Landau, R. H., Páez, M. J., & Bordeianu, C. C. (2015).
Computational physics: Problem solving with Python.
John Wiley & Sons.

12
Fourier Analysis: Signals and Filters

We start this chapter with a discussion of Fourier series and Fourier transforms, the standard tools for decomposing periodic and nonperiodic motions, respectively. We find that, as implemented for numerical computation, both the series and the transform become the same discrete Fourier transform (DFT) algorithm, which has a beautiful simplicity in its realization as a program. We then show how Fourier tools can be used to reduce noise in measured or simulated signals. We end the chapter with a discussion of the fast Fourier transform (FFT), a technique so efficient that it permits nearly instantaneous evaluations of DFTs on various devices.

12.1
Fourier Analysis of Nonlinear Oscillations

Consider a particle oscillating either in the nonharmonic potential of (8.5):

$$V(x) = \frac{1}{p} k|x|^p \,, \quad p \neq 2 \,, \tag{12.1}$$

or in the perturbed harmonic oscillator potential (8.2),

$$V(x) = \frac{1}{2} kx^2 \left(1 - \frac{2}{3}\alpha x\right) \,. \tag{12.2}$$

While free oscillations in these potentials are always periodic, they are not sinusoidal. Your *problem* is to take the solution of one of these nonlinear oscillators and expand it in a Fourier basis:

$$x(t) = A_0 \sin(\omega t + \phi_0) \,. \tag{12.3}$$

For example, if your oscillator is sufficiently nonlinear to behave like the sawtooth function (Figure 12.1a), then the Fourier spectrum you obtain should be similar to that shown in Figure 12.1b.

In general, when we undertake such a spectral analysis we want to analyze the steady-state behavior of a system. This means that we have waited for the initial

Computational Physics, 3rd edition. Rubin H. Landau, Manuel J. Páez, Cristian C. Bordeianu.

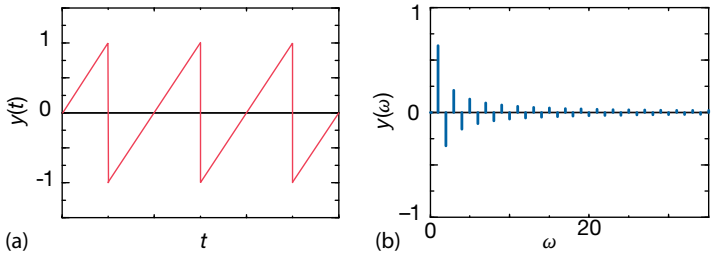

Figure 12.1 (a) A sawtooth function that repeats infinitely in time. (b) The Fourier spectrum of frequencies contained in this function (natural units). Also see Figure 1.9 in which we show the result of summing a finite number of terms of this series.

transient behavior to die out. It is easy to identify just what the initial transient is for linear systems, but may be less so for nonlinear systems in which the "steady state" jumps among a number of configurations. In the latter case, we would have different Fourier spectra at different times.

12.2 Fourier Series (Math)

Part of our interest in nonlinear oscillations arises from their lack of study in traditional physics courses where linear oscillations, despite the fact that they are just a first approximation, are most often studied. If the force on a particle is always toward its equilibrium position (a restoring force), then the resulting motion will be *periodic*, but not necessarily *harmonic*. A good example is the motion in a highly anharmonic potential with $p \simeq 10$ in (12.1) that produces an $x(t)$ looking like a series of pyramids; this motion is periodic but not harmonic.

In a sense, our approach is the inverse of the traditional one in which the *fundamental* oscillation is determined analytically and the higher frequency *overtones* are determined by perturbation theory (Landau and Lifshitz, 1976). We start with the full (numerical) periodic solution and then decompose it into what may be called *harmonics.* When we speak of fundamentals, overtones, and harmonics, we speak of solutions to the linear *boundary-value problem*, for example, of waves on a plucked violin string. In the latter case, and when given the correct conditions (enough musical skill), it is possible to excite individual harmonics or sums of them in the series

$$y(t) = b_0 \sin \omega_0 t + b_1 \sin 2\omega_0 t + \cdots \ . \tag{12.4}$$

Anharmonic oscillators vibrate at a single frequency (which may vary with amplitude) but not with a sinusoidal waveform. Although it is mathematically proper to expand nonlinear oscillations in a Fourier series, this does not imply that the individual harmonics can be excited (played).

You may recall from classical mechanics that the general solution for a vibrating system can be expressed as the sum of the *normal modes* of that system. These expansions are possible only if we have *linear operators* and, subsequently, the *principle of superposition*: If $y_1(t)$ and $y_2(t)$ are solutions of some linear equation, then $\alpha_1 y_1(t) + \alpha_2 y_2(t)$ is also a solution. The principle of linear superposition does not hold when we solve nonlinear problems. Nevertheless, it is always possible to expand a *periodic* solution of a *nonlinear* problem in terms of trigonometric functions with frequencies that are integer multiples of the true frequency of the nonlinear oscillator.[1)] This is a consequence of *Fourier's theorem* being applicable to any single-valued periodic function with only a finite number of discontinuities. We assume we know the period T, that is,

$$y(t + T) = y(t) \ . \tag{12.5}$$

This tells us the *true frequency* ω:

$$\omega \equiv \omega_1 = \frac{2\pi}{T} \ . \tag{12.6}$$

A periodic function (usually designated as the *signal*) can be expanded as a series of harmonic functions with frequencies that are multiples of the true frequency:

$$y(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos n\omega t + b_n \sin n\omega t) \ . \tag{12.7}$$

This equation represents the signal $y(t)$ as the simultaneous sum of pure tones of frequency $n\omega$. The coefficients a_n and b_n measure the amount of $\cos n\omega t$ and $\sin n\omega t$ present in $y(t)$, respectively. The intensity or power at each frequency is proportional to $a_n^2 + b_n^2$.

The Fourier series (12.7) is a "best fit" in the least-squares sense of Chapter 7, because it minimizes $\sum_i [y(t_i) - y_i]^2$, where i denotes different measurements of the signal. This means that the series converges to the *average* behavior of the function, but misses the function at discontinuities (at which points it converges to the mean) or at sharp corners (where it overshoots). A general function $y(t)$ may contain an infinite number of Fourier components, although low-accuracy reproduction is usually possible with a small number of harmonics.

The coefficients a_n and b_n in (12.7) are determined by the standard techniques for orthogonal function expansion. To find the coefficients, multiply both sides of (12.7) by $\cos n\omega t$ or $\sin n\omega t$, integrate over one period, and project a single a_n or b_n:

$$\begin{pmatrix} a_n \\ b_n \end{pmatrix} = \frac{2}{T} \int_0^T \mathrm{d}t \begin{pmatrix} \cos n\omega t \\ \sin n\omega t \end{pmatrix} y(t) \ , \quad \omega \stackrel{\text{def}}{=} \frac{2\pi}{T} \ . \tag{12.8}$$

1) We remind the reader that every periodic system by definition has a period T and consequently a true frequency ω. Nonetheless, this does not imply that the system behaves like $\sin \omega t$. Only harmonic oscillators do that.

As seen in the b_n coefficients (Figure 12.1b), these coefficients usually decrease in magnitude as the frequency increases, and can enter with a negative sign, the negative sign indicating the relative phase.

Awareness of the *symmetry* of the function $y(t)$ may eliminate the need to evaluate all the expansion coefficients. For example,

- a_0 is twice the average value of y:

$$a_0 = 2\langle y(t)\rangle \ . \tag{12.9}$$

- For an *odd function*, that is, one for which $y(-t) = -y(t)$, all a_n coefficients $\equiv 0$, and only half of the integration range is needed to determine b_n:

$$b_n = \frac{4}{T}\int_0^{T/2} \mathrm{d}t\ y(t)\sin n\omega t \ . \tag{12.10}$$

  However, if there is no input signal for $t < 0$, we do not have a truly odd function, and so small values of a_n may occur.
- For an *even function*, that is, one for which $y(-t) = y(t)$, all b_n coefficient $\equiv 0$, and only half the integration range is needed to determine a_n:

$$a_n = \frac{4}{T}\int_0^{T/2} \mathrm{d}t\, y(t)\cos n\omega t \ . \tag{12.11}$$

12.2.1 Examples: Sawtooth and Half-Wave Functions

The sawtooth function (Figure 12.1) is described mathematically as

$$y(t) = \begin{cases} \frac{t}{T/2}, & \text{for } 0 \le t \le \frac{T}{2} \ , \\ \frac{t-T}{T/2}, & \text{for } \frac{T}{2} \le t \le T \ . \end{cases} \tag{12.12}$$

It is clearly periodic, nonharmonic, and discontinuous. Yet it is also odd and so can be represented more simply by shifting the signal to the left:

$$y(t) = \frac{t}{T/2} \ , \quad -\frac{T}{2} \le t \le \frac{T}{2} \ . \tag{12.13}$$

Although the general shape of this function can be reproduced with only a few terms of the Fourier components, many components are needed to reproduce the sharp corners. Because the function is odd, the Fourier series is a sine series and (12.8) determines the b_n values:

$$b_n = \frac{2}{T}\int_{-T/2}^{+T/2} \mathrm{d}t \sin n\omega t \frac{t}{T/2} = \frac{2}{n\pi}(-1)^{n+1} \ , \tag{12.14}$$

$$\Rightarrow y(t) = \frac{2}{\pi}\left[\sin\omega t - \frac{1}{2}\sin 2\omega t + \frac{1}{3}\sin 3\omega t - \cdots\right] . \tag{12.15}$$

The half-wave function

$$y(t) = \begin{cases} \sin\omega t, & \text{for } 0 < t < \frac{T}{2} , \\ 0, & \text{for } \frac{T}{2} < t < T , \end{cases} \tag{12.16}$$

is periodic, nonharmonic (the upper half of a sine wave), and continuous, but with discontinuous derivatives. Because it lacks the sharp corners of the sawtooth function, it is easier to reproduce with a finite Fourier series. Equation 12.8 determines

$$a_n = \begin{cases} \frac{-2}{\pi(n^2-1)}, & n \text{ even or } 0 , \\ 0, & n \text{ odd} , \end{cases} \qquad b_n = \begin{cases} \frac{1}{2}, & n = 1 , \\ 0, & n \neq 1 , \end{cases}$$

$$\Rightarrow \quad y(t) = \frac{1}{2}\sin\omega t + \frac{1}{\pi} - \frac{2}{3\pi}\cos 2\omega t - \frac{2}{15\pi}\cos 4\omega t + \ldots \tag{12.17}$$

12.3 Exercise: Summation of Fourier Series

Hint: The program **FourierMatplot.py** written by Oscar Restrepo performs a Fourier analysis of a sawtooth function and produces the visualization shown in Figure 1.9b. You may want to use this program to help with this exercise.

1. *Sawtooth function:* Sum the Fourier series for the *sawtooth function* up to order $n = 2, 4, 10, 20$, and plot the results over two periods.
   a) Check that in each case, the series gives the mean value of the function *at* the points of discontinuity.
   b) Check that in each case the series *overshoots* by about 9% the value of the function on either side of the discontinuity (the *Gibbs phenomenon*).
2. *Half-wave function:* Sum the Fourier series for the *half-wave function* up to order $n = 2, 4, 10, 50$ and plot the results over two periods. (The series converges quite well, doesn't it?)

12.4 Fourier Transforms (Theory)

Although a Fourier *series* is the right tool for approximating or analyzing periodic functions, the Fourier *transform* or *integral* is the right tool for nonperiodic functions. We convert from series to transform by imagining a system described by a continuum of "fundamental" frequencies. We thereby deal with *wave packets* containing continuous rather than discrete frequencies.[2] While the difference

2) We follow convention and consider time t the function's variable and frequency ω the transform's variable. Nonetheless, these can be reversed or other variables such as position x and wave vector $\boldsymbol{k}$ may also be used.

between series and transform methods may appear clear mathematically, when we approximate the Fourier integral as a finite sum, the two become equivalent.

By analogy with (12.7), we now imagine our function or signal $y(t)$ expressed in terms of a continuous series of harmonics (*inverse Fourier transform*):

$$y(t) = \int_{-\infty}^{+\infty} \mathrm{d}\omega\, Y(\omega) \frac{\mathrm{e}^{\mathrm{i}\omega t}}{\sqrt{2\pi}} \,, \tag{12.18}$$

where for compactness we use a complex exponential function.[3)] The expansion amplitude $Y(\omega)$ is analogous to the Fourier coefficients (a_n, b_n), and is called the *Fourier transform* of $y(t)$. The integral (12.18) is the inverse transform because it converts the transform to the signal. The *Fourier transform* converts the signal $y(t)$ to its transform $Y(\omega)$:

$$Y(\omega) = \int_{-\infty}^{+\infty} \mathrm{d}t \frac{\mathrm{e}^{-\mathrm{i}\omega t}}{\sqrt{2\pi}} y(t) \,. \tag{12.19}$$

The $1/\sqrt{2\pi}$ factor in both these integrals is a common normalization in quantum mechanics, but maybe not in engineering where only a single $1/2\pi$ factor is used. Likewise, the signs in the exponents are also conventions that do not matter as long as you maintain consistency.

If $y(t)$ is the measured response of a system (signal) as a function of time, then $Y(\omega)$ is the *spectral function* that measures the amount of frequency ω present in the signal. In many cases, it turns out that $Y(\omega)$ is a complex function with both positive and negative values, and with powers-of-ten variation in magnitude. Accordingly, it is customary to eliminate some of the complexity of $Y(\omega)$ by making a semilog plot of the squared modulus $|Y(\omega)|^2$ vs. ω. This is called a *power spectrum* and provides an immediate view of the amount of power or strength in each component.

If the Fourier transform and its inverse are consistent with each other, we should be able to substitute (12.18) into (12.19) and obtain an identity:

$$Y(\omega) = \int_{-\infty}^{+\infty} \mathrm{d}t \frac{\mathrm{e}^{-\mathrm{i}\omega t}}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} \mathrm{d}\omega' \frac{\mathrm{e}^{\mathrm{i}\omega' t}}{\sqrt{2\pi}} Y(\omega') \tag{12.20}$$

$$= \int_{-\infty}^{+\infty} \mathrm{d}\omega' \left\{ \int_{-\infty}^{+\infty} \mathrm{d}t \frac{\mathrm{e}^{\mathrm{i}(\omega'-\omega)t}}{2\pi} \right\} Y(\omega') \,. \tag{12.21}$$

3) Recall that $\exp(\mathrm{i}\omega t) = \cos\omega t + \mathrm{i}\sin\omega t$, and with the law of linear superposition this means that the real part of y gives the cosine series, and the imaginary part the sine series.

For this to be an identity, the term in braces must be the *Dirac delta function*:

$$\int_{-\infty}^{+\infty} \mathrm{d}t \mathrm{e}^{\mathrm{i}(\omega'-\omega)t} = 2\pi\delta(\omega' - \omega) \ . \tag{12.22}$$

While the delta function is one of the most common and useful functions in theoretical physics, it is not well behaved in a mathematical sense and misbehaves terribly in a computational sense. While it is possible to create numerical approximations to $\delta(\omega' - \omega)$, they may well be borderline pathological. It is certainly better for you to do the delta function part of an integration analytically and leave the nonsingular leftovers to the computer.

12.5 The Discrete Fourier Transform

If $y(t)$ or $Y(\omega)$ is known analytically or numerically, integrals (12.18) and (12.19) can be evaluated using the integration techniques studied earlier. In practice, the signal $y(t)$ is measured at just a finite number N of times t, and these are all we have as input to approximate the transform. The resultant *discrete Fourier transform* is an approximation both because the signal is not known for all times, and because we must integrate numerically (Briggs and Henson, 1995). Once we have a discrete set of (approximate) transform values, they can be used to reconstruct the signal for any value of the time. In this way, the DFT can be thought of as a technique for interpolating, compressing, and extrapolating the signal.

We assume that the signal $y(t)$ is sampled at $(N + 1)$ discrete times (N time intervals), with a constant spacing $\Delta t = h$ between times:

$$y_k \overset{\text{def}}{=} y(t_k) \ , \quad k = 0, 1, 2, \ldots, N \ , \tag{12.23}$$

$$t_k \overset{\text{def}}{=} kh \ , \quad h = \Delta t \ . \tag{12.24}$$

In other words, we measure the signal $y(t)$ once every hth of a second for a total time of T. This correspondingly define the signal's period T and the *sampling rate s*:

$$T \overset{\text{def}}{=} Nh \ , \quad s = \frac{N}{T} = \frac{1}{h} \ . \tag{12.25}$$

Regardless of the true periodicity of the signal, when we choose a period T over which to sample the signal, the mathematics will inevitably produce a $y(t)$ that is periodic with period T,

$$y(t + T) = y(t) \ . \tag{12.26}$$

We recognize this periodicity, and ensure that there are only N independent measurements used in the transform, by defining the first and last y's to be equal:

$$y_0 = y_N . \tag{12.27}$$

If we are analyzing a truly periodic function, then the N points should span one complete period, but not more. This guarantees their independence. Unless we make further assumptions, the N independent data $y(t_k)$ can determine no more than N independent transform values $Y(\omega_k)$, $k = 0, \ldots, N$.

The time interval T (which should be the period for periodic functions) is the largest time over which we measure the variation of $y(t)$. Consequently, it determines the lowest frequency contained in our Fourier representation of $y(t)$,

$$\omega_1 = \frac{2\pi}{T} \ . \tag{12.28}$$

The full range of frequencies in the spectrum ω_n are determined by the number of samples taken, and by the total sampling time $T = Nh$ as

$$\omega_n = n\omega_1 = n\frac{2\pi}{Nh} \ , \quad n = 0, 1, \ldots, N \ . \tag{12.29}$$

Here $\omega_0 = 0$ corresponds to the zero-frequency or DC component of the transform, that is, the part of the signal that does not oscillate.

The discrete Fourier transform (DFT) algorithm follows from two approximations. First we evaluate the integral in (12.19) from time 0 to time T, over which the signal is measured, and not from $-\infty$ to $+\infty$. Second, the trapezoid rule is used for the integration[4]:

$$Y(\omega_n) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} \mathrm{d}t \frac{\mathrm{e}^{-\mathrm{i}\omega_n t}}{\sqrt{2\pi}} y(t) \simeq \int_0^T \mathrm{d}t \frac{\mathrm{e}^{-\mathrm{i}\omega_n t}}{\sqrt{2\pi}} y(t) \ , \tag{12.30}$$

$$\simeq \sum_{k=1}^{N} h y(t_k) \frac{\mathrm{e}^{-\mathrm{i}\omega_n t_k}}{\sqrt{2\pi}} = h \sum_{k=1}^{N} y_k \frac{\mathrm{e}^{-2\pi \mathrm{i} kn/N}}{\sqrt{2\pi}} \ . \tag{12.31}$$

To keep the final notation more symmetric, the step size h is factored from the transform Y and a discrete function Y_n is defined as

$$\boxed{Y_n \stackrel{\text{def}}{=} \frac{1}{h} Y(\omega_n) = \sum_{k=1}^{N} y_k \frac{\mathrm{e}^{-2\pi \mathrm{i} kn/N}}{\sqrt{2\pi}} \ , \quad n = 0, 1 \ldots, N \ .} \tag{12.32}$$

With this same care in accounting, and with $\mathrm{d}\omega \to 2\pi/Nh$, we invert the Y_n's:

$$y(t) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} \mathrm{d}\omega \frac{\mathrm{e}^{\mathrm{i}\omega t}}{\sqrt{2\pi}} Y(\omega) \tag{12.33}$$

$$\Rightarrow \quad y(t) \simeq \sum_{n=1}^{N} \frac{2\pi}{Nh} \frac{\mathrm{e}^{\mathrm{i}\omega_n t}}{\sqrt{2\pi}} Y(\omega_n) \ . \tag{12.34}$$

4) The alert reader may be wondering what has happened to the $h/2$ with which the trapezoid rule weights the initial and final points. Actually, they are there, but because we have set $y_0 \equiv y_N$, two $h/2$ terms have been added to produce one h term.

Once we know the N values of the transform, we can use (12.34) to evaluate $y(t)$ for any time t. There is nothing illegal about evaluating Y_n and y_k for arbitrarily large values of n and k, yet there is also nothing to be gained either. Because the trigonometric functions are periodic, we just get the old answers:

$$y(t_{k+N}) = y((k+N)h) = y(t_k)\ , \tag{12.35}$$

$$Y(\omega_{n+N}) = Y((n+N)\omega_1) = Y(\omega_n)\ . \tag{12.36}$$

Another way of stating this is to observe that none of the equations change if we replace $\omega_n t$ by $\omega_n t + 2\pi n$. There are still just N independent output numbers for N independent inputs, and so the transform and the reconstituted signal are periodic.

We see from (12.29) that the larger we make the time $T = Nh$ over which we sample the function, the smaller will be the frequency steps or resolution.[5] Accordingly, if you want a smooth frequency spectrum, you need to have a small frequency step $2\pi/T$, which means a longer observation time T. While the best approach would be to measure the input signal for all times, in practice a measured signal $y(t)$ is often extended in time ("padded") by adding zeros for times beyond the last measured signal, which thereby increases the value of T artificially. Although this does not add new information to the analysis, it does build in the experimentalist's view that the signal has no existence, or no meaning, at times after the measurements are stopped.

While periodicity is expected for a Fourier *series*, it is somewhat surprising for Fourier a *integral*, which have been touted as the right tool for nonperiodic functions. Clearly, if we input values of the signal for longer lengths of time, then the inherent period becomes longer, and if the repeat period T is very long, it may be of little consequence for times short compared to the period. If $y(t)$ is actually periodic with period Nh, then the DFT is an excellent way of obtaining Fourier series. If the input function is not periodic, then the DFT can be a bad approximation near the endpoints of the time interval (after which the function will repeat) or, correspondingly, for the lowest frequencies.

The DFT and its inverse can be written in a concise and insightful way, and be evaluated efficiently, by introducing a complex variable Z for the exponential and then raising Z to various powers:

$$\boxed{y_k = \frac{\sqrt{2\pi}}{N} \sum_{n=1}^{N} Z^{-nk} Y_n\ , \quad Z = \mathrm{e}^{-2\pi \mathrm{i}/N}\ ,} \tag{12.37}$$

$$\boxed{Y_n = \frac{1}{\sqrt{2\pi}} \sum_{k=1}^{N} Z^{nk} y_k\ , \quad Z^{nk} \equiv [(Z)^n]^k\ .} \tag{12.38}$$

With this formulation, the computer needs to compute only powers of Z. We give our DFT code in Listing 12.1. If your preference is to avoid complex numbers,

5) See also Section 12.5.1 where we discuss the related phenomenon of aliasing.

we can rewrite (12.37) in terms of separate real and imaginary parts by applying Euler's theorem with $\theta \overset{\text{def}}{=} 2\pi/N$:

$$Z = \mathrm{e}^{-\mathrm{i}\theta}\ , \quad \Rightarrow Z^{\pm nk} = \mathrm{e}^{\mp \mathrm{i}nk\theta} = \cos nk\theta \mp \mathrm{i}\sin nk\theta\ , \tag{12.39}$$

$$\Rightarrow \quad Y_n = \frac{1}{\sqrt{2\pi}} \sum_{k=1}^{N} \big[\cos(nk\theta)\mathrm{Re}\ y_k + \sin(nk\theta)\,\mathrm{Im}\ y_k + \mathrm{i}(\cos(nk\theta)\,\mathrm{Im}\ y_k - \sin(nk\theta)\mathrm{Re}\, y_k)\big]\ , \tag{12.40}$$

$$y_k = \frac{\sqrt{2\pi}}{N} \sum_{n=1}^{N} \big[\cos(nk\theta)\mathrm{Re}\ Y_n - \sin(nk\theta)\mathrm{Im}\ Y_n + \mathrm{i}(\cos(nk\theta)\mathrm{Im}\, Y_n + \sin(nk\theta)\mathrm{Re}\ Y_n)\big]\ . \tag{12.41}$$

Readers new to DFTs are often surprised when they apply these equations to practical situations and end up with transforms Y having imaginary parts, despite the fact that the signal y is real. Equation 12.40 should make it clear that a real signal (Im $y_k \equiv 0$) will yield an imaginary transform unless $\sum_{k=1}^{N} \sin(nk\theta)\mathrm{Re}\ y_k = 0$. This occurs only if $y(t)$ is an *even* function over $-\infty \le t \le +\infty$ *and* we integrate exactly. Because neither condition holds, the DFTs of real, even functions may have small imaginary parts. This is not as a result of an error in programming, and in fact is a good measure of the approximation error in the entire procedure.

The computation time for a DFT can be reduced even further by using *fast Fourier transform* algorithm, as discussed in Section 12.9. An examination of (12.37) shows that the DFT is evaluated as a matrix multiplication of a vector of length N containing the Z values by a vector of length N of y value. The time for this DFT scales like N^2, while the time for the FFT algorithm scales as $N \log_2 N$. Although this may not seem like much of a difference, for $N = 10^{2-3}$, the difference of 10^{3-5} is the difference between a minute and a week. For this reason, it is the FFT is often used for online spectrum analysis.

Listing 12.1 DFTcomplex.py uses the built-in complex numbers of Python to compute the DFT for the signal in method f(signal).

```
# DFTcomplex.py: Discrete Fourier Transform with built in complex from
     visual import * from
visual.graph import * import cmath # complex math

signgr = gdisplay(x=0, y=0, width=600, height=250, title ='Signal',\
             xtitle='x', ytitle = 'signal', xmax = 2.*math.pi, xmin = 0,\
      ymax = 30, ymin = 0)
sigfig = gcurve(color=color.yellow, display=signgr)
imagr = gdisplay(x=0,y=250,width=600,height=250,title ='Imag Fourier TF',
       xtitle = 'x',ytitle='TF.Imag',xmax=10.,xmin=-1,ymax=100,ymin=-0.2)
impart = gvbars(delta = 0.05, color = color.red, display = imagr)

N = 50;                    Np = N
signal = zeros( (N+1), float )
twopi  = 2.*pi;          sq2pi = 1./sqrt(twopi);          h = twopi/N
dftz   = zeros( (Np), complex )                  # Complex elements
```

```
def f(signal):                                                    # Signal
    step = twopi/N;          x = 0.
    for i in range(0, N+1):
        signal[i] = 30*cos(x*x*x*x)
        sigfig.plot(pos = (x, signal[i]))                         # Plot
        x += step

def fourier(dftz):                                                # DFT
    for n in range(0, Np):
      zsum = complex(0.0, 0.0)
      for  k in range(0, N):
          zexpo = complex(0, twopi*k*n/N)            # Complex exponent
          zsum += signal[k]*exp(-zexpo)
      dftz[n] = zsum * sq2pi
      if dftz[n].imag != 0:
          impart.plot(pos = (n, dftz[n].imag) )                   # Plot

f(signal);        fourier(dftz)              # Call signal, transform
```

12.5.1 Aliasing (Assessment)

The sampling of a signal by DFT for only a finite number of times (large Δt) limits the accuracy of the deduced high-frequency components present in the signal. Clearly, good information about very high frequencies requires sampling the signal with small time steps so that all the wiggles can be included. While a poor deduction of the high-frequency components may be tolerable if all we care about are the low-frequency components, the inaccurate high-frequency components remain present in the signal and may contaminate the low-frequency components that we deduce. This effect is called *aliasing* and is the cause of the Moiré pattern distortion in digital images.

As an example, consider Figure 12.2 showing the two functions $\sin(\pi t/2)$ and $\sin(2\pi t)$ for $0 \le t \le 8$, with their points of overlap in bold. If we were unfortunate enough to sample a signal containing these functions at the times $t = 0, 2, 4, 6, 8$, then we would measure $y \equiv 0$ and assume that there was no signal at all. However, if we were unfortunate enough to measure the signal at the filled dots in Figure 12.2 where $\sin(\pi t/2) = \sin(2\pi t)$, specifically, $t = 0, 12/10, 4/3, \ldots$, then our Fourier analysis would completely miss the high-frequency component. In DFT jargon, we would say that the high-frequency component has been *aliased* by the low-frequency component. In other cases, some high-frequency values may be included in our sampling of the signal, but our sampling rate may not be high enough to include enough of them to separate the high-frequency component properly. In this case some high-frequency signals would be included spuriously as part of the low-frequency spectrum, and this would lead to spurious low-frequency oscillations when the signal is synthesized from its Fourier components.

More precisely, aliasing occurs when a signal containing frequency f is sampled at a rate of $s = N/T$ measurements per unit time, with $s \le f/2$. In this case, the frequencies f and $f - 2s$ yield the same DFT, and we would not be able to

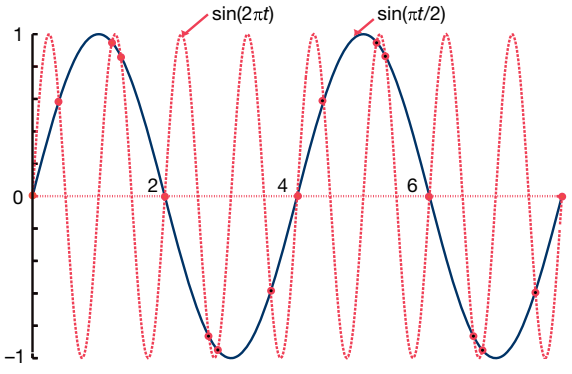

Figure 12.2 A plot of the functions $\sin(\pi t/2)$ and $\sin(2\pi t)$. If the sampling rate is not high enough, these signals may appear indistinguishable in a Fourier decomposition. If the sample rate is too low and if both signals are present in a sample, the deduced low-frequency component may be contaminated by the higher frequency component signal.

determine that there are two frequencies present. That being the case, to avoid aliasing we want no frequencies $f > s/2$ to be present in our input signal. This is known as the *Nyquist criterion*. In practice, some applications avoid the effects of aliasing by filtering out the high frequencies from the signal and then analyzing only the remaining low-frequency part. (The low-frequency *sinc filter* discussed in Section 12.8.1 is often used for this purpose.) Although filtering eliminates some high-frequency information, it lessens the distortion of the low-frequency components, and so may lead to improved reproduction of the signal.

If accurate values for the high frequencies are required, then we will need to increase the sampling rate s by increasing the number N of samples taken within the fixed sampling time $T = Nh$. By keeping the sampling time constant and increasing the number of samples taken, we make the time step h smaller and we pick up the higher frequencies. By increasing the number N of frequencies and that you compute, you move the previous higher frequency components in closer to the middle of the spectrum, and thus away from the error-prone ends.

If we increase the total time sampling time $T = Nh$ and keep h the same, then the sampling rate $s = N/T = 1/h$ remains the same. Because $\omega_1 = 2\pi/T$, this makes ω_1 smaller, which means we have more low frequencies recorded and a smoother frequency spectrum. And as we said, this is often carried out, after the fact, by padding the end of the data set with zeros.

Listing 12.2 DFTreal.py computes the discrete Fourier transform for the signal in method f(signal) using real numbers.

```
# DFTreal.py: Discrete Fourier Transform using real numbers

from visual.graph import *

signgr = gdisplay(x=0,y=0,width=600,height=250, \
        title='Signal y(t)= 3 cos(wt)+2 cos(3wt)+ cos(5wt) ',\
  xtitle='x', ytitle='signal',xmax=2.*math.pi,xmin=0,ymax=7,ymin=-7)
```

```
sigfig = gcurve(color=color.yellow,display=signgr)
imagr = gdisplay(x=0,y=250,width=600,height=250,\
      title='Fourier transform imaginary part',xtitle='x',\
        ytitle='Transf.Imag',xmax=10.0,xmin=-1,ymax=20,ymin=-25)
impart = gvbars(delta=0.05,color=color.red,display=imagr)

N = 200
Np = N
signal = zeros((N+1),float)
twopi = 2.*pi
sq2pi = 1./sqrt(twopi)
h = twopi/N
dftimag = zeros((Np),float)                          # Im. transform

def f(signal):
    step = twopi/N
    t= 0.
    for i in range(0,N+1):
       signal[i] = 3*sin(t*t*t)
       sigfig.plot(pos=(t,signal[i]))
       t += step

def fourier(dftimag):                                # DFT
    for n in range(0,Np):
      imag = 0.
      for k in range(0, N):
        imag += signal[k]*sin((twopi*k*n)/N)
      dftimag[n] = -imag*sq2pi                       # Im transform
      if dftimag[n] !=0:
          impart.plot(pos=(n,dftimag[n]))

f(signal)
fourier(dftimag)
```

12.5.2 Fourier Series DFT (Example)

For simplicity, let us consider the Fourier cosine series:

$$y(t) = \sum_{n=0}^{\infty} a_n \cos(n\omega t) , \quad a_k = \frac{2}{T}\int_0^T \mathrm{d}t \cos(k\omega t) y(t) . \tag{12.42}$$

Here $T \overset{\text{def}}{=} 2\pi/\omega$ is the actual period of the system (not necessarily the period of the simple harmonic motion occurring for a small amplitude). We assume that the function $y(t)$ is sampled for a discrete set of times

$$y(t = t_k) \equiv y_k , \quad k = 0, 1, \ldots, N . \tag{12.43}$$

Because we are analyzing a periodic function, we retain the conventions used in the DFT and require the function to repeat itself with period $T = Nh$; that is, we assume that the amplitude is the same at the first and last points:

$$y_0 = y_N . \tag{12.44}$$

This means that there are only N independent values of y being used as input. For these N independent y_k values, we can determine uniquely only N expansion co-

efficients a_k. If we use the trapezoid rule to approximate the integration in (12.42), we determine the N independent Fourier components as

$$a_n \simeq \frac{2h}{T}\sum_{k=1}^{N}\cos(n\omega t_k)y(t_k) = \frac{2}{N}\sum_{k=1}^{N}\cos\left(\frac{2\pi nk}{N}\right)y_k\ , \quad n = 0,\ldots,N\ . \tag{12.45}$$

Because we can determine only N Fourier components from N independent $y(t)$ values, our Fourier series for the $y(t)$ must be in terms of only these components:

$$y(t) \simeq \sum_{n=0}^{N} a_n\cos(n\omega t) = \sum_{n=0}^{N} a_n\cos\left(\frac{2\pi nt}{Nh}\right)\ . \tag{12.46}$$

In summary, we sample the function $y(t)$ at N times, $t_1,\ldots,t_N$. We see that all the values of y sampled contribute to each a_k. Consequently, if we increase N in order to determine more coefficients, we must recompute all the a_n values. In the wavelet analysis in Chapter 13, the theory is reformulated so that additional samplings determine higher spectral components without affecting lower ones.

12.5.3 Assessments

Simple analytic input It is always good to do simple checks before examining more complex problems, even if you are using a package's Fourier tool.

1. Sample the even signal

$$y(t) = 3\cos(\omega t) + 2\cos(3\omega t) + \cos(5\omega t)\ . \tag{12.47}$$

   a) Decompose this into its components.
   b) Check that the components are essentially real and in the ratio 3 : 2 : 1 (or 9 : 4 : 1 for the power spectrum).
   c) Verify that the frequencies have the expected values (not just ratios).
   d) Verify that the components resum to give the input signal.
   e) Experiment on the separate effects of picking different values of the step size h and of enlarging the measurement period $T = Nh$.
2. Sample the odd signal

$$y(t) = \sin(\omega t) + 2\sin(3\omega t) + 3\sin(5\omega t)\ . \tag{12.48}$$

   Decompose this into its components; then check that they are essentially imaginary and in the ratio 1 : 2 : 3 (or 1 : 4 : 9 if a power spectrum is plotted) and that they resum to give the input signal.
3. Sample the mixed-symmetry signal

$$y(t) = 5\sin(\omega t) + 2\cos(3\omega t) + \sin(5\omega t)\ . \tag{12.49}$$

Decompose this into its components; then check that there are three of them in the ratio 5 : 2 : 1 (or 25 : 4 : 1 if a power spectrum is plotted) and that they resum to give the input signal.

4. Sample the signal

$$y(t) = 5 + 10\sin(t + 2)\ .$$

Compare and explain the results obtained by sampling (a) without the 5, (b) as given but without the 2, and (c) without the 5 and without the 2.

5. In our discussion of aliasing, we examined Figure 12.2 showing the functions $\sin(\pi t/2)$ and $\sin(2\pi t)$. Sample the function

$$y(t) = \sin\left(\frac{\pi}{2}t\right) + \sin(2\pi t) \tag{12.50}$$

and explore how aliasing occurs. Explicitly, we know that the true transform contains peaks at $\omega = \pi/2$ and $\omega = 2\pi$. Sample the signal at a rate that leads to aliasing, as well as at a higher sampling rate at which there is no aliasing. Compare the resulting DFTs in each case and check if your conclusions agree with the Nyquist criterion.

Highly nonlinear oscillator Recall the numerical solution for oscillations of a spring with power $p = 12$ (see (12.1)). Decompose the solution into a Fourier series and determine the number of higher harmonics that contribute at least 10%; for example, determine the n for which $|b_n/b_1| < 0.1$. Check that resuming the components reproduces the signal.

Nonlinearly perturbed oscillator Remember the harmonic oscillator with a nonlinear perturbation (8.2):

$$V(x) = \frac{1}{2}kx^2\left(1 - \frac{2}{3}\alpha x\right)\ , \quad F(x) = -kx(1 - \alpha x)\ . \tag{12.51}$$

For very small amplitudes of oscillation ($x \ll 1/\alpha$), the solution $x(t)$ essentially should be only the first term of a Fourier series.

1. We want the signal to contain "approximately 10% nonlinearity." This being the case, fix your value of α so that $\alpha x_{\max} \simeq 10\%$, where $x_{\max}$ is the maximum amplitude of oscillation. For the rest of the problem, keep the value of α fixed.
2. Decompose your numerical solution into a discrete Fourier spectrum.
3. Plot a graph of the percentage of importance of the first *two*, non-DC Fourier components as a function of the initial displacement for $0 < x_0 < 1/2\alpha$. You should find that higher harmonics are more important as the amplitude increases. Because both even and odd components are present, Y_n should be complex. Because a 10% effect in amplitude becomes a 1% effect in power, make sure that you make a semilog plot of the power spectrum.
4. As always, check that resumations of your transforms reproduce the signal.

(*Warning:* The ω you use in your series must correspond to the *true* frequency of the system, not the ω_0 of small oscillations.)

12.5.4 Nonperiodic Function DFT (Exploration)

Consider a simple model (a wave packet) of a "localized" electron moving through space and time. A good model for an electron initially localized around $x = 5$ is a Gaussian multiplying a plane wave:

$$\psi(x, t = 0) = \exp\left[-\frac{1}{2}\left(\frac{x - 5.0}{\sigma_0}\right)^2\right] e^{ik_0 x} \ . \tag{12.52}$$

This wave packet is not an eigenstate of the momentum operator[6] $p = i\,d/dx$ and in fact contains a spread of momenta. Your *problem* is to evaluate the Fourier transform

$$\psi(p) = \int_{-\infty}^{+\infty} dx \frac{e^{ipx}}{\sqrt{2\pi}} \psi(x) \ , \tag{12.53}$$

as a way of determining the momenta components in (12.52).

12.6 Filtering Noisy Signals

You measure a signal $y(t)$ that obviously contains noise. Your *problem* is to determine the frequencies that would be present in the spectrum of the signal if the signal did not contain noise. Of course, once you have a Fourier transform from which the noise has been removed, you can transform it to obtain a signal $s(t)$ with no noise.

In the process of solving this problem, we examine two simple approaches: the use of autocorrelation functions and the use of filters. Both approaches find wide applications in science, with our discussion not doing the subjects justice. We will see filters again in the discussion of wavelets in Chapter 13.

12.7 Noise Reduction via Autocorrelation (Theory)

We assume that the measured signal is the sum of the true signal $s(t)$, which we wish to determine, plus some unwelcome *noise* $n(t)$:

$$y(t) = s(t) + n(t) \ . \tag{12.54}$$

Our first approach at removing the noise relies on that fact that noise is a random process and thus should not be correlated with the signal. Yet what do we

6) We use natural units in which $\hbar = 1$.

mean when we say that two functions are not *correlated?* Well, if the two tend to oscillate with their nodes and peaks in much the same places, then the two functions are clearly correlated. An analytic measure of the correlation of two arbitrary functions $y(t)$ and $x(t)$ is the *correlation function*

$$c(\tau) = \int_{-\infty}^{+\infty} \mathrm{d}t\; y^*(t)x(t+\tau) \equiv \int_{-\infty}^{+\infty} \mathrm{d}t\, y^*(t-\tau)x(t)\;, \tag{12.55}$$

where τ, the *lag time*, is a variable. Even if the two signals have different magnitudes, if they have similar time dependences except for one lagging or leading the other, then for certain values of τ the integrand in (12.55) will be positive for all values of t. For those values of τ, the two signals interfere constructively and produce a large value for the correlation function. In contrast, if both functions oscillate independently regardless of the value of τ, then it is just as likely for the integrand to be positive as to be negative, in which case the two signals interfere destructively and produce a small value for the integral.

Before we apply the correlation function to our problem, let us study some of its properties. We use (12.18) to express c, y^*, and x in terms of their Fourier transforms:

$$c(\tau) = \int_{-\infty}^{+\infty} \mathrm{d}\omega''\, C(\omega'')\frac{\mathrm{e}^{\mathrm{i}\omega'' t}}{\sqrt{2\pi}}\;, \quad y^*(t) = \int_{-\infty}^{+\infty} \mathrm{d}\omega\, Y^*(\omega)\frac{\mathrm{e}^{-\mathrm{i}\omega t}}{\sqrt{2\pi}}\;,$$
$$x(t+\tau) = \int_{-\infty}^{+\infty} \mathrm{d}\omega'\; X(\omega')\frac{\mathrm{e}^{+\mathrm{i}\omega t}}{\sqrt{2\pi}}\;. \tag{12.56}$$

Because ω, ω', and ω'' are dummy variables, other names may be used for these variables without changing any results. When we substitute these representations into definition (12.55) of the correlation function and assume that the resulting integrals converge well enough to be rearranged, we obtain

$$\begin{aligned}\int_{-\infty}^{+\infty} \mathrm{d}\omega C(\omega)\mathrm{e}^{\mathrm{i}\omega t} &= \int_{-\infty}^{+\infty} \frac{\mathrm{d}\omega}{2\pi} \int_{-\infty}^{+\infty} \mathrm{d}\omega' Y^*(\omega)X(\omega')\mathrm{e}^{\mathrm{i}\omega\tau}2\pi\delta(\omega'-\omega) \\ &= \int_{-\infty}^{+\infty} \mathrm{d}\omega\, Y^*(\omega)X(\omega)\mathrm{e}^{\mathrm{i}\omega\tau}\;, \\ \Rightarrow \quad C(\omega) &= \sqrt{2\pi}\, Y^*(\omega)X(\omega)\;,\end{aligned} \tag{12.57}$$

where the last line follows because ω'' and ω are equivalent dummy variables. Equation 12.57 says that the Fourier transform of the correlation function between two signals is proportional to the product of the transform of one signal and the complex conjugate of the transform of the other. (We shall see a related convolution theorem for filters.)

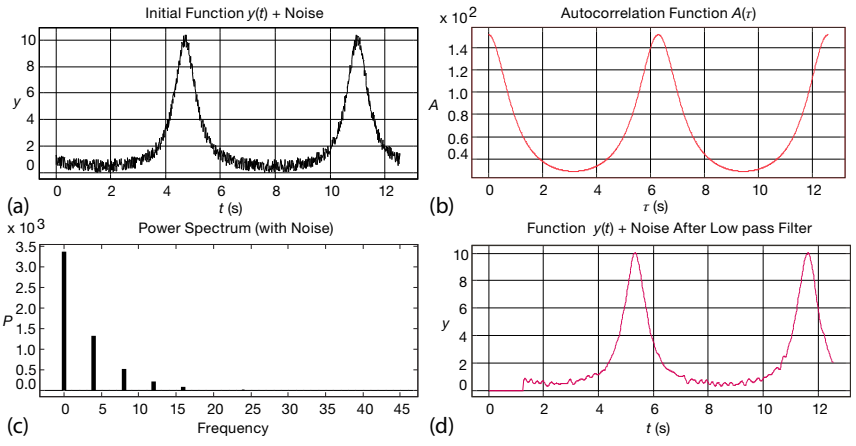

Figure 12.3 (a) A function that is a signal plus noise $s(t)+n(t)$; (b) the autocorrelation function vs. time deduced by processing this signal; (c) the power spectrum obtained from autocorrelation function; (d) the signal plus noise after passage through a low-pass filter.

A special case of the correlation function $c(\tau)$ is the *autocorrelation function* $A(\tau)$. It measures the correlation of a time signal with itself:

$$\boxed{A(\tau) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} \mathrm{d}t\, y^*(t)y(t+\tau) \equiv \int_{-\infty}^{+\infty} \mathrm{d}t\, y(t)y^*(t-\tau)\ .} \tag{12.58}$$

This function is computed by taking a signal $y(t)$ that has been measured over some time period and then averaging it over time using $y(t+\tau)$ as a weighting function. This process is also called *folding* a function onto itself (as might be done with dough) or a *convolution*. To see how this folding removes noise from a signal, we go back to the measured signal (12.54), which was the sum of pure signal plus noise $s(t)+n(t)$. As an example, in Figure 12.3a we show a signal that was constructed by adding random noise to a smooth signal. When we compute the autocorrelation function for this signal, we obtain a function (Figure 12.3b) that looks like a broadened, smoothed version of the signal $y(t)$.

We can understand how the noise is removed by taking the Fourier transform of $s(t)+n(t)$ to obtain a simple sum of transforms:

$$Y(\omega) = S(\omega) + N(\omega)\ , \tag{12.59}$$

$$\begin{bmatrix} S(\omega) \\ N(\omega) \end{bmatrix} = \int_{-\infty}^{+\infty} \mathrm{d}t \begin{bmatrix} s(t) \\ n(t) \end{bmatrix} \frac{\mathrm{e}^{-\mathrm{i}\omega t}}{\sqrt{2\pi}}\ . \tag{12.60}$$

Because the autocorrelation function (12.58) for $y(t) = s(t) + n(t)$ involves the second power of y, is not a linear function, that is, $A_y \neq A_s + A_n$, but instead

$$A_y(\tau) = \int_{-\infty}^{+\infty} \mathrm{d}t[s(t)s(t+\tau) + s(t)n(t+\tau) + n(t)n(t+\tau)]\ . \tag{12.61}$$

If we assume that the noise $n(t)$ in the measured signal is truly random, then it should average to zero over long times and be uncorrelated at times t and $t + \tau$. This being the case, both integrals involving the noise vanish, and so

$$A_y(\tau) \simeq \int_{-\infty}^{+\infty} \mathrm{d}t\, s(t)\, s(t+\tau) = A_s(\tau)\ . \tag{12.62}$$

Thus, the part of the noise that is random tends to be averaged out of the autocorrelation function, and we are left with an approximation of the autocorrelation function of the pure signal.

So how does this help us? Application of (12.57) with $Y(\omega) = X(\omega) = S(\omega)$ tells us that the Fourier transform $A(\omega)$ of the autocorrelation function is proportional to $|S(\omega)|^2$:

$$A(\omega) = \sqrt{2\pi}\,|S(\omega)|^2\ . \tag{12.63}$$

The function $|S(\omega)|^2$ is the *power spectrum* of the pure signal. Thus, evaluation of the autocorrelation function of the noisy signal gives us the pure signal's power spectrum, which is often all that we need to know. For example, in Figure 12.3a we see a noisy signal, the autocorrelation function (Figure 12.3b), which clearly is smoother than the signal, and finally, the deduced power spectrum (Figure 12.3c). Note that the broadband high-frequency components characteristic of noise are absent from the power spectrum.

You can easily modify the sample program `DFTcomplex.py` in Listing 12.1 to compute the autocorrelation function and then the power spectrum from $A(\tau)$. We present a program `NoiseSincFilter.py` on the instructor's site that does this.

12.7.1 Autocorrelation Function Exercises

1. Imagine that you have sampled the pure signal

$$s(t) = \frac{1}{1 - 0.9\sin t}\ . \tag{12.64}$$

Although there is just a single sine function in the denominator, there is an infinite number of overtones as follows from the expansion

$$s(t) \simeq 1 + 0.9\sin t + (0.9\sin t)^2 + (0.9\sin t)^3 + \cdots \tag{12.65}$$

a) Compute the DFT $S(\omega)$. Make sure to sample just one period but to cover the entire period. Make sure to sample at enough times (fine scale) to obtain good sensitivity to the high-frequency components.
b) Make a semilog plot of the power spectrum $|S(\omega)|^2$.
c) Take your input signal $s(t)$ and compute its autocorrelation function $A(\tau)$ for a full range of τ values (an analytic solution is okay too).

d) Compute the power spectrum indirectly by performing a DFT on the autocorrelation function. Compare your results to the spectrum obtained by computing $|S(\omega)|^2$ directly.

2. Add some random noise to the signal using a random number generator:

$$y(t_i) = s(t_i) + \alpha(2r_i - 1) , \quad 0 \le r_i \le 1 , \tag{12.66}$$

where α is an adjustable parameter. Try several values of α, from small values that just add some fuzz to the signal to large values that nearly hide the signal.

a) Plot your noisy data, their Fourier transform, and their power spectrum obtained directly from the transform with noise.
b) Compute the autocorrelation function $A(\tau)$ and its Fourier transform $A(\omega)$.
c) Compare the DFT of $A(\tau)$ to the true power spectrum and comment on the effectiveness of reducing noise by use of the autocorrelation function.
d) For what value of α do you essentially lose all the information in the input?

12.8 Filtering with Transforms (Theory)

A filter (Figure 12.4) is a device that converts an input signal $f(t)$ to an output signal $g(t)$ with some specific property for $g(t)$. More specifically, an *analog filter* is defined (Hartmann, 1998) as integration over the input function:

$$g(t) = \int_{-\infty}^{+\infty} \mathrm{d}\tau \, f(\tau)\, h(t-\tau) \stackrel{\text{def}}{=} f(t) * h(t) . \tag{12.67}$$

The operation indicated in (12.67) occurs often enough that it is given the name *convolution* and is denoted by an asterisk $*$. The function $h(t)$ is called the *response* or *transfer function* of the filter because it is the response of the filter to a unit impulse:

$$h(t) = \int_{-\infty}^{+\infty} \mathrm{d}\tau \, \delta(\tau)\, h(t-\tau) . \tag{12.68}$$

Equation 12.67 states that the output $g(t)$ of a filter equals the input $f(t)$ convoluted with the transfer function $h(t-\tau)$. Because the argument of the response function is delayed by a time τ relative to that of the signal in integral (12.67), τ is called the *lag time*. While the integration is over all times, the response of a good detector usually peaks around zero time. In any case, the response must equal zero for $\tau > t$ because events in the future cannot affect the present (causality).

The *convolution theorem* states that the Fourier transform of the convolution $g(t)$ is proportional to the product of the transforms of $f(t)$ and $h(t)$:

$$G(\omega) = \sqrt{2\pi} F(\omega) H(\omega) . \tag{12.69}$$

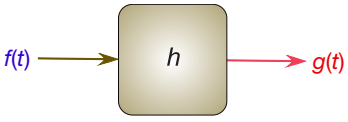

Figure 12.4 A schematic of an input signal $f(t)$ passing through a filter h that outputs the function $g(t)$.

The theorem results from expressing the functions in (12.67) by their transforms and using the resulting Dirac delta function to evaluate an integral (essentially what we did in our discussion of correlation functions).

Regardless of the domain used, filtering as we have defined it is a linear process involving just the first powers of f. This means that the output at one frequency is proportional to the input at that frequency. The constant of proportionality between the two may change with frequency and thus suppress specific frequencies relative to others, but that constant remains fixed in time. Because the law of linear superposition is valid for filters, if the input to a filter is represented as the sum of various functions, then the transform of the output will be the sum of the functions' Fourier transforms.

Filters that remove or decrease high-frequency components more than they do low-frequency ones, are called *low-pass* filters. Those that filter out the low frequencies are called *high-pass filters*. A simple low-pass filter is the RC circuit shown in Figure 12.5a. It produces the transfer function

$$H(\omega) = \frac{1}{1 + \mathrm{i}\omega\tau} = \frac{1 - \mathrm{i}\omega\tau}{1 + \omega^2\tau^2}\,, \tag{12.70}$$

where $\tau = \mathrm{RC}$ is the time constant. The ω^2 in the denominator leads to a decrease in the response at high frequencies and therefore makes this a low-pass filter (the $\mathrm{i}\omega$ affects only the phase). A simple high-pass filter is the RC circuit shown in Figure 12.5b. It produces the transfer function

$$H(\omega) = \frac{\mathrm{i}\omega\tau}{1 + \mathrm{i}\omega\tau} = \frac{\mathrm{i}\omega\tau + \omega^2\tau^2}{1 + \omega^2\tau^2}\,. \tag{12.71}$$

$H = 1$ at large ω, yet H vanishes as $\omega \to 0$, as expected for a high-pass filter.

Filters composed of resistors and capacitors are fine for analog signal processing. For digital processing we want a *digital filter* that has a specific response function for each frequency range. A physical model for a digital filter may be constructed from a delay line with taps at various spacing along the line (Figure 12.6)

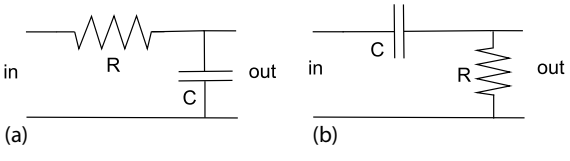

Figure 12.5 (a) An RC circuit arranged as a low-pass filter. (b) An RC circuit arranged as a high-pass filter.

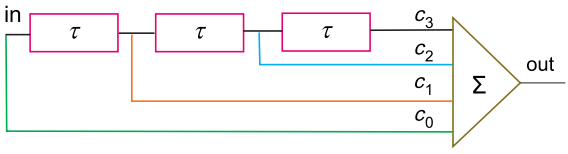

Figure 12.6 A delay-line filter in which the signal at different times is scaled by different amounts c_i.

(Hartmann, 1998). The signal read from tap n is just the input signal delayed by time $n\tau$, where the delay time τ is a characteristic of the particular filter. The output from each tap is described by the transfer function $\delta(t - n\tau)$, possibly with scaling factor c_n. As represented by the triangle in Figure 12.6b, the signals from all taps are ultimately summed together to form the total response function:

$$h(t) = \sum_{n=0}^{N} c_n \delta(t - n\tau) \ . \tag{12.72}$$

In the frequency domain, the Fourier transform of a delta function is an exponential, and so (12.72) results in the transfer function

$$H(\omega) = \sum_{n=0}^{N} c_n \mathrm{e}^{-\mathrm{i}n\omega\tau} \ , \tag{12.73}$$

where the exponential indicates the phase shift from each tap.

If a digital filter is given a continuous time signal $f(t)$ as input, its output will be the discrete sum

$$g(t) = \int_{-\infty}^{+\infty} \mathrm{d}t' \, f(t') \sum_{n=0}^{N} c_n \delta(t - t' - n\tau) = \sum_{n=0}^{N} c_n f(t - n\tau) \ . \tag{12.74}$$

And of course, if the signal's input is a discrete sum, its output will remain a discrete sum. In either case, we see that knowledge of the filter coefficients c_i provides us with all we need to know about a digital filter. If we look back at our work on the DFT in Section 12.5, we can view a digital filter (12.74) as a Fourier transform in which we use an N-point approximation to the Fourier integral. The c_ns then contain both the integration weights and the values of the response function at the integration points. The transform itself can be viewed as a filter of the signal into specific frequencies.

12.8.1
Digital Filters: Windowed Sinc Filters (Exploration) ⊙

Problem Construct digital versions of high-pass and low-pass filters and determine which filter works better at removing noise from a signal.

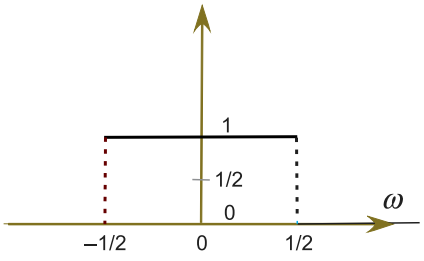

Figure 12.7 The rectangle function rect(ω) that is constant for a finite frequency interval. The Fourier transform of this function is sinc(t).

A popular way to separate the bands of frequencies in a signal is with a *windowed sinc filter* (Smith, 1999). This filter is based on the observation that an ideal *low-pass* filter passes all frequencies below a cutoff frequency ω_c, and blocks all frequencies above this frequency. And because there tends to be more noise at high frequencies than at low frequencies, removing the high frequencies tends to remove more noise than signal, although some signal is inevitably lost. One use for windowed sinc filters is in reducing aliasing in DFTs by removing the high-frequency component of a signal before determining its Fourier components. The graph in Figure 12.1b was obtained by passing our noisy signal through a sinc filter (using the program `NoiseSincFilter.py`).

If both positive and negative frequencies are included, an ideal low-frequency filter will look like the rectangular pulse in frequency space (Figure 12.7):

$$H(\omega, \omega_c) = \text{rect}\left(\frac{\omega}{2\omega_c}\right), \quad \text{rect}(\omega) = \begin{cases} 1, & \text{if } |\omega| \le \frac{1}{2}, \\ 0, & \text{otherwise}. \end{cases} \tag{12.75}$$

Here rect(ω) is the rectangular function. Although maybe not obvious, a rectangular pulse in the frequency domain has a Fourier transform that is proportional to the *sinc function* in the time domain (Smith, 1991)

$$\int_{-\infty}^{+\infty} \mathrm{d}\omega \mathrm{e}^{-\mathrm{i}\omega t} \text{rect}(\omega) = \text{sinc}\left(\frac{t}{2}\right) \stackrel{\text{def}}{=} \frac{\sin(\pi t/2)}{\pi t/2}, \tag{12.76}$$

where the πs are sometimes omitted. Consequently, we can filter out the high-frequency components of a signal by convoluting it with $\sin(\omega_c t)/(\omega_c t)$, a technique also known as the *Nyquist–Shannon* interpolation formula. In terms of discrete transforms, the time-domain representation of the sinc filter is simply

$$h[\mathrm{i}] = \frac{\sin(\omega_c \mathrm{i})}{\mathrm{i}\pi}. \tag{12.77}$$

Because all frequencies below the cutoff frequency ω_c are passed with unit amplitude, while all higher frequencies are blocked, we can see the importance of a sinc filter.

In practice, there are a number of problems in using sinc function as the filter. First, as formulated, the filter is *noncausal*; that is, there are coefficients at negative times, which is nonphysical because we do not start measuring the signal until $t = 0$. Second, in order to produce a perfect rectangular response, we would have to sample the signal at an infinite number of times. In practice, we sample at $(M + 1)$ points (M even) placed symmetrically around the main lobe of $\sin(\pi t)/\pi t$, and then shift times to purely positive values via

$$h[\mathrm{i}] = \frac{\sin[2\pi\omega_c(\mathrm{i} - M/2)]}{\mathrm{i} - M/2} \ , \quad 0 \le t \le M \ . \tag{12.78}$$

As might be expected, a penalty is incurred for making the filter discrete; instead of the ideal rectangular response, we obtain some *Gibbs overshoot*, with rounded corners and oscillations beyond the corner.

There are two ways to reduce the departures from the ideal filter. The first is to increase the length of times for which the filter is sampled, which inevitably leads to longer compute times. The other way is to smooth out the truncation of the sinc function by multiplying it with a smoothly tapered curve like the *Hamming window function*:

$$w[\mathrm{i}] = 0.54 - 0.46\cos(2\pi\mathrm{i}/M) \ . \tag{12.79}$$

In this way, the filter's kernel becomes

$$h[\mathrm{i}] = \frac{\sin[2\pi\omega_c(\mathrm{i} - M/2)]}{\mathrm{i} - M/2}\left[0.54 - 0.46\cos\left(\frac{2\pi\mathrm{i}}{M}\right)\right] \ . \tag{12.80}$$

The cutoff frequency ω_c should be a fraction of the sampling rate. The time length M determines the *bandwidth* over which the filter changes from 1 to 0.

Exercise Repeat the exercise that added random noise to a known signal, this time using the sinc filter to reduce the noise. See how small you can make the signal and still be able to separate it from the noise.

12.9 The Fast Fourier Transform Algorithm ⊙

We have seen in (12.37) that a discrete Fourier transform can be written in the compact form as

$$Y_n = \frac{1}{\sqrt{2\pi}} \sum_{k=1}^{N} Z^{nk} y_k \ , \quad Z = \mathrm{e}^{-2\pi \mathrm{i}/N} \ , \quad n = 0, 1, \ldots, N-1 \ . \tag{12.81}$$

Even if the signal elements y_k to be transformed are real, Z is complex, and therefore we must process both real and imaginary parts when computing transforms. Because both n and k range over N integer values, the $(Z^n)^k y_k$ multiplications in (12.81) require some N^2 multiplications and additions of complex numbers. As N gets large, as happens in realistic applications, this geometric increase in the number of steps leads to long computation times.

In 1965, Cooley and Tukey discovered an algorithm[7] that reduces the number of operations necessary to perform a DFT from N^2 to roughly $N \log_2 N$ (Cooley, 1965; Donnelly and Rust, 2005). Although this may not seem like such a big difference, it represents a 100-fold speedup for 1000 data points, which changes a full day of processing into 15 min of work. Because of its widespread use (including cell phones), the fast Fourier transform algorithm is considered one of the 10 most important algorithms of all time.

The idea behind the FFT is to utilize the periodicity inherent in the definition of the DFT (12.81) to reduce the total number of computational steps. Essentially, the algorithm divides the input data into two equal groups and transforms only one group, which requires $\sim (N/2)^2$ multiplications. It then divides the remaining (nontransformed) group of data in half and transforms them, continuing the process until all the data have been transformed. The total number of multiplications required with this approach is approximately $N \log_2 N$.

Specifically, the FFT's time economy arises from the computationally expensive complex factor $Z^{nk}[= ((Z)^n)^k]$ having values that are repeated as the integers n and k vary sequentially. For instance, for $N = 8$,

$$\begin{aligned}
Y_0 &= Z^0 y_0 + Z^0 y_1 + Z^0 y_2 + Z^0 y_3 + Z^0 y_4 + Z^0 y_5 + Z^0 y_6 + Z^0 y_7 \ , \\
Y_1 &= Z^0 y_0 + Z^1 y_1 + Z^2 y_2 + Z^3 y_3 + Z^4 y_4 + Z^5 y_5 + Z^6 y_6 + Z^7 y_7 \ , \\
Y_2 &= Z^0 y_0 + Z^2 y_1 + Z^4 y_2 + Z^6 y_3 + Z^8 y_4 + Z^{10} y_5 + Z^{12} y_6 + Z^{14} y_7 \ , \\
Y_3 &= Z^0 y_0 + Z^3 y_1 + Z^6 y_2 + Z^9 y_3 + Z^{12} y_4 + Z^{15} y_5 + Z^{18} y_6 + Z^{21} y_7 \ , \\
Y_4 &= Z^0 y_0 + Z^4 y_1 + Z^8 y_2 + Z^{12} y_3 + Z^{16} y_4 + Z^{20} y_5 + Z^{24} y_6 + Z^{28} y_7 \ , \\
Y_5 &= Z^0 y_0 + Z^5 y_1 + Z^{10} y_2 + Z^{15} y_3 + Z^{20} y_4 + Z^{25} y_5 + Z^{30} y_6 + Z^{35} y_7 \ , \\
Y_6 &= Z^0 y_0 + Z^6 y_1 + Z^{12} y_2 + Z^{18} y_3 + Z^{24} y_4 + Z^{30} y_5 + Z^{36} y_6 + Z^{42} y_7 \ , \\
Y_7 &= Z^0 y_0 + Z^7 y_1 + Z^{14} y_2 + Z^{21} y_3 + Z^{28} y_4 + Z^{35} y_5 + Z^{42} y_6 + Z^{49} y_7 \ ,
\end{aligned}$$

7) Actually, this algorithm has been discovered a number of times, for instance, in 1942 by Danielson and Lanczos (Danielson and Lanczos , 1942), as well as much earlier by Gauss.

where we include $Z^0 (\equiv 1)$ for clarity. When we actually evaluate these powers of Z, we find only four independent values:

$$
\begin{aligned}
&Z^0 = \exp(0) = +1\,, && Z^1 = \exp(-\frac{2\pi}{8}) = +\frac{\sqrt{2}}{2} - \mathrm{i}\frac{\sqrt{2}}{2}\,,\\
&Z^2 = \exp\left(-\frac{2\cdot 2\mathrm{i}\pi}{8}\right) = -\mathrm{i}\,, && Z^3 = \exp\left(-\frac{2\pi\cdot 3\mathrm{i}}{8}\right) = -\frac{\sqrt{2}}{2} - \mathrm{i}\frac{\sqrt{2}}{2}\,,\\
&Z^4 = \exp\left(-\frac{2\pi\cdot 4\mathrm{i}}{8}\right) = -Z^0\,, && Z^5 = \exp\left(-\frac{2\pi\cdot 5\mathrm{i}}{8}\right) = -Z^1\,,\\
&Z^6 = \exp\left(-\frac{2\cdot 6\mathrm{i}\pi}{8}\right) = -Z^2\,, && Z^7 = \exp\left(-\frac{2\cdot 7\mathrm{i}\pi}{8}\right) = -Z^3\,,\\
&Z^8 = \exp\left(-\frac{2\pi\cdot 8\mathrm{i}}{8}\right) = +Z^0\,, && Z^9 = \exp\left(-\frac{2\pi\cdot 9\mathrm{i}}{8}\right) = +Z^1\,,\\
&Z^{10} = \exp\left(-\frac{2\pi\cdot 10\mathrm{i}}{8}\right) = +Z^2\,, && Z^{11} = \exp\left(-\frac{2\pi\cdot 11\mathrm{i}}{8}\right) = +Z^3\,,\\
&Z^{12} = \exp\left(-\frac{2\pi\cdot 11\mathrm{i}}{8}\right) = -Z^0\,, && \cdots
\end{aligned}
\tag{12.82}
$$

When substituted into the definitions of the transforms, we obtain

$$Y_0 = Z^0 y_0 + Z^0 y_1 + Z^0 y_2 + Z^0 y_3 + Z^0 y_4 + Z^0 y_5 + Z^0 y_6 + Z^0 y_7\,, \tag{12.83}$$

$$Y_1 = Z^0 y_0 + Z^1 y_1 + Z^2 y_2 + Z^3 y_3 - Z^0 y_4 - Z^1 y_5 - Z^2 y_6 - Z^3 y_7\,, \tag{12.84}$$

$$Y_2 = Z^0 y_0 + Z^2 y_1 - Z^0 y_2 - Z^2 y_3 + Z^0 y_4 + Z^2 y_5 - Z^0 y_6 - Z^2 y_7\,, \tag{12.85}$$

$$Y_3 = Z^0 y_0 + Z^3 y_1 - Z^2 y_2 + Z^1 y_3 - Z^0 y_4 - Z^3 y_5 + Z^2 y_6 - Z^1 y_7\,, \tag{12.86}$$

$$Y_4 = Z^0 y_0 - Z^0 y_1 + Z^0 y_2 - Z^0 y_3 + Z^0 y_4 - Z^0 y_5 + Z^0 y_6 - Z^0 y_7\,, \tag{12.87}$$

$$Y_5 = Z^0 y_0 - Z^1 y_1 + Z^2 y_2 - Z^3 y_3 - Z^0 y_4 + Z^1 y_5 - Z^2 y_6 + Z^3 y_7\,, \tag{12.88}$$

$$Y_6 = Z^0 y_0 - Z^2 y_1 - Z^0 y_2 + Z^2 y_3 + Z^0 y_4 - Z^2 y_5 - Z^0 y_6 + Z^2 y_7\,, \tag{12.89}$$

$$Y_7 = Z^0 y_0 - Z^3 y_1 - Z^2 y_2 - Z^1 y_3 - Z^0 y_4 + Z^3 y_5 + Z^2 y_6 + Z^1 y_7\,, \tag{12.90}$$

$$Y_8 = Y_0\,. \tag{12.91}$$

We see that these transforms now require $8 \times 8 = 64$ multiplications of complex numbers, in addition to some less time-consuming additions. We place these equations in an appropriate form for computing by regrouping the terms into sums and differences of the y's:

$$Y_0 = Z^0(y_0 + y_4) + Z^0(y_1 + y_5) + Z^0(y_2 + y_6) + Z^0(y_3 + y_7)\,, \tag{12.92}$$

$$Y_1 = Z^0(y_0 - y_4) + Z^1(y_1 - y_5) + Z^2(y_2 - y_6) + Z^3(y_3 - y_7)\,, \tag{12.93}$$

$$Y_2 = Z^0(y_0 + y_4) + Z^2(y_1 + y_5) - Z^0(y_2 + y_6) - Z^2(y_3 + y_7)\,, \tag{12.94}$$

$$Y_3 = Z^0(y_0 - y_4) + Z^3(y_1 - y_5) - Z^2(y_2 - y_6) + Z^1(y_3 - y_7)\,, \tag{12.95}$$

$$Y_4 = Z^0(y_0 + y_4) - Z^0(y_1 + y_5) + Z^0(y_2 + y_6) - Z^0(y_3 + y_7)\,, \tag{12.96}$$

$$Y_5 = Z^0(y_0 - y_4) - Z^1(y_1 - y_5) + Z^2(y_2 - y_6) - Z^3(y_3 - y_7)\,, \tag{12.97}$$

$$Y_6 = Z^0(y_0 + y_4) - Z^2(y_1 + y_5) - Z^0(y_2 + y_6) + Z^2(y_3 + y_7)\,, \tag{12.98}$$

$$Y_7 = Z^0(y_0 - y_4) - Z^3(y_1 - y_5) - Z^2(y_2 - y_6) - Z^1(y_3 - y_7)\,, \tag{12.99}$$

$$Y_8 = Y_0\,. \tag{12.100}$$

Note the repeating factors inside the parentheses, with combinations of the form $y_p \pm y_q$. These symmetries are systematized by introducing the *butterfly operation* (Figure 12.8). This operation takes the y_p and y_q data elements from the left wing and converts them to the $y_p + Zy_q$ elements in the right wings. In Figure 12.9 we show what happens when we apply the butterfly operations to an entire FFT process, specifically to the pairs (y_0, y_4), (y_1, y_5), (y_2, y_6), and (y_3, y_7). Note how the number of multiplications of complex numbers has been reduced: For the first butterfly operation there are 8 multiplications by Z^0; for the second butterfly operation there are 8 multiplications, and so forth, until a total of 24 multiplications are made in four butterflies. In contrast, 64 multiplications are required in the original DFT (12.91).

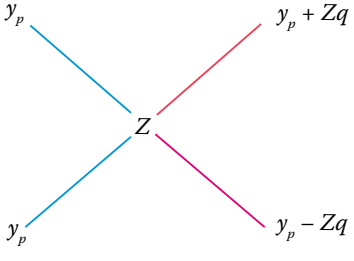

Figure 12.8 The basic butterfly operation in which elements y_p and y_q on the left are transformed into $y_p + Zy_q$ and $y_p - Zy_q$ on the right.

12.9.1 Bit Reversal

The reader may have observed in Figure 12.9 that we started with eight data elements in the order 0–7 and that after three butterfly operators we obtained transforms in the order 0, 4, 2, 6, 1, 5, 3, 7. The astute reader may further have observed that these numbers correspond to the bit-reversed order of 0–7. Let us look into this further. We need 3 bits to give the order of each of the 8 input data elements (the numbers 0–7). Explicitly, on the left in Table 10.1 we give the binary representation for decimal numbers 0–7, their bit reversals, and the corresponding

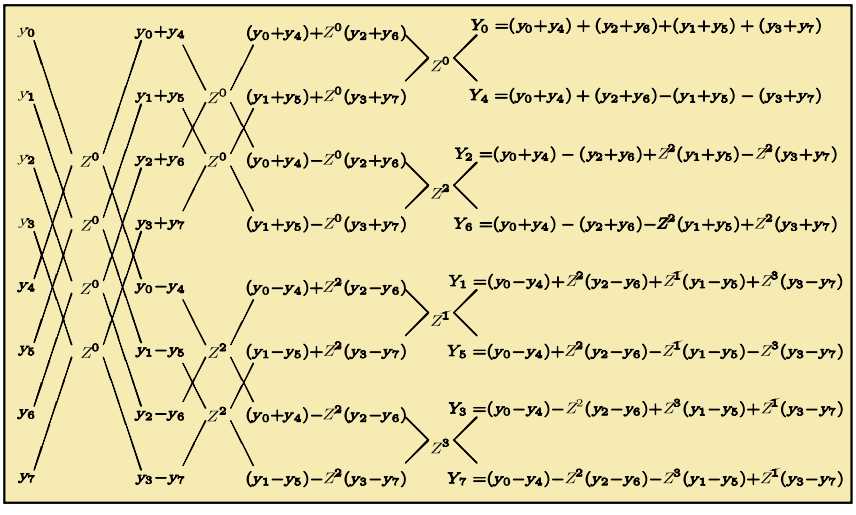

Figure 12.9 The butterfly operations performing a FFT on the eight data on the left leading to eight transforms on the right. The transforms are different linear combinations of the input data.

decimal numbers. On the right we give the ordering for 16 input data elements, where we need 4 bits to enumerate their order. Notice that the order of the first 8 elements differs in the two cases because the number of bits being reversed differs. Also note that after the reordering, the first half of the numbers are all even and the second half are all odd.

The fact that the Fourier transforms are produced in an order corresponding to the bit-reversed order of the numbers 0–7 suggests that if we process the data in the bit-reversed order 0, 4, 2, 6, 1, 5, 3, 7, then the output Fourier transforms will be ordered (see Table 10.1). We demonstrate this conjecture in Figure 12.10, where we see that to obtain the Fourier transform for the eight input data, the butterfly operation had to be applied three times. The number 3 occurs here because it is the power of 2 that gives the number of data; that is, $2^3 = 8$. In general, in order for a FFT algorithm to produce transforms in the proper order, it must reshuffle the input data into bit-reversed order. As a case in point, our sample program starts by reordering the 16 (2^4) data elements given in Table 12.1. Now the four butterfly operations produce sequentially ordered output.

Table 12.1 Reordering for 16 data complex points.

Order	Input data	New order	Order	Input data	New order
0	0.0 + 0.0i	0.0 + 0.0i	8	8.0 + 8.0i	1.0 + 1.0i
1	1.0 + 1.0i	8.0 + 8.0i	9	9.0 + 9.0i	9.0 + 9.0i
2	2.0 + 2.0i	4.0 + 4.0i	10	10.0 + 10.i	5.0 + 5.0i
3	3.0 + 3.0i	12.0 + 12.0i	11	11.0 + 11.0i	13.0 + 13.0i
4	4.0 + 4.0i	2.0 + 2.0i	12	12.0 + 12.0i	3.0 + 3.0i
5	5.0 + 5.0i	10.0 + 10.i	13	13.0 + 13.0i	11.0 + 11.0i
6	6.0 + 6.0i	6.0 + 6.0i	14	14.0 + 14.i	7.0 + 7.0i
7	7.0 + 7.0i	14.0 + 14.0i	15	15.0 + 15.0i	15.0 + 15.0i

		Binary-Reversed 0–7		Binary-Reversed 0–16	
Dec	Bin	Rev	Dec Rev	Rev	Dec Rev
0	000	000	0	0000	0
1	001	100	4	1000	8
2	010	010	2	0100	4
3	011	110	6	1100	12
4	100	001	1	0010	2
5	101	101	5	1010	10
6	110	011	3	0110	6
7	111	111	7	1110	14
8	1000	—	—	0001	1
9	1001	—	—	1001	9
10	1010	—	—	0101	5
11	1011	—	—	1101	13
12	1100	—	—	0011	3
13	1101	—	—	1011	11
14	1101	—	—	0111	7
15	1111	—	—	1111	15

12.10 FFT Implementation

The first FFT program we are aware of was written in Fortran IV by Norman Brenner at MIT's Lincoln Laboratory (Higgins, 1976) and was hard for us to follow. Our (easier-to-follow) Python version is in Listing 12.3. Its input is $N = 2^n$ data to be transformed (FFTs always require that the number of input data are a power of 2). If the number of your input data is not a power of 2, then you can make it so by concatenating some of the initial data to the end of your input until a power of

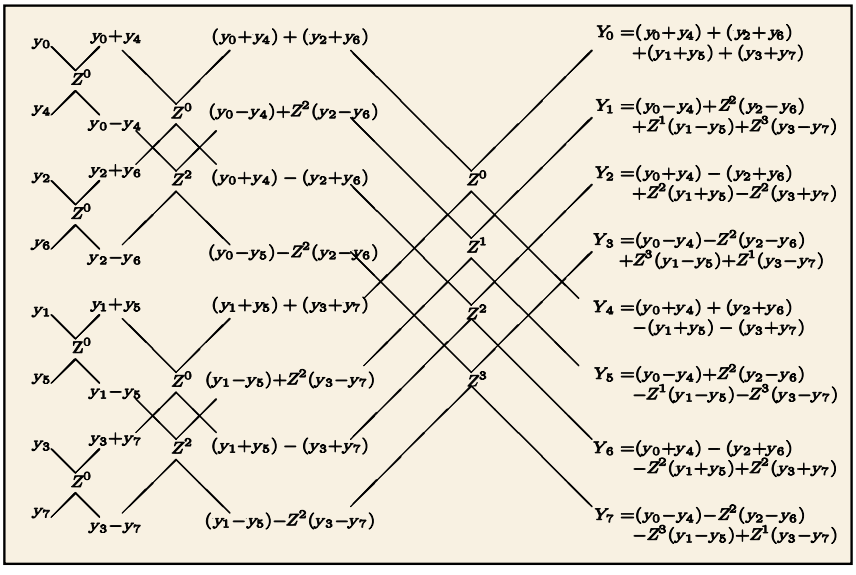

Figure 12.10 A modified FFT in which the eight input data on the left are transformed into eight transforms on the right. The results are the same as in the previous figure, but now the output transforms are in numerical order whereas in the previous figure the input signals were in numerical order.

2 is obtained; because a DFT is always periodic, this just starts the period a little earlier. Our program assigns complex numbers at the 16 data points

$$y_m = m + m\mathrm{i}\ , \quad m = 0, \ldots, 15\ , \tag{12.101}$$

reorders the data via bit reversal, and then makes four butterfly operations. The data are stored in the array `dtr[max,2]`, with the second subscript denoting real and imaginary parts. We increase the speed further by using the 1D array `data` to make memory access more direct:

$$\texttt{data}[1] = \texttt{dtr}[0,1], \quad \texttt{data}[2] = \texttt{dtr}[1,1], \quad \texttt{data}[3] = \texttt{dtr}[1,0], \ldots\ , \tag{12.102}$$

which also provides storage for the output. The FFT transforms `data` using the butterfly operation and stores the results back in `dtr[,]`, where the input data were originally.

12.11 FFT Assessment

1. Compile and execute `FFT.py`. Make sure you understand the output.
2. Take the output from `FFT.py`, inverse-transform it back to signal space, and compare it to your input. (Checking that the double transform is proportional

to itself is adequate, although the normalization factors in (12.37) should make the two equal.)

3. Compare the transforms obtained with a FFT to those obtained with a DFT (you may choose any of the functions studied before). Make sure to compare both precision and execution times.

Listing 12.3 FFT.py computes the FFT or inverse transform depending upon the sign of isign.

```
# FFT.py: FFT for complex numbers in dtr[][2], returned in dtr

from numpy import *
from sys import version
max = 2100
points = 1026                                          # Can be increased
data = zeros((max), float)
dtr  = zeros((points,2), float)

def fft(nn,isign):                                      # FFT of dtr[n,2]
    n = 2*nn
    for i in range(0,nn+1):                  # Original data in dtr to data
         j = 2*i+1
         data[j] = dtr[i,0]                          # Real dtr, odd data[j]
         data[j+1] = dtr[i,1]                      # Imag dtr, even data[j+1]
    j = 1                                    # Place data in bit reverse order
    for i in range(1,n+2, 2):
        if (i-j) < 0 :                      # Reorder equivalent to bit reverse
            tempr = data[j]
            tempi = data[j+1]
            data[j] = data[i]
            data[j+1] = data[i+1]
            data[i] = tempr
            data[i+1] = tempi
        m = n/2;
        while (m-2 > 0):
            if  (j-m) <= 0 :
                break
            j = j-m
            m = m/2
        j = j+m;

    print(" Bit-reversed data ")

    for i in range(1, n+1, 2):
        print("%2d  data[%2d]  %9.5f "%(i,i,data[i]))     # To see reorder
    mmax = 2
    while (mmax-n) < 0 :                                  # Begin transform
       istep = 2*mmax
       theta = 6.2831853/(1.0*isign*mmax)
       sinth = math.sin(theta/2.0)
       wstpr = -2.0*sinth**2
       wstpi = math.sin(theta)
       wr = 1.0
       wi = 0.0
       for m in range(1,mmax +1,2):
           for i in range(m,n+1,istep):
               j = i+mmax
               tempr = wr*data[j]   -wi *data[j+1]
               tempi = wr*data[j+1] +wi *data[j]
               data[j]   = data[i]   -tempr
               data[j+1] = data[i+1] -tempi
               data[i]   = data[i]   +tempr
               data[i+1] = data[i+1] +tempi
           tempr = wr
```

```
            wr = wr*wstpr - wi*wstpi + wr
            wi = wi*wstpr + tempr*wstpi + wi;
        mmax = istep
    for i in range(0,nn):
        j = 2*i+1
        dtr[i,0] = data[j]
        dtr[i,1] = data[j+1]
nn = 16                                                    # Power of 2
isign = -1                              # -1 transform, +1 inverse transform
print('        INPUT')
print("  i   Re part   Im  part")
for i in range(0,nn ):                                     # Form array
    dtr[i,0] = 1.0*i                                        # Real part
    dtr[i,1] = 1.0*i                                          # Im part
    print(" %2d %9.5f %9.5f" %(i,dtr[i,0],dtr[i,1]))
fft(nn, isign)                                  # Call FFT, use global dtr[][]
print('    Fourier transform')
print("  i      Re      Im    ")
for i in range(0,nn):
    print(" %2d  %9.5f  %9.5f "%(i,dtr[i,0],dtr[i,1]))
print("Enter and return any character to quit")
```