10

Wavelet and Principal Components Analysis

A number of techniques can extend Fourier analysis to signals whose time-dependencies change in time. Part I of this chapter introduces wavelet analysis, a field that has seen extensive development and application in areas as diverse as brain waves, stock-market trends, gravitational waves, and compression of photographic images. Part II of this chapter covers the basics of principal components analysis. This is a powerful tool for situations in which there are very large data sets, and especially those with space-time correlated variables.

10.1 Part I: Wavelet Analysis

Problem You have sampled the signal in Figure 10.1 that seems to contain an increasing number of frequencies as time increases. Your **problem** is to undertake a spectral analysis of this signal that tells you, in the most compact way possible, the amount of each frequency present at each instant of time. *Hint*: Although we want the method to be general enough to work with numerical data, for pedagogical purposes it is useful to know that the signal is

$$y(t) = \begin{cases} \sin 2\pi t, & \text{for } 0 \le t \le 2, \\ 5\sin 2\pi t + 10\sin 4\pi t, & \text{for } 2 \le t \le 8, \\ 2.5\sin 2\pi t + 6\sin 4\pi t + 10\sin 6\pi t, & \text{for } 8 \le t \le 12. \end{cases} \tag{10.1}$$

The Fourier analysis we used in Chapter 9 reveals the amount of the harmonic functions $\sin(n\omega t)$ and $\cos(n\omega t)$ that are present in a signal. An expansion in periodic functions is fine for *stationary* signals (those whose forms do not change in time), but has shortcomings for the time dependence of our **problem** signal (10.1). One such problem is that the Fourier reconstruction has all frequencies $n\omega$ occurring simultaneously, and so does not contain *time resolution* information indicating when each frequency occurs. Another shortcoming is that all the Fourier components are correlated, and that results in more information being stored than is needed to reconstruct the signal.

There are a number of techniques that extend simple Fourier analysis to nonstationary signals. The idea behind wavelet analysis is to expand a signal in a complete set of functions (wavelets), each of which oscillates for a finite period of time, and each of which

Computational Physics: Problem Solving with Python, Fourth
Rubin H. Landau, Manuel J. Páez, and Cristian C. Bordeianu.

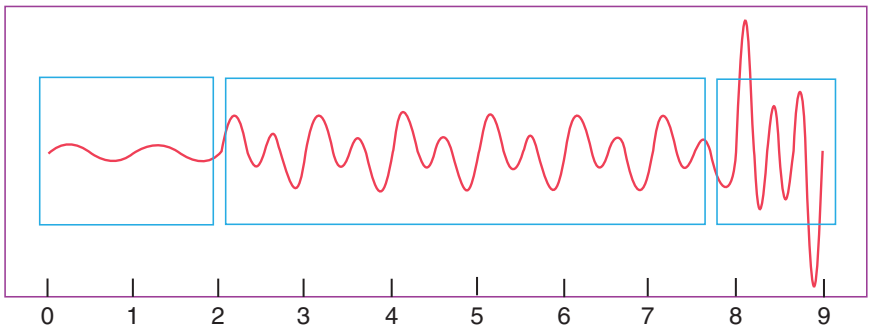

Figure 10.1 The input time signal (10.1) we wish to analyze. The signal is seen to contain additional frequencies as time increases. The boxes are possible placements of windows for short-time Fourier transforms.

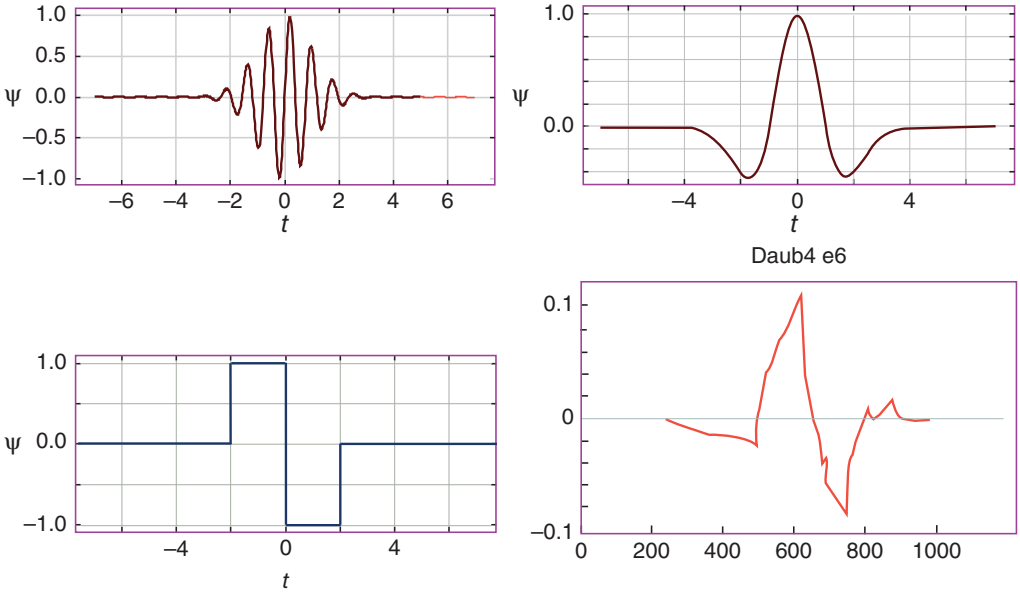

Figure 10.2 Four possible mother wavelets that can be used to generate entire sets of daughter wavelets. *Clockwise from top*: Morlet (real part), Mexican hat, Daub4 e6 (explained later), and Haar. The daughter wavelets are generated by scaling and translating these mother wavelets.

is centered at a different time. To give you a preview before we go into details, we show four sample wavelets in Figure 10.2. Because each wavelet is local in time, it is a wave packet,[1] with its time localization leading to a spectrum with a range of frequencies. These wave packets are called "wavelets" because they exist for only short periods of time [Polikar, 2023].

Although wavelets are required to oscillate in time, they are not restricted to a particular functional form [Addison, 2002; Goswani and Chan, 1999; Graps, 1995]. As a case in point, they may be oscillating Gaussian (Morlet: top left in Figure 10.2),

$$\Psi(t) = e^{2\pi i t} e^{-t^2/2\sigma^2} = (\cos 2\pi t + i \sin 2\pi t) e^{-t^2/2\sigma^2} \quad \text{(Morlet)}, \tag{10.2}$$

the second derivative of a Gaussian (Mexican hat, top right),

$$\Psi(t) = -\sigma^2 \frac{d^2}{dt^2} e^{-t^2/2\sigma^2} = \left(1 - \frac{t^2}{\sigma^2}\right) e^{-t^2/2\sigma^2}, \tag{10.3}$$

1 We discuss wave packets further in Section 10.2.

an up-and-down step function (lower left), or a fractal shape (bottom right). All of these wavelets are *localized* in both time and frequency, that is, they are large for just a finite time and contain a finite range of frequencies. As we shall see, translating and scaling these *mother wavelet* generates an entire set of *child wavelets* basis functions, with the children covering different frequency ranges at different times.

10.2 Wave Packets and Uncertainty Principle

A *wave packet* or *wave train* is a collection of waves of differing frequencies added together in such a way as to produce a pulse of width Δt. As we shall see, the Fourier transform of a wave packet is a pulse in the frequency domain of width $\Delta\omega$. We'll first study wave packets analytically, and then use them numerically. An example of a simple wave packet is a sine wave that oscillates at frequency ω_0 for N periods (Figure 10.3 left) [Arfken and Weber, 2001]:

$$y(t) = \begin{cases} \sin\omega_0 t, & \text{for} \quad |t| < N\frac{\pi}{\omega_0} \equiv N\frac{T}{2}, \\ 0, & \text{for} \quad |t| > N\frac{\pi}{\omega_0} \equiv N\frac{T}{2}, \end{cases} \tag{10.4}$$

where we relate the frequency to the period via the usual $\omega_0 = 2\pi/T$. In terms of these parameters, the width of the wave packet is

$$\Delta t = NT = N\frac{2\pi}{\omega_0}. \tag{10.5}$$

The Fourier transform of the wave packet (10.4) is a straightforward application of the transform formula (9.17):

$$\begin{aligned} Y(\omega) &= \int_{-\infty}^{+\infty} dt\, \frac{e^{-i\omega t}}{\sqrt{2\pi}} y(t) = \frac{-i}{\sqrt{2\pi}} \int_0^{N\pi/\omega_0} dt\, \sin\omega_0 t\, \sin\omega t \\ &= \frac{(\omega_0+\omega)\sin\left[(\omega_0-\omega)\frac{N\pi}{\omega_0}\right] - (\omega_0-\omega)\sin\left[(\omega_0+\omega)\frac{N\pi}{\omega_0}\right]}{\sqrt{2\pi}(\omega_0^2-\omega^2)}, \end{aligned} \tag{10.6}$$

where we have dropped a factor of $-i$ that affects only the phase. While at first glance (10.6) appears to be singular at $\omega = \omega_0$, it actually just peaks there (Figure 10.3 right), reflecting the predominance of frequency ω_0. Note that although the signal $y(t)$ appears to have only one frequency, it does drop off sharply in time (Figure 10.3 left), and these corners give $Y(\omega)$ a finite width $\Delta\omega$.

There is a fundamental relation between the widths Δt and $\Delta\omega$ of a wave packet. Although we use a specific example to determine that relation, it is true in general. While there may not be a precise definition of "width" for all functions, one can usually deduce a good measure of the width (say, within 25%). To illustrate, if we look at the right of Figure 10.3, it makes sense to use the distance between the first zeros of the transform $Y(\omega)$ (10.6) as the frequency width $\Delta\omega$. The zeros occur at

$$\frac{\omega-\omega_0}{\omega_0} = \pm\frac{1}{N} \Rightarrow \Delta\omega \simeq \omega - \omega_0 = \frac{\omega_0}{N}, \tag{10.7}$$

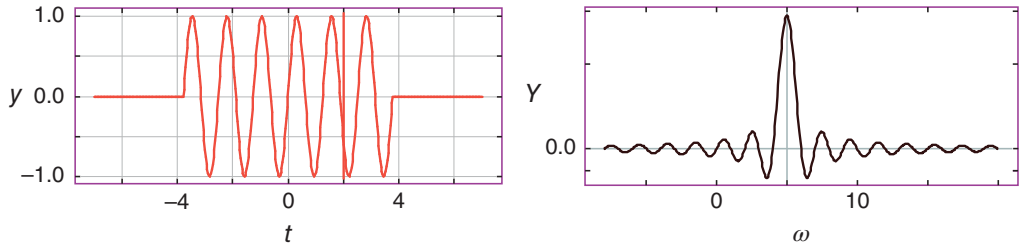

Figure 10.3 *Left*: A wave packet in time corresponding to the functional form (10.4) with $\omega_0 = 5$ and $N = 6$. *Right*: The Fourier transform in frequency of this same wave packet.

where N is the number of cycles in our original wave packet. Because the wave packet in time makes N oscillations each of period T, a reasonable measure of the time width Δt of the signal $y(t)$ is

$$\Delta t = NT = N\frac{2\pi}{\omega_0}. \tag{10.8}$$

When the products of the frequency width (10.7) and the time width (10.8) are combined, we obtain

$$\Delta t\, \Delta\omega \geq 2\pi. \tag{10.9}$$

The greater-than sign is used here to indicate that this is a minimum, that is, that $y(t)$ and $Y(\omega)$ extend beyond Δt and $\Delta\omega$, respectively. Nonetheless, most of the signal and transform should fall within the bounds of (10.9).

A relation of the form (10.9) also occurs in quantum mechanics, where it is known as the *Heisenberg uncertainty principle*, with Δt and $\Delta\omega$ called the uncertainties in t and ω. It is true for transforms in general, and states that as a signal is made more localized in time (smaller Δt), its transform becomes less localized (larger $\Delta\omega$). Conversely, the sine wave $y(t) = \sin\omega_0 t$ is completely localized in frequency, and consequently has an infinite extent in time, $\Delta t \simeq \infty$.

10.2.1 Wave Packet Exercise

Consider the following wave packets:

$$y_1(t) = e^{-t^2/2}, \quad y_2(t) = \sin(8t)e^{-t^2/2}, \quad y_3(t) = (1 - t^2)\, e^{-t^2/2}. \tag{10.10}$$

For each wave packet:

1) Estimate the width Δt. A good measure might be the *full width at half-maxima* (FWHM) of $|y(t)|$.
2) Use your DFT program to evaluate and plot the Fourier transform $Y(\omega)$ for each wave packet. Make *both* a linear and a semilog plot (small components are often important, yet not evident in linear plots). Make sure that your transform has a good number of closely spaced frequency values over a range that is large enough to show the periodicity of $Y(\omega)$.
3) What are the units for $Y(\omega)$ and ω in your DFT?
4) For each wave packet, estimate the width $\Delta\omega$. A good measure might be the *full width at half-maxima* of $|Y(\omega)|$.

5) For each wave packet determine approximate value for the constant C of the uncertainty principle

$$\Delta t\, \Delta\omega \geq 2\pi C. \tag{10.11}$$

10.3 Short-Time Fourier Transforms

The constant amplitude of the functions $\sin n\omega t$ and $\cos n\omega t$ for all times can limit the usefulness of Fourier analysis for reproducing signals whose form changes in time. Seeing that these basis functions extend over all times with a constant amplitude, there is considerable overlap among them, and thus the information present in various Fourier components are correlated. This is undesirable for data storage and compression, where you want to store a minimum amount of information, and also want to adjust the amount of information stored dependent on the desired quality of the reconstructed signal.[2] *Lossless compression* exactly reproduces the original signal. You can save space by storing how many times each data element is repeated, and where each element is located. In *lossy compression*, in addition to removing repeated elements, you also eliminate some transform components consistent with the uncertainty relation (10.9) and with the level of resolution required in the reproduction. This leads to even greater compression.

In Section 9.3 we defined the Fourier transform $Y(\omega)$ of signal $y(t)$ as

$$Y(\omega) = \int_{-\infty}^{+\infty} dt\, \frac{e^{-i\omega t}}{\sqrt{2\pi}}\, y(t) \equiv \langle \omega\, |y\rangle\,. \tag{10.12}$$

As is true for simple vectors, you can think of (10.12) as giving the overlap or scalar product of the basis function $|\omega\rangle = \exp(i\omega t)/\sqrt{2\pi}$ and the signal $y(t)$ [notice that the complex conjugate of the exponential basis function appears in (10.12)]. Another view of (10.12) is the mapping or projection of the signal into ω space. In this latter view, the overlap projects out the amount of the periodic function $\exp(i\omega t)/\sqrt{2\pi}$ in the signal $y(t)$. In other words, the Fourier component $Y(\omega)$ can be thought of as the correlation between the signal $y(t)$ and the basis function $\exp(i\omega t)/\sqrt{2\pi}$. This is the same as what results from filtering the signal $y(t)$ through a frequency filter. If there is no $\exp(i\omega t)$ in the signal, then the integral vanishes and there is no output. If $y(t) = \exp(i\omega t)$, the signal is at only one frequency, and the integral is accordingly singular.

The signal in Figure 10.1 for our problem clearly has different frequencies present at different times, and for different lengths of time. In the past, this signal might have been analyzed with a precursor of wavelet analysis known as the *short-time Fourier transform*. With that technique, the signal $y(t)$ is "chopped up" into different segments along the time axis, with successive segments centered about successive times $\tau_1, \tau_2, \ldots, \tau_N$. For instance, we show three such segments in the boxes of Figure 10.1. Once we have the dissected signal, a Fourier analysis is made for each segment. We are then left with a sequence of transforms $[Y^{(\mathrm{ST})}_{\tau_1}, Y^{(\mathrm{ST})}_{\tau_2}, \ldots, Y^{(\mathrm{ST})}_{\tau_N}]$, one for each short-time interval, where the superscript $^{(\mathrm{ST})}$ indicates short time.

---

2 Wavelets have proven to be a highly effective approach to data compression, with the Joint Photographic Experts Group (JPEG) 2000 standard being based on wavelets.

Rather than chopping up a signal by hand, we can express short-time Fourier transforming mathematically by imagining translating a *window function* $w(t-\tau)$, which is zero outside of some chosen interval, over the signal in Figure 10.1:

$$Y^{(\mathrm{ST})}(\omega,\tau)=\int_{-\infty}^{+\infty} dt\,\frac{e^{i\omega t}}{\sqrt{2\pi}}\,w(t-\tau)\,y(t). \tag{10.13}$$

Here the values of the translation time τ correspond to different locations of window w over the signal, and the window function is essentially a transparent box of small size on an opaque background. Any signal within the width of the window is transformed, while the signal lying outside the window is not seen. Note that in (10.13), the extra variable τ in the Fourier transform indicates the location of the time around which the window was placed. Clearly, because the short-time transform is a function of two variables, a surface or 3D plot is needed to view the amplitude as a function of both ω and τ.

10.4 Wavelet Transforms

The wavelet transform of a time signal $y(t)$ is defined as

$$Y(s,\tau)=\int_{-\infty}^{+\infty} dt\,\psi^{*}_{s,\tau}(t)y(t) \qquad \text{(wavelet transform)}, \tag{10.14}$$

and is similar in concept and notation to a short-time Fourier transform. The difference is rather than using $\exp(i\omega t)$ as the basis functions, here we are using wave packets or wavelets $\psi_{s,\tau}(t)$ localized in time, such as those shown in Figure 10.2. Because each wavelet is localized in time, each acts as its own window function. Because each wavelet is oscillatory, each contains its own limited range of frequencies.

Equation (10.14) says that the wavelet transform $Y(s,\tau)$ is a measure of the amount of basis function $\psi_{s,\tau}(t)$ present in the signal $y(t)$. The τ variable indicates the time portion of the signal being decomposed, while the s variable is equivalent to the frequency present during that time:

$$\omega=\frac{2\pi}{s}, \quad s=\frac{2\pi}{\omega} \qquad \text{(scale-frequency relation)}. \tag{10.15}$$

Seeing that it is key to much that follows, it is a good idea to think about (10.15) for a moment. If we are interested in the time *details* of a signal, then this is another way of saying that we are interested in what is happening at small values of the *scale* s. Equation (10.15) indicates that small values of s correspond to high-frequency components of the signal. That being the case, the time details of the signal are in the high-frequency, or low-scale, components.

10.4.1 Generating Wavelet Basis Functions

The conceptual discussion of wavelets is over, and it is time to get down to work. We first need a technique for generating wavelet basis functions, and then we need to discretize this

technique. As is often the case, the final formulation will turn out to be simple and short, but it will be a while before we get there.

Just as the expansion of an arbitrary function in a complete set of orthogonal functions is not restricted to any particular basis set, so too is the wavelet transform not restricted to any particular wavelet basis set, although some might be better than others for a given signal. The standard way to generate a family of wavelet basis functions starts with $\Psi(t)$, a *mother* or *analyzing* function of the real variable t, and then uses it to generate *daughter* wavelets. As a case in point, we start with the mother wavelet

$$\Psi(t) = \sin(8t)e^{-t^2/2}. \tag{10.16}$$

By scaling, translating, and normalizing this mother wavelet we obtain the set

$$\psi_{s,\tau}(t) \overset{\text{def}}{=} \frac{1}{\sqrt{s}}\Psi\left(\frac{t-\tau}{s}\right) = \frac{1}{\sqrt{s}}\sin\left[\frac{8(t-\tau)}{s}\right] e^{-(t-\tau)^2/2s^2}, \tag{10.17}$$

and with it we generate the four wavelet basis functions displayed in Figure 10.4. We see that larger or smaller values of s, respectively, expand or contract the mother wavelet, while different values of τ shift the center of the wavelet. Because the wavelets are inherently oscillatory, the scaling leads to the same number of oscillations occurring in different time spans, which is equivalent to having basis states with differing frequencies. We see that $s < 1$ produces a higher-frequency wavelet, while $s > 1$ produces a lower-frequency one, both of the same shape. As we shall see, we do not need to store much information to outline the large-time-scale s behavior of a signal (its *smooth envelope*), but we do need more information to specify its short-time-scale s behavior (*details*). And if we want to resolve yet finer features in the signal, then we will need to have more information on yet finer details. Here the division by $\sqrt{s}$ is made to ensure that there is equal "power" (or

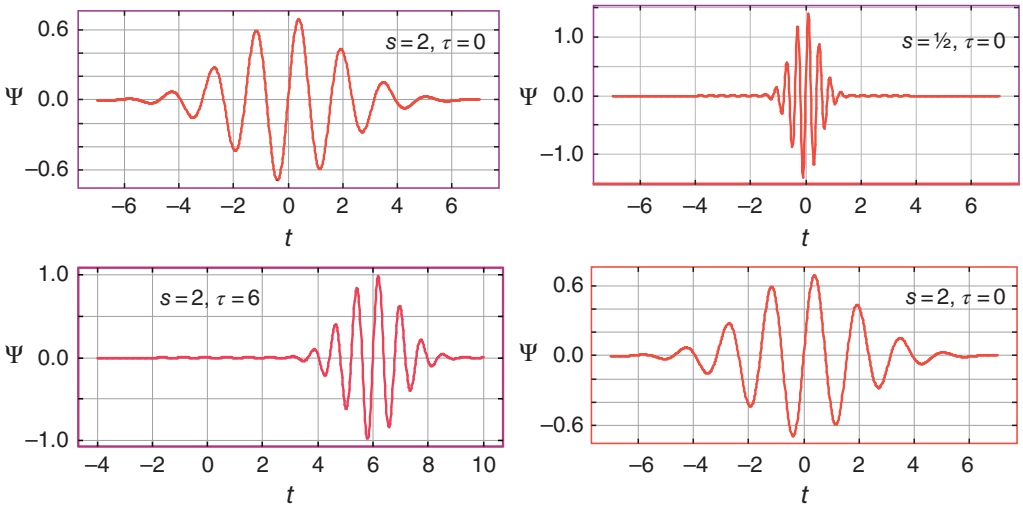

Figure 10.4 Four wavelet basis functions (daughters) generated by scaling (s) and translating (τ) an oscillating Gaussian mother wavelet. *Clockwise from top*: ($s = 1$, $\tau = 0$), ($s = 1/2$, $\tau = 0$), ($s = 1$, $\tau = 6$), and ($s = 2$, $\tau = 60$). Note how $s < 1$ is a wavelet with higher frequency, while $s > 1$ has a lower frequency than the $s = 1$ mother. Likewise, the $\tau = 6$ wavelet is just a translated version of the $\tau = 0$ one directly above it.

energy or intensity) in each region of s, although other normalizations can also be found in the literature. After substituting in the definition of daughters, the wavelet transform (10.14) and its inverse [van den Berg, 1999] are

$$Y(s,\tau) = \frac{1}{\sqrt{s}} \int_{-\infty}^{+\infty} dt\, \Psi^* \left(\frac{t-\tau}{s} \right) y(t) \qquad (\textbf{Wavelet Transform}), \tag{10.18}$$

$$y(t) = \frac{1}{C} \int_{-\infty}^{+\infty} d\tau \int_{0}^{+\infty} ds \frac{\psi^*_{s,\tau}(t)}{s^{3/2}}\, Y(s,\tau) \qquad (\textbf{Inverse Transform}), \tag{10.19}$$

where the normalization constant C depends on the wavelet used.

In summary, wavelet bases are functions of the time variable t, as well as of the two parameters s and τ. The t variable is integrated over to yield a transform that is a function of the time scale s (frequency $2\pi/s$) and window location τ. You can think of scale as being like the scale on a map (also discussed in Section 14.4.1 in relation to fractal analysis) or in terms of *resolution*, as might occur in photographic images. Regardless of the words, as we see in Chapter 14, if we have a fractal, then we have a self-similar object that looks the same at all scales or resolutions. Similarly, each wavelet in a set of basis functions is self-similar to the others, but at a different scale or location.

The general requirements for a mother wavelet Ψ are [Addison, 2002; van den Berg, 1999]:

1) $\Psi(t)$ is real.
2) $\Psi(t)$ oscillates around zero such that its average is zero:

$$\int_{-\infty}^{+\infty} \Psi(t)\, dt = 0. \tag{10.20}$$

3) $\Psi(t)$ is local, that is, a wave packet, and is square-integrable:

$$\Psi(|t| \to \infty) \to 0 \quad \text{(rapidly)}, \qquad \int_{-\infty}^{+\infty} |\Psi(t)|^2\, dt < \infty. \tag{10.21}$$

4) The transforms of low powers of t vanish, that is, the first p moments:

$$\int_{-\infty}^{+\infty} t^0\, \Psi(t)\, dt = \int_{-\infty}^{+\infty} t^1\, \Psi(t)\, dt = \cdots = \int_{-\infty}^{+\infty} t^{p-1}\, \Psi(t)\, dt = 0. \tag{10.22}$$

This makes the transform more sensitive to details than to general shape.

As an example of how we use the s and τ degrees of freedom in a wavelet transform, consider the analysis of a chirp signal $y(t) = \sin(60t^2)$ in Figure 10.5. We see that a slice at the beginning of the signal is compared to our first basis function. (The comparison is carried out via the *convolution* of the wavelet with the signal.) This first comparison is with a narrow version of the wavelet, that is, at low scale, and yields a single coefficient. The comparison at this scale continues with the next signal slice, and eventually ends when the entire signal has been covered (the top row in the figure). Then the wavelet is expanded to larger s values, and the comparisons are repeated. Eventually, the data are processed at all scales and at all time intervals. The narrow signals correspond to a high-resolution analysis, while the broad signals correspond to low resolution. As the scales get larger (lower frequencies, lower resolution), fewer details of the time signal remain visible, but the overall shape or gross features of the signal become clearer.

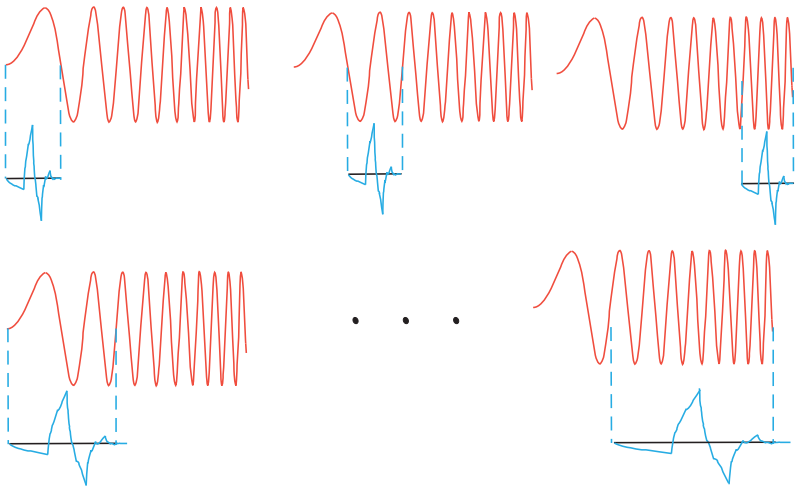

Figure 10.5 A schematic representation of the steps followed in performing a wavelet transformation over all time displacements and scales. The dark grey signal is first analyzed by evaluating its overlap with a narrow wavelet at the signal's beginning. This produces a coefficient that measures the similarity of the signal to the wavelet. The wavelet is successively shifted over the length of the signal and the overlaps are successively evaluated. After the entire signal is covered, the wavelet is expanded and the entire analysis is repeated.

10.4.2 Continuous Wavelet Transforms

We want to develop some intuition as to what wavelet transforms look like before going on to apply them. Accordingly, modify the program you have been using for the Fourier transform so that it now computes the wavelet transform. In contrast to the discrete or digital version, this is a *continuous* wavelet transform.

1) Examine the effect of using different mother wavelets. Accordingly, write a method that calculates the mother wavelet for
   a) a Morlet wavelet (10.2),
   b) a Mexican hat wavelet (10.3),
   c) a Haar wavelet (the square wave in Figure 10.2).
2) Try out your transform for the following input signals and see if the results make sense:
   a) A pure sine wave $y(t) = \sin 2\pi t$,
   b) A sum of sine waves $y(t) = 2.5 \sin 2\pi t + 6 \sin 4\pi t + 10 \sin 6\pi t$,
   c) The nonstationary signal for our problem (10.1)

$$y(t) = \begin{cases} \sin 2\pi t, & \text{for } 0 \le t \le 2, \\ 5 \sin 2\pi t + 10 \sin 4\pi t, & \text{for } 2 \le t \le 8, \\ 2.5 \sin 2\pi t + 6 \sin 4\pi t + 10 \sin 6\pi t, & \text{for } 8 \le t \le 12. \end{cases} \tag{10.23}$$

   d) The half-wave function

$$y(t) = \begin{cases} \sin \omega t, & \text{for } 0 < t < T/2, \\ 0, & \text{for } T/2 < t < T. \end{cases} \tag{10.24}$$

3) ⊙ Use (10.19) to invert your wavelet transform and compare the reconstructed signal to the input signal (you can normalize the two to each other). In Figure 10.6 we show our reconstruction.

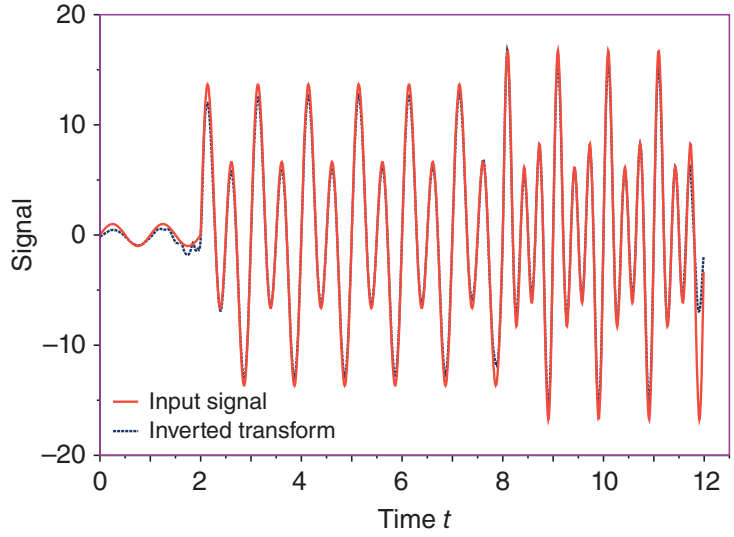

Figure 10.6 Comparison of an input and reconstituted signal (10.23) using Morlet wavelets. The curves overlap nearly perfectly, except at the ends.

In Listing 10.1 we give our *continuous wavelet transformation* `CWT.py` [Lang and Forinash, 1998]. Because wavelets, with their functional dependence on two variables, may be somewhat hard to grasp at first, we suggest that you write your own code and include a portion that does the inverse transform as a check. In Section 10.5, we will describe the *discrete wavelet transformation* that makes optimal discrete choices for the scale and time translation parameters s and τ. Figure 10.7 shows the spectrum produced for the input signal (10.1) in Figure 10.1. And it works! We see predominantly one frequency at short times, two frequencies at intermediate times, and three frequencies at longer times.

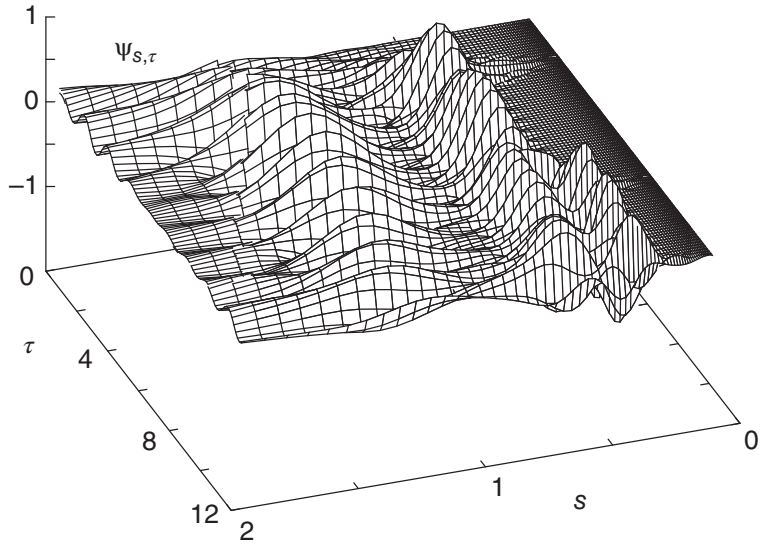

Figure 10.7 The continuous wavelet spectrum obtained by analyzing the input signal with Morlet wavelets. Observe how at small values of time τ there is predominantly one frequency present, how a second, higher-frequency (smaller-scale) component enters at intermediate times, and how at larger times a still higher-frequency components enter. (Figure courtesy of Z. Diabolic.)

10.5 Discrete Wavelet Transforms ⊙

As was true for DFTs, if a time signal is measured at only N discrete times,

$$y(t_m) \equiv y_m, \qquad m = 1, \ldots, N, \tag{10.25}$$

then we can determine only N-independent components of the transform Y. The trick is to remain consistent with the uncertainty principle as we compute only the N-independent components required to reproduce the signal. The *discrete wavelet transform* (DWT) evaluates the transforms with discrete values for the scaling parameter s and the time translation parameter τ:

$$\psi_{j,k}(t) = \frac{\Psi\left[(t - k2^j)/2^j\right]}{\sqrt{2^j}} \equiv \frac{\Psi\left(t/2^j - k\right)}{\sqrt{2^j}} \quad \text{(DWT)}, \tag{10.26}$$

$$s = 2^j, \qquad \tau = \frac{k}{2^j}, \qquad k,\, j = 0, 1, \ldots . \tag{10.27}$$

Here j and k are integers whose maximum values are yet to be determined, and we measure time in integer values. This choice of s and τ, based on powers of 2, is called a *dyadic grid* arrangement, and will be seen to automatically perform the scalings and translations at the different time scales that are at the heart of wavelet analysis.[3] The DWT now becomes

$$Y_{j,k} = \int_{-\infty}^{+\infty} dt\, \psi_{j,k}(t)\, y(t) \simeq \sum_m \psi_{j,k}(t_m) y(t_m) h \qquad \text{(DWT)}, \tag{10.28}$$

where the discreteness here refers to the wavelet basis set and *not* the time variable. For an orthonormal wavelet basis, the inverse discrete transform is then

$$y(t) = \sum_{j,\, k=-\infty}^{+\infty} Y_{j,k}\, \psi_{j,k}(t) \qquad \text{(inverse DWT)}. \tag{10.29}$$

This inversion will exactly reproduce the input signal at the N input points, but only if we sum over an infinite number of terms [Addison, 2002]. Practical calculations will be less exact.

Notice in (10.26) and (10.28) that we have kept the time variable t in the wavelet basis functions continuous, despite the fact that s and τ have been made discrete. This is useful in establishing the orthonormality of the basis functions,

$$\int_{-\infty}^{+\infty} dt\, \psi^*_{j,k}(t)\, \psi_{j',k'}(t) = \delta_{jj'}\, \delta_{kk'}, \tag{10.30}$$

where $\delta_{m,n}$ is the Kronecker delta function. Being normalized to 1 means that each wavelet basis has "unit energy"; being orthogonal means that each basis function is independent of the others. And because wavelets are localized in time, the different transform components have low levels of correlations with each other. Altogether, this leads to efficient and flexible data storage.

The use of a discrete wavelet basis makes it clear that we sample the input signal at the discrete values of time determined by the integers j and k. In general, you want time

3 Note that some references scale down with increasing j, in contrast to our scaling up.

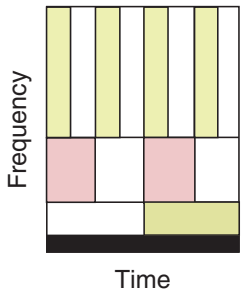

Figure 10.8 A graphical representation of the relation between time and frequency resolutions (the uncertainty relation). Each box represents an equal portion of the time-frequency plane but with different proportions of time and frequency.

steps that sample the signal at enough times in each interval to obtain the desired level of precision. A rule of thumb is to start with 100 steps to cover each major feature. Ideally, the needed times correspond to the times at which the signal was sampled, although this may require some forethought.

Consider Figure 10.8. We measure a signal at a number of discrete times within the intervals (k or τ values) corresponding to the vertical columns of fixed width along the time axis. For each time interval, we want to sample the signal at a number of scales (frequencies or j values). However, as discussed in Section 10.2, the basic mathematics of Fourier transforms indicates that the width Δt of a wave packet $\psi(t)$ and the width $\Delta\omega$ of its Fourier transform $Y(\omega)$ are related by an uncertainty principle

$$\Delta\omega\,\Delta t \geq 2\pi.$$

This relation places a constraint on the time intervals and frequency intervals. Furthermore, while we may want a high-resolution reproduction of our signal, we do not want to store more data than are needed to obtain that reproduction. If we sample the signal for times centered about some τ in an interval of width $\Delta\tau$ (Figure 10.8), and then compute the transform at a number of scales s or frequencies $\omega = 2\pi/s$ covering a range of height $\Delta\omega$, then the relation between the height and width is restricted by the uncertainty relation. All this means that each of the rectangles in Figure 10.8 has the same area $\Delta\omega\,\Delta t = 2\pi$. The increasing heights of the rectangles at higher frequencies means that a larger range of frequencies should be sampled as the frequency increases. The premise here is that the low-frequency components provide the gross or *smooth* outline of the signal which, being smooth, does not require much detail, while the high-frequency components give the details of the signal over a short time interval, and so require many components in order to record these details with high resolution.

Industrial-strength wavelet analyses do not compute explicit integrals, but instead apply a technique known as *multiresolution analysis* (MRA) [Mallat, 1989]. We give an example of this technique in Figure 10.9 and in the code `DWT.py` in Listing 10.2. It is based on a *pyramid algorithm* that samples the signal at a finite number of times, and then passes it successively through a number of *filters*, with each filter representing a digital version of a wavelet.

Filters were discussed in Chapter 9, where in (9.60) we defined the action of a linear filter as a convolution of the filter response function with the signal. A comparison of the definition of a filter to the definition of a wavelet transform (10.14), shows that the two

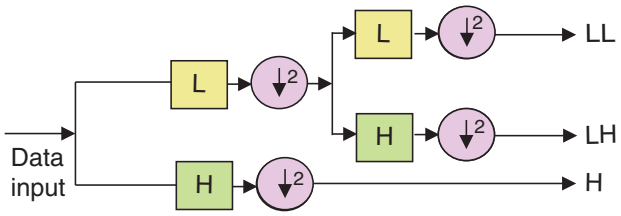

Figure 10.9 A eigenfrequency dyadic (power-of-2) filter tree used for discrete wavelet transformations. The L boxes represent lowpass filters and the H boxes represent highpass filters. Each filter performs a convolution (transform). The circles containing "↓ 2" filter out half of the signal that enters them, which is called *subsampling* or *factor-of-2 decimation*. The signal on the left yields a transform with a single low and two high components (less information is needed about the low components for a faithful reproduction).

are essentially the same. Such being the case, the result of the transform operation is a weighted sum over the input signal values, with each weight the product of the integration weight times the value of the wavelet function at the integration point. Therefore, *rather than tabulate explicit wavelet functions, a set of filter coefficients is all that is needed for DWT*.

In as much as each filter in Figure 10.9 changes the relative strengths of the different frequency components, passing the signal through a series of filters is equivalent, in wavelet language, to analyzing the signal at different scales. This is the origin of the name "multiresolution analysis." Figure 10.9 shows how the pyramid algorithm passes the signal through a series of highpass filters (H) and then through a series of lowpass filters (L). Each filter changes the scale to that of the level below. Notice too, the circles containing ↓ 2 in Figure 10.9. This operation filters out half of the signal and so is called *subsampling* or *factor-of-2 decimation*. It is the way we keep the areas of each box in Figure 10.8 constant as we vary the scale and translation times. We consider subsampling further when we discuss the pyramid algorithm.

In summary, the DWT process decomposes the signal into *smooth* information stored in the low-frequency components and *detailed* information stored in the high-frequency components. Because *high-resolution* reproductions of signals require more information about details than about gross shape, the pyramid algorithm is an effective way to compress data while still maintaining high resolution. In addition, because components of different resolutions are independent of each other, it is possible to lower the number of data stored by systematically eliminating higher-resolution components, if they are not needed. The use of wavelet filters builds in progressive scaling, which is particularly appropriate for fractal-like reproductions.

10.5.1 Pyramid Scheme ⊙

We now implement the pyramid scheme outlined in Figure 10.9. The H and L filters will be represented by matrices, which is an approximate way to perform the integrations or convolutions. Then there is a decimation of the output by one-half, and finally an interleaving of the output for further filtering. This process simultaneously cuts down on the number of points in the data set and changes the scale and the resolution. The decimation

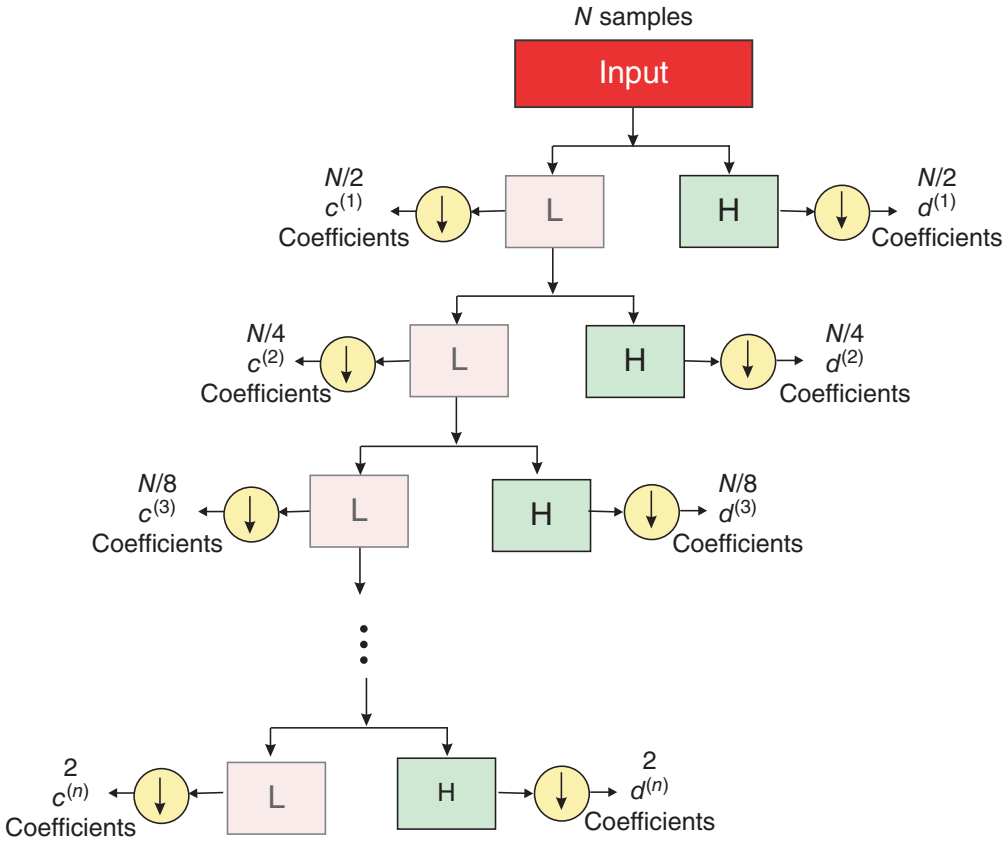

Figure 10.10 An input signal (top) is processed by a tree of high- and low-band filters. The outputs from each filtering are downshifted with half the data kept. The process continues until there are only two data of high-band filtering and two data of low-band filtering.

reduces the number of values of the remaining signal by one-half, with the low-frequency part discarded because the details are in the high-frequency parts.

As indicated in Figure 10.10, the pyramid DWT algorithm follows five steps:

1) Successively applies the (soon-to-be-derived) c matrix (10.41) to the whole N-length vector,

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & -c_0 & c_3 & -c_2 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}. \tag{10.31}$$

2) Applies it to the $N/2$-length smooth vector.
3) Repeats the application until only two smooth components remain.
4) After each filtering, the elements are ordered, with the newest two smooth elements on top, the newest detailed elements below, and the older detailed elements below that.
5) The process continues until there are just two smooth elements left.

To illustrate, here we filter and reorder an initial vector of length $N = 8$:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \overset{\text{filter}}{\rightarrow} \begin{bmatrix} s_1^{(1)} \\ d_1^{(1)} \\ s_2^{(1)} \\ d_2^{(1)} \\ s_3^{(1)} \\ d_3^{(1)} \\ s_4^{(1)} \\ d_4^{(1)} \end{bmatrix} \overset{\text{order}}{\rightarrow} \begin{bmatrix} s_1^{(1)} \\ s_2^{(1)} \\ s_3^{(1)} \\ s_4^{(1)} \\ \hline d_1^{(1)} \\ d_2^{(1)} \\ d_3^{(1)} \\ d_4^{(1)} \end{bmatrix} \overset{\text{filter}}{\rightarrow} \begin{bmatrix} s_1^{(2)} \\ d_1^{(2)} \\ s_2^{(2)} \\ d_2^{(2)} \\ \hline d_1^{(1)} \\ d_2^{(1)} \\ d_3^{(1)} \\ d_4^{(1)} \end{bmatrix} \overset{\text{order}}{\rightarrow} \begin{bmatrix} s_1^{(2)} \\ s_2^{(2)} \\ d_1^{(2)} \\ d_2^{(2)} \\ \hline d_1^{(1)} \\ d_2^{(1)} \\ d_3^{(1)} \\ d_4^{(1)} \end{bmatrix}. \tag{10.32}$$

The discrete inversion of a transform vector back to a signal vector is made using the transpose (inverse) of the transfer matrix at each stage. For instance,

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & -c_2 & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & -c_0 & c_1 & -c_2 \end{bmatrix} \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{bmatrix}. \tag{10.33}$$

As a more realistic example, imagine that we have sampled the chirp signal $y(t) = \sin(60t^2)$ for 1024 times. The filtering process through which we place this signal is illustrated as a passage from the top to the bottom in Figure 10.10. First the original 1024 samples are passed through a single low band and a single high band (which is mathematically equivalent to performing a series of convolutions). As indicated by the down arrows, the output of the first stage is then downshifted, that is, the number is reduced by a factor of 2. This results in 512 points from the high-band filter as well as 512 points from the low-band filter. This produces the first-level output. The output coefficients from the high-band filters are called $\{d_i^{(1)}\}$ to indicate that they show details, and $\{s_i^{(1)}\}$ to indicate that they show smooth features. The superscript indicates that this is the first level of processing. The detail coefficients $\{d^{(1)}\}$ are stored to become part of the final output.

In the next level down, the 512 smooth data $\{s_i^{(1)}\}$ are passed through new low- and high-band filters using a broader wavelet. The 512 outputs from each are downshifted to form a smooth sequence $\{s_i^{(2)}\}$ of size 256 and a detailed sequence $\{d_i^{(2)}\}$ of size 256. Again the detail coefficients $\{d^{(2)}\}$ are stored to become part of the final output. (Note that this is only half the size of the previously stored details.) The process continues until there are only two numbers left for the detail coefficients and two numbers left for the smooth coefficients. Because this last filtering is carried out with the broadest wavelet, it is of the lowest resolution and therefore requires the least information.

In Figure 10.11, we show the actual effects on the chirp signal of pyramid filtering for various levels in the processing. (The processing is carried out with *Daub4* wavelets, which we will discuss soon.) At the uppermost level, the wavelet is narrow, and so convoluting this wavelet with successive sections of the signal results in smooth components that still

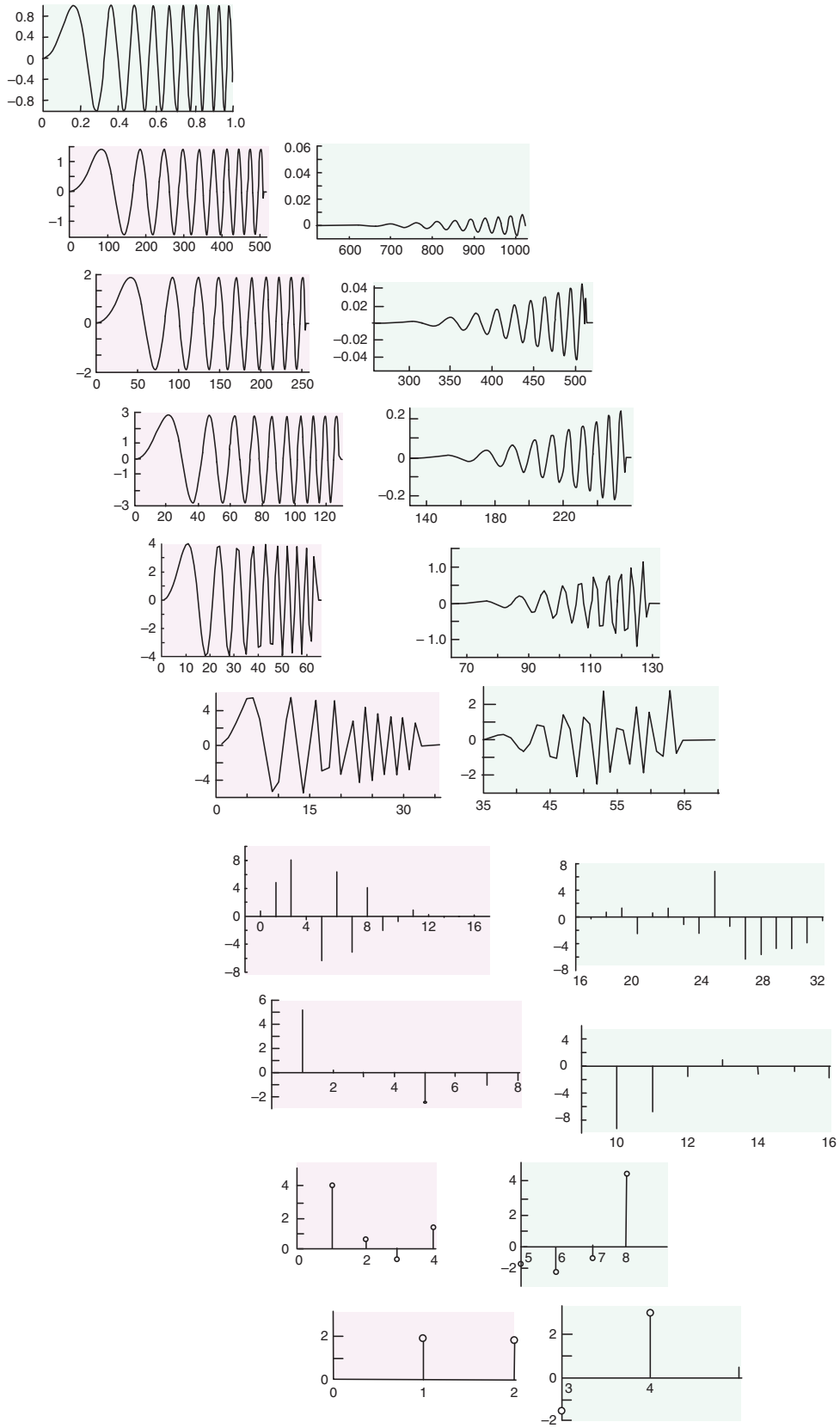

Figure 10.11 In successive passes, the filtering of the original signal at the top goes through the pyramid algorithm and produces the outputs shown. The sampling is reduced by a factor of 2 in each step. Note that in the upper graphs, we have connected the points to emphasize their continuous nature while in the lower graphs, we plot the individual output points as histograms.

contain many large high-frequency parts. The detail components, in contrast, are much smaller in magnitude. In the next stage, the wavelet is dilated to a lower frequency, and the analysis is repeated *on just the smooth (low-band) part*. The resulting output is similar, but with coarser features for the smooth coefficients and larger values for the details. Note that in the upper graphs we have connected the points to make the output look continuous, while in the lower graphs, with fewer points, we have plotted the output as histograms to make the points more evident. Eventually the downshifting leads to just two coefficients output from each filter, at which point the filtering ends.

To reconstruct the original signal (called *synthesis* or *transformation*) a reversed process is followed: Begin with the last sequence of four coefficients, upsample them, pass them through low- and high-band filters to obtain new levels of coefficients, and repeat until all the N values of the original signal are recovered. The inverse scheme is the same as the processing scheme (Figure 10.10), only now the direction of all the arrows is reversed.

10.5.2 Daubechies Wavelets Filters ⊙

We should now be able to understand that digital wavelet analysis has been standardized to the point where classes of wavelet basis functions are specified not by their analytic forms, but rather by their *wavelet filter coefficients*. In 1988, the Belgian mathematician Ingrid Daubechies discovered an important class of such filter coefficients [Daubechies, 1995; Rowe and Abbott, 1995]. We will study just the Daub4 class containing the four coefficients c_0, c_1, c_2, and c_3.

Imagine that our input contains the four elements $\{y_1, y_2, y_3, y_4\}$ corresponding to measurements of a signal at four times. We represent a lowpass filter L and a highpass filter H in terms of the four filter coefficients as

$$L = \begin{bmatrix} c_0 & +c_1 & c_2 & +c_3 \end{bmatrix}, \tag{10.34}$$

$$H = \begin{bmatrix} c_3 & -c_2 & c_1 & -c_0 \end{bmatrix}. \tag{10.35}$$

To see how this works, we form an input vector by placing the four signal elements in a column and then multiply the input by L and H:

$$L \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = c_0 y_0 + c_1 y_1 + c_2 y_2 + c_3 y_3,$$

$$H \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} c_3 & -c_2 & c_1 & -c_0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = c_3 y_0 - c_2 y_1 + c_1 y_2 - c_0 y_3.$$

We see that if we choose the values of the c_i's carefully, the result of L acting on the signal vector is a single number that may be viewed as a weighted average of the four input signal elements. Since an averaging process tends to smooth out data, the lowpass filter may be thought of as a *smoothing filter* that outputs the general shape of the signal.

In turn, we see that if we choose the c_i values carefully, the result of H acting on the signal vector is a single number that may be viewed as the weighted differences of the input signal.

Because a differencing process tends to emphasize the variation in the data, the highpass filter may be thought of as a *detail* filter that produces a large output when the signal varies considerably, and a small output when the signal is smooth.

We have just seen how the individual L and H filters, each represented by a single row of the filter matrix, outputs one number when acting upon an input signal containing four elements in a column. If we want the output of the filtering process Y to contain the same number of elements as the input (four y's in this case), we just stack the L and H filters together:

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} L \\ H \\ L \\ H \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & -c_0 & c_3 & -c_2 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}. \tag{10.36}$$

Of course the first and third rows of the Y vector will be identical, as will the second and fourth, but we will get to that soon.

Now we go about determining the values of the filter coefficients c_i by placing specific demands upon the output of the filter. We start by recalling that in our discussion of discrete Fourier transforms we observed that a transform is equivalent to a rotation from the time domain to the frequency domain. Yet we know from our study of linear algebra that rotations are described by orthogonal matrices, that is, matrices whose inverses are equal to their transposes. In order for the inverse transform to return us to the input signal, the transfer matrix must be orthogonal. For our wavelet transformation to be orthogonal, we must have the 4×4 filter matrix times its transpose equal to the identity matrix:

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & -c_0 & c_3 & -c_2 \end{bmatrix} \begin{bmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & -c_2 & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & -c_0 & c_1 & -c_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\Rightarrow \quad c_0^2 + c_1^2 + c_2^2 + c_3^2 = 1, \quad c_2 c_0 + c_3 c_1 = 0. \tag{10.37}$$

Two equations in four unknowns are not enough for a unique solution, so we now include the further requirement that the detail filter $H = (c_3, -c_0, c_1, -c_2)$ must output a zero if the input is smooth. We define "smooth" to mean that the input is constant or linearly increasing:

$$\begin{bmatrix} y_0 & y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}. \tag{10.38}$$

This is equivalent to demanding that the moments up to order p are zero, that is, that we have an "approximation of order p." Explicitly,

$$\begin{aligned} & H \begin{bmatrix} y_0 & y_1 & y_2 & y_3 \end{bmatrix} = H \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} = H \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix} = 0, \\ \Rightarrow \quad & c_3 - c_2 + c_1 - c_0 = 0, \quad 0 \times c_3 - 1 \times c_2 + 2 \times c_1 - 3 \times c_0 = 0, \\ \Rightarrow \quad & c_0 = \frac{1 + \sqrt{3}}{4\sqrt{2}} \simeq 0.483, \quad c_1 = \frac{3 + \sqrt{3}}{4\sqrt{2}} \simeq 0.836, \end{aligned} \tag{10.39}$$

$$c_2 = \frac{3-\sqrt{3}}{4\sqrt{2}} \simeq 0.224, \quad c_3 = \frac{1-\sqrt{3}}{4\sqrt{2}} \simeq -0.129. \tag{10.40}$$

These are the basic Daub4 filter coefficients. They are used to create larger filter matrices by placing the row versions of L and H along the diagonal, with successive pairs displaced two columns to the right. For example, for eight elements,

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & 0 & 0 \\ c_3 & -c_2 & c_1 & -c_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & 0 & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_0 & c_1 & c_2 & c_3 \\ 0 & 0 & 0 & 0 & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & 0 & 0 & 0 & 0 & c_0 & c_1 \\ c_1 & -c_0 & 0 & 0 & 0 & 0 & c_3 & -c_2 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix}. \tag{10.41}$$

Note that in order not to lose any information, the last pair on the bottom two rows is wrapped over to the left. If you perform the actual multiplications indicated in (10.41), you will note that the output has successive *smooth* and *detailed* information. The output is processed with the pyramid scheme.

The time dependencies of two Daub4 wavelets are displayed in Figure 10.12. To obtain these from our filter coefficients, first imagine that an elementary wavelet $y_{1,1}(t) \equiv \psi_{1,1}(t)$ is input into the filter. This should result in a transform $Y_{1,1} = 1$. Inversely, we obtain $y_{1,1}(t)$ by applying the inverse transform to a Y vector with a 1 in the first position and zeros in all the other positions. Likewise, the ith member of the Daubechies class is obtained by applying the inverse transform to a Y vector with a 1 in the ith position and zeros in all the other positions.

On the left in Figure 10.12 is the wavelet for coefficient 6 (thus the e6 notation). On the right in Figure 10.12 is the sum of two wavelets corresponding to the coefficients 10 and 58. We see that the two wavelets have different levels of scale as well as different time positions.

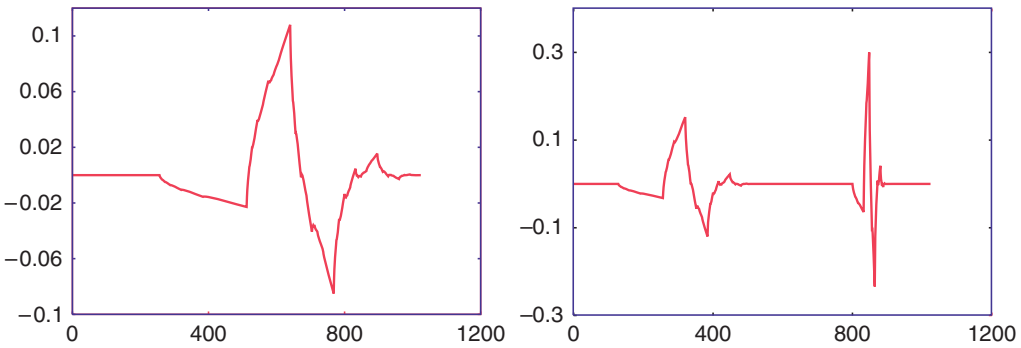

Figure 10.12 *Left*: The Daub4 e6 wavelet constructed by inverse transformation of the wavelet coefficients. This wavelet has been found to be particularly effective in wavelet analyses. *Right*: The sum of Daub4 e10 and Daub4 1e58 wavelets of different scale and time displacements.

So despite the fact that the time dependence of the wavelets is not evident when wavelet (filter) coefficients are used, it is there.

10.5.3 DWT Exercise ⊙

Listing 10.2 gives our program for performing a DWT on the chirp signal $y(t) = \sin(60t^2)$. The method `pyran` calls the `daube4` method to perform the DWT or inverse DWT, depending upon the value of `sign`.

1) Modify the program so that you output to a file the values for the input signal that your code has read in. It is always important to check your input.
2) Try to reproduce the left of Figure 10.11 by using various values for the variable `nend` that controls when the filtering ends. A value `nend=1024` should produce just the first step in the downsampling (top row in Figure 11.10). Selecting `nend=512` should produce the next row, while `nend=4` should output just two smooth and detailed coefficients.
3) Reproduce the scale-time diagram shown on the right in Figure 10.11. This diagram shows the output at different scales and serves to interpret the main components of the signal and the time in which they appear. The timeline at the bottom of the figure corresponds to a signal of length 1 over which 256 samples were recorded. The low-band (smooth) components are shown on the left, and the high-band components on the right.
   a) The bottom most figure results when `nend` = 256.
   b) The figure in the second row up results from `end` = 128, and we have the output from two filterings. The output contains 256 coefficients but divides time into four intervals and shows the frequency components of the original signal in more detail.
   c) Continue with the subdivisions for `end` = 64, 32, 16, 8, and 4.
4) For each of these choices except the topmost, divide the time by 2 and separate the intervals by vertical lines.
5) The topmost spectrum is your final output. Can you see any relation between it and the chirp signal?
6) Change the sign of `sign` and check that the inverse DWT reproduces the original signal.
7) Use the code to visualize the time dependence of the Daubechies mother function at different scales.
   a) Start by performing an inverse transformation on the eight-component signal `[0,0,0,0,1,0,0,0]`. This should yield a function with a width of about 5 units.
   b) Next perform an inverse transformation on a unit vector with $N = 32$ but with all components except the fifth equal to zero. The width should now be about 25 units, a larger scale but still covering the same time interval.
   c) Continue this procedure until you obtain wavelets of 800 units.
   d) Finally, with $N = 1024$, select a portion of the mother wavelet with data in the horizontal interval [590,800]. This should show self-similarity similar to that at the bottom of Figure 10.12.

10.6 Part II: Principal Components Analysis

Problem Given a dataset describing several properties of irises (flowers), separate the data into groups in order of importance. The properties (in cm) are

```
['sepal length', 'sepal width', 'petal length', 'petal width']
array(  [5.1, 3.5, 1.4, 0.2],
        [4.9, 3.0, 1.4, 0.2],
        [4.7, 3.2, 1.3, 0.2],
        [4.6, 3.1, 1.5, 0.2],
        [5.0, 3.6, 1.4, 0.2],
        [5.4, 3.9, 1.7, 0.4],
        [4.6, 3.4, 1.4, 0.3],
        [5.0, 3.4, 1.5, 0.2],
        [4.4, 2.9, 1.4, 0.2],
        [4.9, 3.1, 1.5, 0.1],
        [5.4, 3.7, 1.5, 0.2],
        [4.8, 3.4, 1.6, 0.2],
        [4.8, 3.0, 1.4, 0.1],
        [4.3, 3.0, 1.4, 0.1],
        [5.8, 4.0, 1.2, 0.2],
        [5.7, 4.4, 1.5, 0.4],
        [5.4, 3.9, 1.3, 0.4],
        [5.1, 3.5, 1.4, 0.3]  )
```

We have indicated that a shortcoming of Fourier analysis is that it uses an infinite number of components, that all of these components are correlated, and thus are not independent. Consequently, truncation of some of the components leads to difficulties in compression and reconstitution of the input signal. Wavelet analysis, on the other hand, is excellent at data compression, but not appropriate for high-dimensionality data sets, or for non-temporal signals. *Principal Components Analysis (PCA)* is a powerful analysis tool that uses statistics to provide insight into signals that may be contained within a *high-dimensionality*, multivariate dataset. Examples of high dimensionally data include stellar spectra, brain waves, facial patterns, and ocean currents. In these cases there may be hundreds of detectors in space, each of which records several types of signals for weeks on end. Furthermore, these kinds of data are often noisy, and possibly redundant (different detectors recording correlated signals), and so a statistical approach seems appropriate.

Variations of the PCA approach are used in many fields, where it goes by names such as the Kronen-Loèvet transform, the Hostelling transform, the proper orthogonal decomposition, singular value decomposition, factor analysis, empirical orthogonal functions, empirical component analysis, and empirical modal analysis [Wikipedia, 2014]. The approach combines statistics with transformation theory, the latter familiar from linear algebra, to rotate from the basis vectors used to collect the data into new basis vectors known as *principal components* that lie in the direction of maximal signal strength ("power") in the dataspace. This is analogous to the principal axis theorem of mechanics in which the description of solid object rotations is greatly simplified when moments of inertia relative to the principal axes are used. Our references Jackson [1991], Jolliffe [2002], Smith [2002], and Shlens [2003] tend to view PCA as *unsupervised dimensionality reduction*,

where "unsupervised" refers to the absence of labels on the data, and "reduction" to the small number of principal components that ultimately result. We prefer to view PCA as the way to extract the dominant dynamics contained in complex datasets.

10.6.1 Multi-dimensional Data Space

It's often helpful when dealing with complex data to imagine an abstract vector space in which the data elements lie. It is in this multi-dimensional *dataspace* that the PCA basis vectors lie. As a simple example, consider the four detectors in Figure 10.13 observing a beam of particles passing by. Each detector records its observations over time, with the measurements at each time considered a separate, individual sample. Furthermore, each detector may record a set of M observables, such as position, angle, intensity, thickness of a track, length of a track, *etc*. This produces an M-dimensional dataspace $\mathcal{R}^M$. Specifically, let's say at each instant of time, detector A records the position (x'_A, y'_A), and so on for detectors B–D. The sample of spatial data at that one instant of time is then represented as an 8-D vector:

$$\mathbf{X}' = [x'_a \quad y'_a \quad x'_b \quad y'_b \quad x'_c \quad y'_c \quad x'_d \quad y'_d]. \tag{10.42}$$

To get an idea of the sizes of the dataspaces with which we might be dealing, if the detectors make their recordings for 18 minutes (1080 seconds) at 110 Hz, then $1080 \times 110 = 118\,800$ of these vectors are created in $\mathcal{R}^M$. And this is a small problem.

Experimental data usually contain noise in addition to signals of interest. The *variance* $\sigma^2(z)$ in a dataset of N points is a measure of the dispersion of the datum points from their mean $\bar{z}$:

$$\bar{z} = \frac{1}{N}\sum_i^N z_i, \tag{10.43}$$

$$\sigma^2(z) \equiv \mathrm{Var}(z) \overset{\text{def}}{=} \frac{1}{N-1}\sum_i^N (z_i - \bar{z})^2. \tag{10.44}$$

If the data are of high precision, the signal would be much larger than the noise. In practice, measurements contain random and systematic errors, and therefore the signal-to-noise ratio (SNR),

$$SNR = \frac{\sigma^2_{\text{signal}}}{\sigma^2_{\text{noise}}}, \tag{10.45}$$

may not be large. PCA is a good way to deal with small SNR. On the left of Figure 10.14, we present some made-up 2D data (x_A, y_A) from detector A showing the direction of maximum signal variance σ^2_{signal} along PC_1, and the direction of maximum noise (or secondary signal)

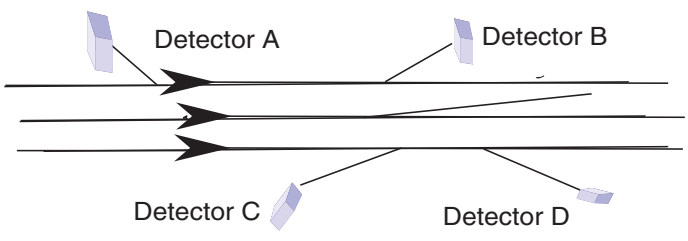

Figure 10.13 A beam of particles being observed by four detectors.

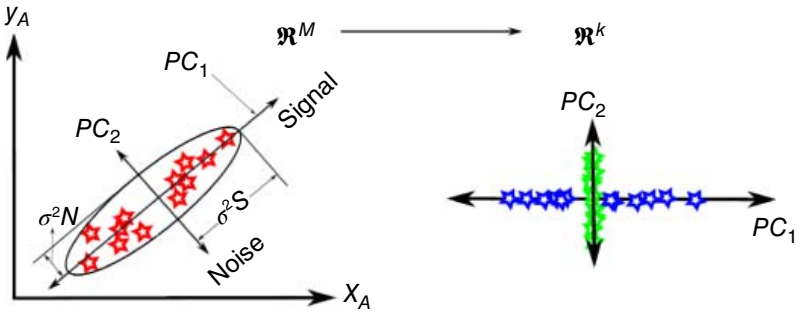

Figure 10.14 *Left*: Samples of 2D data (x_A, y_A) from a detector A showing the direction of maximum signal variance σ_S^2 along the principal component PC_1 basis, and the direction of noise variance σ_N^2 along the secondary PC_2 basis. *Right*: The same data projected onto the principal component axes. (Based on [Shlens, 2003].)

variance σ^2_{noise} along PC_2. Although a traditional view of statistics may be that a large variance indicates high noise, in the PCA view a large variance indicates that some interesting dynamics may be occurring in that direction in dataspace.

The key PCA assumption is that the direction with the largest variance contains most of the dynamics of interest, and that the deviation of the data from maximum variance may be due to noise, or maybe some less important dynamics. The PC_1 and PC_2 directions in Figure 10.14 are the two PCA basis vectors, and are chosen to maximize the SNR measured along PC_1, relative to that along PC_2 (we'll get to how these are determined shortly). On the right of the figure we show the same data projected along the PC_1 and PC_2 axes. Here $\mathcal{R}^k$ is a lower-dimensional space containing the k orthonormal and uncorrelated principal components vectors.

10.6.2 Wonders of the Covariance Matrix

We have just seen graphically how PCA isolates the signal from the noise for a 2D dataset such as (x'_A, y'_A). However, our sample problem has four detectors, and thus a higher-dimensional space to analyze. We generalize one step at a time by extending the approach to also include detector B. We define two datasets A and B, each centered about their means, $\overline{x}, \overline{y}$, and expressed as the row vectors:

$$\mathbf{A} = [a_1(x_{a,1}, y_{a,1})\ a_2(x_{a,2}, y_{a,2})\ \ldots\ a_N(x_{a,N}, y_{a,N})], \tag{10.46}$$

$$\mathbf{B} = [b_1(x_{b,1}, y_{b,1})\ b_2(x_{b,2}, y_{b,2})\ \ldots\ b_N(x_{b,N}, y_{b,N})], \tag{10.47}$$

$$x_{a,i} = x'_{a,i} - \overline{x'}, \qquad y_{a,i} = y'_{a,i} - \overline{y'}. \tag{10.48}$$

Next we compute the variances (10.44) for each of the centered ($\overline{a} = \overline{b} = 0$) sets:

$$\sigma_A^2 = \frac{1}{N-1}\sum_i^N a_i^2, \qquad \sigma_B^2 = \frac{1}{N-1}\sum_i^N b_i^2. \tag{10.49}$$

The variance concept is extended to *covariance*, as a measure of the correlation between the centered data in A and B:

$$\text{cov}(A, B) \overset{\text{def}}{=} \sigma_{AB}^2 \overset{\text{def}}{=} \frac{1}{N-1}\sum_i^N a_i\, b_i. \tag{10.50}$$

A positive covariance indicates that signals within A and B tend to change together and in the same direction. A large covariance indicates a high correlation or redundancy, while a zero value implies no correlation. Note that the variance (10.44) can be viewed as a special case of the covariance, $\text{var}(x) = \text{cov}(x, x)$, and that there is the symmetry $\text{cov}(x, y) = \text{cov}(y, x)$. All of these concepts are combined into the symmetric covariance matrix:

$$C_{AB} = \begin{bmatrix} \text{cov}(A,A) & \text{cov}(A,B) \\ \text{cov}(B,A) & \text{cov}(B,B) \end{bmatrix}. \tag{10.51}$$

These ideas generalize directly to higher dimensions. Start with the sets A (10.46) and B (10.47), where we remind you that the elements may contain a number of measurements. The covariance matrix can be written as the vector direct product (aka dot product, matrix multiplication, or dyadic):

$$C_{AB} \overset{\text{def}}{=} \frac{1}{N-1}\mathbf{AB} \equiv \frac{1}{N-1}\mathbf{A} \otimes \mathbf{B}^T. \tag{10.52}$$

With this notation, we can generalize to higher dimensions by defining new *row* subvectors containing the data from each of the M detectors:

$$\mathbf{x}_1 = \mathbf{A}, \quad \mathbf{x}_2 = \mathbf{B}, \ \ldots, \quad \mathbf{x}_M = \mathbf{M}. \tag{10.53}$$

We combine these row vectors into an extended $M \times N$ data matrix:

$$X = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_M \end{bmatrix} = \begin{bmatrix} \Downarrow \text{All} & \Rightarrow \text{All } A \text{ measurements} \\ \Downarrow \text{one} & \Rightarrow \text{All } B \text{ measurements} \\ \Downarrow \text{time} & \Rightarrow \text{All } C \text{ measurements} \\ \Downarrow \text{measurements} & \Rightarrow \text{All } D \text{ measurements} \end{bmatrix}. \tag{10.54}$$

Each row of this matrix contains all of the measurements from a particular detector, while each column contains all of the measurements for a particular time. With this notation (and $\overline{\mathbf{x}} = 0$), the covariance matrix can be written in the concise form

$$C = \frac{1}{N-1}\mathbf{X}\mathbf{X}^T. \tag{10.55}$$

This can be thought of as a generalization of the familiar dot product of two 2D vectors, $\mathbf{x} \cdot \mathbf{x} = \mathbf{x}^\mathbf{T}\mathbf{x}$, as a measure of their overlap.
In summary:

- The covariant matrix C_{ij} is the dot product of the centered measurements vector from the ith detector ($i = A, B, \ldots$) with the centered measurements vector from the jth detector.
- For any two variables in the data, C is a square symmetric matrix measuring the relationship between those variables.
- The diagonal elements of C are the variances in the measurements from individual detectors.
- The off-diagonal elements of C are the covariances between the measurements from different detectors, that is, the correlations between detectors.

Steps in a Principal Component Analysis (Easy with NumPy)

1) As indicated in Figure 10.14, there is the assumption that the direction in which the variance is largest indicates the "principal" component in the data, PC_1 or $\mathbf{p_1}$.

Table 10.1 PCA demonstration data.

Data		Adjusted data		In PCA basis	
x	y	x	y	x_1	x_2
2.5	2.4	0.69	0.49	−0.828	−0.175
0.5	0.7	−1.31	−1.21	1.78	0.143
2.2	2.9	0.39	0.99	−0.992	0.484
1.9	2.2	0.09	0.29	−0.274	0.130
3.1	3.0	1.29	1.09	−1.68	−0.209
2.3	2.7	0.49	0.79	0.913	0.175
2	1.6	0.19	−0.31	0.0991	−0.350
1.0	1.1	−0.81	−0.81	1.14	0.464
1.6	1.6	−0.31	−0.31	0.438	0.0178
1.1	0.9	−0.71	−1.01	1.22	−0.163

Consequently, PCA searches for the direction in dataspace for which the variance of $\mathbf{X}$ is maximized.

2) Once $\mathbf{p_1}$ has been found, the basis vector $\mathbf{p_2}$ is chosen as the orthonormal to $\mathbf{p_1}$.
3) The process is repeated until there are M orthonormal basis vectors. These are the M principal components of the data.
4) The eigenvectors and eigenvalues are ordered according to their corresponding variances.
5) Explicitly, starting with the $M \times N$ data matrix $\mathbf{X}$, a matrix $\mathbf{P}$ is determined such that

$$\mathbf{C}_y = \frac{1}{N-1}\mathbf{Y}\mathbf{Y}^\mathbf{T} = \text{diagonal}, \qquad \text{where} \quad \mathbf{Y} = \mathbf{P}\mathbf{X}. \tag{10.56}$$

6) The rows of $\mathbf{P}$ are the principal component basis vectors (same as the eigenvectors of $\mathbf{X}\mathbf{X}^\mathbf{T}$).
7) The diagonal elements of $\mathbf{C_Y}$ are the variances of $\mathbf{X}$ along the corresponding $\mathbf{p_i}$'s.

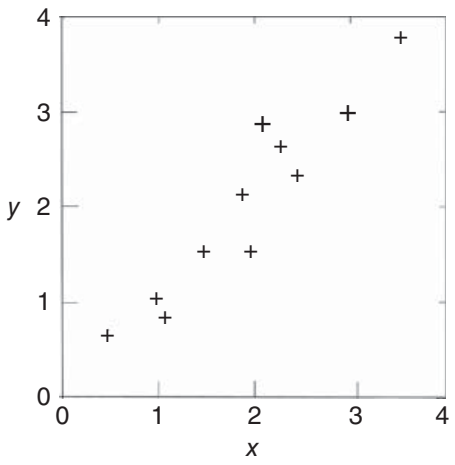

Figure 10.15 Sample data used in our demonstration PCA analysis.

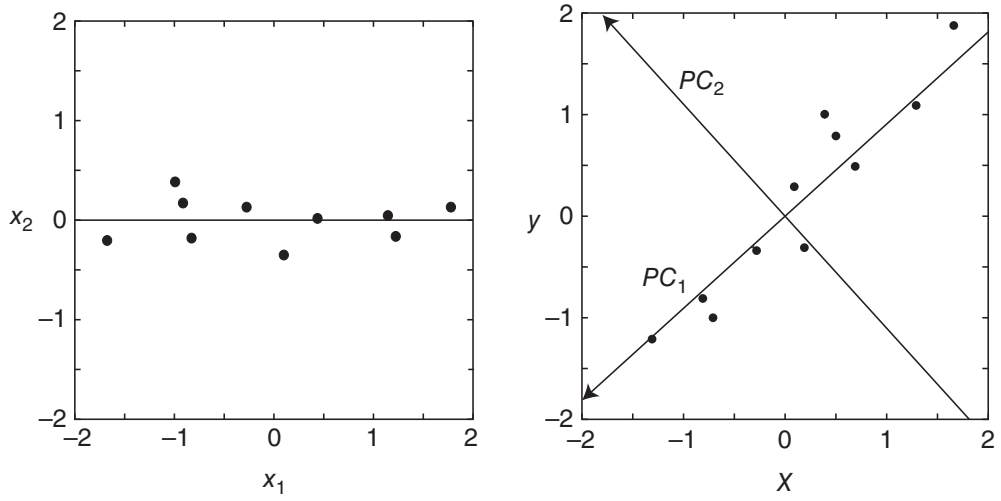

Figure 10.16 *Left*: The PCA basis vectors (eigenvectors of cov(x, y)). *Right*: The normalized data using the PCA eigenvectors as basis.

10.6.3 Demonstration of Principal Component Analysis

We'll leave the analysis of the irises data at the beginning of Part II as *your* **problem**, and, instead, we'll analyze the simpler data in Table 10.1 [Smith, 2002]. These data, are shown in Figure 10.15 as *x versus y*, but they don't have to be spatial. Also shown is the analysis of these data in terms of the first two principal components. As expected, the first eigenvector points in the direction with the largest variance, while the next vector is orthogonal to the first. There clearly is less variance along PC_2 and, consequently, less dynamical importance of that component.

Here are the steps in the analysis:

1) **Enter data as an array**: The first two columns in Table 10.1.
2) **Subtract the mean**: PCA analysis assumes that the data in each dimension has zero mean. Accordingly, as shown in columns two and three in Table 10.1, we calculated the mean for each column, $(\overline{x}, \overline{y})$, and subtracted them from the data. The resulting adjusted data are given in the third and fourth columns of the table.
3) **Calculate the covariance matrix**:

$$\text{var}(x) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2, \tag{10.57}$$

$$\text{cov}(x,y) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})(y_i - \overline{y}), \tag{10.58}$$

$$C = \begin{bmatrix} \text{cov}(x,x) & \text{cov}(x,y) \\ \text{cov}(y,x) & \text{cov}(y,y) \end{bmatrix} = \begin{bmatrix} 0.6166 & 0.6154 \\ 0.6154 & 0.7166 \end{bmatrix}. \tag{10.59}$$

4) **Compute unit eigenvector and eigenvalues of C (easy with NumPy)**:

$$\lambda_1 = 1.284, \qquad \lambda_2 = 0.4908, \tag{10.60}$$

$$\mathbf{PC}_1 = \begin{bmatrix} -0.6779 \\ -0.7352 \end{bmatrix}, \qquad \mathbf{PC}_2 = \begin{bmatrix} -0.7352 \\ 0.6789 \end{bmatrix}, \tag{10.61}$$

where we have ordered the eigenvalues and eigenvectors. The eigenvector with the largest eigenvalue is the principal component in the data, typically with ∼80% of the power in the signal.

In Figure 10.16 we show the translated data and the two PCA eigenvectors of the covariance matrix (scaled to fill the frame). Notice that PC_1, which points in direction of the major variation in the data, is essentially a straight-line fit to the data. The PC_2 eigenvector is clearly orthogonal to PC_1, and contains less signal strength. This is as should be.

5) **Express the data in terms of principal Components**: We next expressed the data in terms of their two principal components by forming two *feature matrices*:

$$F_1 = \begin{bmatrix} -0.6779 \\ -0.7352 \end{bmatrix}, \qquad F_2 = \begin{bmatrix} -0.6779 & -0.7352 \\ -0.7352 & 0.6779 \end{bmatrix}. \tag{10.62}$$

Here F_1 keeps just the major principal component, while F_2 keeps the first two. The matrix gets its name because it focuses on which features of the data are being kept. Next we formed F_2^T, the transpose of the feature matrix, and $\mathbf{X}^T$, the transpose of the translated data matrix $\mathbf{X}$:

$$F_2^T = \begin{bmatrix} -0.6779 & -0.7352 \\ -0.7352 & 0.6779 \end{bmatrix}, \tag{10.63}$$

$$\mathbf{X}^T = \begin{bmatrix} 0.69 & -1.31 & 0.39 & 0.09 & 1.29 & 0.49 & 0.19 & -0.81 & -0.31 & -0.71 \\ 0.49 & -1.21 & 0.99 & 0.29 & 1.09 & 0.79 & -0.31 & -0.81 & -0.31 & -1.01 \end{bmatrix}.$$

We expressed the data in terms of these principal components by multiplying F_2^T and $\mathbf{X}$ together:

$$\begin{aligned} X^{\text{PCA}} &= F_2^T \times X^T \qquad (10.64) \\ &= \begin{bmatrix} -0.6779 & -0.7352 \\ -0.7352 & 0.6779 \end{bmatrix} \\ &\quad \times \begin{bmatrix} 0.69 & -1.31 & 0.39 & 0.09 & 1.29 & 0.49 & 0.19 & -0.81 & -0.31 & -0.71 \\ 0.49 & -1.21 & 0.99 & 0.29 & 1.09 & 0.79 & -0.31 & -0.81 & -0.31 & -1.01 \end{bmatrix} \\ &= \begin{bmatrix} 0.828 & 1.78 & -0.992 & -0.274 & -1.68 & -0.913 & 0.0991 & 1.15 & 0.438 & 1.22 \\ -0.175 & 0.143 & 0.384 & 0.130 & -0.209 & 0.175 & -0.350 & 0.464 & 0.178 & -0.162 \end{bmatrix}. \end{aligned}$$

On the right of Table 10.1, the data are plotted using the PC_1 and PC_2 bases. The plot shows where each datum point sits relative to the trend in the data. If we had plotted only the first principal component, all of the data would fall on a straight line.

10.6.4 PCA Exercises

1) Use just the principal eigenvectors to perform the PCA analysis just completed with two eigenvectors.
2) Store data from 10 cycles of the chaotic pendulum studied in Chapter 8, but do not include transients. Perform a PCA of these data and plot the results using principal component axes.

10.7 Code Listings

Listing 10.1 CWT.py Uses Morlet wavelets to compute the continuous wavelet transform of the sum of sine functions. (Courtesy of Z. Diabolic.)

```
# CWT.py  Continuous Wavelet TF. Based on program by Zlatko Dimcovic

import matplotlib.pylab as p;
from mpl_toolkits.mplot3d import Axes3D ;
from visual.graph import *;

originalsignal=gdisplay(x=0, y=0, width=600, height=200, \
        title='Input Signal',xmin=0,xmax=12,ymin=-20,ymax=20)
orsigraph=gcurve(color=color.yellow)
invtrgr = gdisplay(x=0, y=200, width=600, height=200,
         title='Inverted Transform',xmin=0,xmax=12,ymin=-20,ymax=20)
invtr = gcurve(x=list(range(0,240)), display = invtrgr, color= color.green)
iT =  0.0;          fT =  12.0;         W = fT - iT;
N =  240;           h =  W/N        # Need *very* small s for high f
noPtsSig =  N;      noS =  20;          noTau =  90;
iTau =  0.;         iS =  0.1;          tau =  iTau;        s =  iS
dTau =  W/noTau;    dS =  (W/iS)**(1./noS);
maxY =  0.001;      sig =  zeros((noPtsSig), float)           # Signal

def signal(noPtsSig, y):                                  # Signal function
    t = 0.0;      hs = W/noPtsSig;      t1 = W/6.;     t2 = 4.*W/6.
    for i in range(0, noPtsSig):
        if  t >= iT  and t <=  t1:  y[i] =  sin(2*pi*t)
        elif t >= t1 and t <=  t2: y[i] = 5.*sin(2*pi*t) + 10.*sin(4*pi*t);
        elif t >= t2 and t <=  fT:
            y[i] = 2.5*sin(2*pi*t) + 6.*sin(4*pi*t) + 10.*sin(6*pi*t)
        else:
            print("In signal(...) : t out of range.")
            sys.exit(1)
        yy=y[i]
        orsigraph.plot(pos=(t,yy))
        t += hs
signal(noPtsSig, sig)                                     # Form signal
Yn =  zeros( (noS+1, noTau+1), float)                     # Transform
def morlet(t, s, tau):                                    # Mother
    T =  (t - tau)/s
    return sin(8*T) * exp( - T*T/2. )
def transform(s, tau, sig):                             # Find wavelet TF
    integral = 0.
    t = iT;
    for i in range(0, len(sig) ):
        t += h
        integral += sig[i]*morlet(t, s, tau)*h
    return integral / sqrt(s)
def invTransform(t, Yn):                                 # Compute inverse
    s = iS                                                     # Transform
    tau = iTau
    recSig_t = 0
    for i in range (0, noS):
        s *= dS                                              # Scale graph
        tau = iTau
        for j in range (0, noTau):
            tau += dTau
            recSig_t += dTau*dS *(s**(-1.5))* Yn[i,j] * morlet(t,s,tau)
    return recSig_t
print("working, finding transform, count 20")
for i in range( 0, noS):
    s *= dS                                                # Scaling
    tau = iT
    print(i)
    for j in range(0, noTau):
        tau += dTau                                        # Translate
        Yn[i, j] = transform(s, tau, sig)
```

```
print("transform found")
for i in range( 0, noS):
    for j in range( 0, noTau):
        if Yn[i, j] > maxY or Yn[i, j] < - 1 *maxY :
            maxY = abs( Yn[i, j] )                          # Find max Y
tau =  iT
s =  iS
print("normalize")
for i in range( 0, noS):
     s *= dS
     for j in range( 0, noTau):
         tau +=   dTau                                      # Transform
         Yn[i, j] = Yn[i, j]/maxY
     tau = iT
print("finding inverse transform")                          # Inverse TF
recSigData =  "recSig.dat"
recSig =  zeros(len(sig) )
t =  0.0;
print("count to 10")
kco = 0;            j = 0;              Yinv =  Yn
for rs in range(0, len(recSig) ):
    recSig[rs] = invTransform(t, Yinv)                      # Invert
    xx=rs/20
    yy=4.6*recSig[rs]
    invtr.plot(pos=(xx,yy))
    t += h
    if kco %24 == 0:
        j += 1
        print(j)
    kco += 1
x = list(range(1, noS + 1))
y = list(range(1, noTau + 1))
X,Y = p.meshgrid(x, y)

def functz(Yn):                                     # Transform function
    z = Yn[X, Y]
    return z
Z = functz(Yn)
fig = p.figure()
ax = Axes3D(fig)
ax.plot_wireframe(X, Y, Z, color = 'r')
ax.set_xlabel('s: scale')
ax.set_ylabel('Tau')
ax.set_zlabel('Transform')
p.show()
print("Done")
```

Listing 10.2 DWT.py Uses the Daub4 digital wavelets and the pyramid algorithm to compute the discrete wavelet transform for the chirp signal values stored in `f[ ]`.

```
# DWT.py:  Discrete Wavelet Transform, Daubechies, global variables

from visual import *
from visual.graph import *

sq3 = sqrt(3);           fsq2 = 4.0*sqrt(2); N = 1024     # N = 2^n
c0 = (1+sq3)/fsq2;            c1 = (3+sq3)/fsq2       # Daubechies 4
c2 = (3-sq3)/fsq2;             c3 = (1-sq3)/fsq2
transfgr1 = None                                        # Display

def chirp( xi):                                          # Chirp signal
    y = sin(60.0*xi**2);
    return y;
def daube4(f, n, sign):          # DWT if sign >= 0, inverse if sign < 0
    global transfgr1, transfgr2
    tr = zeros( (n + 1), float)                         # Temporary
    if n < 4 : return
```

```
    mp = n/2
    mp1 = mp + 1                                          # midpoint + 1
    if sign >= 0:                                         # DWT
        j = 1
        i = 1
        maxx  = n/2
        if n > 128:                                       # Scale
            maxy = 3.0
            miny = - 3.0
            Maxy = 0.2
            Miny = - 0.2
            speed = 50                                     # Fast rate
        else:
            maxy = 10.0
            miny = - 5.0
            Maxy = 7.5
            Miny = - 7.5
            speed = 8                                        # Lower rate
        if transfgr1:
            transfgr1.display.visible = False
            transfgr2.display.visible = False
            del transfgr1
            del transfgr2
        transfgr1 = gdisplay(x=0, y=0, width=600, height=400,\
                    title='Wavelet TF, down sample + low pass', xmax=maxx,\
                    xmin=0, ymax=maxy, ymin=miny)
        transf  = gvbars(delta=2.*n/N,color=color.cyan,display=transfgr1)
        transfgr2 = gdisplay(x=0, y=400, width=600, height=400,\
                        title='Wavelet TF, down sample + high pass',\
                        xmax=2*maxx, xmin=0, ymax=Maxy, ymin=Miny)
        transf2 = gvbars(delta=2.*n/N,color=color.cyan,display=transfgr2)
        while j <= n - 3:
            rate(speed)
            tr[i] = c0*f[j] + c1*f[j+1] + c2*f[j+2] + c3*f[j+3]# low-pass
            transf.plot(pos = (i, tr[i]) )                 # c coefficients
            tr[i+mp] = c3*f[j] - c2*f[j+1] + c1*f[j+2] - c0*f[j+3] # high
            transf2.plot(pos = (i + mp, tr[i + mp]) )
            i += 1                                          # d coefficents
            j += 2                                          # Downsampling
        tr[i] = c0*f[n-1] + c1*f[n] + c2*f[1] + c3*f[2]        # low-pass
        transf.plot(pos = (i, tr[i]) )                     # c coefficients
        tr[i+mp] = c3*f[n-1] - c2*f[n] + c1*f[1] - c0*f[2]    # High-pass
        transf2.plot(pos = (i+mp, tr[i+mp]) )
    else:                                                  # Inverse DWT
        tr[1] = c2*f[mp] + c1*f[n] + c0*f[1] + c3*f[mp1]       # Low-pass
        tr[2] = c3*f[mp] - c0*f[n] + c1*f[1] - c2*f[mp1]       # High-pass
        j = 3
        for i in range (1, mp):
           tr[j] = c2*f[i] + c1*f[i+mp] + c0*f[i+1] + c3*f[i+mp1]   # Low
           j += 1                                                  # Upsample
           tr[j] = c3*f[i] - c0*f[i+mp] + c1*f[i+1] - c2*f[i+mp1]  # High
           j += 1;                                                 # Upsampling
    for i in range(1, n+1):
        f[i] = tr[i]                                       # Copy TF to array
def pyram(f, n, sign):                                    # DWT, replaces f by TF
    if (n < 4): return                                       # Too few data
    nend = 4                                                 # When to stop
    if sign >= 0 :                                             # Transform
        nd = n
        while nd >= nend:                                  # Downsample filtering
            daube4(f, nd, sign)
            nd //= 2
    else:                                                    # Inverse TF
        while nd <= n:                                       # Upsampling
            daube4(f, nd, sign)
            nd *= 2
f = zeros( (N + 1), float)                                   # Data vector
inxi = 1.0/N                                              # For chirp signal
xi = 0.0
```

```
for i in range(1, N + 1):
    f[i] = chirp(xi)                                    # Function to TF
    xi   += inxi;
n = N                                                   # Must be 2^m
pyram(f, n, 1)                                          # TF
# pyram(f, n, -1)                                       # Inverse TF
```