2 Fourier and Wavelet Transforms

A central concern of mathematical physics and engineering mathematics involves the transformation of equations into a coordinate system where expressions simplify, decouple, and are amenable to computation and analysis. This is a common theme throughout this book, in a wide variety of domains, including data analysis (e.g., the SVD), dynamical systems (e.g., spectral decomposition into eigenvalues and eigenvectors), and control (e.g., defining coordinate systems by controllability and observability). Perhaps the most foundational and ubiquitous coordinate transformation was introduced by J.-B. Joseph Fourier in the early 1800s to investigate the theory of heat [185]. Fourier introduced the concept that sine and cosine functions of increasing frequency provide an orthogonal *basis* for the space of solution functions. Indeed, the Fourier transform basis of sines and cosines serve as eigenfunctions of the heat equation, with the specific frequencies serving as the eigenvalues, determined by the geometry, and amplitudes determined by the boundary conditions.

Fourier's seminal work provided the mathematical foundation for Hilbert spaces, operator theory, approximation theory, and the subsequent revolution in analytical and computational mathematics. Fast forward two hundred years, and the fast Fourier transform has become the cornerstone of computational mathematics, enabling real-time image and audio compression, global communication networks, modern devices and hardware, numerical physics and engineering at scale, and advanced data analysis. Simply put, the fast Fourier transform has had a more significant and profound role in shaping the modern world than any other algorithm to date.

With increasingly complex problems, data sets, and computational geometries, simple Fourier sine and cosine bases have given way to *tailored* bases, such as the data-driven SVD. In fact, the SVD basis can be used as a direct analogue of the Fourier basis for solving PDEs with complex geometries, as will be discussed later. In addition, related functions, called wavelets, have been developed for advanced signal processing and compression efforts. In this chapter, we will demonstrate a few of the many uses of Fourier and wavelet transforms.

2.1 Fourier Series and Fourier Transforms

Before describing the computational implementation of Fourier transforms on vectors of data, here we introduce the analytic Fourier series and Fourier transform, defined for continuous functions. Naturally, the discrete and continuous formulations should match in the limit of data with infinitely fine resolution. The Fourier series and transform are intimately related to the geometry of infinite-dimensional function spaces, or *Hilbert* spaces, which generalize the notion of vector spaces to include functions with infinitely many degrees of freedom. Thus, we begin with an introduction to function spaces.

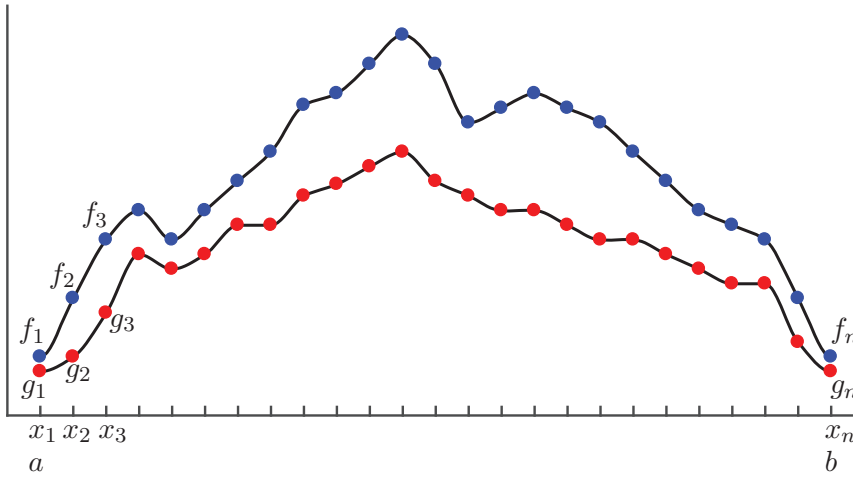

Figure 2.1 Discretized functions used to illustrate the inner product.

Inner Products of Functions and Vectors

In this section, we will make use of inner products and norms of functions. In particular, we will use the common Hermitian inner product for functions $f(x)$ and $g(x)$ defined for x on a domain $x \in [a, b]$:

$$\langle f(x), g(x) \rangle = \int_a^b f(x)\bar{g}(x)\, dx \tag{2.1}$$

where $\bar{g}$ denotes the complex conjugate.

The inner product of functions may seem strange or unmotivated at first, but this definition becomes clear when we consider the inner product of vectors of data. In particular, if we discretize the functions $f(x)$ and $g(x)$ into vectors of data, as in Fig. 2.1, we would like the vector inner product to converge to the function inner product as the sampling resolution is increased. The inner product of the data vectors $\mathbf{f} = \begin{bmatrix} f_1 & f_2 & \cdots & f_n \end{bmatrix}^T$ and $\mathbf{g} = \begin{bmatrix} g_1 & g_2 & \cdots & g_n \end{bmatrix}^T$ is defined by:

$$\langle \mathbf{f}, \mathbf{g} \rangle = \mathbf{g}^* \mathbf{f} = \sum_{k=1}^{n} f_k \bar{g}_k = \sum_{k=1}^{n} f(x_k)\bar{g}(x_k). \tag{2.2}$$

The magnitude of this inner product will grow as more data points are added; i.e., as n increases. Thus, we may normalize by $\Delta x = (b-a)/(n-1)$:

$$\frac{b-a}{n-1} \langle \mathbf{f}, \mathbf{g} \rangle = \sum_{k=1}^{n} f(x_k)\bar{g}(x_k)\Delta x, \tag{2.3}$$

which is the Riemann approximation to the continuous function inner product. It is now clear that as we take the limit of $n \to \infty$ (i.e., infinite data resolution, with $\Delta x \to 0$), the vector inner product converges to the inner product of functions in (2.1).

This inner product also induces a norm on functions, given by

$$\|f\|_2 = (\langle f, f\rangle)^{1/2} = \sqrt{\langle f, f\rangle} = \left(\int_a^b f(x)\bar{f}(x)\,dx\right)^{1/2}. \tag{2.4}$$

The set of all functions with bounded norm define the set of square integrable functions, denoted by $L^2([a, b])$; this is also known as the set of Lebesgue integrable functions. The interval $[a, b]$ may also be chosen to be infinite (e.g., $(-\infty, \infty)$), semi-infinite (e.g., $[a, \infty)$), or periodic (e.g., $[-\pi, \pi)$). A fun example of a function in $L^2([1, \infty))$ is $f(x) = 1/x$. The square of f has finite integral from 1 to ∞, although the integral of the function itself diverges. The shape obtained by rotating this function about the x-axis is known as Gabriel's horn, as the volume is finite (related to the integral of f^2), while the surface area is infinite (related to the integral of f).

As in finite-dimensional vector spaces, the inner product may be used to *project* a function into an new coordinate system defined by a basis of orthogonal functions. A Fourier series representation of a function f is precisely a projection of this function onto the orthogonal set of sine and cosine functions with integer period on the domain $[a, b]$. This is the subject of the following sections.

Fourier Series

A fundamental result in Fourier analysis is that if $f(x)$ is periodic and piecewise smooth, then it can be written in terms of a Fourier series, which is an infinite sum of cosines and sines of increasing frequency. In particular, if $f(x)$ is 2π-periodic, it may be written as:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos(kx) + b_k \sin(kx)\right). \tag{2.5}$$

The coefficients a_k and b_k are given by

$$a_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(kx)dx \tag{2.6a}$$

$$b_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(kx)dx, \tag{2.6b}$$

which may be viewed as the coordinates obtained by projecting the function onto the orthogonal cosine and sine basis $\{\cos(kx), \sin(kx)\}_{k=0}^{\infty}$. In other words, the integrals in (2.6) may be re-written in terms of the inner product as:

$$a_k = \frac{1}{\|\cos(kx)\|^2}\langle f(x), \cos(kx)\rangle \tag{2.7a}$$

$$b_k = \frac{1}{\|\sin(kx)\|^2}\langle f(x), \sin(kx)\rangle, \tag{2.7b}$$

where $\|\cos(kx)\|^2 = \|\sin(kx)\|^2 = \pi$. This factor of $1/\pi$ is easy to verify by numerically integrating $\cos(x)^2$ and $\sin(x)^2$ from $-\pi$ to π.

The Fourier series for an L-periodic function on $[0, L)$ is similarly given by:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty}\left(a_k \cos\left(\frac{2\pi kx}{L}\right) + b_k \sin\left(\frac{2\pi kx}{L}\right)\right), \tag{2.8}$$

with coefficients a_k and b_k given by

$$a_k = \frac{2}{L}\int_0^L f(x)\cos\left(\frac{2\pi kx}{L}\right)dx \tag{2.9a}$$

$$b_k = \frac{2}{L}\int_0^L f(x)\sin\left(\frac{2\pi kx}{L}\right)dx. \tag{2.9b}$$

Because we are expanding functions in terms of sine and cosine functions, it is also natural to use Euler's formula $e^{ikx} = \cos(kx) + i\sin(kx)$ to write a Fourier series in complex form with complex coefficients $c_k = \alpha_k + i\beta_k$:

$$\begin{aligned} f(x) = \sum_{k=-\infty}^{\infty} c_k e^{ikx} &= \sum_{k=-\infty}^{\infty} (\alpha_k + i\beta_k)\left(\cos(kx) + i\sin(kx)\right) \\ &= (\alpha_0 + i\beta_0) + \sum_{k=1}^{\infty}\Big[(\alpha_{-k} + \alpha_k)\cos(kx) + (\beta_{-k} - \beta_k)\sin(kx)\Big] \\ &\quad + i\sum_{k=1}^{\infty}\Big[(\beta_{-k} + \beta_k)\cos(kx) - (\alpha_{-k} - \alpha_k)\sin(kx)\Big]. \end{aligned} \tag{2.10}$$

If $f(x)$ is real-valued, then $\alpha_{-k} = \alpha_k$ and $\beta_{-k} = -\beta_k$, so that $c_{-k} = \bar{c}_k$.

Thus, the functions $\psi_k = e^{ikx}$ for $k \in \mathbb{Z}$ (i.e., for integer k) provide a basis for periodic, complex-valued functions on an interval $[0, 2\pi)$. It is simple to see that these functions are orthogonal:

$$\langle \psi_j, \psi_k \rangle = \int_{-\pi}^{\pi} e^{ijx}e^{-ikx}dx = \int_{-\pi}^{\pi} e^{i(j-k)x}dx = \left[\frac{e^{i(j-k)x}}{i(j-k)}\right]_{-\pi}^{\pi} = \begin{cases} 0 & \text{if } j \neq k \\ 2\pi & \text{if } j = k. \end{cases}$$

So $\langle \psi_j, \psi_k \rangle = 2\pi\delta_{jk}$, where δ is the Kronecker delta function. Similarly, the functions $e^{i2\pi kx/L}$ provide a basis for $L^2([0, L))$, the space of square integrable functions defined on $x \in [0, L)$.

In principle, a Fourier series is just a change of coordinates of a function $f(x)$ into an infinite-dimensional orthogonal function space spanned by sines and cosines (i.e., $\psi_k = e^{ikx} = \cos(kx) + i\sin(kx)$):

$$f(x) = \sum_{k=-\infty}^{\infty} c_k\psi_k(x) = \frac{1}{2\pi}\sum_{k=-\infty}^{\infty}\langle f(x), \psi_k(x)\rangle\psi_k(x). \tag{2.11}$$

The coefficients are given by $c_k = \frac{1}{2\pi}\langle f(x), \psi_k(x)\rangle$. The factor of $1/2\pi$ normalizes the projection by the square of the norm of ψ_k; i.e., $\|\psi_k\|^2 = 2\pi$. This is consistent with our standard finite-dimensional notion of change of basis, as in Fig. 2.2. A vector $\vec{f}$ may be written in the $(\vec{x}, \vec{y})$ or $(\vec{u}, \vec{v})$ coordinate systems, via projection onto these orthogonal bases:

$$\vec{f} = \langle \vec{f}, \vec{x}\rangle\frac{\vec{x}}{\|\vec{x}\|^2} + \langle \vec{f}, \vec{y}\rangle\frac{\vec{y}}{\|\vec{y}\|^2} \tag{2.12a}$$

$$= \langle \vec{f}, \vec{u}\rangle\frac{\vec{u}}{\|\vec{u}\|^2} + \langle \vec{f}, \vec{v}\rangle\frac{\vec{v}}{\|\vec{v}\|^2}. \tag{2.12b}$$

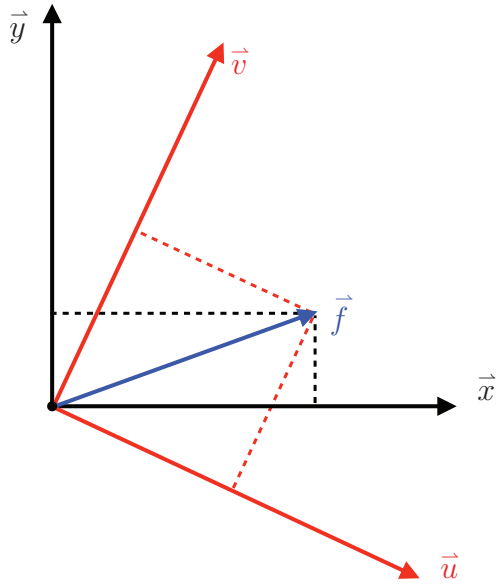

Figure 2.2 Change of coordinates of a vector in two dimensions.

Example: Fourier Series for a Continuous Hat Function

As a simple example, we demonstrate the use of Fourier series to approximate a continuous hat function, defined from $-\pi$ to π:

$$f(x) = \begin{cases} 0 & \text{for } x \in [-\pi, \pi/2) \\ 1 + 2x/\pi & \text{for } x \in [-\pi/2, 0) \\ 1 - 2x/\pi & \text{for } x \in [0, \pi/2) \\ 0 & \text{for } x \in [\pi/2, \pi). \end{cases} \tag{2.13}$$

Because this function is even, it may be approximated with cosines alone. The Fourier series for $f(x)$ is shown in Fig. 2.3 for an increasing number of cosines.

Figure 2.4 shows the coefficients a_k of the even cosine functions, along with the approximation error, for an increasing number of modes. The error decreases monotonically, as expected. The coefficients b_k corresponding to the odd sine functions are not shown, as they are identically zero since the hat function is even.

Code 2.1 Fourier series approximation to a hat function.

```
% Define domain
dx = 0.001;
L = pi;
x = (-1+dx:dx:1)*L;
n = length(x);   nquart = floor(n/4);

% Define hat function
f = 0*x;
f(nquart:2*nquart) = 4*(1:nquart+1)/n;
f(2*nquart+1:3*nquart) = 1-4*(0:nquart-1)/n;
plot(x,f,'-k','LineWidth',1.5), hold on

% Compute Fourier series
```

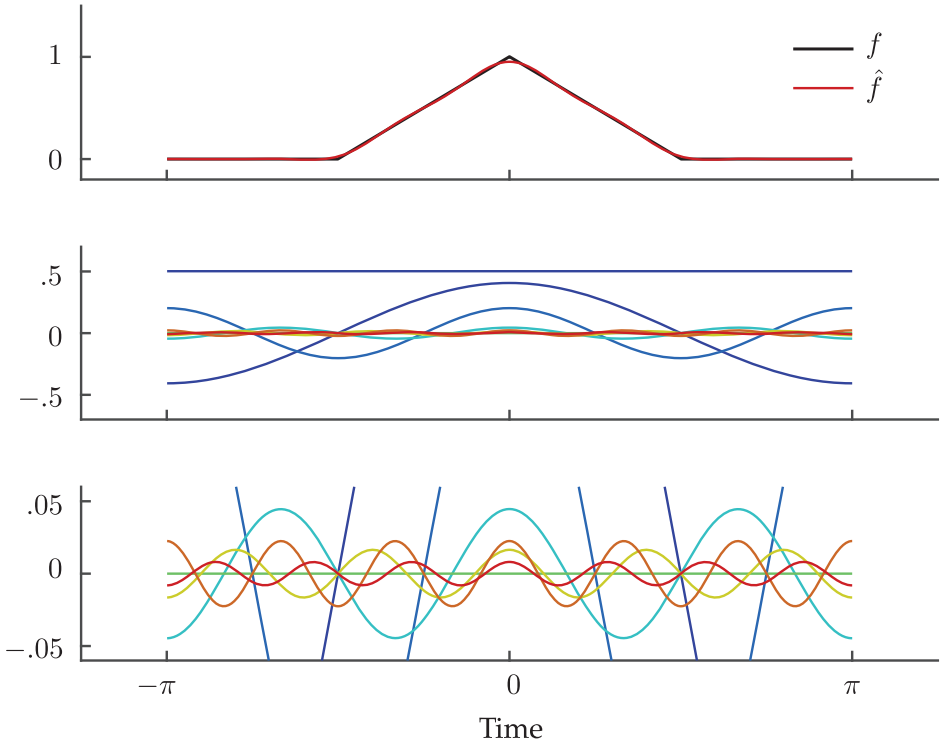

Figure 2.3 (top) Hat function and Fourier cosine series approximation for $n = 7$. (middle) Fourier cosines used to approximate the hat function, and (bottom) zoom in of modes with small amplitude and high frequency.

```
CC = jet(20);
A0 = sum(f.*ones(size(x)))*dx;
fFS = A0/2;
for k=1:20
    A(k) = sum(f.*cos(pi*k*x/L))*dx; % Inner product
    B(k) = sum(f.*sin(pi*k*x/L))*dx;
    fFS = fFS + A(k)*cos(k*pi*x/L) + B(k)*sin(k*pi*x/L);
    plot(x,fFS,'-','Color',CC(k,:),'LineWidth',1.2)
end
```

Example: Fourier Series for a Discontinuous Hat Function

We now consider the discontinuous square hat function, defined on $[0, L)$, shown in Fig. 2.5. The function is given by:

$$f(x) = \begin{cases} 0 & \text{for } x \in [0, L/4) \\ 1 & \text{for } x \in [L/4, 3L/4) \\ 0 & \text{for } x \in [3L/4, L). \end{cases} \tag{2.14}$$

The truncated Fourier series is plagued by ringing oscillations, known as Gibbs phenomena, around the sharp corners of the step function. This example highlights the challenge of applying the Fourier series to discontinuous functions:

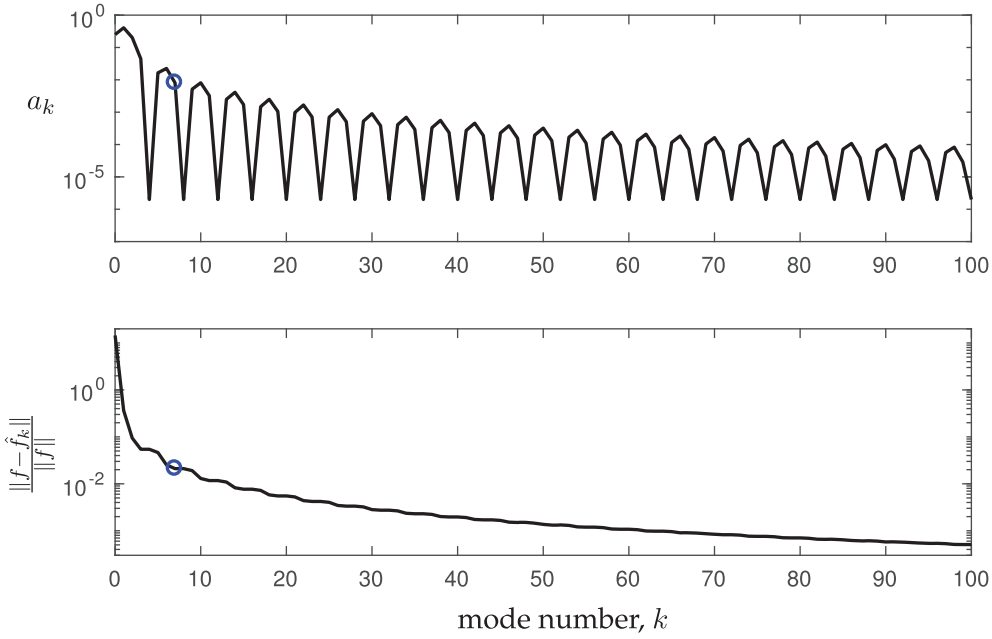

Figure 2.4 Fourier coefficients (top) and relative error of Fourier cosine approximation with true function (bottom) for hat function in Fig. 2.3. The $n = 7$ approximation is highlighted with a blue circle.

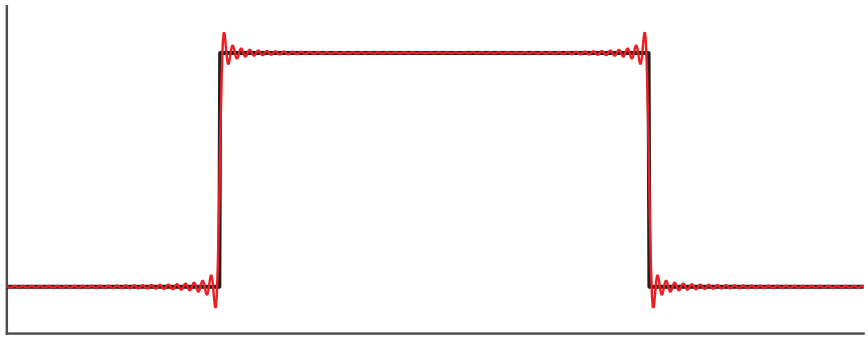

Figure 2.5 Gibbs phenomena is characterized by high-frequency oscillations near discontinuities. The black curve is discontinuous, and the red curve is the Fourier approximation.

```
dx = 0.01;  L = 10;
x = 0:dx:L;
n = length(x); nquart = floor(n/4);

f = zeros(size(x));
f(nquart:3*nquart) = 1;

A0 = sum(f.*ones(size(x)))*dx*2/L;
fFS = A0/2;
for k=1:100
    Ak = sum(f.*cos(2*pi*k*x/L))*dx*2/L;
    Bk = sum(f.*sin(2*pi*k*x/L))*dx*2/L;
```

```
    fFS = fFS + Ak*cos(2*k*pi*x/L) + Bk*sin(2*k*pi*x/L);
end

plot(x,f,'k','LineWidth',2), hold on
plot(x,fFS,'r-','LineWidth',1.2)
```

Fourier Transform

The Fourier series is defined for periodic functions, so that outside the domain of definition, the function repeats itself forever. The Fourier transform integral is essentially the limit of a Fourier series as the length of the domain goes to infinity, which allows us to define a function defined on $(-\infty, \infty)$ without repeating, as shown in Fig. 2.6. We will consider the Fourier series on a domain $x \in [-L, L)$, and then let $L \to \infty$. On this domain, the Fourier series is:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left[a_k \cos\left(\frac{k\pi x}{L}\right) + b_k \sin\left(\frac{k\pi x}{L}\right) \right] = \sum_{k=-\infty}^{\infty} c_k e^{ik\pi x/L} \tag{2.15}$$

with the coefficients given by:

$$c_k = \frac{1}{2L} \langle f(x), \psi_k \rangle = \frac{1}{2L} \int_{-L}^{L} f(x) e^{-ik\pi x/L} \, dx. \tag{2.16}$$

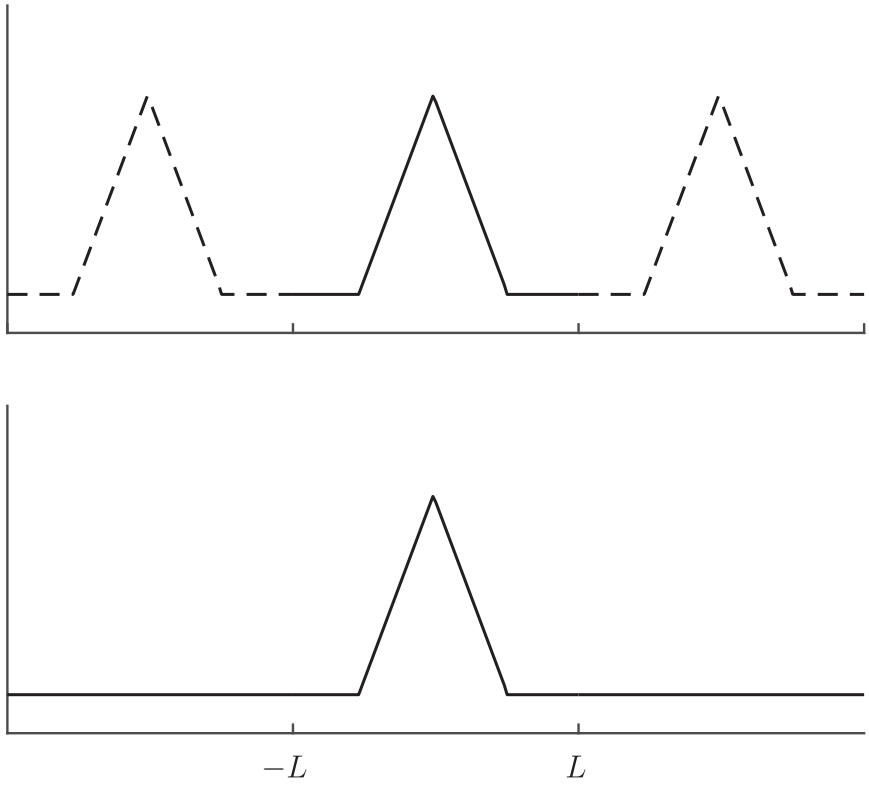

Figure 2.6 (top) Fourier series is only valid for a function that is periodic on the domain $[-L, L)$. (bottom) The Fourier transform is valid for generic nonperiodic functions.

Restating the previous results, $f(x)$ is now represented by a sum of sines and cosines with a discrete set of frequencies given by $\omega_k = k\pi/L$. Taking the limit as $L \to \infty$, these discrete frequencies become a continuous range of frequencies. Define $\omega = k\pi/L$, $\Delta\omega = \pi/L$, and take the limit $L \to \infty$, so that $\Delta\omega \to 0$:

$$f(x) = \lim_{\Delta\omega \to 0} \sum_{k=-\infty}^{\infty} \frac{\Delta\omega}{2\pi} \underbrace{\int_{-\pi/\Delta\omega}^{\pi/\Delta\omega} f(\xi) e^{-ik\Delta\omega\xi}\, d\xi}_{\langle f(x), \psi_k(x)\rangle} \; e^{ik\Delta\omega x}. \tag{2.17}$$

When we take the limit, the expression $\langle f(x), \psi_k(x)\rangle$ will become the Fourier transform of $f(x)$, denoted by $\hat{f}(\omega) \triangleq \mathcal{F}(f(x))$. In addition, the summation with weight $\Delta\omega$ becomes a Riemann integral, resulting in the following:

$$f(x) = \mathcal{F}^{-1}\left(\hat{f}(\omega)\right) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega x}\, d\omega \tag{2.18a}$$

$$\hat{f}(\omega) = \mathcal{F}\left(f(x)\right) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x}\, dx. \tag{2.18b}$$

These two integrals are known as the *Fourier transform pair*. Both integrals converge as long as $\int_{-\infty}^{\infty} |f(x)|\, dx < \infty$ and $\int_{-\infty}^{\infty} |\hat{f}(\omega)|\, d\omega < \infty$; i.e., as long as both functions belong to the space of Lebesgue integrable functions, $f, \hat{f} \in L^1(-\infty, \infty)$.

The Fourier transform is particularly useful because of a number of properties, including linearity, and how derivatives of functions behave in the Fourier transform domain. These properties have been used extensively for data analysis and scientific computing (e.g., to solve PDEs accurately and efficiently), as will be explored throughout this chapter.

Derivatives of Functions The Fourier transform of the derivative of a function is given by:

$$\mathcal{F}\left(\frac{d}{dx} f(x)\right) = \int_{-\infty}^{\infty} \overbrace{f'(x)}^{dv} \overbrace{e^{-i\omega x}}^{u}\, dx \tag{2.19a}$$

$$= \Big[\underbrace{f(x) e^{-i\omega x}}_{uv}\Big]_{-\infty}^{\infty} - \int_{-\infty}^{\infty} \underbrace{f(x)}_{v} \Big[\underbrace{-i\omega e^{-i\omega x}}_{du}\Big] dx \tag{2.19b}$$

$$= i\omega \int_{-\infty}^{\infty} f(x) e^{-i\omega x}\, dx \tag{2.19c}$$

$$= i\omega \mathcal{F}(f(x)). \tag{2.19d}$$

This is an extremely important property of the Fourier transform, as it will allow us to turn PDEs into ODEs, closely related to the separation of variables:

$$\underset{\text{(PDE)}}{u_{tt} = c u_{xx}} \quad \xrightarrow{\mathcal{F}} \quad \underset{\text{(ODE)}}{\hat{u}_{tt} = -c\omega^2 \hat{u}}. \tag{2.20}$$

Linearity of Fourier Transforms The Fourier transform is a linear operator, so that:

$$\mathcal{F}(\alpha f(x) + \beta g(x)) = \alpha \mathcal{F}(f) + \beta \mathcal{F}(g). \tag{2.21}$$

$$\mathcal{F}^{-1}(\alpha \hat{f}(\omega) + \beta \hat{g}(\omega)) = \alpha \mathcal{F}^{-1}(\hat{f}) + \beta \mathcal{F}^{-1}(\hat{g}). \tag{2.22}$$

Parseval's Theorem

$$\int_{-\infty}^{\infty} |\hat{f}(\omega)|^2 \, d\omega = 2\pi \int_{-\infty}^{\infty} |f(x)|^2 \, dx. \tag{2.23}$$

In other words, the Fourier transform preserves the L_2 norm, up to a constant. This is closely related to unitarity, so that two functions will retain the same inner product before and after the Fourier transform. This property is useful for approximation and truncation, providing the ability to bound error at a given truncation.

Convolution The convolution of two functions is particularly well-behaved in the Fourier domain, being the product of the two Fourier transformed functions. Define the convolution of two functions $f(x)$ and $g(x)$ as $f * g$:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x - \xi) g(\xi) \, d\xi. \tag{2.24}$$

If we let $\hat{f} = \mathcal{F}(f)$ and $\hat{g} = \mathcal{F}(g)$, then:

$$\mathcal{F}^{-1}\left(\hat{f}\hat{g}\right)(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega)\hat{g}(\omega) e^{i\omega x} \, d\omega \tag{2.25a}$$

$$= \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega x} \left(\frac{1}{2\pi} \int_{-\infty}^{\infty} g(y) e^{-i\omega y} \, dy \right) d\omega \tag{2.25b}$$

$$= \frac{1}{2\pi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(y) \hat{f}(\omega) e^{i\omega(x-y)} \, d\omega \, dy \tag{2.25c}$$

$$= \int_{-\infty}^{\infty} g(y) \Big(\underbrace{\frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega(x-y)} \, d\omega}_{f(x-y)} \Big) dy \tag{2.25d}$$

$$= \int_{-\infty}^{\infty} g(y) f(x - y) \, dy = g * f = f * g. \tag{2.25e}$$

Thus, multiplying functions in the frequency domain is the same as convolving functions in the spatial domain. This will be particularly useful for control systems and transfer functions with the related Laplace transform.

2.2 Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT)

Until now, we have considered the Fourier series and Fourier transform for continuous functions $f(x)$. However, when computing or working with real-data, it is necessary to approximate the Fourier transform on discrete vectors of data. The resulting discrete Fourier transform (DFT) is essentially a discretized version of the Fourier series for vectors of data $\mathbf{f} = \begin{bmatrix} f_1 & f_2 & f_3 & \cdots & f_n \end{bmatrix}^T$ obtained by discretizing the function $f(x)$ at a regular spacing, Δx, as shown in Fig. 2.7.

The DFT is tremendously useful for numerical approximation and computation, but it does not scale well to very large $n \gg 1$, as the simple formulation involves multiplication by a dense $n \times n$ matrix, requiring $\mathcal{O}(n^2)$ operations. In 1965, James W. Cooley (IBM) and John W. Tukey (Princeton) developed the revolutionary *fast* Fourier transform (FFT)

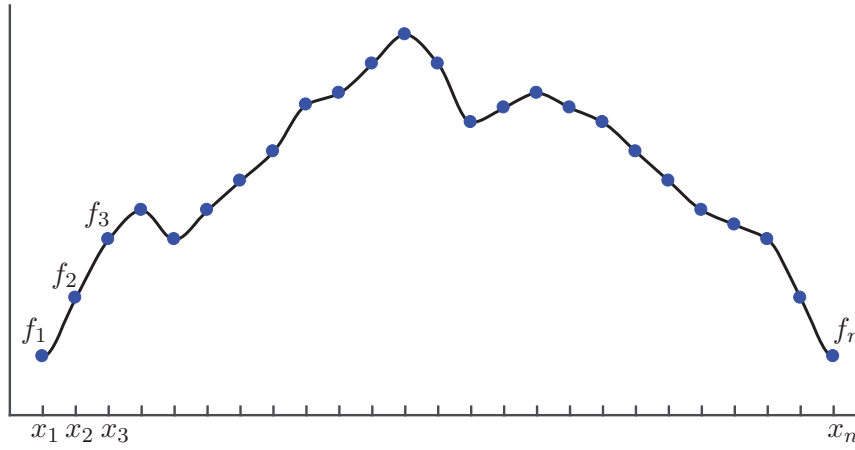

Figure 2.7 Discrete data sampled for the discrete Fourier transform.

algorithm [137, 136] that scales as $\mathcal{O}(n\log(n))$. As n becomes very large, the $\log(n)$ component grows slowly, and the algorithm approaches a linear scaling. Their algorithm was based on a fractal symmetry in the Fourier transform that allows an n dimensional DFT to be solved with a number of smaller dimensional DFT computations. Although the different computational scaling between the DFT and FFT implementations may seem like a small difference, the fast $\mathcal{O}(n\log(n))$ scaling is what enables the ubiquitous use of the FFT in real-time communication, based on audio and image compression [539].

It is important to note that Cooley and Tukey did not invent the idea of the FFT, as there were decades of prior work developing special cases, although they provided the general formulation that is currently used. Amazingly, the FFT algorithm was formulated by Gauss over 150 years earlier in 1805 to approximate the orbits of the asteroids Pallas and Juno from measurement data, as he required a highly accurate interpolation scheme [239]. As the computations were performed by Gauss in his head and on paper, he required a fast algorithm, and developed the FFT. However, Gauss did not view this as a major breakthrough and his formulation only appeared later in 1866 in his compiled notes [198]. It is interesting to note that Gauss's discovery even predates Fourier's announcement of the Fourier series expansion in 1807, which was later published in 1822 [186].

Discrete Fourier Transform

Although we will always use the FFT for computations, it is illustrative to begin with the simplest formulation of the DFT. The discrete Fourier transform is given by:

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{-i2\pi jk/n}, \tag{2.26}$$

and the inverse discrete Fourier transform (iDFT) is given by:

$$f_k = \frac{1}{n}\sum_{j=0}^{n-1} \hat{f}_j e^{i2\pi jk/n}. \tag{2.27}$$

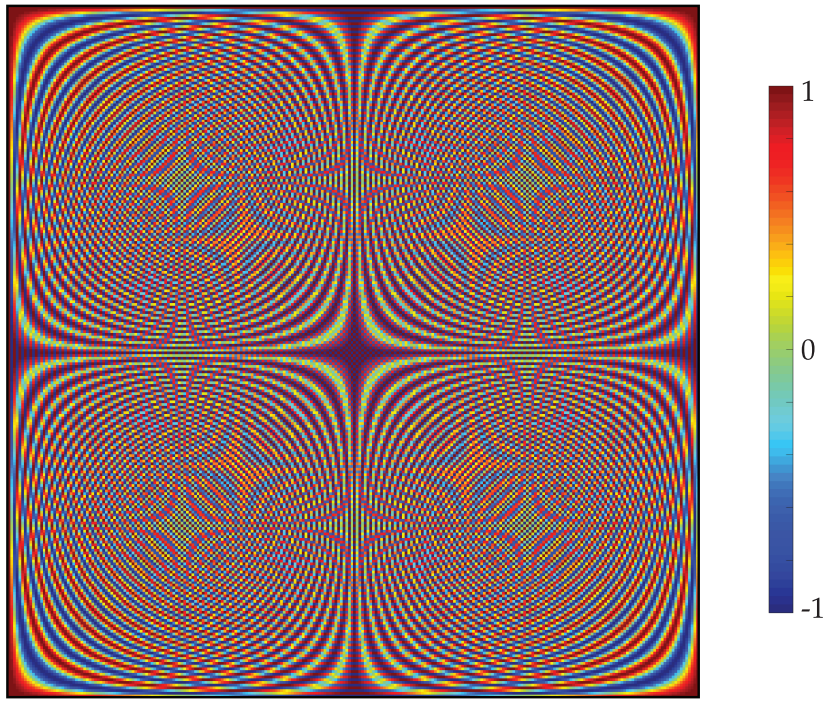

Figure 2.8 Real part of DFT matrix for $n = 256$.

Thus, the DFT is a linear operator (i.e., a *matrix*) that maps the data points in $\mathbf{f}$ to the frequency domain $\hat{\mathbf{f}}$:

$$\{f_1, f_2, \cdots, f_n\} \quad \stackrel{\text{DFT}}{\Longrightarrow} \quad \{\hat{f}_1, \hat{f}_2, \cdots \hat{f}_n\}. \tag{2.28}$$

For a given number of points n, the DFT represents the data using sine and cosine functions with integer multiples of a fundamental frequency, $\omega_n = e^{-2\pi i/n}$. The DFT may be computed by matrix multiplication:

$$\begin{bmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{bmatrix}. \tag{2.29}$$

The output vector $\hat{\mathbf{f}}$ contains the Fourier coefficients for the input vector $\mathbf{f}$, and the DFT matrix $\mathbf{F}$ is a unitary Vandermonde matrix. The matrix $\mathbf{F}$ is complex-valued, so the output $\hat{\mathbf{f}}$ has both a magnitude and a phase, which will both have useful physical interpretations.

The real part of the DFT matrix $\mathbf{F}$ is shown in Fig. 2.8 for $n = 256$. Code 2.2 generates and plots this matrix. It can be seen from this image that there is a hierarchical and highly symmetric multiscale structure to $\mathbf{F}$. Each row and column is a cosine function with increasing frequency.

Code 2.2 Generate discrete Fourier transform matrix.

```
clear all, close all, clc
n = 256;
w = exp(-i*2*pi/n);

% Slow
for i=1:n
    for j=1:n
        DFT(i,j) = w^((i-1)*(j-1));
    end
end

% Fast
[I,J] = meshgrid(1:n,1:n);
DFT = w.^((I-1).*(J-1));
imagesc(real(DFT))
```

Fast Fourier Transform

As mentioned earlier, multiplying by the DFT matrix $\mathbf{F}$ involves $\mathcal{O}(n^2)$ operations. The fast Fourier transform scales as $\mathcal{O}(n\log(n))$, enabling a tremendous range of applications, including audio and image compression in MP3 and JPG formats, streaming video, satellite communications, and the cellular network, to name only a few of the myriad applications. For example, audio is generally sampled at 44.1 kHz, or 44, 100 samples per second. For 10 seconds of audio, the vector $\mathbf{f}$ will have dimension $n = 4.41 \times 10^5$. Computing the DFT using matrix multiplication involves approximately 2×10^{11}, or 200 billion, multiplications. In contrast, the FFT requires approximately 6×10^6, which amounts to a speed-up factor of over 30, 000. Thus, the FFT has become synonymous with the DFT, and FFT libraries are built in to nearly every device and operating system that performs digital signal processing.

To see the tremendous benefit of the FFT, consider the transmission, storage, and decoding of an audio signal. We will see later that many signals are highly compressible in the Fourier transform domain, meaning that most of the coefficients of $\hat{\mathbf{f}}$ are small and can be discarded. This enables much more efficient storage and transmission of the compressed signal, as only the non-zero Fourier coefficients must be transmitted. However, it is then necessary to rapidly encode and decode the compressed Fourier signal by computing the FFT and inverse FFT (iFFT). This is accomplished with the one-line commands:

```
>>fhat = fft(f);    % Fast Fourier transform
>>f = ifft(fhat); % Inverse fast Fourier transform
```

The basic idea behind the FFT is that the DFT may be implemented much more efficiently if the number of data points n is a power of 2. For example, consider $n = 1024 = 2^{10}$. In this case, the DFT matrix $\mathbf{F}_{1024}$ may be written as:

$$\hat{\mathbf{f}} = \mathbf{F}_{1024}\mathbf{f} = \begin{bmatrix} \mathbf{I}_{512} & -\mathbf{D}_{512} \\ \mathbf{I}_{512} & -\mathbf{D}_{512} \end{bmatrix} \begin{bmatrix} \mathbf{F}_{512} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_{512} \end{bmatrix} \begin{bmatrix} \mathbf{f}_{\text{even}} \\ \mathbf{f}_{\text{odd}} \end{bmatrix}, \tag{2.30}$$

where $\mathbf{f}_{\text{even}}$ are the even index elements of $\mathbf{f}$, $\mathbf{f}_{\text{odd}}$ are the odd index elements of $\mathbf{f}$, $\mathbf{I}_{512}$ is the 512×512 identity matrix, and $\mathbf{D}_{512}$ is given by

$$\mathbf{D}_{512} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & \omega & 0 & \cdots & 0 \\ 0 & 0 & \omega^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \omega^{511} \end{bmatrix}. \tag{2.31}$$

This expression can be derived from a careful accounting and reorganization of the terms in (2.26) and (2.29). If $n = 2^p$, this process can be repeated, and $\mathbf{F}_{512}$ can be represented by $\mathbf{F}_{256}$, which can then be represented by $\mathbf{F}_{128} \rightarrow \mathbf{F}_{64} \rightarrow \mathbf{F}_{32} \rightarrow \cdots$. If $n \neq 2^p$, the vector can be padded with zeros until it is a power of 2. The FFT then involves an efficient interleaving of even and odd indices of sub-vectors of $\mathbf{f}$, and the computation of several smaller 2×2 DFT computations.

FFT Example: Noise Filtering

To gain familiarity with how to use and interpret the FFT, we will begin with a simple example that uses the FFT to denoise a signal. We will consider a function of time $f(t)$:

$$f(t) = \sin(2\pi f_1 t) + \sin(2\pi f_2 t) \tag{2.32}$$

with frequencies $f_1 = 50$ and $f_2 = 120$. We then add a large amount of Gaussian white noise to this signal, as shown in the top panel of Fig. 2.9.

It is possible to compute the fast Fourier transform of this noisy signal using the **fft** command. The power spectral density (PSD) is the normalized squared magnitude of $\hat{\mathbf{f}}$, and indicates how much *power* the signal contains in each frequency. In Fig. 2.9 (middle), it is clear that the noisy signal contains two large peaks at 50 Hz and 120 Hz. It is possible to zero out components that have power below a threshold to remove noise from the signal. After inverse transforming the filtered signal, we find the clean and filtered time-series match quite well (Fig. 2.9, bottom). Code 2.3 performs each step and plots the results.

Code 2.3 Fast Fourier transform to denoise signal.

```
dt = .001;
t = 0:dt:1;
f = sin(2*pi*50*t) + sin(2*pi*120*t); % Sum of 2 frequencies
f = f + 2.5*randn(size(t));  %  Add some noise

%% Compute the Fast Fourier Transform FFT
n = length(t);
fhat = fft(f,n);        % Compute the fast Fourier transform
PSD = fhat.*conj(fhat)/n; % Power spectrum (power per freq)
freq = 1/(dt*n)*(0:n); % Create x-axis of frequencies in Hz
L = 1:floor(n/2);    % Only plot the first half of freqs

%% Use the PSD to filter out noise
indices = PSD>100;  % Find all freqs with large power
PSDclean = PSD.*indices;  % Zero out all others
fhat = indices.*fhat;  % Zero out small Fourier coeffs. in Y
ffilt = ifft(fhat); % Inverse FFT for filtered time signal
```

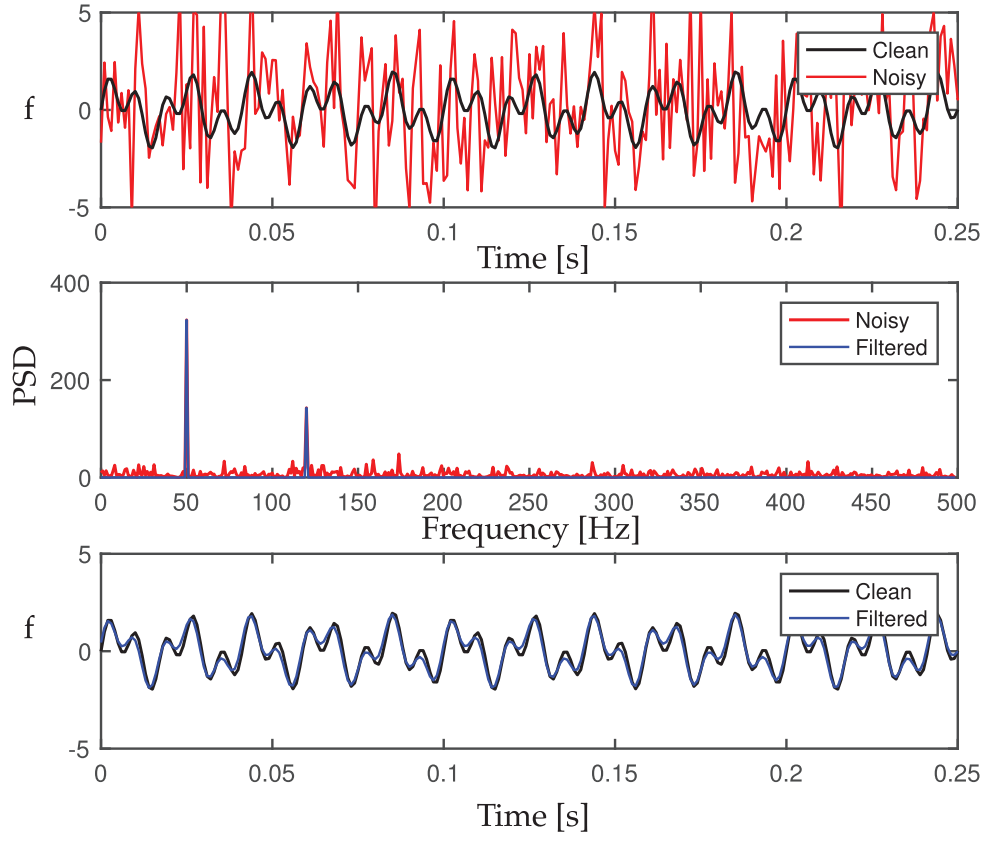

Figure 2.9 De-noising with FFT. (top) Noise is added to a simple signal given by a sum of two sine waves. (middle) In the Fourier domain, dominant peaks may be selected and the noise filtered. (bottom) The de-noised signal is obtained by inverse Fourier transforming the two dominant peaks.

```
%% PLOTS
subplot(3,1,1)
plot(t,f,'r','LineWidth',1.2), hold on
plot(t,f,'k','LineWidth',1.5)
legend('Noisy','Clean')

subplot(3,1,2)
plot(t,f,'k','LineWidth',1.5), hold on
plot(t,ffilt,'b','LineWidth',1.2)
legend('Clean','Filtered')

subplot(3,1,3)
plot(freq(L),PSD(L),'r','LineWidth',1.5), hold on
plot(freq(L),PSDclean(L),'-b','LineWidth',1.2)
legend('Noisy','Filtered')
```

FFT Example: Spectral Derivatives

For the next example, we will demonstrate the use of the FFT for the fast and accurate computation of derivatives. As we saw in (2.19), the continuous Fourier transform has

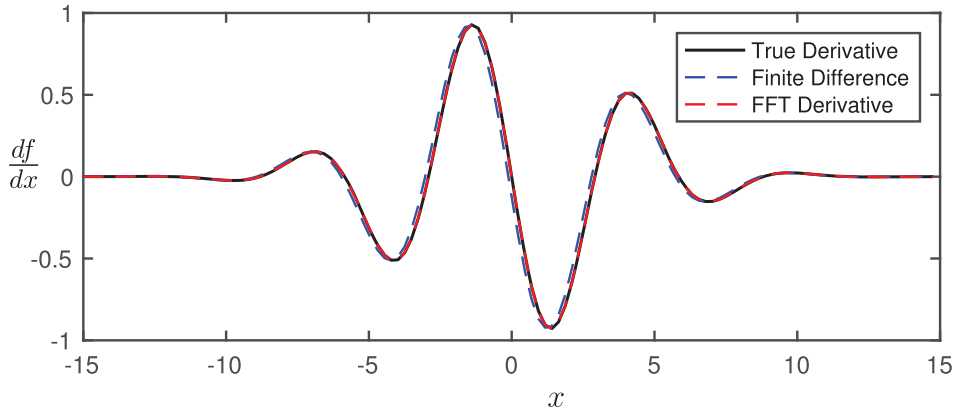

Figure 2.10 Comparison of the spectral derivative, computed using the FFT, with the finite-difference derivative.

the property that $\mathcal{F}(df/dx) = i\omega\mathcal{F}(f)$. Similarly, the numerical derivative of a vector of discretized data can be well approximated by multiplying each component of the discrete Fourier transform of the vector $\hat{\mathbf{f}}$ by $i\kappa$, where $\kappa = 2\pi k/n$ is the discrete wavenumber associated with that component. The accuracy and efficiency of the spectral derivative makes it particularly useful for solving partial differential equations, as explored in the next section.

To demonstrate this so-called *spectral* derivative, we will start with a function $f(x)$ where we can compute the analytic derivative for comparison:

$$f(x) = \cos(x)e^{-x^2/25} \quad \Longrightarrow \quad \frac{df}{dx}(x) = -\sin(x)e^{-x^2/25} - \frac{2}{25}xf(x). \tag{2.33}$$

Fig. 2.10 compares the spectral derivative with the analytic derivative and the forward Euler finite-difference derivative using $n = 128$ discretization points:

$$\frac{df}{dx}(x_k) \approx \frac{f(x_{k+1}) - f(x_k)}{\Delta x}. \tag{2.34}$$

The error of both differentiation schemes may be reduced by increasing n, which is the same as decreasing Δx. However, the error of the spectral derivative improves more rapidly with increasing n than finite-difference schemes, as shown in Fig. 2.11. The forward Euler differentiation is notoriously inaccurate, with error proportional to $\mathcal{O}(\Delta x)$; however, even increasing the order of a finite-difference scheme will not yield the same accuracy trend as the spectral derivative, which is effectively using information on the whole domain. Code 2.4 computes and compares the two differentiation schemes.

Code 2.4 Fast Fourier transform to compute derivatives.

```
n = 128;
L = 30;
dx = L/(n);
x = -L/2:dx:L/2-dx;
f = cos(x).*exp(-x.^2/25);                    % Function
df = -(sin(x).*exp(-x.^2/25) + (2/25)*x.*f);  % Derivative

%% Approximate derivative using finite Difference...
```

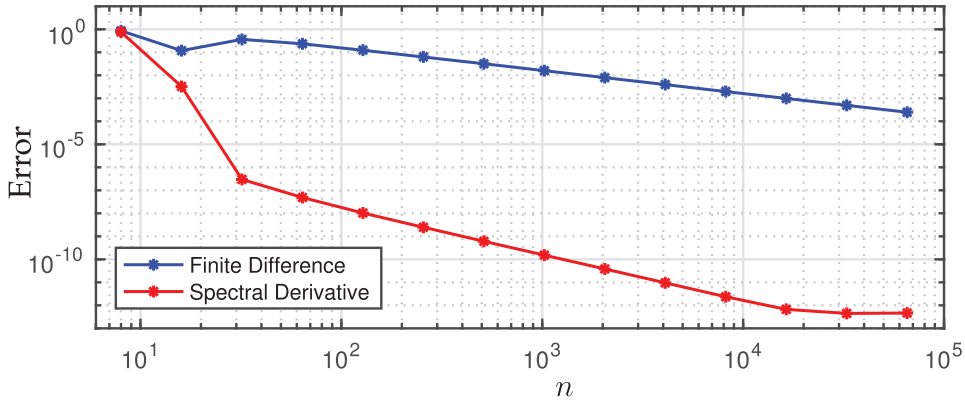

Figure 2.11 Benchmark of spectral derivative for varying data resolution.

```
for kappa=1:length(df)-1
    dfFD(kappa) = (f(kappa+1)-f(kappa))/dx;
end
dfFD(end+1) = dfFD(end);

%% Derivative using FFT (spectral derivative)
fhat = fft(f);
kappa = (2*pi/L)*[-n/2:n/2-1];
kappa = fftshift(kappa);  % Re-order fft frequencies
dfhat = i*kappa.*fhat;
dfFFT = real(ifft(dfhat));

%% Plotting commands
plot(x,df,'k','LineWidth',1.5), hold on
plot(x,dfFD,'b--','LineWidth',1.2)
plot(x,dfFFT,'r--','LineWidth',1.2)
legend('True Derivative','Finite Diff.','FFT Derivative')
```

If the derivative of a function is discontinuous, then the spectral derivative will exhibit Gibbs phenomena, as shown in Fig. 2.12.

2.3 Transforming Partial Differential Equations

The Fourier transform was originally formulated in the 1800s as a change of coordinates for the heat equation into an eigenfunction coordinate system where the dynamics decouple. More generally, the Fourier transform is useful for transforming partial differential equations (PDEs) into ordinary differential equations (ODEs), as in (2.20). Here, we will demonstrate the utility of the FFT to numerically solve a number of PDEs. For an excellent treatment of spectral methods for PDEs, see Trefethen [523]; extensions also exist for stiff PDEs [282].

Heat Equation

The Fourier transform basis is ideally suited to solve the heat equation. In one spatial dimension, the heat equation is given by

$$u_t = \alpha^2 u_{xx} \tag{2.35}$$

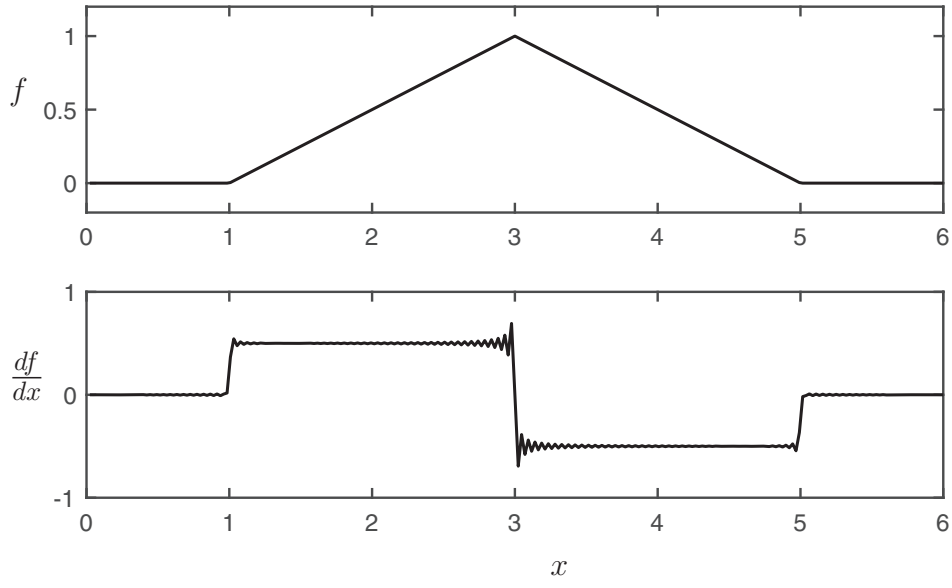

Figure 2.12 Gibbs phenomena for spectral derivative of function with discontinuous derivative.

where $u(t, x)$ is the temperature distribution in time and space. If we Fourier transform in space, then $\mathcal{F}(u(t, x)) = \hat{u}(t, \omega)$. The PDE in (2.35) becomes:

$$\hat{u}_t = -\alpha^2 \omega^2 \hat{u} \tag{2.36}$$

since the two spatial derivatives contribute $(i\omega)^2 = -\omega^2$ in the Fourier transform domain. Thus, by taking the Fourier transform, the PDE in (2.35) becomes an ODE for each fixed frequency ω. The solution is given by:

$$\hat{u}(t, \omega) = e^{-\alpha^2 \omega^2 t} \hat{u}(0, \omega). \tag{2.37}$$

The function $\hat{u}(0, \omega)$ is the Fourier transform of the initial temperature distribution $u(0, x)$. It is now clear that higher frequencies, corresponding to larger values of ω, decay more rapidly as time evolves, so that sharp corners in the temperature distribution rapidly smooth out. We may take the inverse Fourier transform using the convolution property in (2.24), yielding:

$$u(t, x) = \mathcal{F}^{-1}(\hat{u}(t, \omega)) = \mathcal{F}^{-1}\left(e^{-\alpha^2 \omega^2 t}\right) * u(0, x) = \frac{1}{2\alpha\sqrt{\pi t}} e^{-\frac{x^2}{4\alpha^2 t}} * u(0, x). \tag{2.38}$$

To simulate this PDE numerically, it is simpler and more accurate to first transform to the frequency domain using the FFT. In this case (2.36) becomes

$$\hat{u}_t = -\alpha^2 \kappa^2 \hat{u} \tag{2.39}$$

where κ is the discretized frequency. It is important to use the **fftshift** command to re-order the wavenumbers according to the Matlab convention.

Code 2.5 simulates the 1D heat equation using the FFT, as shown in Figs. 2.13 and 2.14. In this example, because the PDE is linear, it is possible to advance the system using **ode45**

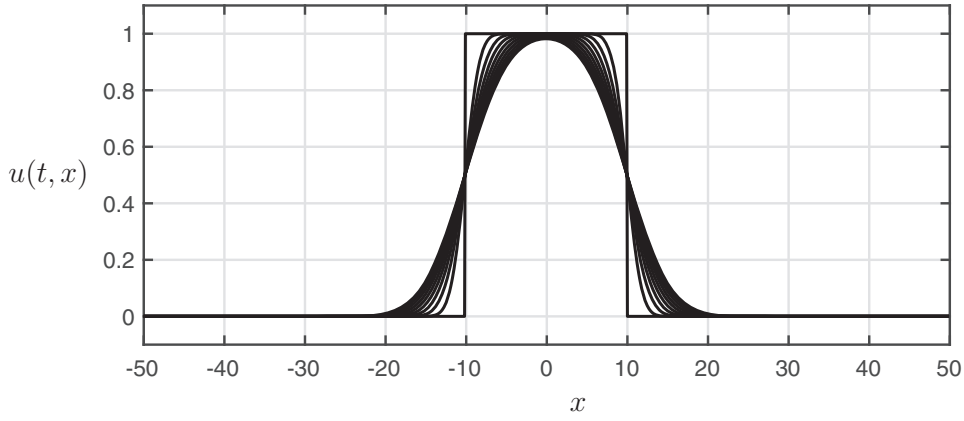

Figure 2.13 Solution of the 1D heat equation in time for an initial condition given by a square hat function. As time evolves, the sharp corners rapidly smooth and the solution approaches a Gaussian function.

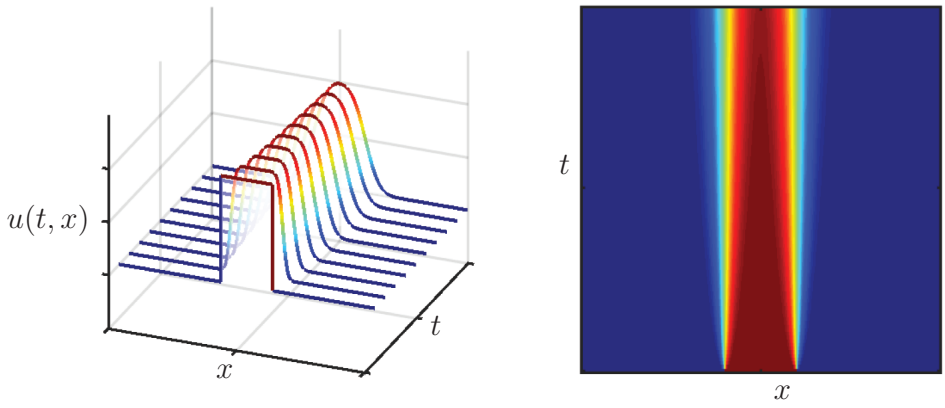

Figure 2.14 Evolution of the 1D heat equation in time, illustrated by a waterfall plot (left) and an x-t diagram (right).

directly in the frequency domain, using the vector field given in Code 2.6. Finally, the plotting commands are given in Code 2.7.

Figs. 2.13 and 2.14 show several different views of the temperature distribution $u(t, x)$ as it evolves in time. Fig. 2.13 shows the distribution at several times overlayed, and this same data is visualized in Fig. 2.14 in a waterfall plot (left) and in an x-t diagram (right). In all of the figures, it becomes clear that the sharp corners diffuse rapidly, as these correspond to the highest wavenumbers. Eventually, the lowest wavenumber variations will also decay, until the temperature reaches a constant steady state distribution, which is a solution of Laplace's equation $u_{xx} = 0$. When solving this PDE using the FFT, we are implicitly assuming that the solution domain is periodic, so that the right and left boundaries are identified and the domain forms a ring. However, if the domain is large enough, then the effect of the boundaries is small.

Code 2.5 Code to simulate the 1D heat equation using the Fourier transform.

```
a = 1;          % Thermal diffusivity constant
L = 100;        % Length of domain
N = 1000;       % Number of discretization points
dx = L/N;
x = -L/2:dx:L/2-dx; % Define x domain

% Define discrete wavenumbers
kappa = (2*pi/L)*[-N/2:N/2-1];
kappa = fftshift(kappa);     % Re-order fft wavenumbers

% Initial condition
u0 = 0*x;
u0((L/2 - L/10)/dx:(L/2 + L/10)/dx) = 1;

% Simulate in Fourier frequency domain
t = 0:0.1:10;
[t,uhat]=ode45(@(t,uhat)rhsHeat(t,uhat,kappa,a),t,fft(u0));

for k = 1:length(t) % iFFT to return to spatial domain
    u(k,:) = ifft(uhat(k,:));
end
```

Code 2.6 Right-hand side for 1D heat equation in Fourier domain, $d\hat{u}/dt$.

```
function duhatdt = rhsHeat(t,uhat,kappa,a)
duhatdt = -a^2*(kappa.^2)'.*uhat;  % Linear and diagonal
```

Code 2.7 Code to plot the solution of the 1D heat equation.

```
figure, waterfall((u(1:10:end,:)));
figure, imagesc(flipud(u));
```

One-Way Wave Equation

As second example is the simple linear PDE for the one-way equation:

$$u_t + cu_x = 0. \tag{2.40}$$

Any initial condition $u(0, x)$ will simply propagate to the right in time with speed c, as $u(t, x) = u(0, x - ct)$ is a solution. Code 2.8 simulates this PDE for an initial condition given by a Gaussian pulse. It is possible to integrate this equation in the Fourier transform domain, as before, using the vector field given by Code 2.9. However, it is also possible to integrate this equation in the spatial domain, simply using the FFT to compute derivatives and then transform back, as in Code 2.10. The solution $u(t, x)$ is plotted in Figs. 2.15 and 2.16, as before.

Code 2.8 Code to simulate the 1D wave equation using the Fourier transform.

```
c = 2;              % Wave speed
L = 20;             % Length of domain
N = 1000;           % Number of discretization points
dx = L/N;
x = -L/2:dx:L/2-dx; % Define x domain

% Define discrete wavenumbers
kappa = (2*pi/L)*[-N/2:N/2-1];
```

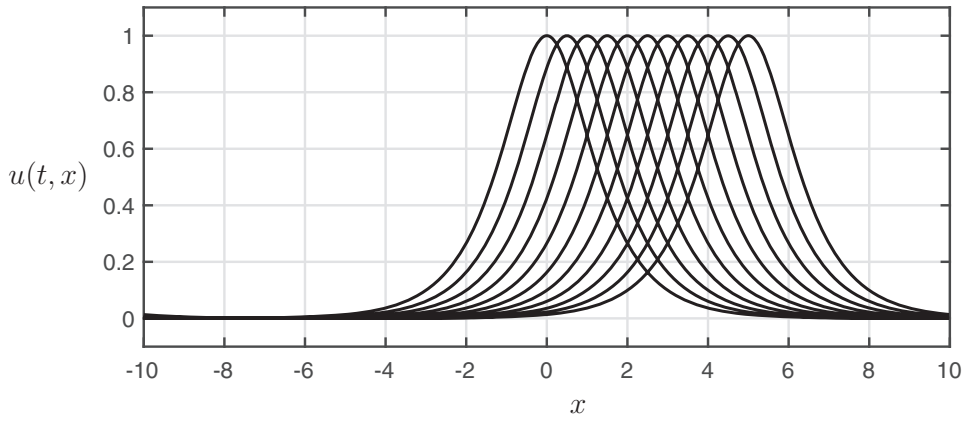

Figure 2.15 Solution of the 1D wave equation in time. As time evolves, the Gaussian initial condition moves from left to right at a constant wave speed.

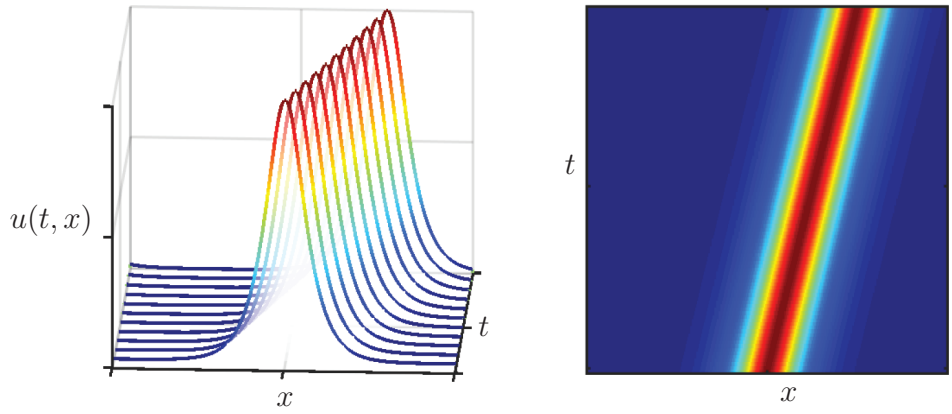

Figure 2.16 Evolution of the 1D wave equation in time, illustrated by a waterfall plot (left) and an x-t diagram (right).

```
kappa = fftshift(kappa');     % Re-order fft wavenumbers

% Initial condition
u0 = sech(x);
uhat0 = fft(u0);

% Simulate in Fourier frequency domain
dt = 0.025;
t = 0:dt:100*dt;
[t,uhat] = ode45(@(t,uhat)rhsWave(t,uhat,kappa,c),t,uhat0);

% Alternatively, simulate in spatial domain
[t,u] = ode45(@(t,u)rhsWaveSpatial(t,u,kappa,c),t,u0);
```

Code 2.9 Right hand side for 1D wave equation in Fourier transform domain.

```
function duhatdt = rhsWave(t,uhat,kappa,c)
duhatdt = -c*i*kappa.*uhat;
```

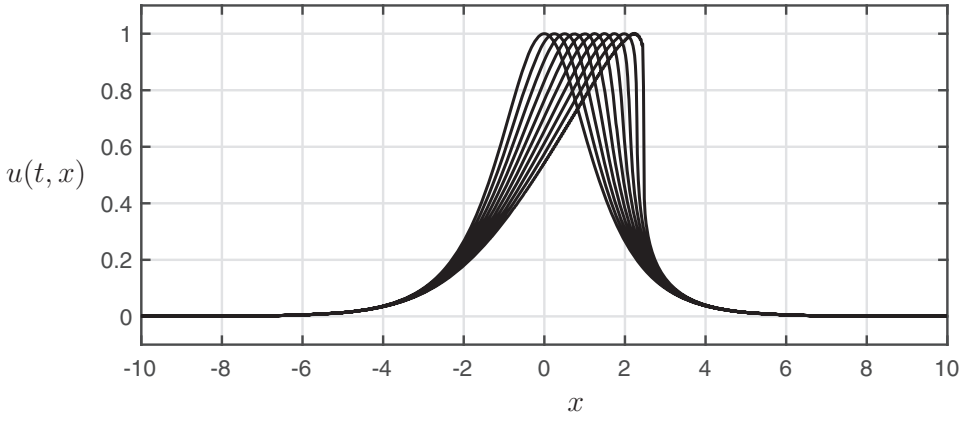

Figure 2.17 Solution of Burgers' equation in time. As time evolves, the leading edge of the Gaussian initial condition steepens, forming a shock front.

Code 2.10 Right hand side for 1D wave equation in spatial domain.

```
function dudt = rhsWaveSpatial(t,u,kappa,c)
uhat = fft(u);
duhat = i*kappa.*uhat;
du = ifft(duhat);
dudt = -c*du;
```

Burgers' Equation

For the final example, we consider the nonlinear Burgers' equation

$$u_t + uu_x = vu_{xx} \tag{2.41}$$

which is a simple 1D example for the nonlinear convection and diffusion that gives rise to shock waves in fluids [253]. The nonlinear convection uu_x essentially gives rise to the behavior of wave steepening, where portions of u with larger amplitude will convect more rapidly, causing a shock front to form.

Code 2.11 simulates the Burgers' equation, giving rise to Figs. 2.17 and 2.18. Burgers' equation is an interesting example to solve with the FFT, because the nonlinearity requires us to map into and out of the Fourier domain at each time step, as shown in the vector field in Code 2.12. In this example, we map into the Fourier transform domain to compute u_x and u_{xx}, and then map back to the spatial domain to compute the product uu_x. Figs. 2.17 and 2.18 clearly show the wave steepening effect that gives rise to a shock. Without the damping term u_{xx}, this shock would become infinitely steep, but with damping, it maintains a finite width.

Code 2.11 Code to simulate Burgers' equation using the Fourier transform.

```
clear all, close all, clc
nu=0.001;   % Diffusion constant

% Define spatial domain
L = 20;              % Length of domain
N = 1000;            % Number of discretization points
```

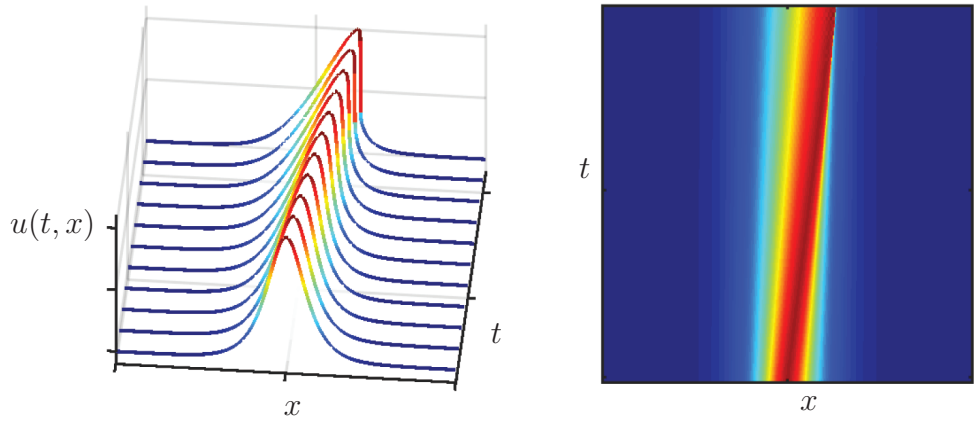

Figure 2.18 Evolution of Burgers' equation in time, illustrated by a waterfall plot (left) and an x-t diagram (right).

```
dx = L/N;
x = -L/2:dx:L/2-dx; % Define x domain

% Define discrete wavenumbers
kappa = (2*pi/L)*[-N/2:N/2-1];
kappa = fftshift(kappa');     % Re-order fft wavenumbers

% Initial condition
u0 = sech(x);

% Simulate PDE in spatial domain
dt = 0.025;
t = 0:dt:100*dt;
[t,u] = ode45(@(t,u)rhsBurgers(t,u,kappa,nu),t,u0);
```

Code 2.12 Right hand side for Burgers' equation in Fourier transform domain.

```
function dudt = rhsBurgers(t,u,kappa,nu)
uhat = fft(u);
duhat = i*kappa.*uhat;
dduhat = -(kappa.^2).*uhat;
du = ifft(duhat);
ddu = ifft(dduhat);
dudt = -u.*du + nu*ddu;
```

2.4 Gabor Transform and the Spectrogram

Although the Fourier transform provides detailed information about the frequency content of a given signal, it does not give any information about when in time those frequencies occur. The Fourier transform is only able to characterize truly periodic and stationary signals, as time is stripped out via the integration in (2.18a). For a signal with nonstationary frequency content, such as a musical composition, it is important to simultaneously characterize the frequency content and its evolution in time.

The Gabor transform, also known as the short-time Fourier transform (STFT), computes a windowed FFT in a moving window [437, 262, 482], as shown in Fig. 2.19. This STFT

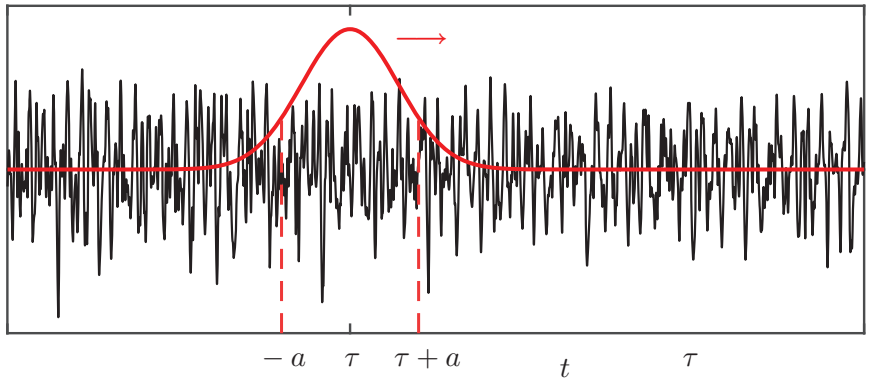

Figure 2.19 Illustration of the Gabor transform with a translating Gaussian window for the short-time Fourier transform.

enables the localization of frequency content in time, resulting in the *spectrogram*, which is a plot of frequency versus time, as demonstrated in Figs. 2.21 and 2.22. The STFT is given by:

$$\mathcal{G}(f)(t,\omega) = \hat{f}_g(t,\omega) = \int_{-\infty}^{\infty} f(\tau) e^{-i\omega\tau} \bar{g}(\tau - t)\, d\tau = \langle f, g_{t,\omega} \rangle \tag{2.42}$$

where $g_{t,\omega}(\tau)$ is defined as

$$g_{t,\omega}(\tau) = e^{i\omega\tau} g(\tau - t). \tag{2.43}$$

The function $g(t)$ is the kernel, and is often chosen to be a Gaussian:

$$g(t) = e^{-(t-\tau)^2/a^2}. \tag{2.44}$$

The parameter a determines the spread of the short-time window for the Fourier transform, and τ determines the center of the moving window.

The inverse STFT is given by:

$$f(t) = \mathcal{G}^{-1}\left(\hat{f}_g(t,\omega)\right) = \frac{1}{2\pi \|g\|^2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \hat{f}_g(\tau,\omega) g(t-\tau) e^{i\omega t}\, d\omega\, dt. \tag{2.45}$$

Discrete Gabor Transform

Generally, the Gabor transform will be performed on discrete signals, as with the FFT. In this case, it is necessary to discretize both time and frequency:

$$\nu = j\Delta\omega \tag{2.46}$$

$$\tau = k\Delta t. \tag{2.47}$$

The discretized kernel function becomes:

$$g_{j,k} = e^{i2\pi j \Delta\omega t} g(t - k\Delta t) \tag{2.48}$$

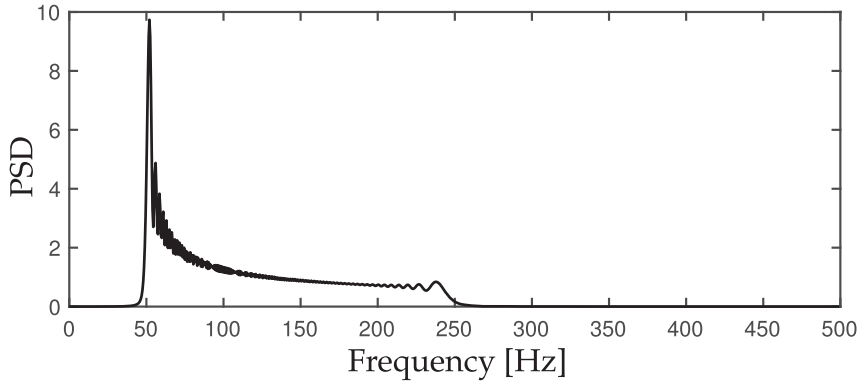

Figure 2.20 Power spectral density of quadratic chirp signal.

and the discrete Gabor transform is:

$$\hat{f}_{j,k} = \langle f, g_{j,k} \rangle = \int_{-\infty}^{\infty} f(\tau) \bar{g}_{j,k}(\tau)\, d\tau. \tag{2.49}$$

This integral can then be approximated using a finite Riemman sum on discretized functions f and $\bar{g}_{j,k}$.

Example: Quadratic Chirp

As a simple example, we construct an oscillating cosine function where the frequency of oscillation increases as a quadratic function of time:

$$f(t) = \cos(2\pi t \omega(t)) \quad \text{where} \quad \omega(t) = \omega_0 + (\omega_1 - \omega_0) t^2 / 3t_1^2. \tag{2.50}$$

The frequency shifts from ω_0 at $t = 0$ to ω_1 at $t = t_1$.

Fig. 2.20 shows the power spectral density obtained from the FFT of the quadratic chirp signal. Although there is a clear peak at 50 Hz, there is no information about the progression of the frequency in time. The code to generate the spectrogram is given in Code 2.13, and the resulting spectrogram is plotted in Fig. 2.21, where it can be seen that the frequency content shifts in time.

Code 2.13 Spectrogram of quadratic chirp, shown in Fig. 2.21.

```
t = 0:0.001:2;
f0 = 50;
f1 = 250;
t1 = 2;
x = chirp(t,f0,t1,f1,'quadratic');
x = cos(2*pi*t.*(f0 + (f1-f0)*t.^2/(3*t1^2)));
% There is a typo in Matlab documentation...
% ... divide by 3 so derivative amplitude matches frequency

spectrogram(x,128,120,128,1e3,'yaxis')
```

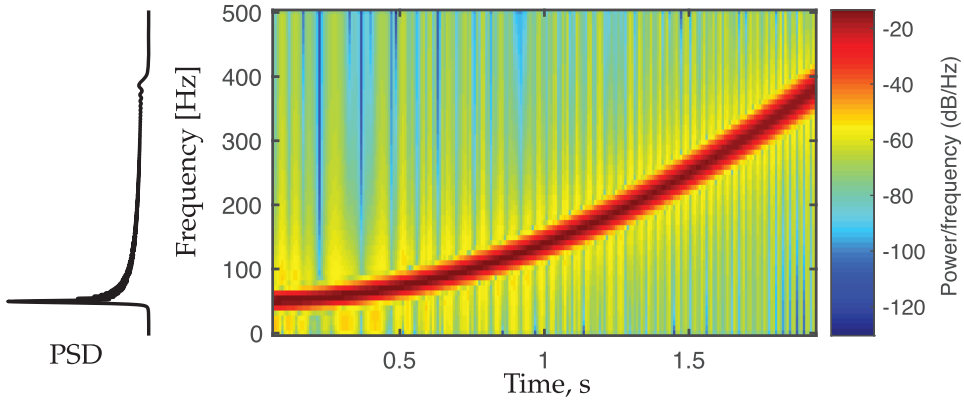

Figure 2.21 Spectrogram of quadratic chirp signal. The PSD is shown on the left, corresponding to the integrated power across rows of the spectrogram.

Example: Beethoven's Sonata Pathétique

It is possible to analyze richer signals with the spectrogram, such as Beethoven's Sonata Pathétique, shown in Fig. 2.22. The spectrogram is widely used to analyze music, and has recently been leveraged in the Shazam algorithm, which searches for key point markers in the spectrogram of songs to enable rapid classification from short clips of recorded music [545].

Fig. 2.22 shows the first two bars of Beethoven's Sonata Pathétique, along with the spectrogram. In the spectrogram, the various chords and harmonics can be seen clearly. A zoom-in of the frequency shows two octaves, and how cleanly the various notes are excited. Code 2.14 loads the data, computes the spectrogram, and plots the result.

Code 2.14 Compute spectrogram of Beethoven's Sonata Pathétique (Fig. 2.22).

```
% Download mp3read from http://www.mathworks.com/matlabcentral/
    fileexchange/13852-mp3read-and-mp3write
[Y,FS,NBITS,OPTS] = mp3read('beethoven.mp3');

%% Spectrogram using 'spectrogram' comand
T = 40;              % 40 seconds
y=Y(1:T*FS);         % First 40 seconds
spectrogram(y,5000,400,24000,24000,'yaxis');

%% Spectrogram using short-time Fourier transform 'stft'
wlen = 5000;    % Window length
h=400;          % Overlap is wlen - h
% Perform time-frequency analysis
[S,f,t_stft] = stft(y, wlen, h, FS/4, FS); % y axis 0-4000HZ

imagesc(log10(abs(S)));    % Plot spectrogram (log-scaled)
```

To invert the spectrogram and generate the original sound:

```
[x_istft, t_istft] = istft(S, h, FS/4, FS);
sound(x_istft,FS);
```

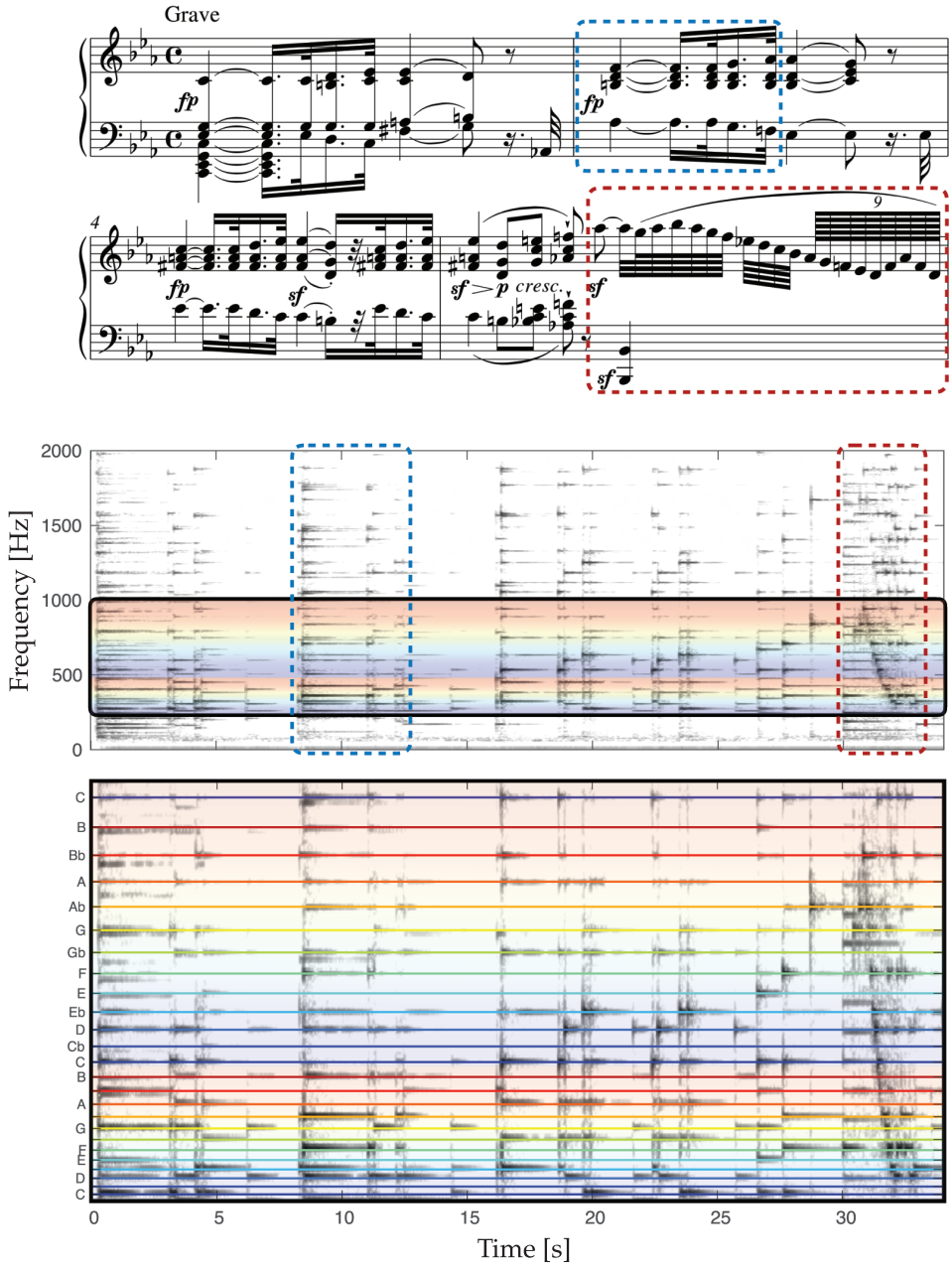

Figure 2.22 First two bars of Beethoven's Sonata Pathétique (No. 8 in C minor, Op. 13), along with annotated spectrogram.

Artists, such as Aphex Twin, have used the inverse spectrogram of images to generate music. The frequency of a given piano key is also easily computed. For example, the 40th key frequency is given by:

```
freq = @(n)(((2^(1/12))^(n-49))*440);
freq(40) % frequency of 40th key = C
```

Uncertainty Principles

In time-frequency analysis, there is a fundamental uncertainty principle that limits the ability to simultaneously attain high resolution in both the time and frequency domains. In the extreme limit, a time series is perfectly resolved in time, but provides no information about frequency content, and the Fourier transform perfectly resolves frequency content, but provides no information about when in time these frequencies occur. The spectrogram resolves both time and frequency information, but with lower resolution in each domain, as illustrated in Fig. 2.23. An alternative approach, based on a multi-resolution analysis, will be the subject of the next section.

Stated mathematically, the time-frequency uncertainty principle [429] may be written as:

$$\left(\int_{-\infty}^{\infty} x^2 |f(x)|^2 \, dx\right)\left(\int_{-\infty}^{\infty} \omega^2 |\hat{f}(\omega)|^2 \, d\omega\right) \geq \frac{1}{16\pi^2}. \tag{2.51}$$

This is true if $f(x)$ is absolutely continuous and both $xf(x)$ and $f'(x)$ are square integrable. The function $x^2|f(x)|^2$ is the dispersion about $x = 0$. For real-valued functions, this is the second moment, which measures the variance if $f(x)$ is a Gaussian function. In other words, a function $f(x)$ and its Fourier transform cannot both be arbitrarily localized. If the

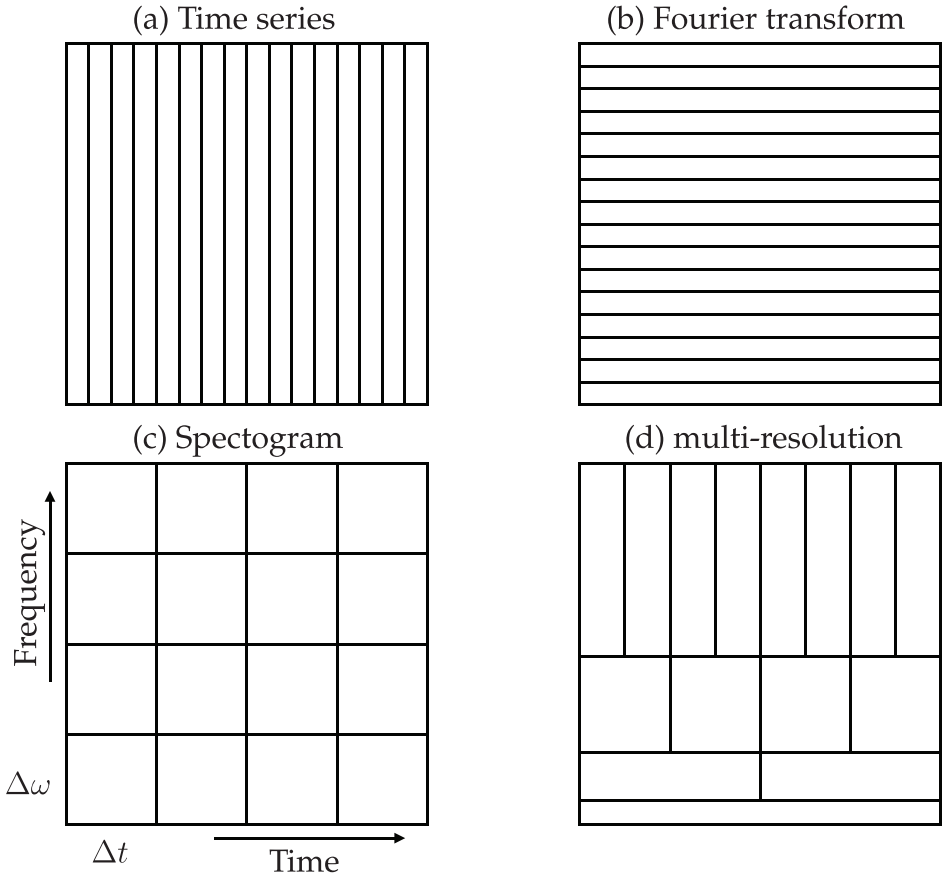

Figure 2.23 Illustration of resolution limitations and uncertainty in time-frequency analysis.

function f approaches a delta function, then the Fourier transform must become broadband, and vice versa. This has implications for the Heisenberg uncertainty principle [240], as the position and momentum wave functions are Fourier transform pairs.

In time-frequency analysis, the uncertainty principle has implication for the ability to localize the Fourier transform in time. These uncertainty principles are known as the Gabor limit. As the frequency content of a signal is resolved more finely, we lose information about when in time these events occur, and vice versa. Thus, there is a fundamental tradeoff between the simultaneously attainable resolutions in the time and frequency domains. Another implication is that a function f and its Fourier transform cannot both have finite support, meaning that they are localized, as stated in Benedick's theorem [8, 51].

2.5 Wavelets and Multi-Resolution Analysis

Wavelets [359, 145] extend the concepts in Fourier analysis to more general orthogonal bases, and partially overcome the uncertainty principle discussed above by exploiting a multi-resolution decomposition, as shown in Fig. 2.23 (d). This multi-resolution approach enables different time and frequency fidelities in different frequency bands, which is particularly useful for decomposing complex signals that arise from multi-scale processes such as are found in climatology, neuroscience, epidemiology, finance, and turbulence. Images and audio signals are also amenable to wavelet analysis, which is currently the leading method for image compression [16], as will be discussed in subsequent sections and chapters. Moreover, wavelet transforms may be computed using similar fast methods [58], making them scalable to high-dimensional data. There are a number of excellent books on wavelets [521, 401, 357], in addition to the primary references [359, 145].

The basic idea in wavelet analysis is to start with a function $\psi(t)$, known as the *mother* wavelet, and generate a family of scaled and translated versions of the function:

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right). \tag{2.52}$$

The parameters a and b are responsible for scaling and translating the function ψ, respectively. For example, one can imagine choosing a and b to scale and translate a function to fit in each of the segments in Fig. 2.23 (d). If these functions are orthogonal then the basis may be used for projection, as in the Fourier transform.

The simplest and earliest example of a wavelet is the *Haar* wavelet, developed in 1910 [227]:

$$\psi(t) = \begin{cases} 1 & 0 \le t < 1/2 \\ -1 & 1/2 \le t < 1 \\ 0 & \text{otherwise.} \end{cases} \tag{2.53}$$

The three Haar wavelets, $\psi_{1,0}$, $\psi_{1/2,0}$, and $\psi_{1/2,1/2}$, are shown in Fig. 2.24, representing the first two layers of the multi-resolution in Fig. 2.23 (d). Notice that by choosing each higher frequency layer as a bisection of the next layer down, the resulting Haar wavelets are orthogonal, providing a hierarchical basis for a signal.

The orthogonality property of wavelets described above is critical for the development of the discrete wavelet transform (DWT) below. However, we begin with the continuous

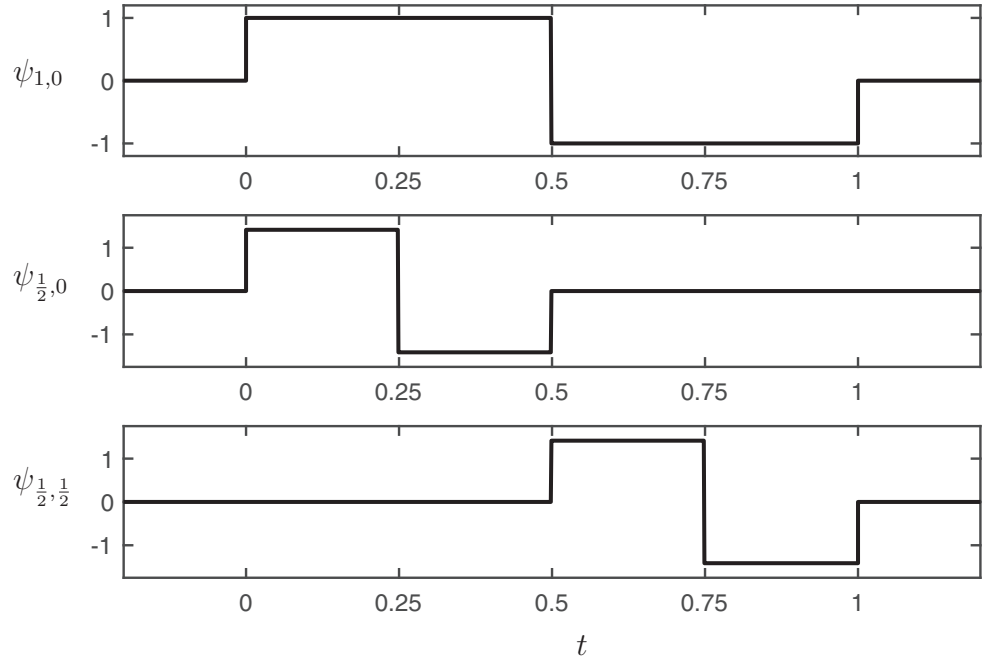

Figure 2.24 Three Haar wavelets for the first two levels of the multi-resolution in Fig. 2.23 (d).

wavelet transform (CWT), which is given by:

$$\mathcal{W}_\psi(f)(a,b) = \langle f, \psi_{a,b}\rangle = \int_{-\infty}^{\infty} f(t)\bar{\psi}_{a,b}(t)\,dt, \tag{2.54}$$

where $\bar{\psi}_{a,b}$ denotes the complex conjugate of $\psi_{a,b}$. This is only valid for functions $\psi(t)$ that satisfy the boundedness property that

$$C_\psi = \int_{-\infty}^{\infty} \frac{|\hat{\psi}(\omega)|^2}{|\omega|}\,d\omega < \infty. \tag{2.55}$$

The inverse continuous wavelet transform (iCWT) is given by:

$$f(t) = \frac{1}{C_\psi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathcal{W}_\psi(f)(a,b)\psi_{a,b}(t)\frac{1}{a^2}\,da\,db. \tag{2.56}$$

New wavelets may also be generated by the convolution $\psi * \phi$ if ψ is a wavelet and ϕ is a bounded and integrable function. There are many other popular mother wavelets ψ beyond the Haar wavelet, designed to have various properties. For example, the Mexican hat wavelet is given by:

$$\psi(t) = (1-t^2)e^{-t^2/2} \tag{2.57a}$$

$$\hat{\psi}(\omega) = \sqrt{2\pi}\omega^2 e^{-\omega^2/2}. \tag{2.57b}$$

Discrete Wavelet Transform

As with the Fourier transform and Gabor transform, when computing the wavelet transform on data, it is necessary to introduce a discretized version. The discrete wavelet transform (DWT) is given by:

$$\mathcal{W}_\psi(f)(j,k) = \langle f, \psi_{j,k} \rangle = \int_{-\infty}^{\infty} f(t)\bar{\psi}_{j,k}(t)\,dt \tag{2.58}$$

where $\psi_{j,k}(t)$ is a discrete family of wavelets:

$$\psi_{j,k}(t) = \frac{1}{a^j}\psi\left(\frac{t-kb}{a^j}\right). \tag{2.59}$$

Again, if this family of wavelets is orthogonal, as in the case of the discrete Haar wavelets described earlier, it is possible to expand a function $f(t)$ uniquely in this basis:

$$f(t) = \sum_{j,k=-\infty}^{\infty} \langle f(t), \psi_{j,k}(t) \rangle \psi_{j,k}(t). \tag{2.60}$$

The explicit computation of a DWT is somewhat involved, and is the subject of several excellent papers and texts [359, 145, 521, 401, 357]. However, the goal here is not to provide computational details, but rather to give a high-level idea of what the wavelet transform accomplishes. By scaling and translating a given shape across a signal, it is possible to efficiently extract multi-scale structures in an efficient hierarchy that provides an optimal tradeoff between time and frequency resolution. This general procedure is widely used in audio and image processing, compression, scientific computing, and machine learning, to name a few examples.

2.6 2D Transforms and Image Processing

Although we analyzed both the Fourier transform and the wavelet transform on one-dimensional signals, both methods readily generalize to higher spatial dimensions, such as two-dimensional and three-dimensional signals. Both the Fourier and wavelet transforms have had tremendous impact on image processing and compression, which provides a compelling example to investigate higher-dimensional transforms.

2D Fourier Transform for Images

The two-dimensional Fourier transform of a matrix of data $\mathbf{X} \in \mathbb{R}^{n\times m}$ is achieved by first applying the one-dimensional Fourier transform to every row of the matrix, and then applying the one-dimensional Fourier transform to every column of the intermediate matrix. This sequential row-wise and column-wise Fourier transform is shown in Fig. 2.25. Switching the order of taking the Fourier transform of rows and columns does not change the result.

Code 2.15 Two-dimensional Fourier transform via one-dimensional row-wise and column-wise FFTs.

```
A = imread('../../CH01_SVD/DATA/dog.jpg');
B = rgb2gray(A);     % Convert to grayscale image
subplot(1,3,1), imagesc(B);          % Plot image

for j=1:size(B,1);   % Compute row-wise FFT
```

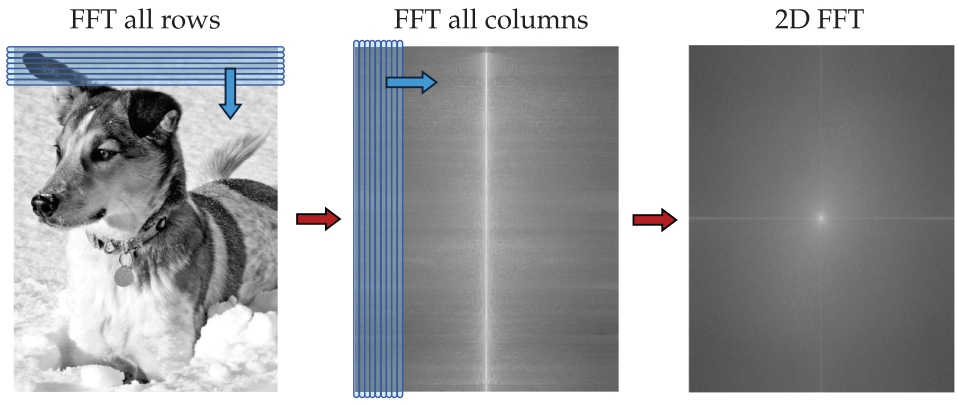

Figure 2.25 Schematic of 2D FFT. First, the FFT is taken of each row, and then the FFT is taken of each column of the resulting transformed matrix.

```
    Cshift(j,:) = fftshift(fft(B(j,:)));
    C(j,:) = (fft(B(j,:)));
end
subplot(1,3,2), imagesc(log(abs(Cshift)))

for j=1:size(C,2);  % Compute column-wise FFT
    D(:,j) = fft(C(:,j));
end
subplot(1,3,3), imagesc(fftshift(log(abs(D))))

D = fft2(B); % Much more efficient to use fft2
```

The two-dimensional FFT is effective for image compression, as many of the Fourier coefficients are small and may be neglected without loss in image quality. Thus, only a few large Fourier coefficients must be stored and transmitted.

Code 2.16 Image compression via the FFT.

```
Bt=fft2(B);     % B is grayscale image from above
Btsort = sort(abs(Bt(:)));  % Sort by magnitude

% Zero out all small coefficients and inverse transform
for keep=[.1 .05 .01 .002];
    thresh = Btsort(floor((1-keep)*length(Btsort)));
    ind = abs(Bt)>thresh;       % Find small indices
    Atlow = Bt.*ind;            % Threshold small indices
    Alow=uint8(ifft2(Atlow));   % Compressed image
    figure, imshow(Alow)       % Plot Reconstruction
end
```

Finally, the FFT is extensively used for denoising and filtering signals, as it is straightforward to isolate and manipulate particular frequency bands. Code 2.17 and Fig. 2.27 demonstrate the use of a FFT threshold filter to denoise an image with Gaussian noise added. In this example, it is observed that the noise is especially pronounced in high frequency modes, and we therefore zero out any Fourier coefficient outside of a given radius containing low frequencies.

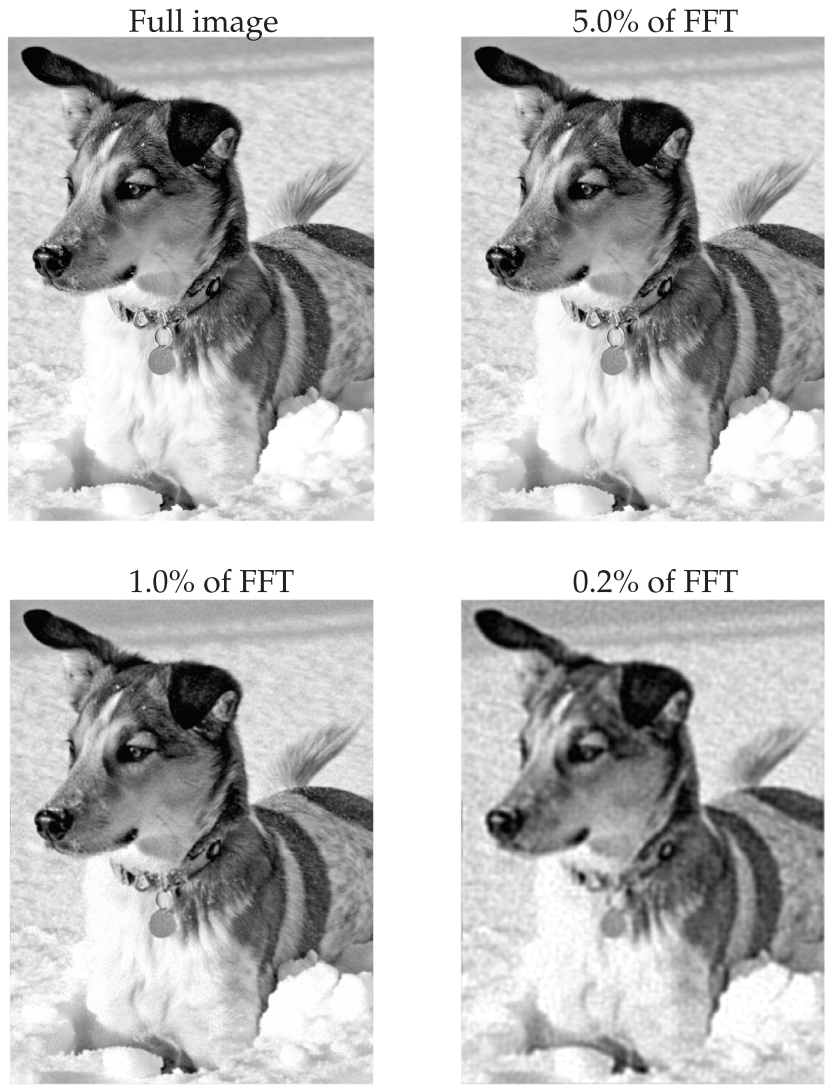

Figure 2.26 Compressed image using various thresholds to keep 5%, 1%, and 0.2% of the largest Fourier coefficients.

Code 2.17 Image denoising via the FFT.

```
Bnoise = B + uint8(200*randn(size(B)));  % Add some noise
Bt=fft2(Bnoise);
F = log(abs(Btshift)+1);        % Put FFT on log-scale

subplot(2,2,1), imagesc(Bnoise) % Plot image
subplot(2,2,2), imagesc(F)      % Plot FFT

[nx,ny] = size(B);
[X,Y] = meshgrid(-ny/2+1:ny/2,-nx/2+1:nx/2);
R2 = X.^2+Y.^2;
ind = R2<150^2;
```

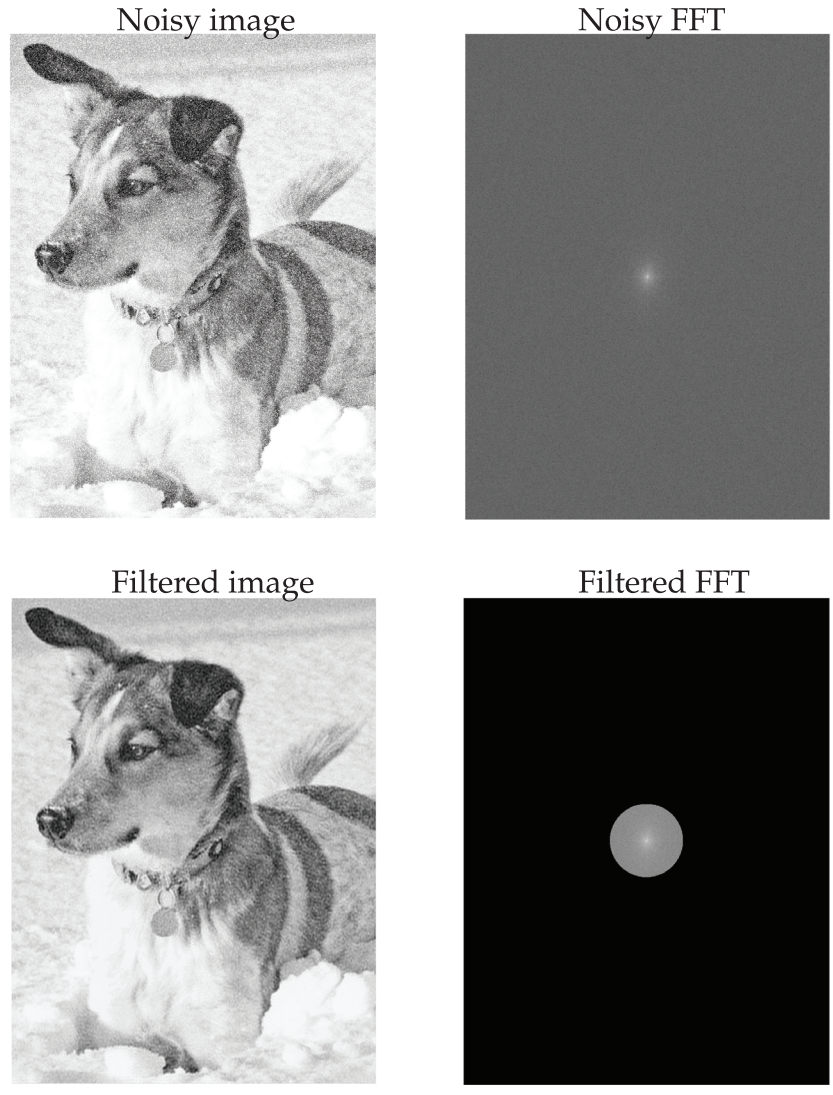

Figure 2.27 Denoising image by eliminating high-frequency Fourier coefficients outside of a given radius (bottom right).

```
Btshiftfilt = Btshift.*ind;
Ffilt = log(abs(Btshiftfilt)+1);   % Put FFT on log-scale
subplot(2,2,4), imagesc(Ffilt)     % Plot filtered FFT

Btfilt = ifftshift(Btshiftfilt);
Bfilt = ifft2(Btfilt);
subplot(2,2,3), imagesc(uint8(real(Bfilt))) % Filtered image
```

2D wavelet Transform for Images

Similar to the FFT, the discrete wavelet transform is extensively used for image processing and compression. Code 2.18 computes the wavelet transform of an image, and the first

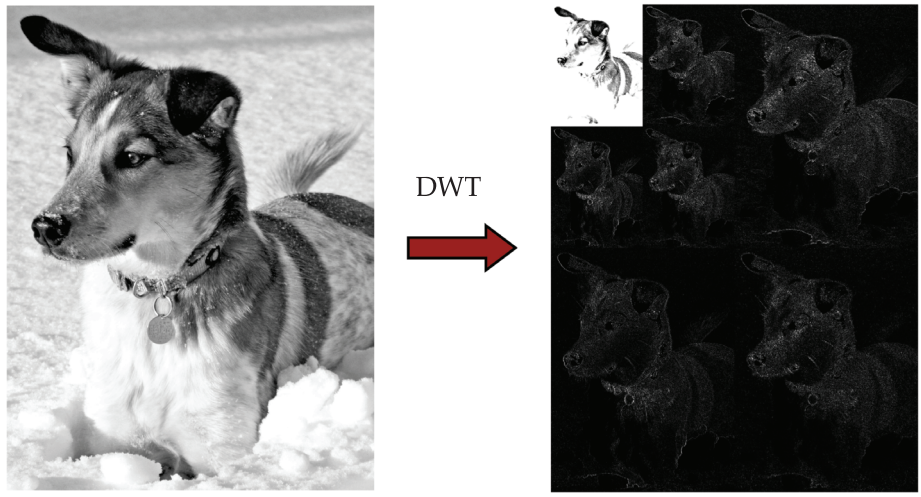

Figure 2.28 Illustration of three level discrete wavelet transform.

three levels are illustrated in Fig. 2.28. In this figure, the hierarchical nature of the wavelet decomposition is seen. The upper left corner of the DWT image is a low-resolution version of the image, and the subsequent features add fine details to the image.

Code 2.18 Example of a two level wavelet decomposition.

```
%% Wavelet decomposition (2 level)
n = 2; w = 'db1'; [C,S] = wavedec2(B,n,w);

% LEVEL 1
A1 = appcoef2(C,S,w,1); % Approximation
[H1 V1 D1] = detcoef2('a',C,S,k); % Details
A1 = wcodemat(A1,128);
H1 = wcodemat(H1,128);
V1 = wcodemat(V1,128);
D1 = wcodemat(D1,128);

% LEVEL 2
A2 = appcoef2(C,S,w,1); % Approximation
[H2 V2 D2] = detcoef2('a',C,S,k); % Details
A2 = wcodemat(A2,128);
H2 = wcodemat(H2,128);
V2 = wcodemat(V2,128);
D2 = wcodemat(D2,128);

dec2 = [A2 H2; V2 D2];
dec1 = [imresize(dec2,size(H1)) H1 ; V1 D1];
image(dec1);
```

Fig. 2.29 shows several versions of the compressed image for various compression ratios, as computed by Code 2.19. The hierarchical representation of data in the wavelet transform is ideal for image compression. Even with an aggressive truncation, retaining only 0.5% of the DWT coefficients, the coarse features of the image are retained. Thus,

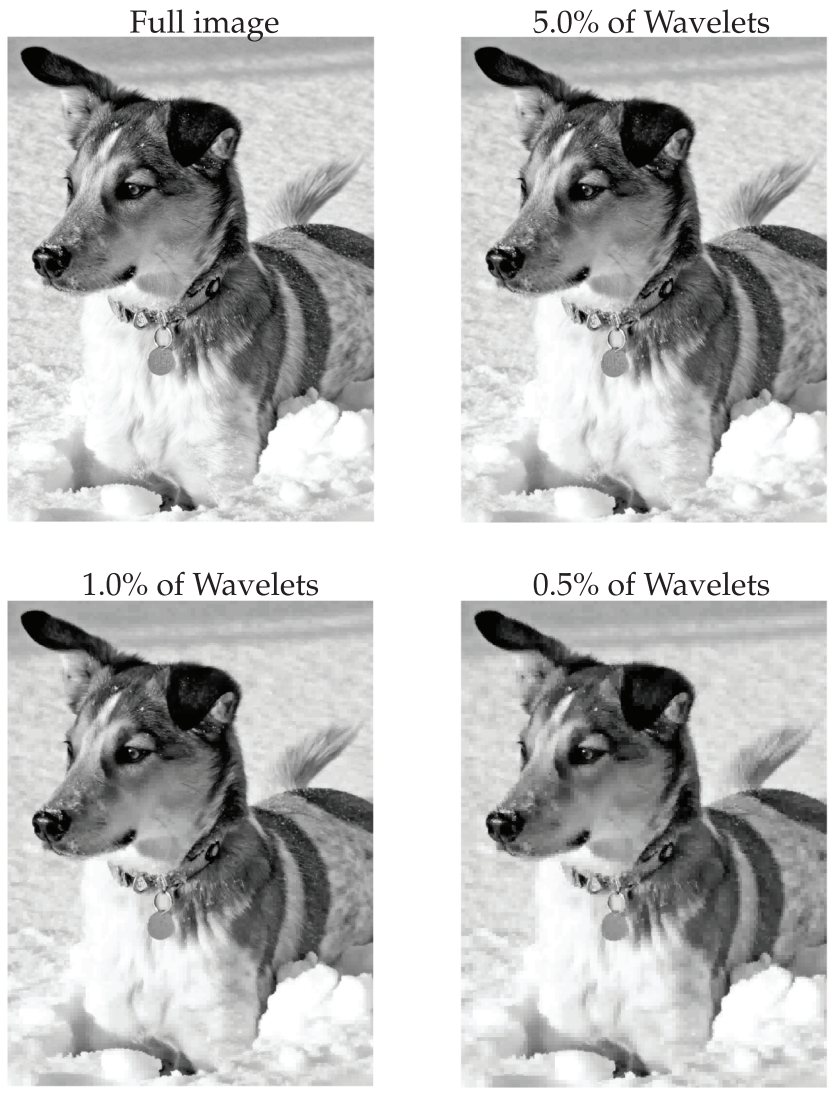

Figure 2.29 Compressed image using various thresholds to keep 5%, 1%, and 0.5% of the largest wavelet coefficients.

when transmitting data, even if bandwidth is limited and much of the DWT information is truncated, the most important features of the data are transferred.

Code 2.19 Wavelet decomposition for image compression.

```
[C,S] = wavedec2(B,4,'db1');
Csort = sort(abs(C(:)));  % Sort by magnitude

for keep =  [.1 .05 .01 .005]
    thresh = Csort(floor((1-keep)*length(Csort)));
    ind = abs(C)>thresh;
    Cfilt = C.*ind;       % Threshold small indices
```

```
    % Plot Reconstruction
    Arecon=uint8(waverec2(Cfilt,S,'db1'));
    figure, imagesc(uint8(Arecon))
end
```

Suggested Reading

Texts

(1) **The analytical theory of heat**, by J.-B. J. Fourier, 1978 [185].

(2) **A wavelet tour of signal processing**, by S. Mallat, 1999 [357].

(3) **Spectral methods in MATLAB**, by L. N. Trefethen, 2000 [523].

Papers and reviews

(1) **An algorithm for the machine calculation of complex Fourier series**, by J. W. Cooley and J. W. Tukey, *Mathematics of Computation*, 1965 [137].

(2) **The wavelet transform, time-frequency localization and signal analysis**, by I. Daubechies, *IEEE Transactions on Information Theory*, 1990 [145].

(3) **An industrial strength audio search algorithm**, by A. Wang et al., *Ismir*, 2003 [545].