Lab 11

Introduction to Wavelets

Lab Objective: *In the context of Fourier analysis, one seeks to represent a function as a sum of sinusoids. A drawback to this approach is that the Fourier transform only captures global frequency information, and local information is lost; we can know which frequencies are the most prevalent, but not when or where they occur. The Wavelet transform provides an alternative approach that avoids this shortcoming and is often a superior analysis technique for many types of signals and images.*

The Discrete Wavelet Transform

In wavelet analysis, we seek to analyze a function by considering its *wavelet decomposition*. The wavelet decomposition of a function is a way of expressing the function as a linear combination of a particular family of basis functions. In this way, we can represent a function by the sequence of coefficients (called *wavelet coefficients*) defining this linear combination. The mapping from a function to its sequence of wavelet coefficients is called the *discrete wavelet transform*.

This situation is entirely analogous to the discrete Fourier transform. Instead of using trigonometric functions as our basis, we use a different family of basis functions. In Wavelet analysis, we determine the family of basis functions by first starting off with a function ψ called the *wavelet* and a function ϕ called the *scaling function* (these functions are also called the mother and father wavelets, respectively). We then generate countably many basis functions (sometimes called baby wavelets) from these two functions:

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

where $m, k \in \mathbb{Z}$. The historically first, and most basic, wavelet is called the *Haar Wavelet*, given by

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

The associated scaling function is given by

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

In the case of finitely-sampled signals and images, only finitely many wavelet coefficients are nonzero. Depending on the application, we are often only interested in the coefficients corresponding to a subset of the basis functions. Since a given family of wavelets forms an orthogonal set, we can compute the wavelet coefficients by taking inner products (i.e. by integrating). This direct approach is not particularly efficient, however. Just as there are fast algorithms for computing the fourier transform (e.g. the FFT), we can efficiently calculate wavelet coefficients using techniques from signal processing. In particular, we will use an *iterative filterbank* to compute the transform.

Let's launch into an implementation of the one-dimensional discrete wavelet transform. The key operations in the algorithm are the discrete convolution ($*$) and down-sampling (DS). The inputs to the algorithm are a one-dimensional array X (the signal that we want to transform), a one-dimensional array L (called the *low-pass filter*), a one-dimensional array H (the *high-pass filter*), and a positive integer n (controlling to what degree we wish to transform the signal, i.e. how many wavelet coefficients we wish to compute). The low-pass and high-pass filters can be derived from the wavelet and scaling function. The low-pass filter extracts low frequency information, which gives us an approximation of the signal. This approximation highlights the overall (slower-moving) pattern without paying too much attention to the high frequency details, which to the eye (or ear) may be unhelpful noise. However, we also need to extract the high-frequency details with the high-pass filter. While they may sometimes be nothing more than unhelpful noise, there are applications where they are the most important part of the signal; for example, details are very important if we are sharpening a blurry image or increasing contrast.

For the Haar Wavelet, our filters are given by

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

See Algorithm 11.1 and Figure 11.1 for the specifications.

Algorithm 11.1 The one-dimensional discrete wavelet transform.

```
1: procedure DWT(X, L, H, n)
2:     A_i ← X                                  ▷ Some initialization steps
3:     for i = 0 ... n − 1 do
4:         D_{i+1} ← DS(A_i ∗ H)                ▷ High-pass filtering
5:         A_{i+1} ← DS(A_i ∗ L)                ▷ Low-pass filtering
6:     return A_n, D_n, D_{n−1}, ..., D_1.
```

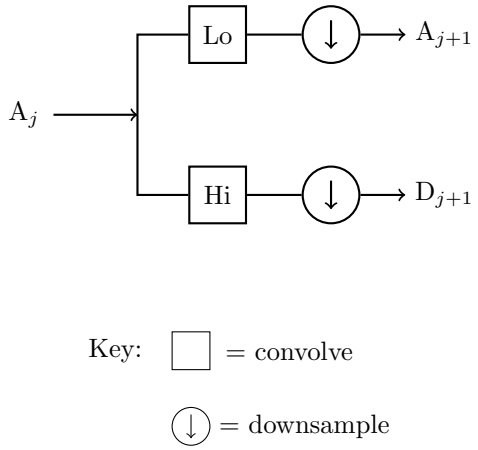

Figure 11.1: The one-dimensional discrete wavelet transform implemented as a filter bank.

At each stage of the algorithm, we filter the signal into an approximation and its details. Note that the algorithm returns a sequence of one dimensional arrays

$$A_n, D_n, D_{n-1}, \ldots, D_1.$$

If the input signal X has length 2^m for some $m \geq n$ and we are using the Haar wavelet, then A_n has length 2^{m-n}, and D_i has length 2^{m-i} for $i = 1, \ldots, n$. The arrays D_i are outputs of the high-pass filter, and thus represent high-frequency details. Hence, these arrays are known as *details*. The array A_n is computed by recursively passing the signal through the low-pass filter, and hence it represents the low-frequency structure in the signal. In fact, A_n can be seen as a smoothed approximation of the original signal, and is called the *approximation*.

As noted earlier, the key mathematical operations are convolution and down-sampling. To accomplish the convolution, we simply use a function in SciPy.

```
>>> import numpy as np
>>> from scipy.signal import fftconvolve
>>> # initialize the filters
>>> L = np.ones(2)/np.sqrt(2)
>>> H = np.array([-1,1])/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 us redundant information, so we down-sample to keep only what we need. In particular, we will down-sample by a factor of two, which means keeping only every other entry:

```
>>> # down-sample an array X
>>> sampled = X[1::2]
```

Putting these two operations together, we can obtain the approximation coefficients in one line of code:

```
>>> A = fftconvolve(X,L)[1::2]
```

Computing the detail coefficients is done in exactly the same way, replacing L with H.

Problem 1. Write a function that calculates the discrete wavelet transform as described above. The output should be a list of one-dimensional NumPy arrays in the following form: $[A_n, D_n, \dots, D_1]$.

The main body of your function should be a loop in which you calculate two arrays: the i-th approximation and detail coefficients. Append the detail coefficients array to your list, and feed the approximation array back into the loop. When the loop is finished, append the approximation array. Finally, reverse the order of your list to adhere to the required return format.

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

```
>>> 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 your results and verify that they match the plots in Figure 11.2.

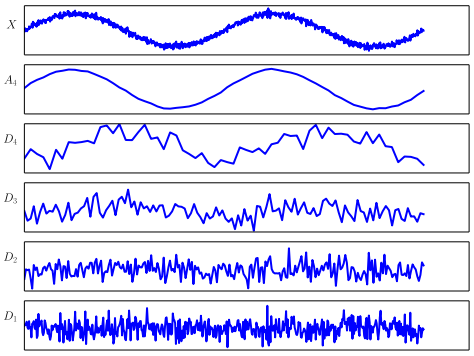

Figure 11.2: A level 4 wavelet decomposition of a signal. The top panel is the original signal, the next panel down is the approximation, and the remaining panels are the detail coefficients. Notice how the approximation resembles a smoothed version of the original signal, while the details capture the high-frequency oscillations and noise.

We can now transform a one-dimensional signal into its wavelet coefficients, but the reverse transformation is just as important. Luckily, we can reconstruct a signal from the approximation and detail coefficients. We reverse the effects of the filterbank, using slightly modified filters, essentially adding the details back into the signal at each stage until we reach the original. 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 we have the wavelet coefficients A_n and D_n. Consulting Figure 11.1, we can recreate A_{n-1} by tracing the schematic backwards: A_n and D_n are first *up-sampled*, then they are convolved with L and H, respectively, and finally added together to obtain A_{n-1}. Up-sampling means doubling the length of an array by inserting a 0 at every other position.

```
>>> # up-sample the coefficient arrays A, D
>>> up_A = np.zeros(2*A.size)
>>> up_A[::2] = A
>>> up_D = np.zeros(2*D.size)
>>> up_D[::2] = D
>>> # now convolve and add, but discard last entry
>>> A = fftconvolve(up_A,L)[:-1] + fftconvolve(up_D,H)[:-1]
```

Now that we have A_{n-1}, we repeat the process with A_{n-1} and D_{n-1} to obtain A_{n-2}. Proceed for a total of n steps (one for each $D_n, D_{n-1}, \ldots, D_1$) until we have obtained A_0. Since A_0 is defined to be the original signal, we have finished the inverse transformation.

Problem 2. Write a function that calculates the inverse wavelet transform as described above. The inputs should be a list of arrays (of the same form as the output of your discrete wavelet transform function), the low-pass filter, and the high-pass filter. The output should be a single array, 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.

In order to check your work, compute the discrete wavelet transform of a random array for different values of n, then compute the inverse transform. Compare the original signal with the recovered signal using `np.allclose`.

The PyWavelets Module

Having implemented our own version of the basic 1-dimensional wavelet transform, we now turn to PyWavelets, a Python library for Wavelet Analysis. It provides convenient and efficient methods to calculate the one- and two-dimensional discrete Wavelet transform, as well as much more.

If you have the Anaconda distribution, then you can install PyWavelets simply with the command:

```
$ conda install -c ioos pywavelets=0.4.0
```

Once the package has been installed on your machine, type the following to get started:

```
>>> import pywt
```

Performing the discrete Wavelet transform is very simple. Below, we compute the one-dimensional transform for a sinusoidal signal.

```
>>> import numpy as np
>>> f = np.sin(np.linspace(0,8*np.pi, 256)) # build the sine wave
>>> fw = pywt.wavedec(f, 'haar') # compute the wavelet coefficients of f
```

The variable `fw` is now a list of arrays, starting with the final approximation frame, followed by the various levels of detail coefficients, just like the output of the wavelet transform function that you already coded. Plot the level 2 detail and verify that it resembles a blocky sinusoid.

```
>>> from matplotlib import pyplot as plt
>>> plt.plot(fw[-2], linestyle='steps')
>>> plt.show()
```

To reconstruct the signal, we simply call the function `waverec`:

```
>>> f_prime = pywt.waverec(fw, 'haar') # reconstruct the signal
>>> np.allclose(f_prime, f) # compare with the original
True
```

The second positional argument, as you will notice, is a string that gives the name of the wavelet to be used. We first used the Haar wavelet, with which you are already familiar. PyWavelets supports a number of different Wavelets, however, which you can list by executing the following code:

```
>>> # list the available Wavelet families
>>> print pywt.families()
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey']
>>> # list the available wavelets in the coif family
>>> print pywt.wavelist('coif')
['coif1', 'coif2', 'coif3', 'coif4', 'coif5']
```

Different wavelets have different properties; the most suitable wavelet is dependent on the specific application. See Figure 11.3 for the plots of a couple of additional wavelets.

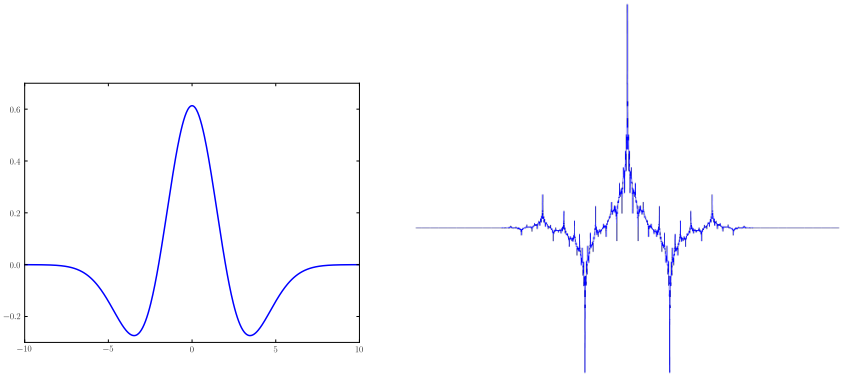

Figure 11.3: Examples of different mother wavelets.

The 2-dimensional Wavelet Transform

We can generalize the wavelet transform for two dimensions much as we generalized the Fourier transform. This allows us to perform wavelet analysis on, for example, digital images. In particular, we can calculate the wavelet transform of a two-dimensional array by first transforming the rows, and then the columns of the array.

When implemented as an iterative filterbank, each pass through the filterbank 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 filterbank, we obtain arrays LL, LH, HL, and HH, where LL is a smoothed approximation of X and the other three arrays contain wavelet coefficients capturing high-frequency oscillations in vertical, horizontal, and diagonal directions. In the jargon of signal processing, the arrays LL, LH, HL, and HH are called *subbands*. By recursively feeding any or all of the subbands back into the filterbank, we can decompose an input array into a collection of many subbands. This decomposition can be represented schematically by a dyadic partition of a rectangle, called a *subband pattern*. The subband pattern for one pass of the filterbank is shown in Figure 11.4, with a concrete example given in Figure 11.5.

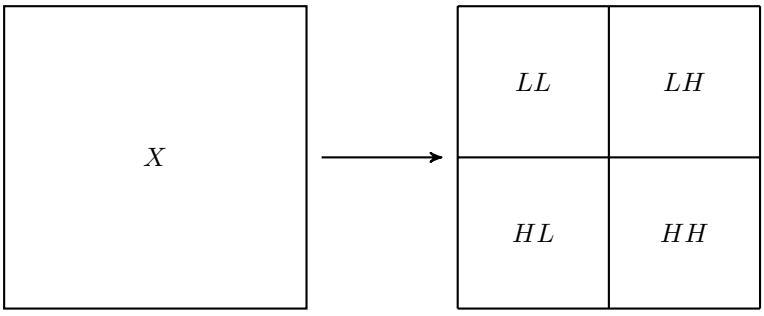

Figure 11.4: The subband pattern for one step in the 2-dimensional wavelet transform.

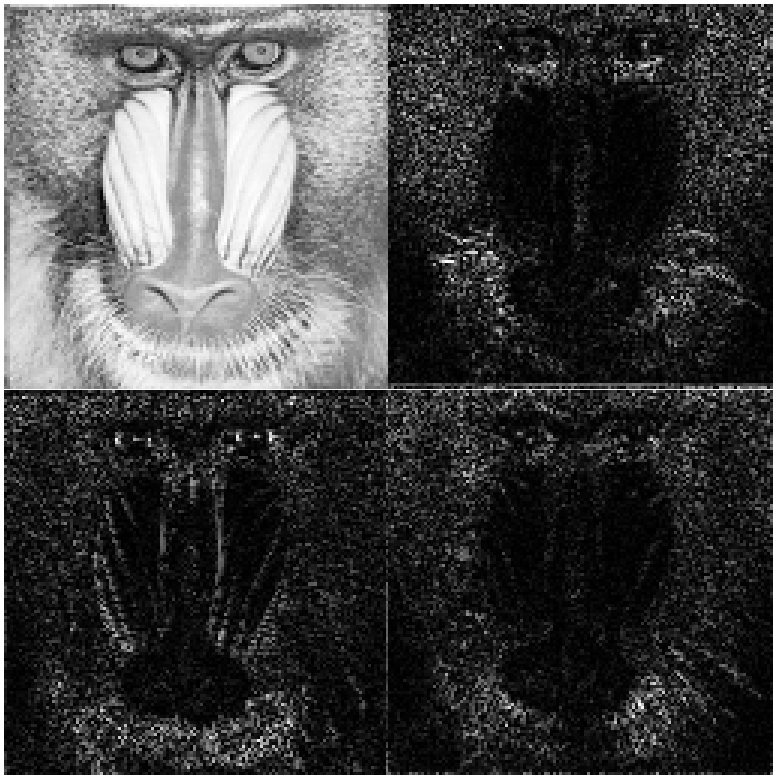

Figure 11.5: Subbands for the Mandrill image after one pass through the filterbank. Note how the upper left subband (LL) is an approximation of the original Mandrill image, while the other three subbands highlight the stark vertical, horizontal, and diagonal changes in the image.
Original image source: `http://sipi.usc.edu/database/`.

The wavelet coefficients that we obtain from a two-dimensional wavelet transform are very useful in a variety of image processing tasks. They allow us to analyze and manipulate images in terms of both their frequency and spatial properties, and at differing levels of resolution. Furthermore, wavelet bases often have the remarkable ability to represent images in a very *sparse* manner – 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.

PyWavelets provides a simple way to calculate the subbands resulting from one pass through the filterbank.

```
>>> from scipy.misc import imread
>>> # flag True produces a grayscale image
>>> mandrill = imread('mandrill1.png', True)
>>> # use the db4 wavelet with periodic extension
>>> lw = pywt.dwt2(mandrill, 'db4', mode='per')
```

Note that the `mode` keyword argument determines the type of extension mode (required for the convolution operation). The variable `lw` is 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). Plot these subbands as follows:

```
>>> plt.subplot(221)
>>> plt.imshow(np.abs(lw[0]), cmap='gray')
>>> plt.subplot(222)
>>> plt.imshow(np.abs(lw[1][0]), cmap='gray')
>>> plt.subplot(223)
>>> plt.imshow(np.abs(lw[1][1]), cmap='gray')
>>> plt.subplot(224)
>>> plt.imshow(np.abs(lw[1][2]), cmap='gray')
>>> plt.show()
```

Problem 3. Plot the subbands of the file `swanlake_polluted.png` as described above. Compare this with the subbands the mandrill image shown in Figure 11.5.

Image Processing

We are now ready to use the two-dimensional wavelet transform for image processing. Wavelets are especially good at filtering out high-frequency noise from an image. Just as we were able to pinpoint the noise added to the sine wave in Figure 11.2, the majority of the noise added to an image will be contained in the final LH, HL, and HH detail subbands of our wavelet decomposition. If we decompose our image and reconstruct it with all subbands except these final subbands, we will eliminate most of the troublesome noise while preserving the primary aspects of the image.

We perform this cleaning as follows:

```
image = imread(filename,True)
wavelet = pywt.Wavelet('haar')
WaveletCoeffs = pywt.wavedec2(image,wavelet)
new_image = pywt.waverec2(WaveletCoeffs[:-1], wavelet)
```

Problem 4. Write a function called `clean_image()` which accepts the name of a grayscale image file and cleans high-frequency noise out of the image. Load the image as an ndarray, and perform a wavelet decomposition using PyWavelets. Reconstruct the image using all subbands except the last set of detail coefficients, and return this cleaned image as an ndarray.

Additional Material

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, rounding 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 rounding), allowing us to almost perfectly reconstruct the image from the compressed bitstream. See Figure 11.6.

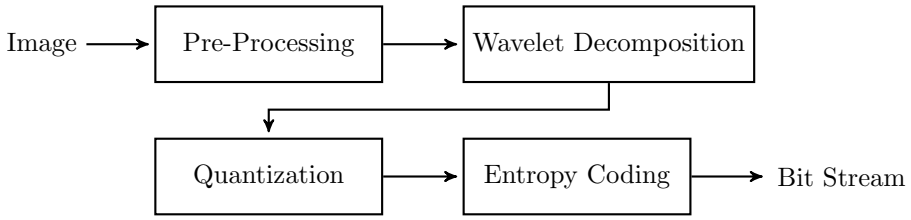

Figure 11.6: Wavelet Image Compression Schematic