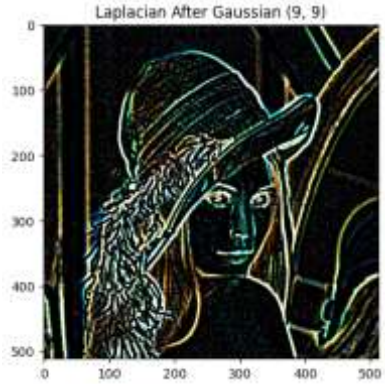
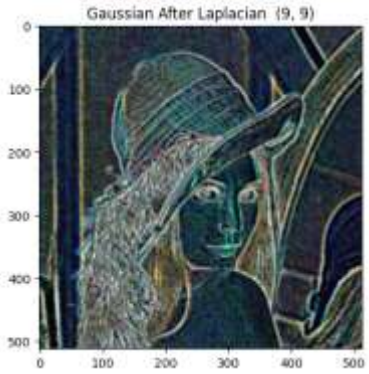
IMAGE_PROCESSING_EXERCISES

1. For three kernel sizes: (3, 3), (9, 9), (15, 15), I first applied the Gaussian and Laplacian filters to the image in different orders, using the cv2.GaussianBlur and cv2.Laplacian functions. The results are as follows.

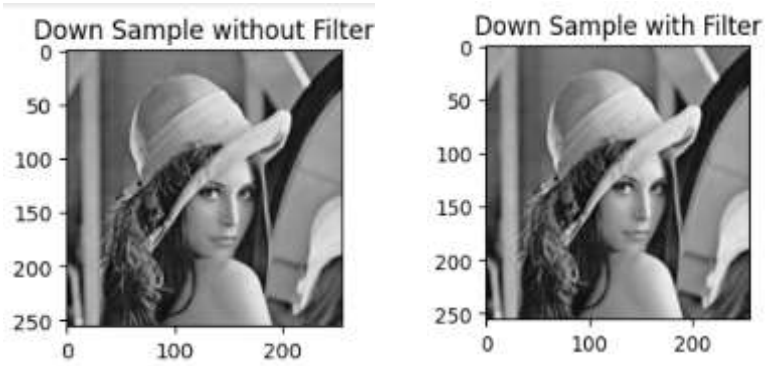




If we apply the Laplacian filter first, the image noise is amplified, as can be seen in the output. Applying the Gaussian filter then blurs the edges created by the Gaussian filter, and ultimately does not output edges with high frequency. On the other hand, if we apply the Gaussian filter first, the image noise is removed, and we can obtain better edges from the image by applying the Laplacian filter.

2. I first down sampled the image by a factor of 2, without using the averaging by using cv2.resize method and also with using the averaging by using cv2.pyrdown method:

```
# Down sample image without using average filter
downsampled_without_filter = cv2.resize(image, (w // 2, h // 2))
plt.figure(figsize=(fh/2, fw/2))
plt.imshow(downsampled_without_filter, cmap='gray')
plt.title('Down Sample without Filter')
plt.grid(False)
plt.show()
print(downsampled_without_filter.shape)

# Down sample image by using average filter
downsampled_avg_filter = cv2.pyrDown(image, (w // 2, h // 2))
plt.figure(figsize=(fh/2, fw/2))
plt.imshow(downsampled_avg_filter, cmap='gray')
plt.title('Down Sample with Filter')
plt.grid(False)
plt.show()
```

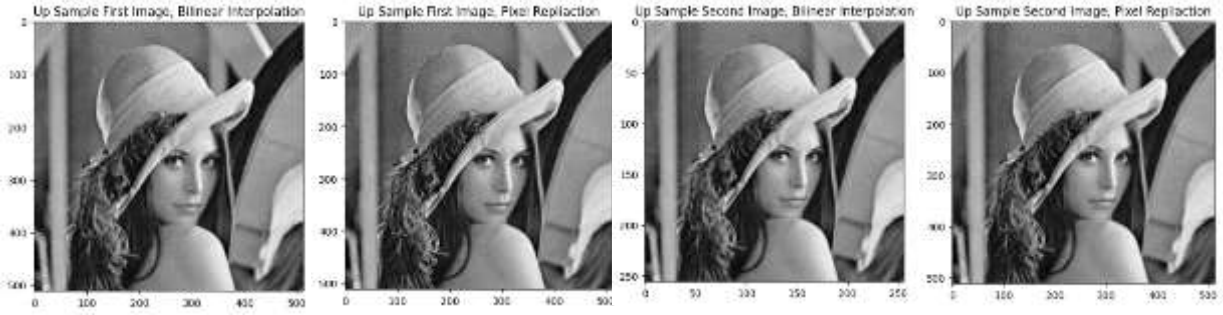


Then I up sampled each of these two down sampled images by a factor of 2, using the cv2.resize method with parameter “cv2.INTER_LINEAR” as interpolation.

```
# Up sample  the first down_sampled image using Bilinear Interpolation
uimage  = cv2.resize(downsampled_without_filter, (w , h), interpolation=cv2.INTER_LINEAR)
plt.figure(figsize=(fh, fw))
plt.imshow(uimage, cmap='gray')
plt.title('Up Sample First Image, Bilinear Interpolation')
plt.grid(False)
plt.show()
```

I also up sampled each down sampled image by a factor of 2 by repeating the pixels:

```
upsampled_image = np.zeros((downsampled_without_filter.shape[0] * 2, downsampled_without_filter.shape[1] * 2), dtype=np.uint8)
for x in range(downsampled_without_filter.shape[0]):
    for y in range(downsampled_without_filter.shape[1]):
        for i in range(2):
            for j in range(2):
                upsampled_image[x * 2 + i, y * 2 + j] = downsampled_without_filter[x, y]
plt.figure(figsize=(fh, fw))
plt.imshow(upsampled_image, cmap='gray')
plt.title('Up Sample First Image, Pixel Repliaction')
plt.grid(False)
plt.show()
```



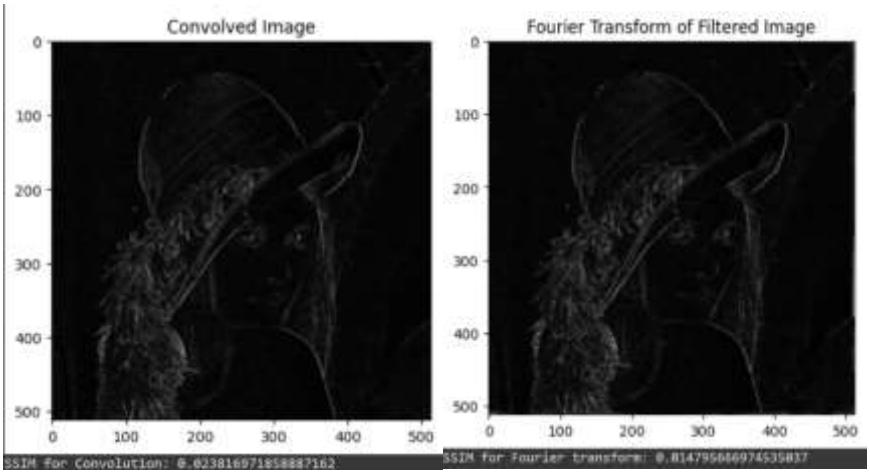
3. The convolve_fft function first calculates the Fourier transform of the kernel and the image using the fft2 function. Then, after multiplying the kernel and the image in the frequency domain using the ifft2 function, we take the inverse Fourier transform and then shift it with the ifftshift function. The output of the function is the image with the kernel applied to it in the space domain.

```python
def convolve_fft(image, filter):
    # Convert image and filter to Fourier domain
    F_image = fft2(image)
    F_filter = fft2(filter)

    # Multiply the padded filter with the image's Fourier transform
    F_filtered = F_image * F_filter

    # Apply inverse Fourier transform
    filtered_image = ifft2(F_filtered)
    filtered_image = np.fft.ifftshift(filtered_image)
    # Return filtered image
    abs_F_filtered = np.abs(filtered_image)

    # Return absolute values of filtered image
    return abs_F_filtered
```

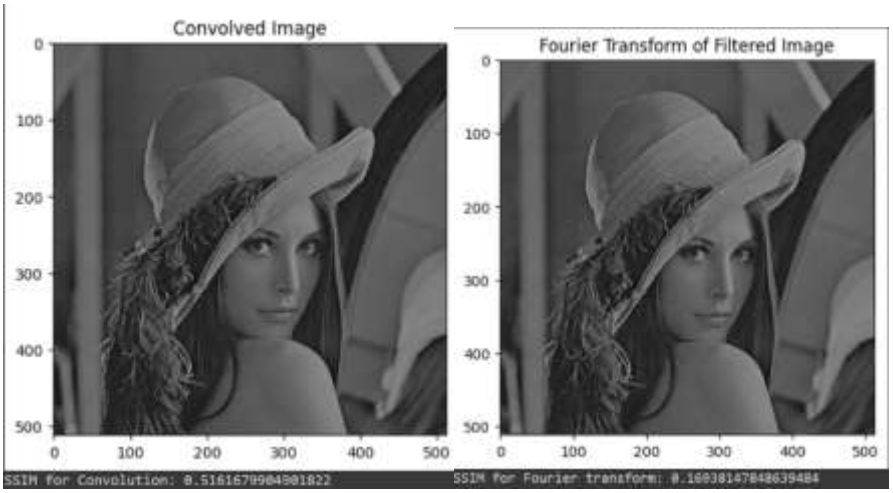
Now, I convolve the image with each of the desired kernels in the question statement, using the cv2.filter2D function. Then, we apply them in the frequency domain on the image, using the convolve_fft function. Also, the SSIM value is calculated for the output of each kernel.

First Kernel:

Second Kernel:

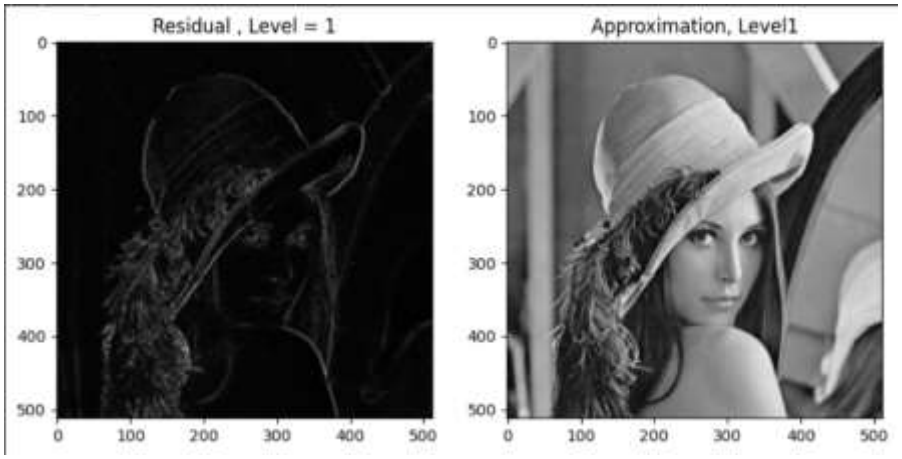


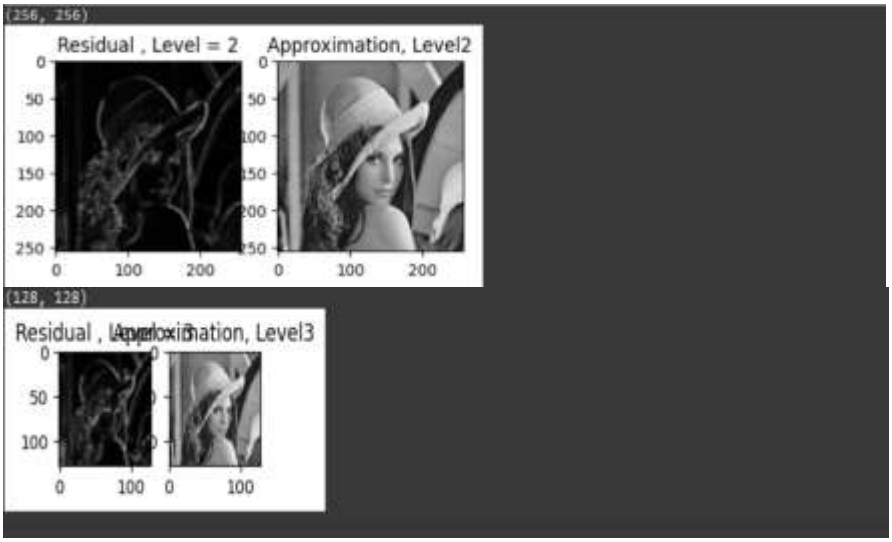
Third Kernel:

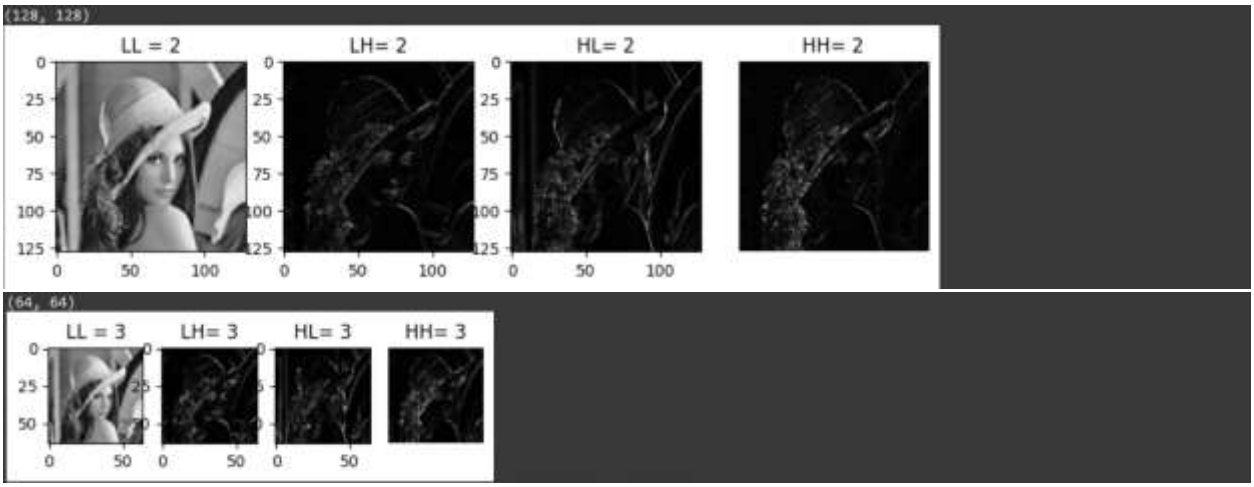


4. First I computed 3-level approximation pyramid cv2.resze method by down sampling in each iteration and stored them in approx_pyramid array. Then, I constructed the Gaussian pyramid corresponding to each approximation using the function cv2.pyrDown. Then, I constructed the corresponding prediction residual pyramid by upsampling the Gaussian pyramid with pyrUp and subtracting each image inside the Gaussian pyramid from the level before it. Finally, I plotted the approximation and corresponding prediction residual in each level using subplot.

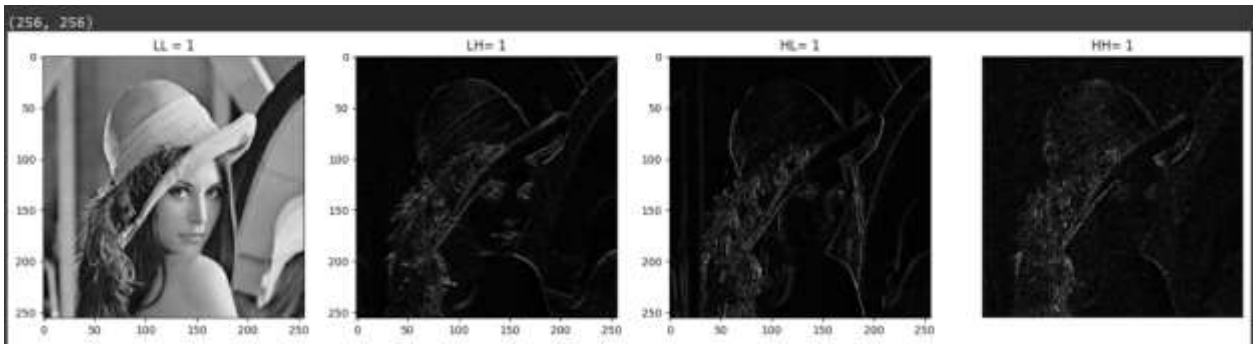
5. I performed a three-level discrete wavelet transform (DWT) on the image using the Haar wavelet and pywt.dwt2 method. Then, I stored the resulting approximation and detail coefficients and plotted all of them in three level by subplots.







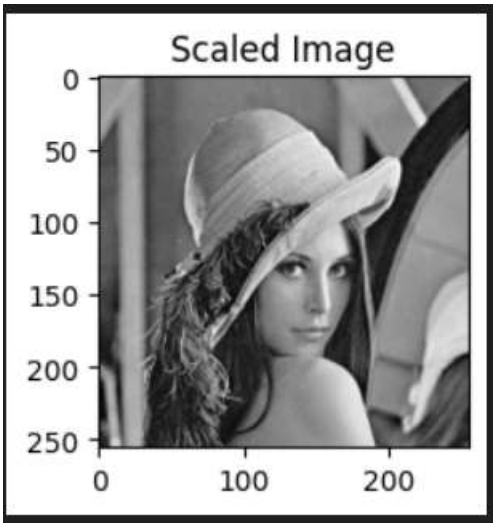
The difference is that in the wavelet pyramid, there are three details in three different directions in each level, while in the Laplacian pyramid, there is only one detail. Also, the



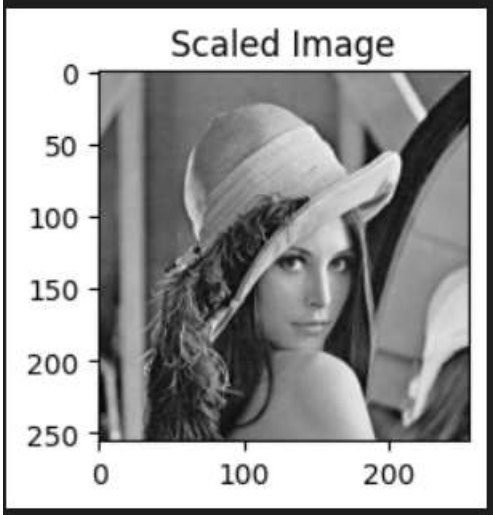
wavelet is calculated using complex mathematical calculations, while in the Laplacian, the corresponding prediction residual is simply calculated by subtracting the approximation.

6. Fist I defined three points in the original image and its corresponding location to the output image to determine the Affine transformation type which is scaling transform. Then I used cv2.getAffineTransform method to compute the transform using Bilinear Interpolation and Nearest-Neighbor. Finally, by up sampling them I computed the SSIM.

```
# Define the source and destination points with the correct shape and data type
pts1 = np.float32([[50,50],[200,50],[50,200]])

# define three points corresponding location to output image
pts2 = np.float32([[25,25],[100,25],[25,100]])

# get the affine transformation Matrix
M = cv2.getAffineTransform(pts1,pts2)

# apply affine transformation on the input image using Nearest-Neighbor
dst = cv2.warpAffine(image,M,(image.shape[0]//2,image.shape[1]//2), flags=cv2.INTER_NEAREST)
plt.figure(figsize=(fh/2, fw/2))
plt.imshow(dst, cmap='gray')
plt.title('Scaled Image')
plt.grid(False)
plt.show()
up_sampled  = cv2.resize(dst, (image.shape[0] , image.shape[1]))
print(up_sampled.shape)
print("SSIM for Nearest-Neighbor:", ssim(image, up_sampled))

# apply affine transformation on the input image using Bilinear Interpolation
dst = cv2.warpAffine(image,M,(image.shape[0]//2,image.shape[1]//2), flags=cv2.INTER_LINEAR)
plt.figure(figsize=(fh/2, fw/2))
plt.imshow(dst, cmap='gray')
plt.title('Scaled Image')
plt.grid(False)
plt.show()
print(dst.shape)
up_sampled  = cv2.resize(dst, (image.shape[0] , image.shape[1]))
print("SSIM for Bilinear Interpolation:", ssim(image, up_sampled))
```




```
512, 512)
SIM for Nearest-Neighbor: 0.6519208671225432
```




```
256, 256)
SIM for Bilinear Interpolation: 0.6519208671225432
```