DSA5203 Assignment 1 Technical Report

Name: Ong Jian Ying Gary

Student ID: A0155664X

The assignment requires us to write 2 functions, namely haar2d which performs 2D discrete Haar wavelet transform of a given ndarray im, and ihar2d which performs 2D discrete inverse Haar wavelet transform from the ndarray of wavelet coefficients.

To start the assignment, we first define the necessary functions required for the task, such as scipy.signal.convolve2d. We then define the 2D Haar wavelet transform filter banks as shown below, and they are required for 2D wavelet reconstruction.

```
# Defining the filter banks for 2D Haar wavelet transform - required for
wavelet reconstruction
H = np.array([[1, 1], [1, 1]]) / 2
G1 = np.array([[1, 1], [-1, -1]]) / 2
G2 = np.array([[1, -1], [1, -1]]) / 2
G3 = np.array([[1, -1], [-1, 1]]) / 2
```

And also define the reverse filter banks for 2D Haar wavelet transform, by using np.fliplr and np.flipud to reverse the orders in both row and column of the original filter banks. This is required for 2D wavelet decomposition.

```
# Defining the reverse order filter banks for 2D Haar wavelet transform -
required for wavelet decomposition
# reverse in both the up&down and left&right direction to obtain the reverse
filter banks for 2D Haar Wavelet
H_rev = np.fliplr(np.flipud(H))
G1_rev = np.fliplr(np.flipud(G1))
G2_rev = np.fliplr(np.flipud(G2))
G3_rev = np.fliplr(np.flipud(G3))
```

One comment to be made here is that, the inverse wavelet transform should perfectly reconstruct the original image from its corresponding output from the wavelet transform, no matter how many levels were used in the wavelet transform. This is a fundamental property and is known as the perfect reconstruction property of wavelet transform. This means that all of the information in the original image is preserved in the wavelet coefficients, and it can be exactly recovered from the coefficients by the inverse transform. The wavelet coefficients capture information about the image's edges, textures and other features, and they could be used for various image processing tasks.

Now we will start writing the function for haar2d.

```
  # initialize a wavelet coefficient matrices with the same initial size as
the image, that we will continuously update with values from our convolve2d
and downsampling.
  out = np.zeros([len(im), len(im)])
  # initialize the length of the image, which will be adjused within the for
loop for required number of levels
  im_len = len(im)
```

We first initialize a wavelet coefficient matrix with zeroes, which has the same initial size as the original image. The image that was used for testing was the image_512.png that was provided, which had a size of 512x512. Since the required output has to be of same size, we initialize the zeroes matrix and update this matrix at every wavelet decomposition level being performed. We also initialize the length of im_len = len(im), which will be adjusted within the for loop which loops through the 'lvl' argument of wavelet decomposition required.

Now we will loop through the number of times that was specified by 'lvl' argument to perform wavelet decomposition. We run convolve2d with reverse order filter banks for wavelet decomposition as defined in the lecture notes. For convolve2d, we make use of mode='same', which will keep the output coefficient size to be same as that of the input, and use circular convolution for the boundary treatment and define boundary='wrap' for the argument.

Thereafter, we perform downsampling by taking a step size of 2. Care has to be taken to use a start index of 1 for both rows and columns, so that we always take the second element and downsample from the original matrix, in step size of 2.

```
    # run the downsampling by taking a step size of 2, taking note of the
starting index for downsampling for row & column to be index at 1 (2nd
element)
    im_h = im_h[1::2, 1::2]
    im_g1 = im_g1[1::2, 1::2]
    im_g2 = im_g2[1::2, 1::2]
    im_g3 = im_g3[1::2, 1::2]
```

Thereafter, we will update the output ndarray 'out' with the current iteration's wavelet coefficient. We update the im_h coefficients to be at the top left of the 'out' array, im_g1 coefficient to be on the right of H, im_g2 coefficient to be below H, and im_g3 coefficient to be diagonal from im_h, as this ordering was discussed in lectures.

```
    out[0:im_len//2 , 0:im_len//2] = im_h # Scale Coefficient - H is on the
top left as in the question paper
    out[0:im_len//2 , im_len//2:im_len] = im_g1 # Wavelet Coefficient - G1 is
on the right of H as in lecture notes
    out[im_len//2:im_len , 0:im_len//2] = im_g2 # Wavelet Coefficient - G2 is
directly below H as in lecture notes
    out[im_len//2:im_len , im_len//2:im_len] = im_g3 # Wavelet Coefficient -
G3 is diagonally opposite H as in lecture notes
```

We have now completed one iteration of wavelet decomposition, and need to prepare for the next iteration if lvl > 1.

```
    # Now we need to prepare for the next iteration, if any, if lvl > 1
    # In decomposition, the subsequent wavelet decompositions are performed on
im_h as shown in lecture notes
    # Update im to be im_h
    im = im_h

    # Update our length to be divided by 2
    im_len //= 2
```

Hence, we update im = im_h, since scale coefficient im_h will be further decomposed in the next wavelet decomposition, and also update the length of the image to be operated on as half, so that we can work on the next image grid of interest.

After all of the wavelet decompositions are completed, we will return 'out', which is the numpy array representing the 2D Haar wavelet coefficients after 'lvl' times of wavelet decomposition.

Next, we proceed to define the ihaar2d 2D Haar wavelet reconstruction.

We first define start from the smallest set of images in our coefficient matrix by finding the length of the smallest image. We can know this by using 'lvl' input, so that we can find the smallest grid to reconstruct backwards from.

```
  # Start with the smallest set of images in our coefficient matrix by finding
the length
  im_len_smallest = len(coef)//2**(lvl-1)
```

Next, we also locate the last level (lvl'th) scale coefficients (im_h) at the extreme top left of our coefficient matrix and initialize it outside of the for loop, as we will iteratively reconstruct the (lvl'th - 1) scale coefficients by a summation of coefficients.

```
   # locate s.x (im_h_coef) outside the for loop, as we will get im_h_coef
after summing up the rest of the coefficients for the rest of the loop.

  out = coef[0:im_len_smallest//2, 0:im_len_smallest//2]
```

Now we will start to loop through the levels for wavelet reconstruction.

Within the loop, we will locate our im_g1_coef, im_g2_coef, im_g3_coef for the lvl'th wavelet coefficients as follows:

```
    # locate our locate our w.x.1, w.x.2, w.x.3 from our coefficient matrix
with im_len_smallest
    im_g1_coef = coef[0:im_len_smallest//2 ,
im_len_smallest//2:im_len_smallest]
    im_g2_coef = coef[im_len_smallest//2:im_len_smallest,
0:im_len_smallest//2]
```

```
    im_g3_coef = coef[im_len_smallest//2:im_len_smallest ,
im_len_smallest//2:im_len_smallest]
```

Then, to perform upsampling by a factor of 2, we first initialize the zeroes matrix to have the same length as im_len_smallest. These are the matrices to be updated throughout the wavelet reconstruction procedure.

```
    # To perform upsampling by a factor of 2, first initialize zeroes matrix
to be updated at the current iteration.
    h_coef_up = np.zeros([im_len_smallest, im_len_smallest])
    g1_coef_up = np.zeros([im_len_smallest, im_len_smallest])
    g2_coef_up = np.zeros([im_len_smallest, im_len_smallest])
    g3_coef_up = np.zeros([im_len_smallest, im_len_smallest])
```

We then update our matrix at step size of 2 intervals to complete upsizing. Key point to note is to use starting index of 0 for upsampling as we want to insert zeroes at every in between every element, and this is also demonstrated in lecture notes for 1D wavelet reconstruction. This will also ensure that our coefficient matrix is aligned to achieve the perfect reconstructed image that resembles the original.

```
    # updating our matrix at step size of 2 intervals to complete upsizing
    # starting index = 0 for upsampling as in lecture notes.
    h_coef_up[0::2, 0::2] = out
    g1_coef_up[0::2, 0::2] = im_g1_coef
    g2_coef_up[0::2, 0::2] = im_g2_coef
    g3_coef_up[0::2, 0::2] = im_g3_coef
```

Now, we perform convolve2d using the same settings of mode= ‘same’ and boundary= ‘wrap’ as before.

```
    # Perform convolve2d with the 2D Haar wavelet filter banks
    out = convolve2d(h_coef_up, H, mode='same', boundary='wrap')
    im_g1_coef = convolve2d(g1_coef_up, G1, mode='same', boundary='wrap')
    im_g2_coef = convolve2d(g2_coef_up, G2, mode='same', boundary='wrap')
    im_g3_coef = convolve2d(g3_coef_up, G3, mode='same', boundary='wrap')
```

Lastly, we perform summation to get im_h_coef for the next iteration, if required, and also double the length of im_len_smallest to prepare for the next iteration of reconstruction.

```
    # Perform summation to get im_h_coef for the current larger cell
    out = out + im_g1_coef + im_g2_coef + im_g3_coef

    # Update for next iteration, double the im_len_smallest
    im_len_smallest *= 2
```

After finish looping through all levels of wavelet reconstruction, we must realize that the current reconstructed image 'out' will store numpy array of floats. This might cause issues with the display of the recovery image.

```
recovery =  Image.fromarray(ihaar2d(haar2d_coef,level),mode='L')
```

Before we output our numpy array, we have to convert the data type to uint8 to ensure correct saved image.

```
  # After the end of the final loop, we will obtain our perfectly
reconstructed image result.
  # However, since Image.fromarray is utilized, we need to make sure we
convert the numpy array from floating to uint8, so that Image.fromarray will
save the correct image
  out = out.astype(np.uint8)
```

We now have finished defining haar2d and ihaar2d for 2D Haar wavelet transform and inverse transform for a 2**N x 2**N image.

By using the same test image and specifying the image_512.png as the test image path, the following is the 4-level visualization of the wavelet transform:

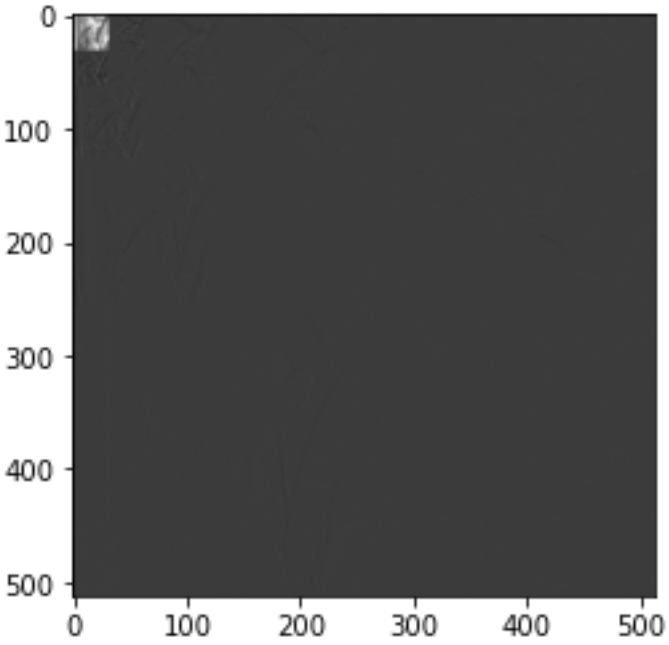

Figure 1: Visualization of 4-level wavelet coefficients

We can also look at the scale and wavelet coefficients of the 4th level in separate subplots below:

```
In [82]:
1 plt.subplot(1,4,1); plt.imshow(wavelet_coefficients[0:64//2, 0:64//2], cmap='gray')
2 plt.subplot(1,4,2); plt.imshow(wavelet_coefficients[0:64//2, 64//2:64], cmap='gray')
3 plt.subplot(1,4,3); plt.imshow(wavelet_coefficients[64//2:64, 0:64//2], cmap='gray')
4 plt.subplot(1,4,4); plt.imshow(wavelet_coefficients[64//2:64, 64//2:64], cmap='gray')

Out[82]: <matplotlib.image.AxesImage at 0x2e4c2692fd0>
```

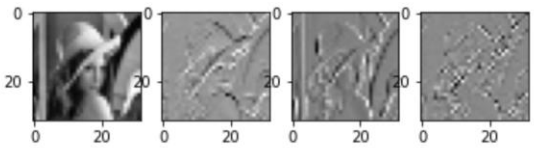

Figure 2: 4th level scale and wavelet coefficients depiction

Thereafter, we perform inverse wavelet transform to get the recovered image from performing 4 level wavelet decomposition and reconstruction:

Figure 3: Reconstructed image recovery.png from image_512.png after 4-level wavelet transform.

We can see that the image clearly resembles the original image as shown:

Figure 4: Original image_512.png

Doing an additional check on the output numpy arrays in a separate jupyter notebook by using np.array_equal() method, we can also see that the result is True, which means that the code perfectly reconstructs the original image regardless of the number of levels of wavelet transform and inverse transform.

In [10]:
```
1 np.array_equal(recon_image,im)
```

Out[10]: True

End of Assignment.