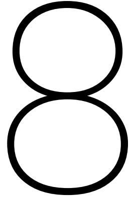


Introduction to Wavelets

Lab Objective: *Wavelets are used to sparsely represent information. This makes them useful in a variety of applications. We explore both the one- and two-dimensional discrete wavelet transforms using various types of wavelets. We then use a Python package called PyWavelets for further wavelet analysis including image cleaning and image compression.*

Wavelet Functions

Wavelets families are sets of orthogonal functions (wavelets) designed to decompose nonperiodic, piecewise continuous functions. These families have four types of wavelets: mother, daughter, father, and son functions. Father and son wavelets contain information related to the general movement of the function, while mother and daughter wavelets contain information related to the details of the function. The father and mother wavelets are the basis of a family of wavelets. Son and daughter wavelets are just scaled translates of the father and mother wavelets, respectively.

Haar Wavelets

The *Haar Wavelet* family is one of the most widely used wavelet families in wavelet analysis. This set includes the father, mother, son, and daughter wavelets defined below. The Haar father (scaling) function is given by

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

The Haar son wavelets are scaled and translated versions of the father wavelet:

$$\varphi_{jk}(x) = \varphi(2^j x - k) = \begin{cases} 1 & \text{if } \frac{k}{2^j} \leq x < \frac{k+1}{2^j} \\ 0 & \text{otherwise.} \end{cases}$$

The Haar mother wavelet function is defined as

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

The Haar daughter wavelets are scaled and translated versions of the mother wavelet

$$\psi_{jk} = \psi(2^j x - k)$$

Wavelet Decompositions

Information (such as a mathematical function or signal) can be stored and analyzed by considering its *wavelet decomposition*. A *wavelet decomposition* is a linear combination of wavelets. For example, a mathematical function f can be approximated as a combination of Haar son and daughter wavelets as follows:

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

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 the detail coefficients capture smaller details and noise.

A wavelet decomposition can be done with any family of wavelet functions. Depending on the properties of the wavelet and the function (or signal) f, f can be approximated to an arbitrary level of accuracy. Each arbitrary wavelet family has a mother wavelet ψ and a father wavelet φ which are the basis of the family. A countably infinite set of wavelet functions (daughter and son wavelets) can be generated using dilations and shifts of the first two functions where $m, k \in \mathbb{Z}$:

$$\psi_{m,k}(x) = \psi(2^m x - k)$$
$$\varphi_{m,k}(x) = \varphi(2^m x - k).$$

The Discrete Wavelet Transform

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.

In the case of finitely-sampled signals and images, there exists an efficient algorithm for computing the wavelet decomposition. 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 is similar to turning up the bass on a speaker, which extracts the low frequencies of a sound wave. This filter highlights the overall (slower-moving) pattern without paying too much attention to the high frequency details and extracts the approximation coefficients.

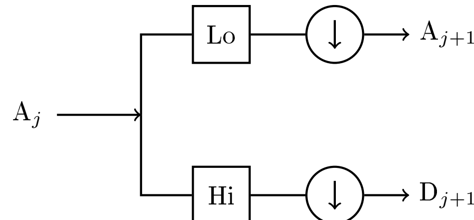
When the high-pass filter is convolved with the sampled signal, high frequency information (also known as detail) is extracted. This is similar to turning up the treble on a speaker, which extracts the high frequencies of a sound wave. This filter highlights the small changes found in the signal and extracts the detail coefficients.

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. The resulting arrays will be twice the size of the original signal because the frequency of the sample will have changed by a factor of 2. To remove this redundant information, the resulting arrays are *downsampled*. In the context of this lab, a *filter bank* is the combined process of convolving with a filter, and then downsampling. The result will be an array of approximation coefficients A and an array of detail coefficients D. This process can be repeated on the new approximation to obtain another layer of approximation and detail coefficients. See Figure 8.1.

A common lowpass filter is the averaging filter. Given an array **x**, the averaging filter produces an array **y** where y_n is the average of x_n and x_{n-1}. In other words, the averaging filter convolves an array with the array $L = \begin{bmatrix}\frac{1}{2} & \frac{1}{2}\end{bmatrix}$. This filter preserves the main idea of the data. The corresponding highpass filter is the distance filter. Given an array **x**, the distance filter produces an array **y** where y_n is the distance between x_n and x_{n-1}. In other words, the difference filter convolves an array with the array $H = \begin{bmatrix}-\frac{1}{2} & \frac{1}{2}\end{bmatrix}$. This filter preserves the details of the data.

For the Haar Wavelet, we will use the lowpass and highpass filters mentioned. In order for this filters to have inverses, the filters must be normalized (for more on why this is, see Additional Materials). The resulting lowpass and highpass filters for the Haar Wavelets are the following:

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



Key: □ = convolve (↓) = downsample

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

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. The array will be downsampled by a factor of 2, which means keeping only every other entry:

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

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. Implementation of a filter bank is found in Algorithm 8.1.

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** $i = 0 \ldots n-1$ **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** $A_n, D_n, D_{n-1}, \ldots, D_1$.

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: $[A_n, D_n, \ldots, D_1]$.

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.
(Hint: Use array broadcasting).

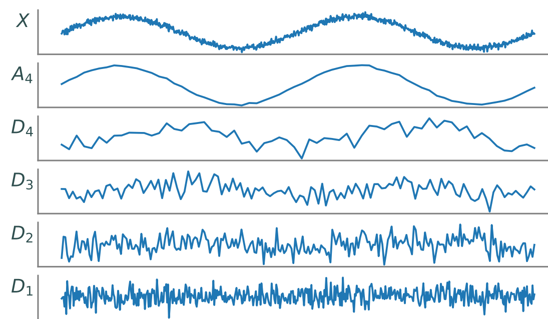


Figure 8.2: A level four wavelet decomposition of a signal. The top panel is the original signal, the next panel down is the approximation, and the remaining panels are the detail coefficients. Notice how the approximation resembles a smoothed version of the original signal, while the details capture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

Inverse Discrete Wavelet Transform

The process of the discrete wavelet transform is reversible. Using modified filters, a set of detail and approximation coefficients can be manipulated and combined to recreate a signal. The Haar wavelet filters for the inverse transformation are found by reversing the operations for each filter. The Haar inverse filters are given below:

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

The first row refers to the inverse high-pass filter and the second row refers to the inverse low-pass filter.

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 the inverse low-pass and high-pass filters, 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 convolved and 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 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 $D_n, D_{n-1}, \ldots, D_1$), 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()`.

Achtung!

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 a different 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.

The Two-dimensional Wavelet Transform

The generalization of the wavelet transform to two dimensions is similar to one dimensional transforms. Again, the two primary operations used 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, as in the one dimensional case. 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.

When implemented as an iterative filter bank, each pass through the filter bank yields one set of approximation coefficients plus three sets of detail coefficients. 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 detail coefficients that capture 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 additional 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 of an image decomposition given in Figure 8.5.

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

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

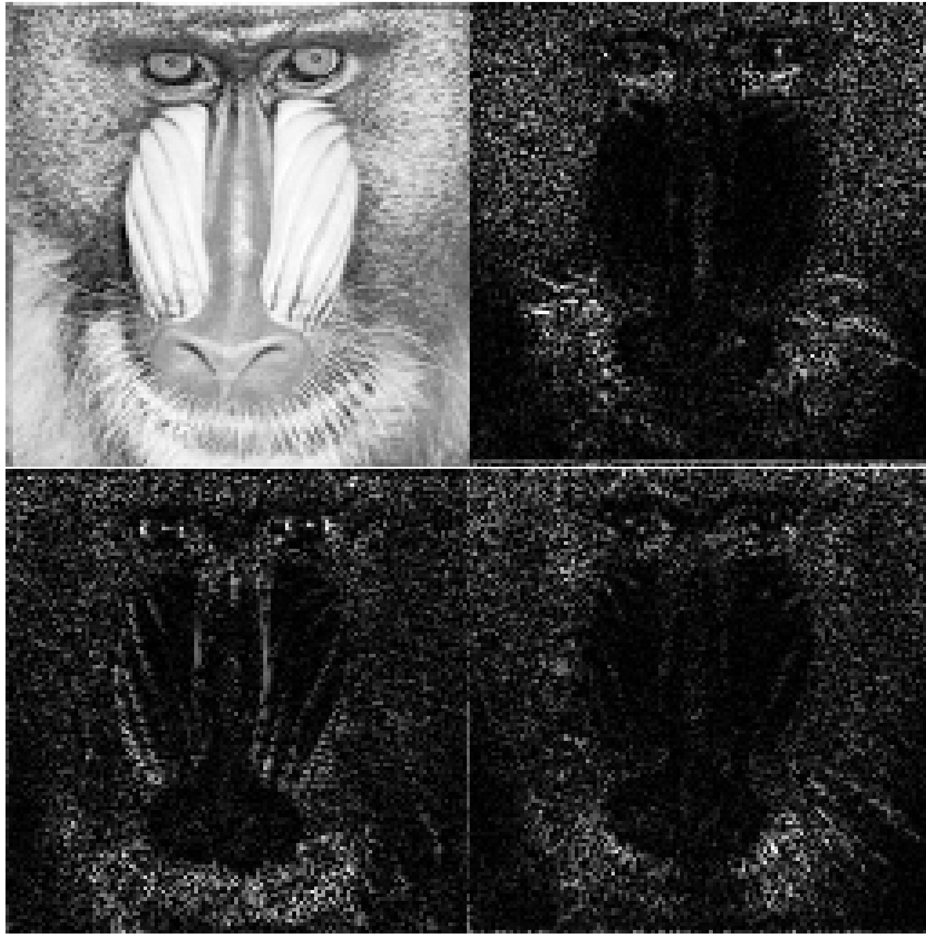


Figure 8.5: Subbands for the mandrill image after one pass through the filter bank. Note how the upper left subband (LL) is an approximation of the original Mandrill image, while the other three subbands highlight the stark vertical, horizontal, and diagonal changes in the image.
Original image source: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

The wavelet coefficients obtained from a two-dimensional wavelet transform are used to analyze and manipulate images at differing levels of resolution. Images are often sparsely represented by wavelets; that is, most of the image information is captured by a small subset of the wavelet coefficients. This is a 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 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 of an image.

```
>>> from imageio import imread
>>> import pywt                              # The PyWavelets package.
# The True parameter produces a grayscale image.
>>> mandrill = imread('mandrill1.png', True)
# 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 second positional argument specifies the type of wavelet to be used in the transform. 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 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).

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. For example, the morlet wavelet is closely related to human hearing and vision. Note that not all of these families work with the function `pywt.dwt2()`, because they are continuous wavelets. Choosing which wavelet is used is 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 without some type of testing.

Note

The numerical value in a wavelets name refers to the filter length. This value is multiplied by the standard filter length of the given wavelet, resulting in the new filter length. For example, `coif1` has filter length 6 and `coif2` has filter length 12.

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

1. Save a picture of a raccoon with the following code

```
>>> from scipy.misc import face
>>> racoon = face(True)
```

2. Plot the subbands of raccoon 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.
3. Compare the subband patterns of the haar, symlet, and coiflet wavelets of the raccoon picture by plotting the subbands after one pass through the filter bank. The haar subband should have more detail than the symlet subband, and the symlet subband should have more detail than the coiflet wavelet.

The function `pywt.wavedec2()` is similar to `pywt.dwt2()`, but 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 the maximum level where the decomposition is still useful. 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, `https://pywavelets.readthedocs.io/en/latest/index.html`.

Applications

Noise Reduction

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

Gaussian white noise causes every pixel in an image to be perturbed by a small amount. Many types of noise, including Gaussian white noise, are very high-frequency. Since many images are relatively sparse in high-frequency domains, noise in an image can be safely removed from the high frequency subbands while minimally distorting the true image. A basic, but effective, approach to reducing Gaussian white noise in an image is thresholding. Thresholding can be done in two ways, referred to as hard and soft thresholding.

Given a positive threshold value τ, hard thresholding sets every detail coefficient whose magnitude is less than τ to zero, while leaving the remaining coefficients untouched. Soft thresholding also zeros out all coefficients of magnitude less than τ, but in addition maps the remaining positive coefficients β to $\beta - \tau$ and the remaining negative coefficients α 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 that accept a list of wavelet coefficients in the usual form, as well as a threshold value. Each function returns the thresholded wavelet coefficients (also in the usual form). The first function should implement hard thresholding and the second should implement soft thresholding. While writing these two functions, remember that only the detail coefficients are thresholded, so 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

Transform methods based on Fourier and wavelet analysis play an important role in image compression; 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 as follows. 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 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.

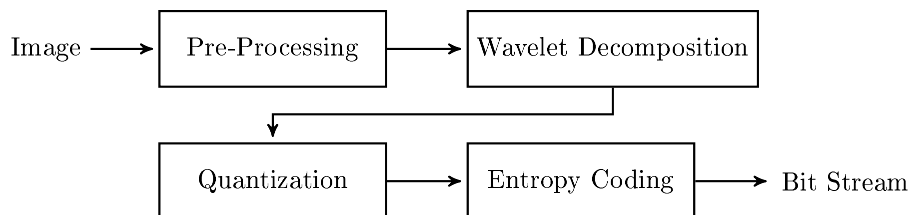
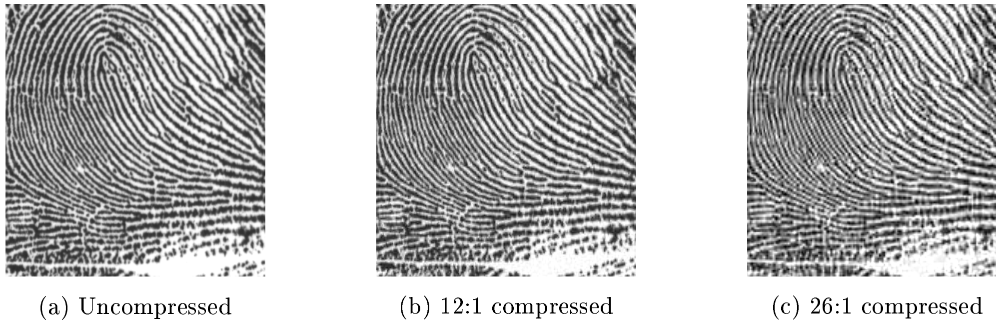


Figure 8.6: Wavelet Image Compression Schematic

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. Differences between this simplified algorithm and the complete algorithm are found in the Additional Material section at the end of this lab. Most of the methods of the class have already been implemented. The following problems will detail the methods you will need to implement yourself.



(a) Uncompressed (b) 12:1 compressed (c) 26:1 compressed

Figure 8.7: Fingerprint scan at different levels of compression. Original image source: `http://www.nist.gov/itl/iad/ig/wsq.cfm`.

WSQ: Preprocessing

Preprocessing in this algorithm ensures that roughly half of the new pixel values are negative, while the other half are positive, and all fall in the range $[-128, 128]$. 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).

Problem 5. Implement the preprocessing step as well as its inverse by implementing the class methods `pre_process()` and `post_process()`. Each method accepts a NumPy array (the image) and returns the processed image as a NumPy array. In the `pre_process()` method, calculate the values of m and s given above and 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 $[LL_5, LH_5, HL_5, HH_5, LH_4, HL_4, \ldots, HH_1]$).

Problem 6. Implement 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()`. 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()`.

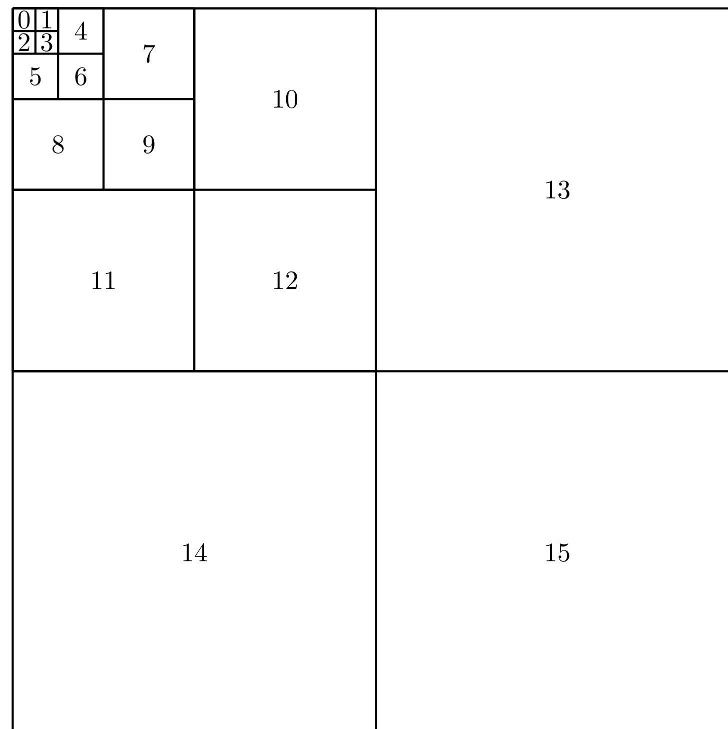


Figure 8.8: Subband Pattern for simplified WSQ algorithm.

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

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

where Z_k and Q_k are dependent on the subband. They 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. 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

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

where C is a new dequanitization parameter. This process is called *dequantization*. 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.

(Hint: Masking and array slicing will help keep your code short and fast when implementing both of these methods. Remember the case for $Q_k = 0$. Test your functions 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 original image is just eight times the number of pixels (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. The compression ratio should be approximately 18.

```
# 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()
```

Additional Material

Haar Wavelet Transform

The Haar Wavelet Transform is a general matrix transform used to convolve Haar Wavelets. It is found by combining the convolution matrices for a lowpass and highpass filter such that one is directly on top of the other. The lowpass filter is taking the average of every two elements in an array and the highpass filter is taking the difference of every two elements in an array. Redundant information given in the new matrix is then removed via downsampling. However, in order for the transform matrix to have the property $A^T = A^{-1}$, the columns of the matrix must be normalized. Thus, each column is normalized (and subsequently the filters) and the resulting matrix is the Haar Wavelet Transform.

For more on the Haar Wavelet Transform, see *Discrete Wavelet Transformations: An Elementary Approach with Applications* by Patrick J. Van Fleet.

WSQ Algorithm

The official standard for the WSQ algorithm is slightly different from the version implemented in this lab. One of the largest differences is the subband pattern that is used in the official algorithm; this pattern is demonstrated in Figure 8.9. The pattern used may seem complicated and somewhat arbitrary, but it is used because of the relatively good empirical results when used in compression. This pattern can be obtained by performing a single pass of the 2-dimensional filter bank on the image then passing each of the resulting subbands through the filter bank resulting in 16 total subbands. This same process is then repeated with the LL, LH and HL subbands of the original approximation subband creating 46 additional subbands. Finally, the subband corresponding to the top left of Figure 8.9 should be passed through the 2-dimensional filter bank a single time.

As in the implementation given above, the subbands of the official algorithm are divided into three groups. The subbands 0 through 18 are grouped together, as are 19 through 51 and 52 through 63. The official algorithm also uses a wavelet specialized for image compression that is not included in the PyWavelets distribution. There are also some slight modifications made to the implementation of the discrete wavelet transform that do not drastically affect performance.

0 1 4 7 8 19 20 23 24
2 3
5 6 9 10 21 22 25 26
52 53
11 12 15 16 27 28 31 32
13 14 17 18 29 30 33 34
35 36 39 40
37 38 41 42
51 54 55
43 44 47 48
45 46 49 50
56 57 60 61
58 59 62 63

Figure 8.9: True subband pattern for WSQ algorithm.