

# Introduction to Wavelets

**Lab Objective:** *Wavelets are useful in a variety of applications because of their ability to represent some types of information in a very sparse manner. We will explore both the one- and two-dimensional discrete wavelet transforms using various types of wavelets. We will then use a Python package called PyWavelets for further wavelet analysis including image cleaning and image compression.*

## The Discrete Wavelet Transform

In wavelet analysis, information (such as a mathematical function or image) can be stored and analyzed by considering its *wavelet decomposition*. The wavelet decomposition is a way of expressing information as a linear combination of a particular set of wavelet functions. Once a wavelet is chosen, information can be represented by a sequence of coefficients (called *wavelet coefficients*) that define the linear combination. The mapping from a function to a sequence of wavelet coefficients is called the *discrete wavelet transform*.

The discrete wavelet transform is analogous to the discrete Fourier transform. Now, instead of using trigonometric functions, different families of basis functions are used. A function called the *wavelet* or *mother wavelet*, usually denoted  $\psi$ , and another function called the *scaling* or *father scaling function*, typically represented by  $\phi$ , are the basis of a wavelet family. A countably infinite set of wavelet functions (commonly known as *daughter wavelets*) can be generated using dilations and shifts of the first two functions:

$$\begin{aligned}\psi_{m,k}(x) &= \psi(2^m x - k) \\ \phi_{m,k}(x) &= \phi(2^m x - k),\end{aligned}$$

where  $m, k \in \mathbb{Z}$ .

Given a wavelet family, a function  $f$  can be approximated as a combination of father and daughter wavelets as follows:

$$f(x) = \sum_{k=-\infty}^{\infty} a_k \phi_{m,k}(x) + \sum_{k=-\infty}^{\infty} b_{m,k} \psi_{m,k}(x) + \cdots + \sum_{k=-\infty}^{\infty} b_{n,k} \psi_{n,k}(x)$$

where  $m < n$  and all but a finite number of the  $a_k$  and  $b_{j,k}$  terms are nonzero. The  $a_k$  terms are often referred to as *approximation coefficients* while the  $b_{j,k}$  terms are known as *detail coefficients*. The approximation coefficients typically capture the broader, more general features of a signal while

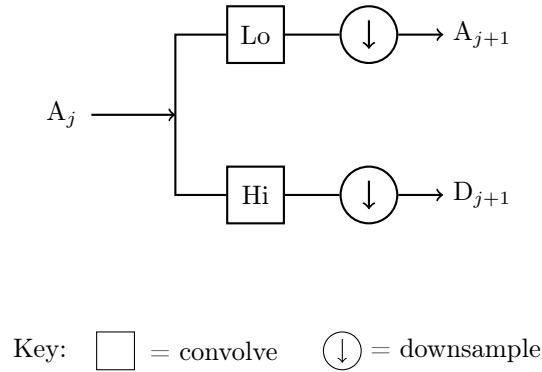


Figure 8.1: The one-dimensional discrete wavelet transform implemented as a filter bank.

the detail coefficients capture smaller details and noise. Depending on the properties of the wavelet and the function (or signal),  $f$  can be approximated to an arbitrary level of accuracy.

In the case of finitely-sampled signals and images, there exists an efficient algorithm for computing the wavelet coefficients. Most commonly used wavelets have associated high-pass and low-pass filters which are derived from the wavelet and scaling functions, respectively. When the low-pass filter is convolved with the sampled signal, low frequency (also known as approximation) information is extracted. This approximation highlights the overall (slower-moving) pattern without paying too much attention to the high frequency details. The high frequency details are extracted by the high-pass filter. These detail coefficients highlight the small changes found in the signal. The two primary operations of the algorithm are the discrete convolution and downsampling, denoted  $*$  and  $DS$ , respectively. First, a signal is convolved with both filters. Then the resulting arrays are downsampled to remove redundant information. In the context of this lab, a *filter bank* is the combined process of convolving with a filter then downsampling. This process can be repeated on the new approximation to obtain another layer of approximation and detail coefficients.

See Algorithm 8.1 and Figure 8.1 for the specifications.

---

**Algorithm 8.1** The one-dimensional discrete wavelet transform.  $X$  is the signal to be transformed,  $L$  is the low-pass filter,  $H$  is the high-pass filter and  $n$  is the number of filter bank iterations.

---

```

1: procedure DWT( $X, L, H, n$ )
2:    $A_0 \leftarrow X$  ▷ Initialization.
3:   for  $i = 0 \dots n - 1$  do
4:      $D_{i+1} \leftarrow DS(A_i * H)$  ▷ High-pass filter and downsample.
5:      $A_{i+1} \leftarrow DS(A_i * L)$  ▷ Low-pass filter and downsample.
6:   return  $A_n, D_n, D_{n-1}, \dots, D_1$ .
```

---

The *Haar Wavelet* is one of the most widely used wavelets in wavelet analysis. Its wavelet function is defined

$$\psi(x) = \begin{cases} 1 & \text{if } 0 \leq x < \frac{1}{2} \\ -1 & \text{if } \frac{1}{2} \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

The associated scaling function is given by

$$\phi(x) = \begin{cases} 1 & \text{if } 0 \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

For the Haar Wavelet, the low-pass and high-pass filters are given by

$$L = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$H = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

As noted earlier, the key mathematical operations of the discrete wavelet transform are convolution and downsampling. Given a filter and a signal, the convolution can be obtained using `scipy.signal.fftconvolve()`.

```
>>> from scipy.signal import fftconvolve
>>> # Initialize a filter.
>>> L = np.ones(2)/np.sqrt(2)
>>> # Initialize a signal X.
>>> X = np.sin(np.linspace(0,2*np.pi,16))
>>> # Convolve X with L.
>>> fftconvolve(X, L)
[ -1.84945741e-16  2.87606238e-01  8.13088984e-01  1.19798126e+00
  1.37573169e+00  1.31560561e+00  1.02799937e+00  5.62642704e-01
  7.87132986e-16 -5.62642704e-01 -1.02799937e+00 -1.31560561e+00
 -1.37573169e+00 -1.19798126e+00 -8.13088984e-01 -2.87606238e-01
 -1.84945741e-16]
```

The convolution operation alone gives redundant information, so it is downsampled to keep only what is needed. In the case of the Haar wavelet, the array will be downsampled by a factor of two, which means keeping only every other entry:

```
>>> # Downsample an array X.
>>> sampled = X[1::2]
```

Both the approximation and detail coefficients are computed in this manner. The approximation uses the low-pass filter while the detail uses the high-pass filter.

**Problem 1.** Write a function that calculates the discrete wavelet transform using Algorithm 8.1. The function should return a list of one-dimensional NumPy arrays in the following form:  $[A_n, D_n, \dots, D_1]$ .

Test your function by calculating the Haar wavelet coefficients of a noisy sine signal with  $n = 4$ :

```
domain = np.linspace(0, 4*np.pi, 1024)
noise = np.random.randn(1024)*.1
noisysin = np.sin(domain) + noise
coeffs = dwt(noisysin, L, H, 4)
```

Plot the original signal with the approximation and detail coefficients and verify that they match the plots in Figure 8.2.

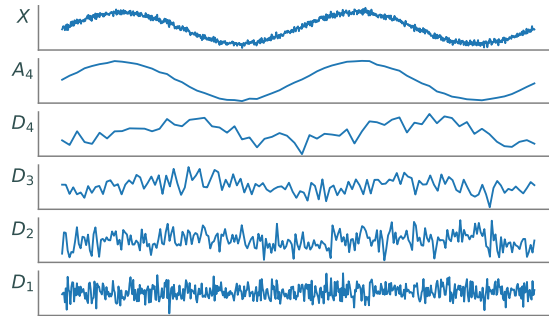


Figure 8.2: A level four wavelet decomposition of a signal. The top panel is the original signal, the next panel down is the approximation, and the remaining panels are the detail coefficients. Notice how the approximation resembles a smoothed version of the original signal, while the details capture the high-frequency oscillations and noise.

The process of the discrete wavelet transform is reversible. Using modified filters, a set of detail coefficients and a set of approximation coefficients can be manipulated and added together to recreate a signal. The Haar wavelet filters for the inverse transformation are

$$L = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$H = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}.$$

Suppose the wavelet coefficients  $A_n$  and  $D_n$  have been computed.  $A_{n-1}$  can be recreated by tracing the schematic in Figure 8.1 backwards:  $A_n$  and  $D_n$  are first upsampled, and then are convolved with  $L$  and  $H$ , respectively. In the case of the Haar wavelet, *upsampling* involves doubling the length of an array by inserting a 0 at every other position. To complete the operation, the new arrays are added together to obtain  $A_{n-1}$ .

```
>>> # Upsample the coefficient arrays A and D.
>>> up_A = np.zeros(2*A.size)
>>> up_A[::2] = A
>>> up_D = np.zeros(2*D.size)
>>> up_D[::2] = D
>>> # Convolve and add, discarding the last entry.
>>> A = fftconvolve(up_A, L)[: -1] + fftconvolve(up_D, H)[: -1]
```

This process is continued with the newly obtained approximation coefficients and with the next detail coefficients until the original signal is recovered.

**Problem 2.** Write a function that performs the inverse wavelet transform. The function should accept a list of arrays (of the same form as the output of the function written in Problem 1), a reverse low-pass filter, and a reverse high-pass filter. The function should return a single array,

which represents the recovered signal.

Note that the input list of arrays has length  $n + 1$  (consisting of  $A_n$  together with  $D_n, D_{n-1}, \dots, D_1$ ), so your code should perform the process given above  $n$  times.

To test your function, first perform the inverse transform on the noisy sine wave that you created in the first problem. Then, compare the original signal with the signal recovered by your inverse wavelet transform function using `np.allclose()`.

## NOTE

Although Algorithm 8.1 and the preceding discussion apply in the general case, the code implementations apply only to the Haar wavelet. Because of the nature of the discrete convolution, when convolving with longer filters, the signal to be transformed needs to undergo some type of lengthening in order to avoid information loss during the convolution. As such, the functions written in Problems 1 and 2 will only work correctly with the Haar filters and would require modifications to be compatible with more wavelets.

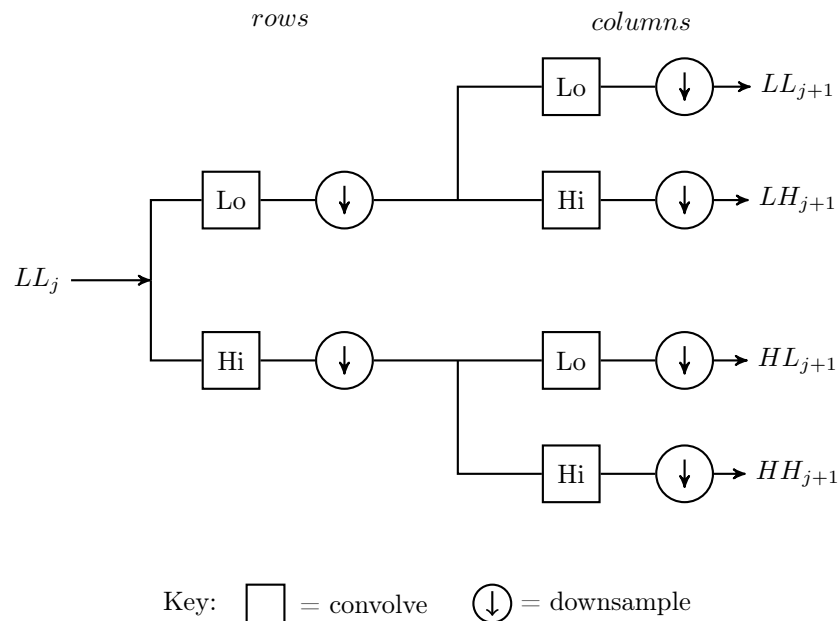


Figure 8.3: The two-dimensional discrete wavelet transform implemented as a filter bank.

## The Two-dimensional Wavelet Transform

The generalization of the wavelet transform to two dimensions is fairly straightforward. Once again, the two primary operations are convolution and downsampling. The main difference in the two-dimensional case is the number of convolutions and downsamples per iteration. First, the convolution and downsampling are performed along the rows of an array. This results in two new arrays. Then,

convolution and downsampling are performed along the columns of the two new arrays. This results in four final arrays that make up the new approximation and detail coefficients. See Figure 8.3 for an illustration of this concept.

When implemented as an iterative filter bank, each pass through the filter bank yields an approximation plus three sets of detail coefficients rather than just one. More specifically, if the two-dimensional array  $X$  is the input to the filter bank, the arrays  $LL$ ,  $LH$ ,  $HL$ , and  $HH$  are obtained.  $LL$  is a smoothed approximation of  $X$  (similar to  $A_n$  in the one-dimensional case) and the other three arrays contain wavelet coefficients capturing high-frequency oscillations in vertical, horizontal, and diagonal directions. The arrays  $LL$ ,  $LH$ ,  $HL$ , and  $HH$  are known as *subbands*. Any or all of the subbands can be fed into a filter bank to further decompose the signal into different subbands. This decomposition can be represented by a partition of a rectangle, called a *subband pattern*. The subband pattern for one pass of the filter bank is shown in Figure 8.4, with an example given in Figure 8.5.

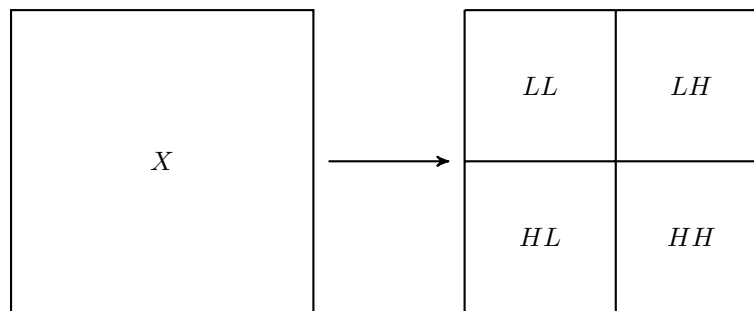


Figure 8.4: The subband pattern for one step in the 2-dimensional wavelet transform.

The wavelet coefficients that are obtained from a two-dimensional wavelet transform are very useful in a variety of image processing tasks. They allow images to be analyzed and manipulated in terms of both their frequency and spatial properties, and at differing levels of resolution. Furthermore, images are often represented in a very sparse manner by wavelets; that is, most of the image information is captured by a small subset of the wavelet coefficients. This is the key fact for wavelet-based image compression and will be discussed in further detail later in the lab.

## The PyWavelets Module

PyWavelets is a Python package designed for use in wavelet analysis. Although it has many other uses, in this lab it will primarily be used for image manipulation. PyWavelets can be installed using the following command:

```
$ pip install PyWavelets
```

PyWavelets provides a simple way to calculate the subbands resulting from one pass through the filter bank. The following code demonstrates how to find the approximation and detail subbands and plot them in a manner similar to Figure 8.5.

```
>>> from scipy.misc import imread
>>> import pywt                                     # The PyWavelets package.
# The True parameter produces a grayscale image.
>>> mandrill = imread('mandrill1.png', True)
```

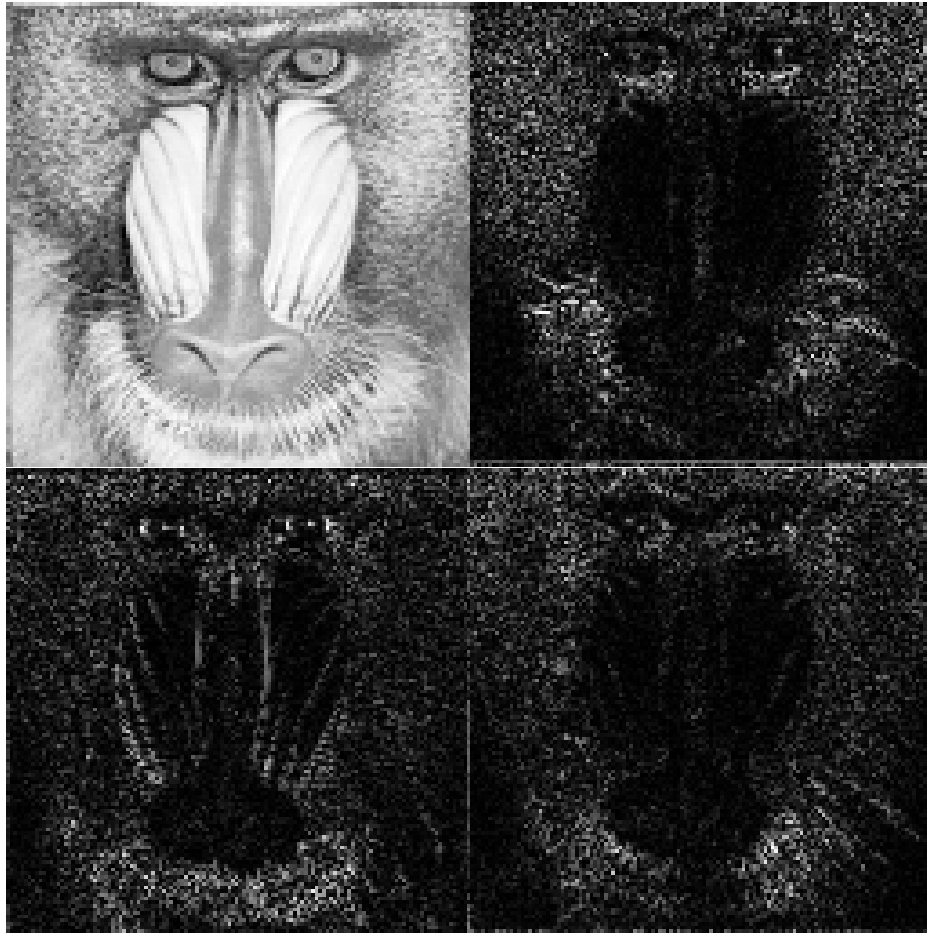


Figure 8.5: Subbands for the mandrill image after one pass through the filter bank. Note how the upper left subband ( $LL$ ) is an approximation of the original Mandrill image, while the other three subbands highlight the stark vertical, horizontal, and diagonal changes in the image. Original image source: <http://sipi.usc.edu/database/>.

```
# Use the Daubechies 4 wavelet with periodic extension.
>>> lw = pywt.dwt2(mandrill, 'db4', mode='per')
```

The function `pywt.dwt2()` calculates the subbands resulting from one pass through the filter bank. The `mode` keyword argument sets the extension mode, which determines the type of padding used in the convolution operation. For the problems in this lab, always use `mode='per'` which is the periodic extension. The second positional argument specifies the type of wavelet to be used in the transform. The function `dwt2()` returns a list. The first entry of the list is the  $LL$ , or approximation, subband. The second entry of the list is a tuple containing the remaining subbands,  $LH$ ,  $HL$ , and  $HH$  (in that order). These subbands can be plotted as follows:

```
>>> plt.subplot(221)
>>> plt.imshow(lw[0], cmap='gray')
>>> plt.axis('off')
>>> plt.subplot(222)
```

```
# The absolute value of the detail subbands is plotted to highlight contrast.
>>> plt.imshow(np.abs(lw[1][0]), cmap='gray')
>>> plt.axis('off')
>>> plt.subplot(223)
>>> plt.imshow(np.abs(lw[1][1]), cmap='gray')
>>> plt.axis('off')
>>> plt.subplot(224)
>>> plt.imshow(np.abs(lw[1][2]), cmap='gray')
>>> plt.axis('off')
>>> plt.subplots_adjust(wspace=0, hspace=0) # Remove space between plots.
```

As noted, the second positional argument is a string that gives the name of the wavelet to be used. PyWavelets supports a number of different wavelets which are divided into different classes known as families. The supported families and their wavelet instances can be listed by executing the following code:

```
>>> # List the available wavelet families.
>>> print(pywt.families())
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey', 'gaus', 'mexh', 'morl', '↵
cgau', 'shan', 'fbsp', 'cmor']
>>> # List the available wavelets in a given family.
>>> print(pywt.wavelist('coif'))
['coif1', 'coif2', 'coif3', 'coif4', 'coif5', 'coif6', 'coif7', 'coif8', 'coif9↵
', 'coif10', 'coif11', 'coif12', 'coif13', 'coif14', 'coif15', 'coif16', '↵
coif17']
```

Different wavelets have different properties; the most suitable wavelet is dependent on the specific application. The best wavelet to use in a particular application is rarely known beforehand. A choice about which wavelet to use can be partially based on the properties of a wavelet, but since many wavelets share desirable properties, the best wavelet for a particular application is often not known until some type of testing is done.

**Problem 3.** Explore the two-dimensional wavelet transform by completing the following:

1. Plot the subbands of the file `woman_darkhair.png` as described above (using the Daubechies 4 wavelet with periodic extension). Compare this with the subbands of the mandrill image shown in Figure 8.5.
2. Compare the subband patterns of different wavelets by plotting the *LH* subband pattern for the Haar wavelet and two other wavelets of your choice using `woman_darkhair.png`. Note that not all wavelets included in PyWavelets are compatible with every function. For example, only the first seven families listed by `pywt.families()` are compatible with `dwt2()`.

The function `pywt.wavedec2()` is similar to `pywt.dwt2()`; however it also includes a keyword argument `level`, which specifies the number of times to pass an image through the filter bank. It will return a list of subbands, the first of which is the final approximation subband, while the remaining elements are tuples which contain sets of detail subbands (*LH*, *HL*, and *HH*). If `level` is not



specified, the number of passes through the filter bank will be determined at runtime. The function `pywt.waverec2()` accepts a list of subband patterns (like the output of `pywt.wavedec2()` or `pywt.dwt2()`), a name string denoting the wavelet, and a keyword argument `mode` for the extension mode. It returns a reconstructed image using the reverse filter bank. When using this function, be sure that the wavelet and mode match the deconstruction parameters. PyWavelets has many other useful functions including `dwt()`, `idwt()` and `idwt2()` which can be explored further in the documentation for PyWavelets <http://pywavelets.readthedocs.io/en/latest/contents.html>.

## Applications

### Noise Reduction

Noise in an image can be defined as unwanted visual artifacts that obscure the true image. Images can acquire noise from a variety of sources, including the camera, data transfer, and image processing algorithms. This section will focus on reducing a particular type of random noise in images called *Gaussian white noise*.

An image that is distorted by Gaussian white noise is one in which every pixel has been perturbed by a small amount. Many types of noise, including Gaussian white noise, are very high-frequency. Since many images are relatively sparse in the high-frequency domains, noise in an image can be safely removed from the high frequency subbands without distorting the true image very much. A basic but effective approach to reducing Gaussian white noise in an image is thresholding.

Given a positive threshold value  $\tau$ , hard thresholding sets every wavelet coefficient whose magnitude is less than  $\tau$  to zero, while leaving the remaining coefficients untouched. Soft thresholding also zeros out all coefficients of magnitude less than  $\tau$ , but in addition maps the remaining positive coefficients  $\beta$  to  $\beta - \tau$  and the remaining negative coefficients  $\alpha$  to  $\alpha + \tau$ .

Once the coefficients have been thresholded, the inverse wavelet transform is used to recover the denoised image. The threshold value is generally a function of the variance of the noise, and in real situations, is not known. In fact, noise variance estimation in images is a research area in its own right, but that goes beyond the scope of this lab.

**Problem 4.** Write two functions, one of which implements the hard thresholding technique and one of which implements the soft. While writing these two functions, remember the following:

- The functions should accept a list of wavelet coefficients in the usual form, as well as a threshold value.
- The functions should return the thresholded wavelet coefficients (also in the usual form).
- Since only the detail coefficients are thresholded, the first entry of the input coefficient list should remain unchanged.

To test your functions, perform hard and soft thresholding on `noisy_darkhair.png` and plot the resulting images together. When testing your function, use the Daubechies 4 wavelet and four sets of detail coefficients (`level=4` when using `wavedec2()`). For soft thresholding use  $\tau = 20$ , and for hard thresholding use  $\tau = 40$ .

## Image Compression

Numerous image compression techniques have been developed over the years to reduce the cost of storing large quantities of images. Transform methods based on Fourier and wavelet analysis have long played an important role in these techniques; for example, the popular JPEG image compression standard is based on the discrete cosine transform. The JPEG2000 compression standard and the FBI Fingerprint Image database, along with other systems, take the wavelet approach.

The general framework for compression is fairly straightforward. First, the image to be compressed undergoes some form of preprocessing, depending on the particular application. Next, the discrete wavelet transform is used to calculate the wavelet coefficients, and these are then *quantized*, i.e. mapped to a set of discrete values (for example, rounded to the nearest integer). The quantized coefficients are then passed through an entropy encoder (such as Huffman Encoding), which reduces the number of bits required to store the coefficients. What remains is a compact stream of bits that can then be saved or transmitted much more efficiently than the original image. The steps above are nearly all invertible (the only exception being quantization), allowing the original image to be almost perfectly reconstructed from the compressed bitstream. See Figure 8.6.

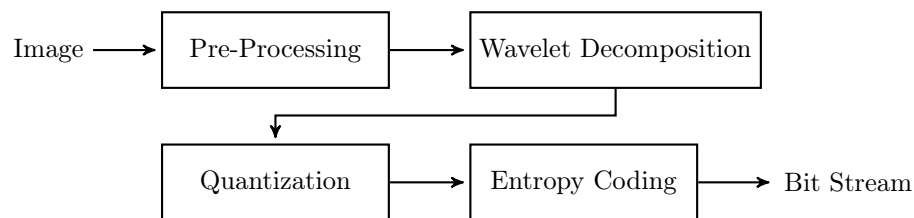


Figure 8.6: Wavelet Image Compression Schematic

## WSQ: The FBI Fingerprint Image Compression Algorithm

The Wavelet Scalar Quantization (WSQ) algorithm is among the first successful wavelet-based image compression algorithms. It solves the problem of storing millions of fingerprint scans efficiently while meeting the law enforcement requirements for high image quality. This algorithm is capable of achieving compression ratios in excess of 10-to-1 while retaining excellent image quality; see Figure 8.7. This section of the lab steps through a simplified version of this algorithm by writing a Python class that performs both the compression and decompression. Some of the difference between this simplified algorithm and the complete algorithm are found in the Additional Material section at the end of this lab. Also included in Additional Materials are the methods of the WSQ class that are needed to complete the algorithm.

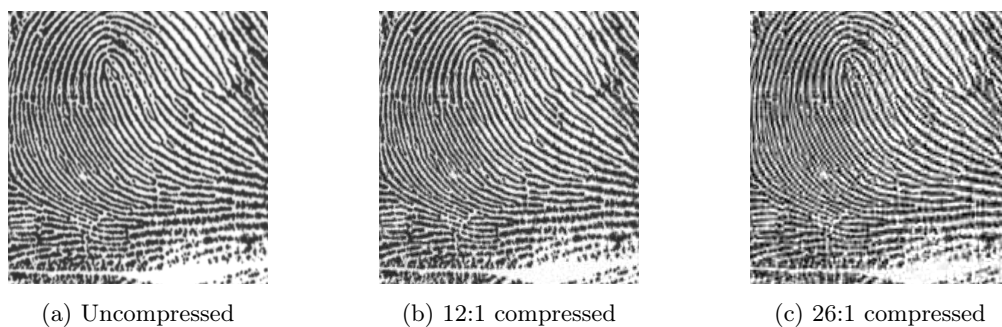


Figure 8.7: Fingerprint scan at different levels of compression.

Original image source: <http://www.nist.gov/itl/iad/ig/wsqa.cfm>.

### WSQ: Preprocessing

The input to the algorithm is a matrix of nonnegative 8-bit integer values giving the grayscale pixel values for the fingerprint image. The image is processed by the following formula:

$$M' = \frac{M - m}{s},$$

where  $M$  is the original image matrix,  $M'$  is the processed image,  $m$  is the mean pixel value, and  $s = \max\{\max(M) - m, m - \min(M)\}/128$  (here  $\max(M)$  and  $\min(M)$  refer to the maximum and minimum pixel values in the matrix). This preprocessing ensures that roughly half of the new pixel values are negative, while the other half are positive, and all fall in the range  $[-128, 128]$ .

**Problem 5.** Implement the preprocessing step as well as its inverse by implementing the class methods `pre_process()` and `post_process()`. These methods should accept a NumPy array (the image) and return the processed image as a NumPy array. In the `pre_process()` method, calculate the values of  $m$  and  $s$  given above. These values are needed later on for decompression, so store them in the class attributes `_m` and `_s`.

### WSQ: Calculating the Wavelet Coefficients

The next step in the compression algorithm is decomposing the image into subbands of wavelet coefficients. In this implementation of the WSQ algorithm, the image is decomposed into five sets of detail coefficients (`level=5`) and one approximation subband, as shown in Figure 8.8. Each of these subbands should be placed into a list in the same ordering as in Figure 8.8 (another way to consider this ordering is the approximation subband followed by each level of detail coefficients  $[LL_5, LH_5, HL_5, HH_5, LH_4, HL_4, \dots, HH_1]$ ).

**Problem 6.** Implement the subband decomposition as described above by implementing the class method `decompose()`. This function should accept an image to decompose and should return a list of ordered subbands. Use the function `pywt.wavedec2()` with the `'coif1'` wavelet to obtain the subbands. These subbands should then be ordered in a single list as described above.

Implement the inverse of the decomposition by writing the class method `recreate()`.

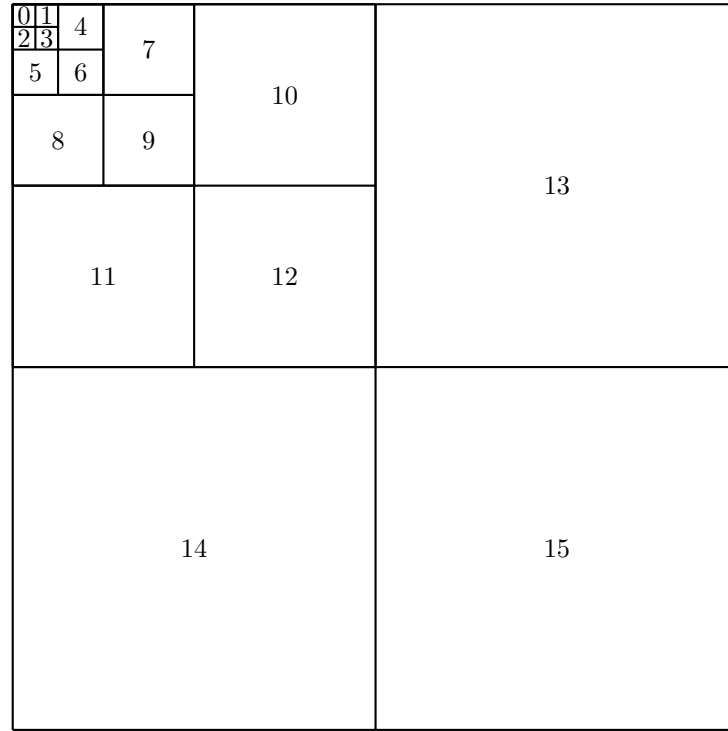


Figure 8.8: Subband Pattern for simplified WSQ algorithm.

This function should accept a list of 16 subbands (ordered like the output of `decompose()`) and should return a reconstructed image. Use `pywt.waverec2()` to reconstruct an image from the subbands. Note that you will need to adjust the accepted list in order to adhere to the required input for `waverec2()`.

### WSQ: Quantization

Quantization is the process of mapping each wavelet coefficient to an integer value, and is the main source of compression in the algorithm. By mapping the wavelet coefficients to a relatively small set of integer values, the complexity of the data is reduced, which allows for efficient encoding of the information in a bit string. Further, a large portion of the wavelet coefficients will be mapped to 0 and discarded completely. The fact that fingerprint images tend to be very nearly sparse in the wavelet domain means that little information is lost during quantization. Care must be taken, however, to perform this quantization in a manner that achieves good compression without discarding so much information that the image cannot be accurately reconstructed.

Given a wavelet coefficient  $a$  in subband  $k$ , the corresponding quantized coefficient  $p$  is given by

$$p = \begin{cases} \left\lfloor \frac{a - Z_k/2}{Q_k} \right\rfloor + 1, & a > Z_k/2 \\ 0, & -Z_k/2 \leq a \leq Z_k/2 \\ \left\lceil \frac{a + Z_k/2}{Q_k} \right\rceil - 1, & a < -Z_k/2. \end{cases}$$

The values  $Z_k$  and  $Q_k$  are dependent on the subband, and determine how much compression is achieved. If  $Q_k = 0$ , all coefficients are mapped to 0.

Selecting appropriate values for these parameters is a tricky problem in itself, and relies on heuristics based on the statistical properties of the wavelet coefficients. Therefore, the methods that calculate these values have already been initialized.

Quantization is not a perfectly invertible process. Once the wavelet coefficients have been quantized, some information is permanently lost. However, wavelet coefficients  $\hat{a}_k$  in subband  $k$  can be roughly reconstructed from the quantized coefficients  $p$  using the following formula. This process is called *dequantization*.

$$\hat{a}_k = \begin{cases} (p - C)Q_k + Z_k/2, & p > 0 \\ 0, & p = 0 \\ (p + C)Q_k - Z_k/2, & p < 0 \end{cases}$$

Note the inclusion of a new dequantization parameter  $C$ . Again, if  $Q_k = 0$ ,  $\hat{a}_k = 0$  should be returned.

**Problem 7.** Implement the quantization step by writing the `quantize()` method of your class. This method should accept a NumPy array of coefficients and the quantization parameters  $Q_k$  and  $Z_k$ . The function should return a NumPy array of the quantized coefficients.

Also implement the `dequantize()` method of your class using the formula given above. This function should accept the same parameters as `quantize()` as well as a parameter  $C$  which defaults to .44. The function should return a NumPy array of dequantized coefficients.

Masking and array slicing will help keep your code short and fast when implementing both of these methods. Remember the case for  $Q_k = 0$ . You can check that your functions are working correctly by comparing the output of your functions to a hand calculation on a small matrix.

## WSQ: The Rest

The remainder of the compression and decompression methods have already been implemented in the WSQ class. The following discussion explains the basics of what happens in those methods. Once all of the subbands have been quantized, they are divided into three groups. The first group contains the smallest ten subbands (positions zero through nine), while the next two groups contain the three subbands of next largest size (positions ten through twelve and thirteen through fifteen, respectively). All of the subbands of each group are then flattened and concatenated with the other subbands in the group. These three arrays of values are then mapped to Huffman indices. Since the wavelet coefficients for fingerprint images are typically very sparse, special indices are assigned to lists of sequential zeros of varying lengths. This allows large chunks of information to be stored as a single index, greatly aiding in compression. The Huffman indices are then assigned a bit string representation through a Huffman map. Python does not natively include all of the tools necessary to work with bit strings, but the Python package `bitstring` does have these capabilities. Download `bitstring` using the following command:

```
$ pip install bitstring
```

Import the package with the following line of code:

```
>>> import bitstring as bs
```

### WSQ: Calculating the Compression Ratio

The methods of compression and decompression are now fully implemented. The final task is to verify how much compression has taken place. The compression ratio is the ratio of the number of bits in the original image to the number of bits in the encoding. Assuming that each pixel of the input image is an 8-bit integer, the number of bits in the image is just eight times the number of pixels (recall that the number of pixels in the original source image is stored in the class attribute `_pixels`). The number of bits in the encoding can be calculated by adding up the lengths of each of the three bit strings stored in the class attribute `_bitstrings`.

**Problem 8.** Implement the method `get_ratio()` by calculating the ratio of compression. The function should not accept any parameters and should return the compression ratio.

Your compression algorithm is now complete! You can test your class with the following code:

```
# Try out different values of r between .1 to .9.
r = .5
finger = imread('uncompressed_finger.png', True)
wsq = WSQ()
wsq.compress(finger, r)
print(wsq.get_ratio())
new_finger = wsq.decompress()
plt.subplot(211)
plt.imshow(finger, cmap=plt.cm.Greys_r)
plt.subplot(212)
plt.imshow(np.abs(new_finger), cmap=plt.cm.Greys_r)
plt.show()
```



actually perform the compression and decompression, use the `_compress` and `_decompress` methods respectively. Note that all class attributes are set to `None` in `__init__`, but their values are initialized in the `compress` method.

Attributes:

```

    _pixels (int): Number of pixels in source image.
    _s (float): Scale parameter for image preprocessing.
    _m (float): Shift parameter for image preprocessing.
    _Q ((16, ), ndarray): Quantization parameters q for each subband.
    _Z ((16, ), ndarray): Quantization parameters z for each subband.
    _bitstrings (list): List of 3 BitArrays, giving bit encodings for
        each group.
    _tvals (tuple): Tuple of 3 lists of bools, indicating which
        subbands in each groups were encoded.
    _shapes (tuple): Tuple of 3 lists of tuples, giving shapes of each
        subband in each group.
    _huff_maps (list): List of 3 dictionaries, mapping huffman index to
        bit pattern.
"""

```

```
def __init__(self):
```

```

    self._pixels = None
    self._s = None
    self._m = None
    self._Q = None
    self._Z = None
    self._bitstrings = None
    self._tvals = None
    self._shapes = None
    self._huff_maps = None
    self._info loss = None

```

```
def compress(self, img, r, gamma=2.5):
```

```

    """The main compression routine. It computes and stores a bitstring
    representation of a compressed image, along with other values
    needed for decompression.

```

```
    Parameters:
```

```

        img ((m,n), ndarray): Numpy array containing 8-bit integer
            pixel values.
        r (float): Defines compression ratio. Between 0 and 1, smaller
            numbers mean greater levels of compression.
        gamma (float): A parameter used in quantization.
    """

```

```

    self._pixels = img.size    # Store image size.
    # Process then decompose image into subbands.
    mprime = self.pre_process(img)
    subbands = self.decompose(img)

```



```

# Calculate quantization parameters, quantize the image then group.
self._Q, self._Z = self.get_bins(subbands, r, gamma)
q_subbands = [self.quantize(subbands[i], self._Q[i], self._Z[i])
               for i in range(16)]
groups, self._shapes, self._tvals = self.group(q_subbands)

# Complete the Huffman encoding and transfer to bitstring.
huff_maps = []
bitstrings = []
for i in range(3):
    inds, freqs, extra = self.huffman_indices(groups[i])
    huff_map = huffman(freqs)
    huff_maps.append(huff_map)
    bitstrings.append(self.encode(inds, extra, huff_map))

# Store the bitstrings and the huffman maps.
self._bitstrings = bitstrings
self._huff_maps = huff_maps

def pre_process(self, img):
    """Preprocessing routine that takes an image and shifts it so that
    roughly half of the values are on either side of zero and fall
    between -128 and 128.

    Parameters:
        img ((m,n), ndarray): Numpy array containing 8-bit integer
            pixel values.

    Returns:
        ((m,n), ndarray): Processed numpy array containing 8-bit
            integer pixel values.
    """
    pass

def post_process(self, img):
    """Postprocess routine that reverses pre_process().

    Parameters:
        img ((m,n), ndarray): Numpy array containing 8-bit integer
            pixel values.

    Returns:
        ((m,n), ndarray): Unprocessed numpy array containing 8-bit
            integer pixel values.
    """
    pass

def decompose(self, img):
    """Decompose an image into the WSQ subband pattern using the

```

```

    Coiflet1 wavelet.

    Parameters:
        img ((m,n) ndarray): Numpy array holding the image to be
            decomposed.

    Returns:
        subbands (list): List of 16 numpy arrays containing the WSQ
            subbands in order.
    """
    pass

def recreate(self, subbands):
    """Recreate an image from the 16 WSQ subbands.

    Parameters:
        subbands (list): List of 16 numpy arrays containing the WSQ
            subbands in order.

    Returns:
        img ((m,n) ndarray): Numpy array, the image recreated from the
            WSQ subbands.
    """
    pass

def get_bins(self, subbands, r, gamma):
    """Calculate quantization bin widths for each subband. These will
    be used to quantize the wavelet coefficients.

    Parameters:
        subbands (list): List of 16 WSQ subbands.
        r (float): Compression parameter, determines the degree of
            compression.
        gamma(float): Parameter used in compression algorithm.

    Returns:
        Q ((16, ) ndarray): Array of quantization step sizes.
        Z ((16, ) ndarray): Array of quantization coefficients.
    """
    subband_vars = np.zeros(16)
    fracs = np.zeros(16)

    for i in range(len(subbands)): # Compute subband variances.
        X,Y = subbands[i].shape
        fracs[i]=(X*Y)/(np.float(finger.shape[0]*finger.shape[1]))
        x = np.floor(X/8.).astype(int)
        y = np.floor(9*Y/32.).astype(int)
        Xp = np.floor(3*X/4.).astype(int)
        Yp = np.floor(7*Y/16.).astype(int)

```

```

        mu = subbands[i].mean()
        sigsq = (Xp*Yp-1.)*(-1)*((subbands[i][x:x+Xp, y:y+Yp]-mu)**2).sum()
        subband_vars[i] = sigsq

A = np.ones(16)
A[13], A[14] = [1.32]*2

Qprime = np.zeros(16)
mask = subband_vars >= 1.01
Qprime[mask] = 10./(A[mask]*np.log(subband_vars[mask]))
Qprime[:4] = 1
Qprime[15] = 0

K = []
for i in range(15):
    if subband_vars[i] >= 1.01:
        K.append(i)

while True:
    S = fracs[K].sum()
    P = ((np.sqrt(subband_vars[K])/Qprime[K])**fracs[K]).prod()
    q = (gamma**(-1))*(2**(r/S-1))*(P**(-1./S))
    E = []
    for i in K:
        if Qprime[i]/q >= 2*gamma*np.sqrt(subband_vars[i]):
            E.append(i)
    if len(E) > 0:
        for i in E:
            K.remove(i)
        continue
    break

Q = np.zeros(16) # Final bin widths.
for i in K:
    Q[i] = Qprime[i]/q
Z = 1.2*Q

return Q, Z

def quantize(self, coeffs, Q, Z):
    """Implementation of a uniform quantizer which maps wavelet
    coefficients to integer values using the quantization parameters
    Q and Z.

    Parameters:
        coeffs ((m,n) ndarray): Contains the floating-point values to
            be quantized.
        Q (float): The step size of the quantization.

```

```

        Z (float): The null-zone width (of the center quantization bin).

Returns
    out ((m,n) ndarray): Numpy array of the quantized values.
"""
pass

def dequantize(self, coeffs, Q, Z, C=0.44):
    """Given quantization parameters, approximately reverses the
    quantization effect carried out in quantize().

    Parameters:
        coeffs ((m,n) ndarray): Array of quantized coefficients.
        Q (float): The step size of the quantization.
        Z (float): The null-zone width (of the center quantization bin).
        C (float): Centering parameter, defaults to .44.

    Returns:
        out ((m,n) ndarray): Array of dequantized coefficients.
    """
    pass

def group(self, subbands):
    """Split the quantized subbands into 3 groups.

    Parameters:
        subbands (list): Contains 16 numpy arrays which hold the
            quantized coefficients.

    Returns:
        gs (tuple): (g1,g2,g3) Each gi is a list of quantized coeffs
            for group i.
        ss (tuple): (s1,s2,s3) Each si is a list of tuples which
            contain the shapes for group i.
        ts (tuple): (t1,t2,t3) Each ti is a list of bools indicating
            which subbands were included.
    """
    g1 = [] # This will hold the group 1 coefficients.
    s1 = [] # Keep track of the subband dimensions in group 1.
    t1 = [] # Keep track of which subbands were included.
    for i in range(10):
        s1.append(subbands[i].shape)
        if subbands[i].any(): # True if there is any nonzero entry.
            g1.extend(subbands[i].ravel())
            t1.append(True)
        else: # The subband was not transmitted.
            t1.append(False)

    g2 = [] # This will hold the group 2 coefficients.

```

```

s2 = [] # Keep track of the subband dimensions in group 2.
t2 = [] # Keep track of which subbands were included.
for i in range(10, 13):
    s2.append(subbands[i].shape)
    if subbands[i].any(): # True if there is any nonzero entry.
        g2.extend(subbands[i].ravel())
        t2.append(True)
    else: # The subband was not transmitted.
        t2.append(False)

g3 = [] # This will hold the group 3 coefficients.
s3 = [] # Keep track of the subband dimensions in group 3.
t3 = [] # Keep track of which subbands were included.
for i in range(13,16):
    s3.append(subbands[i].shape)
    if subbands[i].any(): # True if there is any nonzero entry.
        g3.extend(subbands[i].ravel())
        t3.append(True)
    else: # The subband was not transmitted.
        t3.append(False)

return (g1,g2,g3), (s1,s2,s3), (t1,t2,t3)

def ungroup(self, gs, ss, ts):
    """Re-create the subband list structure from the information stored
    in gs, ss and ts.

    Parameters:
        gs (tuple): (g1,g2,g3) Each gi is a list of quantized coeffs
            for group i.
        ss (tuple): (s1,s2,s3) Each si is a list of tuples which
            contain the shapes for group i.
        ts (tuple): (t1,t2,t3) Each ti is a list of bools indicating
            which subbands were included.

    Returns:
        subbands (list): Contains 16 numpy arrays holding quantized
            coefficients.
    """
    subbands1 = [] # The reconstructed subbands in group 1.
    i = 0
    for j, shape in enumerate(ss[0]):
        if ts[0][j]: # True if the j-th subband was included.
            l = shape[0]*shape[1] # Number of entries in the subband.
            subbands1.append(np.array(gs[0][i:i+l]).reshape(shape))
            i += l
        else: # The j-th subband wasn't included, so all zeros.
            subbands1.append(np.zeros(shape))

```

```

subbands2 = [] # The reconstructed subbands in group 2.
i = 0
for j, shape in enumerate(ss[1]):
    if ts[1][j]: # True if the j-th subband was included.
        l = shape[0]*shape[1] # Number of entries in the subband.
        subbands2.append(np.array(gs[1][i:i+l]).reshape(shape))
        i += 1
    else: # The j-th subband wasn't included, so all zeros.
        subbands2.append(np.zeros(shape))

subbands3 = [] # the reconstructed subbands in group 3
i = 0
for j, shape in enumerate(ss[2]):
    if ts[2][j]: # True if the j-th subband was included.
        l = shape[0]*shape[1] # Number of entries in the subband.
        subbands3.append(np.array(gs[2][i:i+l]).reshape(shape))
        i += 1
    else: # The j-th subband wasn't included, so all zeros.
        subbands3.append(np.zeros(shape))

subbands1.extend(subbands2)
subbands1.extend(subbands3)
return subbands1

def huffman_indices(self, coeffs):
    """Calculate the Huffman indices from the quantized coefficients.

    Parameters:
        coeffs (list): Integer values that represent quantized
            coefficients.

    Returns:
        inds (list): The Huffman indices.
        freqs (ndarray): Array whose i-th entry gives the frequency of
            index i.
        extra (list): Contains zero run lengths and coefficient
            magnitudes for exceptional cases.
    """
    N = len(coeffs)
    i = 0
    inds = []
    extra = []
    freqs = np.zeros(254)

    # Sweep through the quantized coefficients.
    while i < N:

        # First handle zero runs.
        zero_count = 0

```

```

while coeffs[i] == 0:
    zero_count += 1
    i += 1
    if i >= N:
        break

if zero_count > 0 and zero_count < 101:
    inds.append(zero_count - 1)
    freqs[zero_count - 1] += 1
elif zero_count >= 101 and zero_count < 256: # 8 bit zero run.
    inds.append(104)
    freqs[104] += 1
    extra.append(zero_count)
elif zero_count >= 256: # 16 bit zero run.
    inds.append(105)
    freqs[105] += 1
    extra.append(zero_count)
if i >= N:
    break

# now handle nonzero coefficients
if coeffs[i] > 74 and coeffs[i] < 256: # 8 bit pos coeff.
    inds.append(100)
    freqs[100] += 1
    extra.append(coeffs[i])
elif coeffs[i] >= 256: # 16 bit pos coeff.
    inds.append(102)
    freqs[102] += 1
    extra.append(coeffs[i])
elif coeffs[i] < -73 and coeffs[i] > -256: # 8 bit neg coeff.
    inds.append(101)
    freqs[101] += 1
    extra.append(abs(coeffs[i]))
elif coeffs[i] <= -256: # 16 bit neg coeff.
    inds.append(103)
    freqs[103] += 1
    extra.append(abs(coeffs[i]))
else: # Current value is a nonzero coefficient in the range [-73, ←
    74].
    inds.append(179 + coeffs[i])
    freqs[179 + coeffs[i].astype(int)] += 1
    i += 1

return list(map(int, inds)), list(map(int, freqs)), list(map(int, extra))

def indices_to_coeffs(self, indices, extra):
    """Calculate the coefficients from the Huffman indices plus extra
    values.

```

```

Parameters:
    indices (list): List of Huffman indices.
    extra (list): Indices corresponding to exceptional values.

Returns:
    coeffs (list): Quantized coefficients recovered from the indices.
"""
coeffs = []
j = 0 # Index for extra array.

for s in indices:
    if s < 100: # Zero count of 100 or less.
        coeffs.extend(np.zeros(s+1))
    elif s == 104 or s == 105: # Zero count of 8 or 16 bits.
        coeffs.extend(np.zeros(extra[j]))
        j += 1
    elif s in [100, 102]: # 8 or 16 bit pos coefficient.
        coeffs.append(extra[j]) # Get the coefficient from the extra ←
                                list.
        j += 1
    elif s in [101, 103]: # 8 or 16 bit neg coefficient.
        coeffs.append(-extra[j]) # Get the coefficient from the extra ←
                                list.
        j += 1
    else: # Coefficient from -73 to +74.
        coeffs.append(s-179)
return coeffs

def encode(self, indices, extra, huff_map):
    """Encodes the indices using the Huffman map, then returns
    the resulting bitstring.

    Parameters:
        indices (list): Huffman Indices.
        extra (list): Indices corresponding to exceptional values.
        huff_map (dict): Dictionary that maps Huffman index to bit
            pattern.

    Returns:
        bits (BitArray object): Contains bit representation of the
            Huffman indices.
    """
    bits = bs.BitArray()
    j = 0 # Index for extra array.
    for s in indices: # Encode each huffman index.
        bits.append('0b' + huff_map[s])

        # Encode extra values for exceptional cases.
        if s in [104, 100, 101]: # Encode as 8-bit ints.

```



```

        bits.append('uint:8={}'.format(int(extra[j])))
        j += 1
    elif s in [102, 103, 105]: # Encode as 16-bit ints.
        bits.append('uint:16={}'.format(int(extra[j])))
        j += 1
    return bits

def decode(self, bits, huff_map):
    """Decodes the bits using the given huffman map, and returns
    the resulting indices.

    Parameters:
        bits (BitArray object): Contains bit-encoded Huffman indices.
        huff_map (dict): Maps huffman indices to bit pattern.

    Returns:
        indices (list): Decoded huffman indices.
        extra (list): Decoded values corresponding to exceptional indices.
    """
    indices = []
    extra = []

    # Reverse the huffman map to get the decoding map.
    dec_map = {v:k for k, v in huff_map.items()}

    # Wrap the bits in an object better suited to reading.
    bits = bs.ConstBitStream(bits)

    # Read each bit at a time, decoding as we go.
    i = 0 # The index of the current bit.
    pattern = '' # The current bit pattern.
    while i < bits.length:
        pattern += bits.read('bin:1') # Read in another bit.
        i += 1

        # Check if current pattern is in the decoding map.
        if pattern in dec_map:
            indices.append(dec_map[pattern]) # Insert huffman index.

            # If an exceptional index, read next bits for extra value.
            if dec_map[pattern] in (100, 101, 104): # 8-bit int or 8-bit ↵
                zero run length.
                extra.append(bits.read('uint:8'))
                i += 8
            elif dec_map[pattern] in (102, 103, 105): # 16-bit int or 16-↵
                bit zero run length.
                extra.append(bits.read('uint:16'))
                i += 16
            pattern = '' # Reset the bit pattern.

```

```

        return indices, extra

def decompress(self):
    """Return the uncompressed image recovered from the compressed
        bistring representation.

    Returns:
        img ((m,n) ndarray): The recovered, uncompressed image.
    """
    # For each group, decode the bits, map from indices to coefficients.
    groups = []
    for i in range(3):
        indices, extras = self.decode(self._bitstrings[i],
                                      self._huff_maps[i])
        groups.append(self.indices_to_coeffs(indices, extras))

    # Recover the subbands from the groups of coefficients.
    q_subbands = self.ungroup(groups, self._shapes, self._tvals)

    # Dequantize the subbands.
    subbands = [self.dequantize(q_subbands[i], self._Q[i], self._Z[i])
                 for i in range(16)]

    # Recreate the image.
    img = self.recreate(subbands)

    # Post-process, return the image.
    return self.post_process(img)

def get_ratio(self):
    """Calculate the compression ratio achieved.

    Returns:
        ratio (float): Ratio of number of bytes in the original image
                       to the number of bytes contained in the bitstrings.
    """
    pass

```

The following code includes the methods used in the WSQ class to perform the Huffman encoding.

```

# Helper functions and classes for the Huffman encoding portions of WSQ ↵
algorithm.

import queue
class huffmanLeaf():
    """Leaf node for Huffman tree."""
    def __init__(self, symbol):
        self.symbol = symbol

```

```

def makeMap(self, huff_map, path):
    huff_map[self.symbol] = path

def __str__(self):
    return str(self.symbol)

def __lt__(self, other):
    return False

class huffmanNode():
    """Internal node for Huffman tree."""
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def makeMap(self, huff_map, path):
        """Traverse the huffman tree to build the encoding map."""
        self.left.makeMap(huff_map, path + '0')
        self.right.makeMap(huff_map, path + '1')

    def __lt__(self, other):
        return False

def huffman(freqs):
    """
    Generate the huffman tree for the given symbol frequencies.
    Return the map from symbol to bit pattern.
    """
    q = queue.PriorityQueue()
    for i in range(len(freqs)):
        leaf = huffmanLeaf(i)
        q.put((freqs[i], leaf))
    while q.qsize() > 1:
        l1 = q.get()
        l2 = q.get()
        weight = l1[0] + l2[0]
        node = huffmanNode(l1[1], l2[1])
        q.put((weight, node))
    root = q.get()[1]
    huff_map = dict()
    root.makeMap(huff_map, '')
    return huff_map

```