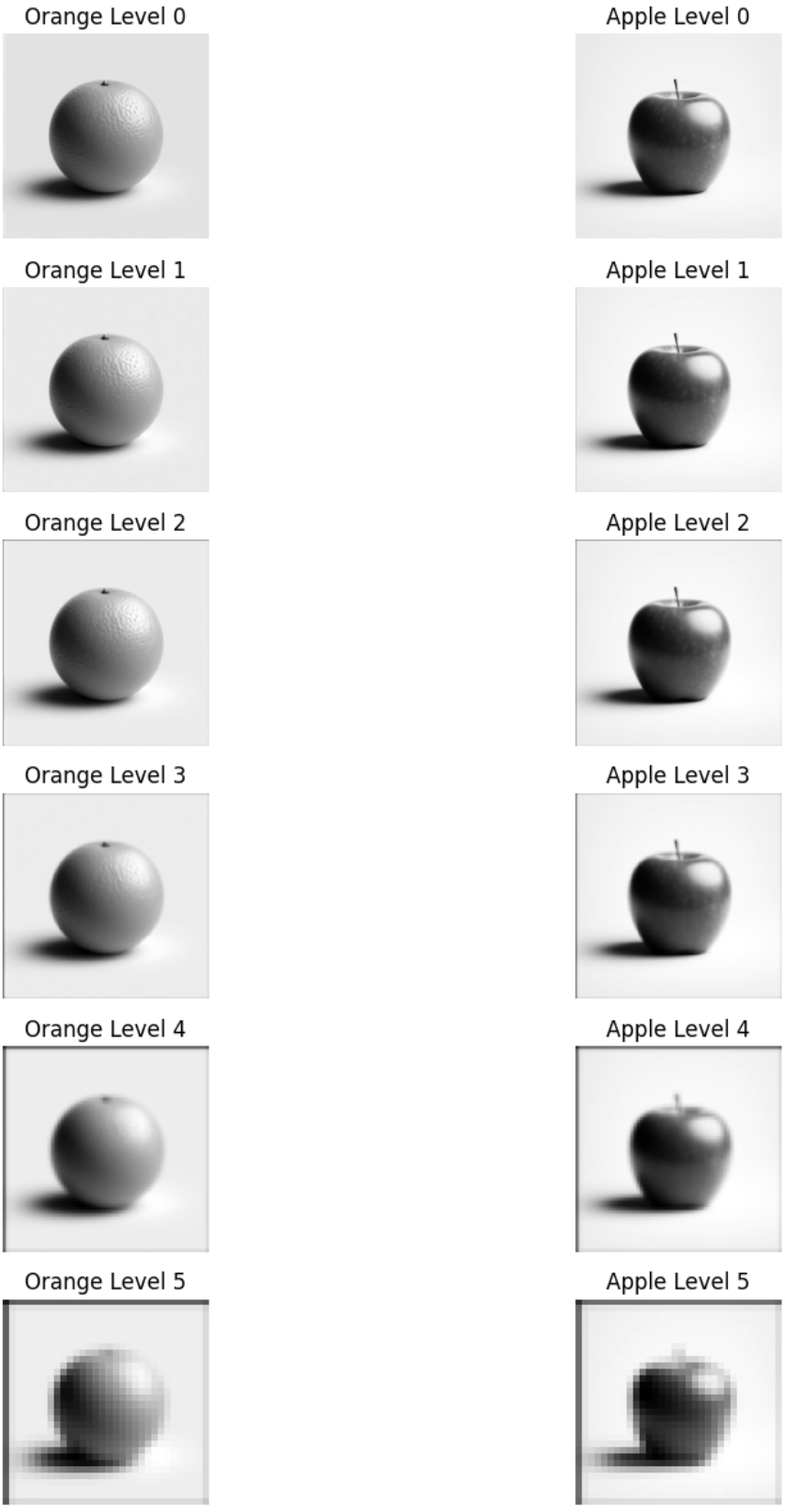

Figure 12: Gausssian Pyramid

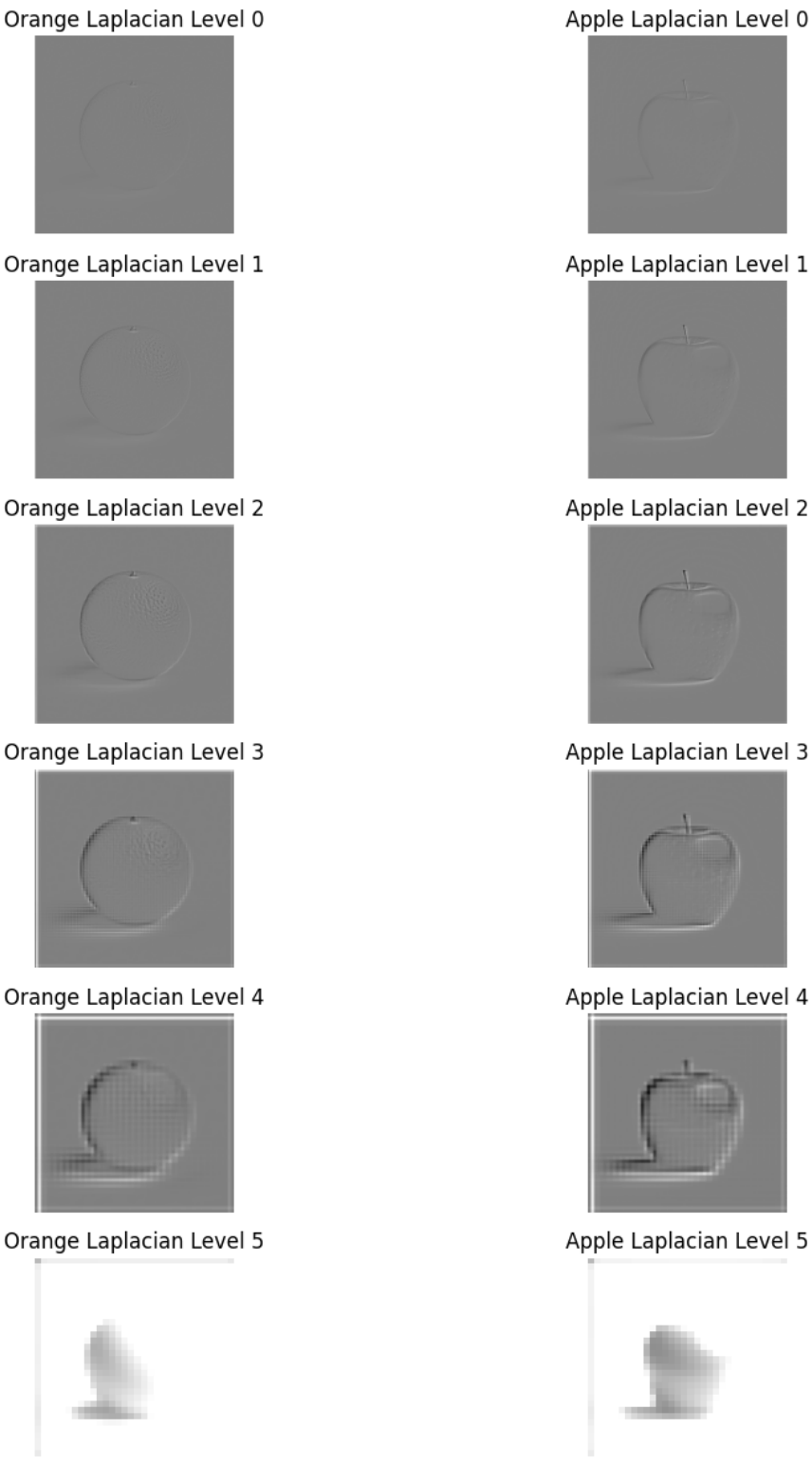

Figure 13: Laplacian Pyramid

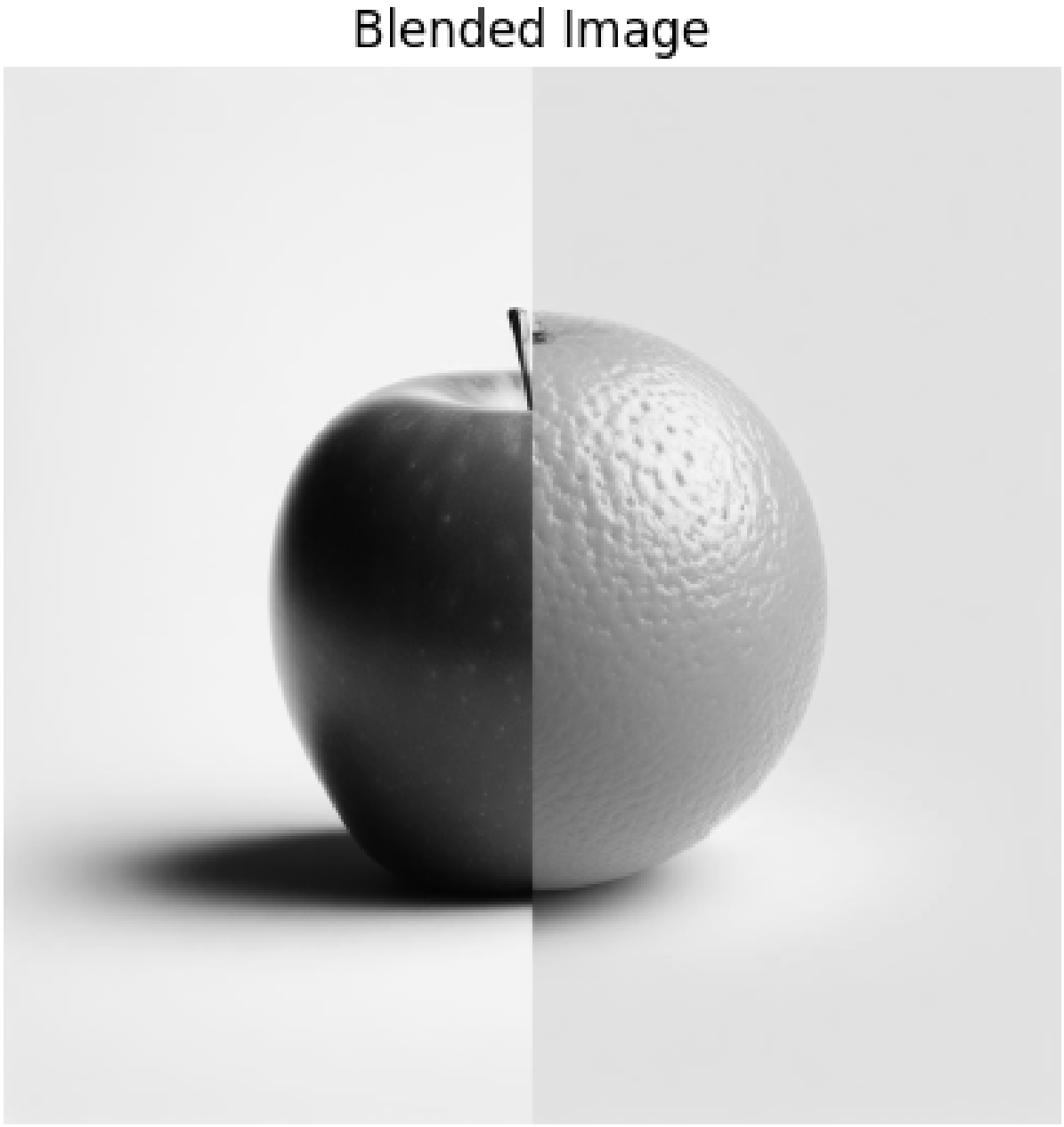

Figure 14: Combined Image