# Chapter 9

# Normal Modes and Waves

We discuss the physics of wave phenomena and the motivation and use of Fourier transforms.

## 9.1 Coupled Oscillators and Normal Modes

Terms such as period, amplitude, and frequency are used to describe both waves and oscillatory motion. To understand the relation between waves and oscillatory motion, consider a flexible rope that is under tension with one end fixed. If we flip the free end, a pulse propagates along the rope with a speed that depends on the tension and on the inertial properties of the rope. At the *macroscopic* level, we observe a transverse wave that moves along the length of the rope. In contrast, at the *microscopic* level we see discrete particles undergoing oscillatory motion in a direction perpendicular to the motion of the wave. One goal of this chapter is to use simulations to understand the relation between the microscopic dynamics of a simple mechanical model and the macroscopic wave motion that the model can support.

For simplicity, we first consider a one-dimensional chain of $N$ particles each of mass $m$. The particles are coupled by massless springs with force constant $k$. The equilibrium separation between the particles is $a$. We denote the displacement of particle $j$ from its equilibrium position at time $t$ by $u_j(t)$ (see Figure 9.1). For many purposes the most realistic boundary conditions are to attach particles $j = 1$ and $j = N$ to springs which are attached to fixed walls. We denote the walls by $j = 0$ and $j = N + 1$, and require that $u_0(t) = u_{N+1}(t) = 0$.

The force on an individual particle is determined by the compression or extension of its adjacent springs. The equation of motion of particle $j$ is given by

$$\begin{aligned} m\frac{d^2u_j(t)}{dt^2} &= -k\big[u_j(t) - u_{j+1}(t)\big] - k\big[u_j(t) - u_{j-1}(t)\big] \\ &= -k\big[2u_j(t) - u_{j+1}(t) - u_{j-1}(t)\big]. \end{aligned} \tag{9.1}$$

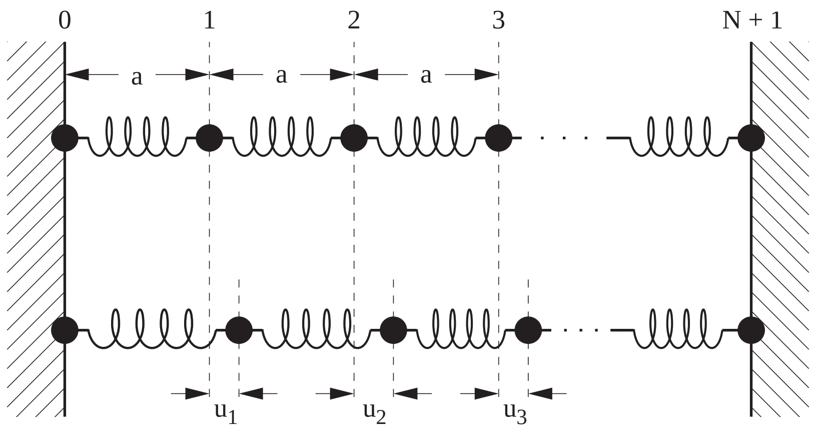

Figure 9.1: A one-dimensional chain of $N$ particles of mass $m$ coupled by massless springs with force constant $k$. The first and last particles (0 and $N+1$) are attached to fixed walls. The top chain shows the oscillators in equilibrium. The bottom chain shows the oscillators displaced from equilibrium.

Equation (9.1) couples the motion of particle $j$ to its two nearest neighbors and describes *longitudinal* oscillations, that is, motion along the length of the system. It is straightforward to show that identical equations hold for the *transverse* oscillations of $N$ identical mass points equally spaced on a stretched massless string (cf. French).

Because the equations of motion (9.1) are linear, that is, only terms proportional to the displacements appear, it is straightforward to obtain analytical solutions of (9.1). We first discuss these solutions because they will help us interpret the nature of the numerical solutions. To find the *normal modes*, we look for oscillatory solutions for which the displacement of each particle is proportional to $\sin\omega t$ or $\cos\omega t$. We write

$$u_j(t) = u_j \cos\omega t, \tag{9.2}$$

where $u_j$ is the amplitude of the displacement of the $j$th particle. If we substitute the form (9.2) into (9.1), we obtain

$$-\omega^2 u_j = -\frac{k}{m}[2u_j - u_{j+1} - u_{j-1}]. \tag{9.3}$$

We next assume that the amplitude $u_j$ depends sinusoidally on the distance $ja$:

$$u_j = C \sin qja, \tag{9.4}$$

where the constants $q$ and $C$ will be determined. If we substitute (9.4) into (9.3), we find the following condition for $\omega$:

$$-\omega^2 \sin qja = -\frac{k}{m}\big[2\sin qja - \sin q(j-1)a - \sin q(j+1)a\big]. \tag{9.5}$$

We write $\sin q(j \pm 1)a = \sin qja \cos qa \pm \cos qja \sin qa$ and find that (9.4) is a solution if

$$\omega^2 = 2\frac{k}{m}\big(1 - \cos qa\big). \tag{9.6}$$

We need to find the values of the wavenumber $q$ that satisfy the boundary conditions $u_0 = 0$ and $u_{N+1} = 0$. The former condition is automatically satisfied by assuming a sine instead of a cosine solution in (9.4). The latter boundary condition implies that

$$q = q_n = \frac{\pi n}{a(N+1)}, \qquad \text{(fixed boundary conditions)} \tag{9.7}$$

where $n = 1, \ldots, N$. The corresponding possible values of the wavelength $\lambda$ are related to $q$ by $q = 2\pi/\lambda$, and the corresponding values of the angular frequencies are given by

$$\omega_n{}^2 = 2\frac{k}{m}[1 - \cos q_n a] = 4\frac{k}{m}\sin^2\frac{q_n a}{2}, \tag{9.8}$$

or

$$\omega_n = 2\sqrt{\frac{k}{m}}\sin\frac{q_n a}{2}. \tag{9.9}$$

The relation (9.9) between $\omega_n$ and $q_n$ is known as a *dispersion relation.*

A particular value of the integer $n$ corresponds to the $n$th normal mode. We write the (time-independent) normal mode solutions as

$$u_{j,n} = C \sin q_n ja. \tag{9.10}$$

The linear nature of the equation of motion (9.1) implies that the time dependence of the displacement of the $j$th particle can be written as a superposition of normal modes:

$$u_j(t) = C\sum_{n=1}^{N}\big(A_n \cos\omega_n t + B_n \sin\omega_n t\big)\sin q_n ja. \tag{9.11}$$

The coefficients $A_n$ and $B_n$ are determined by the initial conditions:

$$u_j(t=0) = C\sum_{n=1}^{N} A_n \sin q_n ja, \tag{9.12a}$$

$$v_j(t=0) = C\sum_{n=1}^{N} \omega_n B_n \sin q_n ja. \tag{9.12b}$$

To solve (9.12) for $A_n$ and $B_n$, we note that the normal mode solutions, $u_{j,n}$, are orthogonal, that is, they satisfy the condition

$$\sum_{j=1}^{N} u_{j,n}\, u_{j,m} \propto \delta_{n,m}. \tag{9.13}$$

The Kronecker $\delta$ symbol $\delta_{n,m} = 1$ if $n = m$ and is zero otherwise. It is convenient to normalize the $u_{j,n}$ so that they are orthonormal, that is,

$$\sum_{j=1}^{N} u_{j,n}\, u_{j,m} = \delta_{n,m}. \tag{9.14}$$

It is easy to show that the choice, $C = 1/\sqrt{(N+1)/2}$, in (9.4) and (9.10) insures that (9.14) is satisfied.

We now use the orthonormality condition (9.14) to determine the $A_n$ and $B_n$ coefficients. If we multiply both sides of (9.12) by $C \sin q_m ja$, sum over $j$, and use the orthogonality condition (9.14), we obtain

$$A_n = C \sum_{j=1}^{N} u_j(0) \sin q_n ja, \tag{9.15a}$$

$$B_n = C \sum_{j=1}^{N} (v_j(0)/\omega_n) \sin q_n ja. \tag{9.15b}$$

For example, if the initial displacement of every particle is zero, and the initial velocity of every particle is zero except for $v_1(0) = 1$, we find $A_n = 0$ for all $n$, and

$$B_n = \frac{C}{\omega_n} \sin q_n a. \tag{9.16}$$

The corresponding solution for $u_j(t)$ is

$$u_j(t) = \frac{2}{N+1} \sum_{n=1}^{N} \frac{1}{\omega_n} \cos \omega_n t \sin q_n a \sin q_n ja. \tag{9.17}$$

What is the solution if the particles start in a normal mode, that is, $u_j(t=0) \propto \sin q_2 ja$?

The `Oscillators` class in Listing 9.1 displays the analytic solution (9.11) of the oscillator displacements. The `draw` method uses a single circle that is repeatedly set equal to the appropriate world coordinates. The initial positions are calculated and stored in the `y` array in the `Oscillators` constructor. When an oscillator is drawn, the position array is multiplied by the given mode's sinusoidal (phase) factor to produce a time-dependent displacement.

Listing 9.1: The `Oscillators` class models the time evolution of a normal mode of a chain of coupled oscillators.

```
package org.opensourcephysics.sip.ch09;
import java.awt.Graphics;
import org.opensourcephysics.display.*;

public class Oscillators implements Drawable {
   OscillatorsMode normalMode;
   Circle circle = new Circle();
   double[] x; // drawing positions
   double[] u; // displacement
   double time = 0;

   public Oscillators(int mode, int N) {
      u = new double[N+2]; // includes the two ends of the chain
      x = new double[N+2]; // includes the two ends of the chain
      normalMode = new OscillatorsMode(mode, N);
      double xi = 0;
      for(int i = 0;i<N+2;i++) {
         x[i] = xi;
         u[i] = normalMode.evaluate(xi); // initial displacement
         xi++;                           // increment x[i] by lattice spacing of one
      }
   }

   public void step(double dt) {
      time += dt;
   }

   public void draw(DrawingPanel drawingPanel, Graphics g) {
      normalMode.draw(drawingPanel, g); // draw initial condition
      double phase = Math.cos(time*normalMode.omega);
      for(int i = 0, n = x.length;i<n;i++) {
         circle.setXY(x[i], u[i]*phase);
         circle.draw(drawingPanel, g);
      }
   }
}
```

The `OscillatorsMode` class in Listing 9.2 instantiates a normal mode. This class stores the mode frequency and implements the `Function` interface to evaluate the analytic solution. It draws a light gray outline of the initial analytic solution using a `FunctionDrawer`.

Listing 9.2: The `OscillatorsMode` class models a normal mode of a chain of coupled oscillators.

```
package org.opensourcephysics.sip.ch09;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class OscillatorsMode implements Drawable, Function {
   static final double OMEGA_SQUARED = 1; // equals k/m
   FunctionDrawer functionDrawer;         // draws the initial condition
   double omega;                          // oscillation frequency of mode
   double wavenumber;                     // wavenumber = 2*pi/wavelength
   double amplitude;

   OscillatorsMode(int mode, int N) {
      amplitude = Math.sqrt(2.0/(N+1));
      omega = 2*Math.sqrt(OMEGA_SQUARED)*Math.abs(Math.sin(mode*Math.PI/N/2));
      wavenumber = Math.PI*mode/(N+1);
      functionDrawer = new FunctionDrawer(this);
      functionDrawer.initialize(0, N+1, 300, false); // draws the initial displacement
      functionDrawer.color = Color.LIGHT_GRAY;
   }

   public double evaluate(double x) {
      return amplitude*Math.sin(x*wavenumber);
   }

   public void draw(DrawingPanel panel, Graphics g) {
      functionDrawer.draw(panel, g);
   }
}
```

The `OscillatorsApp` target class extends `AbstractSimulation`, creates an `Oscillators` object, and displays the particle displacements as transverse oscillations. The complete listing is available in the ch09 code package, but it is not given here because it is similar to other animations.

**Problem 9.1.** Normal modes

a. How many modes are there for a chain of 16 oscillators? Predict the initial positions of the oscillators for modes 1 and 14 and compare your prediction to the program's output.

b. Because a normal mode is a standing wave, it can be written as the sum of right and left traveling sinusoidal waves, $f_+ = A\sin(kx - \omega t)$ and $f_- = A\sin(kx + \omega t)$, respectively. Does the phase velocity $v = \omega/k$ depend on the mode (wavelength)? In other words, how does the speed depend on the wavelength (see (9.9))?

c. Determine the wavelength and frequency for modes with the largest and smallest wavelengths. Note that you can click-drag within the window to measure position. The time is displayed in the yellow message box. Compare your measured values with (9.9).

d. Are negative mode numbers acceptable? Give two different listings of mode numbers that contain a complete set of modes for $N = 16$.

e. Write a short stand-alone program to verify that the normal mode solutions (9.10) are orthonormal.

f. Modify your program to show the evolution of the superposition of two or more normal modes. Compare this evolution to that of a single mode.

As seen in Problem 9.1 the number of unique nontrivial solutions is equal to the number of oscillators. Modes with mode numbers $1, 2, 3, \ldots, N$ form a complete set and all other modes are indistinguishable from them. (The number of modes is related to the Nyquist frequency and is discussed further in Section 9.3.)

The analytical solution (9.11), together with the initial conditions, represent the complete solution of the displacement of the particles. We can use a computer to calculate the sum in (9.11) and plot the time dependence of the displacements $u_j(t)$. There are many interesting extensions that are amenable to an analytical solution. What is the effect of changing the boundary conditions? What happens if the spring constants are not all equal, but are chosen from a probability distribution? What happens if we vary the masses of the particles? For these cases we can follow a similar approach and look for the eigenvalues $\omega_n$ and eigenvectors $u_{j,n}$ of the matrix equation

$$\mathbf{T}\,\mathbf{u} = \omega^2\,\mathbf{u}. \tag{9.18}$$

The matrix elements $T_{i,j}$ are zero except for

$$T_{i,i} = \frac{1}{m_i}[k_{i,i+1} + k_{i,i-1}] \tag{9.19a}$$

$$T_{i,i+1} = -\frac{k_{i,i+1}}{m_i} \tag{9.19b}$$

$$T_{i,i-1} = -\frac{k_{i,i-1}}{m_i}, \tag{9.19c}$$

where $k_{i,j}$ is the spring constant between particles $i$ and $j$. The solution of matrix equations such as (9.18) is a well studied problem in linear programming, and an open source library such as LINPAC available from NetLIB (<www.netlib.org>) or a stand-alone program such as Octave (<www.octave.org>) can be used to obtain the solutions.

## 9.2 Numerical Solutions

Because we also are interested in the effects of nonlinear forces between the particles, for which the matrix approach is inapplicable, we study the numerical solution of the equations of motion (9.1) directly.

To use the ODE interface, we need to remember that the ordering of the variables in the coupled oscillator state array is important because the implementations of some ODE solvers, such as Verlet and Euler-Richardson, make explicit assumptions about the ordering. Our standard ordering is to follow a variable by its derivative. For example, the state vector of an $N$ oscillator chain is ordered as $\{u_0, v_0, u_1, v_1, \ldots, u_N, v_N, u_{N+1}, v_{N+1}, t\}$. Note that the state array includes variables for the chain's end points although the velocity rate corresponding to the end points is always zero. We include the time as the last variable because we will sometimes model time-dependent external forces. With this ordering, the `getRate` method is implemented as follows:

```java
static final double OMEGA_SQUARED = 1;       // equals k/m
public void getRate(double[] state, double[] rate) {
   for(int i = 1, N = x.length-1; i<N; i++) {   // skip ends
      rate[2*i] = state[2*i+1]; // displacement rate
      rate[2*i+1] = -OMEGA_SQUARED*(2*state[2*i]-state[2*i-2]-state[2*i+2]);
   }
   rate[state.length-1] = 1;
}
```

**Problem 9.2.** Numerical solution

a. Modify the `Oscillators` class to solve the dynamical equations of motion by implementing the `ODE` interface. Compare the numerical and the analytical solution for $N = 10$ using an algorithm that is well suited to oscillatory problems.

b. What is the maximum deviation between the analytical and numerical solution of $u_j(t)$? How well is the total energy conserved in the numerical solution? How does the maximum deviation and the conservation of the total energy change when the time step $\Delta t$ is reduced? Justify your choice of numerical algorithm.

**Problem 9.3.** Dynamics of coupled oscillators

a. Use your program for Problem 9.2 for $N = 2$ using units such that the ratio $k/m = 1$. Choose the initial values of $u_1$ and $u_2$ so that the system is in one of its two normal modes, for example, $u_1 = u_2 = 0.5$ and set the initial velocities equal to zero. Describe the displacement of the particles. Is the motion of each particle periodic in time? To answer this question, add code that plots the displacement of each particle versus the time. Then consider the other normal mode, for example, $u_1 = 0.5$, $u_2 = -0.5$. What is the period in this case? Does the system remain in a normal mode indefinitely? Finally, choose the initial particle displacements equal to random values between $-0.5$ and $+0.5$. Is the motion of each particle periodic in this case?

b. Consider the same questions as in part (a), but with $N = 4$ and $N = 10$. Consider the $n = 2$ mode for $N = 4$ and the $n = 3$ and $n = 8$ modes for $N = 10$. (See (9.10) for the form of the normal mode solutions.) Also consider random initial displacements.

**Problem 9.4.** Different boundary conditions

a. Modify your program from Problem 9.3 so that periodic boundary conditions are used, that is, $u_0 = u_N$ and $u_1 = u_{N+1}$. Choose $N = 10$, and the initial condition corresponding to the normal mode (9.10) with $n = 2$. Does this initial condition yield a normal mode solution for periodic boundary conditions? (It might be easier to answer this question by plotting $u_i$ versus time for two or more particles.) For fixed boundary conditions there are $N + 1$ springs, but for periodic boundary conditions there are $N$ springs. Why? Choose the initial condition corresponding to the $n = 2$ normal mode, but replace $N+1$ by $N$ in (9.7). Does this initial condition correspond to a normal mode? Now try $n = 3$, and other values of $n$. Which values of $n$ give normal modes? Only sine functions can be normal modes for fixed boundary conditions (see (9.4)). Can there be normal modes with cosine functions if we use periodic boundary conditions?

b. Modify your program so that free boundary conditions are used, which means that the masses at the end points are connected to only one nearest neighbor. A simple way to implement this boundary condition is to set $u_0 = u_1$ and $u_N = u_{N+1}$. Choose $N = 10$, and use the initial condition corresponding to the $n = 3$ normal mode found using fixed boundary conditions. Does this condition correspond to a normal mode for free boundary conditions? Is $n = 2$ a normal mode for free boundary conditions? Are the normal modes purely sinusoidal?

c. Choose free boundary conditions and $N \geq 10$. Let the initial condition be a pulse of the form, $u_1 = 0.2, u_2 = 0.6, u_3 = 1.0, u_4 = 0.6, u_5 = 0.2$, and all other $u_j = 0$. After the pulse reaches the right end, what is the phase of the reflected pulse, that is, are the displacements in the reflected pulse in the same direction as the incoming pulse (a phase shift of zero degrees) or in the opposite direction (a phase shift of 180 degrees)? What happens for fixed boundary conditions? Choose $N$ to be as large as possible so that it is easy to distinguish the incident and reflected waves.

d. Choose $N \geq 20$ and let the spring constants on the right half of the system be four times greater than the spring constants on the left half. Use fixed boundary conditions. Set up a pulse on the left side. Is there a reflected pulse at the boundary between the two types of springs? If so, what is its relative phase? Compare the amplitude of the reflected and transmitted pulses. Consider the same questions with a pulse that is initially on the right side.

**Problem 9.5.** Motion of coupled oscillators with external forces

a. Modify your program from Problem 9.4 so that an external force $F_{\rm ext}$ is exerted on the first particle,
$$F_{\rm ext}/m = 0.5\cos\omega t, \tag{9.20}$$
where $\omega$ is the angular frequency of the external force. Let the initial displacements and velocities of all $N$ particles be zero. Choose $N = 3$ and consider the response of the system to an external force for $\omega = 0.5$ to 4.0 in steps of 0.5. Record $A(\omega)$, the maximum amplitude of any particle, for each value of $\omega$. Repeat the simulation for $N = 10$.

b. Choose $\omega$ to be one of the normal mode frequencies. Does the maximum amplitude remain constant or does it increase with time? How can you use the response of the system to an external force to determine the normal mode frequencies? Discuss your results in terms of the power input, $F_{\rm ext}v_1$?

c. In addition to the external force exerted on the first particle, add a damping force equal to $-\gamma v_i$ to all the oscillators. Choose the damping constant $\gamma = 0.05$. How do you expect the system to behave? How does the maximum amplitude depend on $\omega$? Are the normal mode frequencies changed when $\gamma \neq 0$?

In Problem 9.5 we saw that a boundary produces reflections that lead to resonances. Although it is not possible to eliminate all reflections, it can be shown analytically that it is possible to absorb waves at a single frequency $\omega$ by using a free boundary and a judicious choice of the mass and damping coefficient for the last oscillator (see the text by Main). These conditions are

$$m_{N-1} = \frac{m}{2} \tag{9.21a}$$
$$\gamma_{N-1} = \frac{k}{\omega}\sin qa, \tag{9.21b}$$

where $a$ is the separation between oscillators, $q$ is the wavenumber, and $\omega$ is the angular frequency. Problem 9.6 shows that this choice leads to a transparent boundary (at the selected frequency) and is akin to impedance matching in electrical transmission lines and enables us to study traveling waves. In Section 9.7 we will study the classical wave equation in detail.

**Problem 9.6.** Traveling waves

a. Modify the oscillator program by imposing a sinusoidal amplitude on the first oscillator

$$u_1(t) = 0.5 \sin \omega t. \tag{9.22}$$

Run the program with $\omega = 1.0$ and observe the right-traveling wave, but note that reflections soon produce a left traveling wave.

b. Implement transparent boundary conditions on the right-hand side. Is the first wave crest totally transmitted at the end or is there some residual reflection? Explain. Add a small overall damping to the chain to remove transient effects.

c. The dispersion relation (9.9) predicts a cutoff frequency

$$\omega_c^2 = 4\frac{k}{m}. \tag{9.23}$$

What happens if we apply an external driving force above this frequency?

**Problem 9.7.** Evanescent waves

Increase $\omega$ past the cutoff frequency in the traveling wave simulation that was found in Problem 9.6. Do you still observe waves? Is energy being transported along the chain?

Waves above the cutoff frequency are known as *evanescent waves*. In Problem 9.8 we show how these waves lead to a classical counterpart of quantum mechanical tunneling.

**Problem 9.8.** Tunneling

a. Model a traveling wave on a $N = 64$ particle chain with mass $m = 1$ and $k = 1$, but assign $m = 4$ to eight oscillators near the center. Drive the first particle in the chain with a frequency of 0.113. (This value is slightly higher than the frequency above which the wave amplitude falls off exponentially.) Describe the steady state motion in the left region, the central region of heavier masses, and the right region.

b. Lower the frequency in part (a) until you observe maximum transmission through the barrier. Describe the steady state motion in the left region, the central barrier, and the right region. Explain how this system can be used as a frequency filter.

## 9.3 Fourier Series

In Section 9.1, we showed that the displacement of a single particle can be written as a linear combination of normal modes, that is, a linear superposition of sinusoidal terms. In general, an arbitrary periodic function $f(t)$ of period $T$ can be expressed as a sum of sines and cosines:

$$f(t) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty} \big(a_k \cos \omega_k t + b_k \sin \omega_k t\big), \tag{9.24}$$

where

$$\omega_k = k\omega_0 \text{ and } \omega_0 = \frac{2\pi}{T}. \tag{9.25}$$

The quantity $\omega_0$ is the fundamental frequency. Such a sum is called a Fourier series. The sine and cosine terms in (9.24) for $k = 2, 3, \ldots$ represent the second, third, ..., and higher order harmonics. The Fourier coefficients $a_k$ and $b_k$ are given by

$$a_k = \frac{2}{T}\int_0^T f(t)\cos\omega_k t\,dt \tag{9.26a}$$

$$b_k = \frac{2}{T}\int_0^T f(t)\sin\omega_k t\,dt. \tag{9.26b}$$

The constant term $\frac{1}{2}a_0$ in (9.24) is the average value of $f(t)$. The expressions in (9.26) for the coefficients follow from the orthogonality conditions:

$$\frac{2}{T}\int_0^T \sin\omega_k t\sin\omega_{k'} t\,dt = \delta_{k,k'} \tag{9.27a}$$

$$\frac{2}{T}\int_0^T \cos\omega_k t\cos\omega_{k'} t\,dt = \delta_{k,k'}. \tag{9.27b}$$

$$\frac{2}{T}\int_0^T \sin\omega_k t\cos\omega_{k'} t\,dt = 0. \tag{9.27c}$$

In general, an infinite number of terms is needed to represent an arbitrary periodic function exactly. In practice, a good approximation usually can be obtained by including a relatively small number of terms. Unlike a power series, which can approximate a function only near a particular point, a Fourier series can approximate a function at almost every point. The `Synthesize` class in Listing 9.3 evaluates such a series given the Fourier coefficients $a$ and $b$.

Listing 9.3: A class that synthesizes a function using a Fourier series.

```java
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.numerics.Function;

public class Synthesize implements Function {
   double[] cosCoefficients, sinCoefficients; // cosine and sine coefficients
   double a0;                                  // the constant term
   double omega0;

   public Synthesize(double period, double a0, double[] cosCoef, double[] sinCoef) {
      omega0 = Math.PI*2/period;
      cosCoefficients = cosCoef;
      sinCoefficients = sinCoef;
      this.a0 = a0;
   }

   public double evaluate(double x) {
      double f = a0/2;
      // sum the cosine terms
      for(int i = 0, n = cosCoefficients.length;i<n;i++) {
         f += cosCoefficients[i]*Math.cos(omega0*x*(i+1));
      }
      // sum the sine terms
      for(int i = 0, n = sinCoefficients.length;i<n;i++) {
         f += sinCoefficients[i]*Math.sin(omega0*x*(i+1));
      }
      return f;
   }
}
```

The `SynthesizeApp` class creates a `Synthesize` object by defining the values of the nonzero Fourier coefficients and draws the result of the Fourier series. Because the `Synthesize` class implements the `Function` interface, we can plot the Fourier series and see how the sum can represent an arbitrary periodic function. An easy way to do so is to create a `FunctionDrawer` and add it to a drawing frame as shown in Listing 9.4. The `FunctionDrawer` handles the routine task of generating a curve from the given function to produce a drawing.

Listing 9.4: A program that displays a Fourier series.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.FunctionDrawer;
import org.opensourcephysics.frames.DisplayFrame;

public class SynthesizeApp extends AbstractCalculation {
   DisplayFrame frame = new DisplayFrame("x", "f(x)", "Fourier Synthesis");

   public void calculate() {
      double xmin = control.getDouble("xmin");
      double xmax = control.getDouble("xmax");
      int N = control.getInt("N");
      double period = control.getDouble("period");
      double[] sinCoefficients = (double[]) control.getObject("sin coefficients");
      double[] cosCoefficients = (double[]) control.getObject("cos coefficients");
      FunctionDrawer functionDrawer = new FunctionDrawer(new Synthesize(period, 0, cosCoeffic
                                       sinCoefficients));
      functionDrawer.initialize(xmin, xmax, N, false);
      frame.clearDrawables();              // remove old function drawer
      frame.addDrawable(functionDrawer); // add new function drawer
   }

   public void reset() {
      control.setValue("xmin", -1);
      control.setValue("xmax", 1);
      control.setValue("N", 300);
      control.setValue("period", 1);
      control.setValue("sin coefficients", new double[] {
         1.0, 0, 1.0/3.0, 0, 1.0/5.0, 0, 0
      });
      control.setValue("cos coefficients", new double[] {
         0, 0, 0, 0, 0, 0, 0
      });
      calculate();
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new SynthesizeApp());
   }
}
```

**Problem 9.9.** Fourier synthesis

a. The process of approximating a function by adding together a fundamental frequency and harmonics of various amplitudes is *Fourier synthesis*. The `SynthesizeApp` class shows how a sum of harmonic functions can represent an arbitrary periodic function. Consider the series

$$f(t) = \frac{2}{\pi}(\sin t + \frac{1}{3}\sin 3t + \frac{1}{5}\sin 5t + \cdots). \tag{9.28}$$

Describe the nature of the plot of $f(t)$ when only the first three terms in (9.28) are retained. Increase the number of terms until you are satisfied that (9.28) represents the desired function sufficiently accurately. What function is represented by the infinite series?

b. Modify `SynthesizeApp` so that you can create an initial state with an arbitrary number of terms. (Do not use the control to input each coefficient. Input only the total number of terms.) Consider the series (9.28) with at least 32 terms. For what values of $t$ does the finite sum most faithfully represent the exact function? For what values of $t$ does it not? Why is it necessary to include a large number of terms to represent $f(t)$ where it has sharp edges? The small oscillations that increase in amplitude as a sharp edge is approached are known as the *Gibbs phenomenon*.

c. Modify `SynthesizeApp` and determine the function that is represented by the Fourier series with coefficients $a_k = 0$ and $b_k = (2/k\pi)(-1)^{k-1}$ for $k = 1, 2, 3, \ldots$ Approximately how many terms in the series are required?

So far we have considered how a sum of sines and cosines can approximate a known periodic function. More typically, we generate a time series consisting of $N$ data points, $f(t_i)$, where $t_i = 0, \Delta, 2\Delta, \ldots, (N-1)\Delta$. The Fourier series approximation to this data assumes that the data repeats itself with a period $T$ given by $T = N\Delta$. Our goal is to determine the Fourier coefficients $a_k$ and $b_k$. We will see that these coefficients contain important physical information.

Because we know only a finite number of data points $f_i \equiv f(t_i)$, it is possible to find only a finite set of Fourier coefficients. For a given value of $\Delta$, what is the largest frequency component we can extract? In the following, we give a plausibility argument that suggests that the maximum frequency we can analyze is given by

$$\omega_Q \equiv \frac{\pi}{\Delta} \quad \text{or} \quad f_Q \equiv \frac{1}{2\Delta}, \qquad \text{(Nyquist frequency)} \tag{9.29}$$

where $f_Q$ is the Nyquist frequency.

One way to understand (9.29) is to analyze a sine wave, $\sin\omega t$. If we choose $\Delta = \pi/\omega$, that is, sample at the Nyquist frequency, we would sample two points per cycle. If we sample at or below this frequency, we sample during the sine's positive and negative displacement. If we sample above this frequency, we would sample during the positive displacement of one crest and the positive displacement of another crest, thereby completely missing the negative displacement. In Problem 9.10, we explore sampling at frequencies higher than the Nyquist frequency.

**Problem 9.10.** Nyquist demonstration

Run `OscillatorsApp` with 16 particles in normal mode 27. What is the nature of the motion that you observe? Because the particles sample the analytic function $f(x)$ which sets the initial position at intervals $\Delta$ that are larger than $1/(2\lambda)$, the particle positions cannot accurately represent the high spatial frequencies. What mode number corresponds to sampling at the Nyquist frequency?

The Nyquist frequency is very important in signal processing because the sampling theorem due to Nyquist and Shannon states that a continuous function is completely determined by $N$ sampling points if the signal has passed though a filter with a cutoff frequency less than $f_Q$. This theorem is easy to understand. The filter will take out all the high frequency modes, and allow only $N$ modes of the signal to pass through. With $N$ sampling points we can solve for the amplitudes of each mode using (9.35) (also see Section 9.6).

One consequence of (9.29) is that there are $\omega_Q/\omega_0+1$ independent coefficients for $a_k$ (including $a_0$), and $\omega_Q/\omega_0$ independent coefficients for $b_k$, a total of $N+1$ independent coefficients. (Recall that $\omega_Q/\omega_0 = N/2$, where $\omega_0 = 2\pi/T$ and $T = N\Delta$.) Because $\sin\omega_Q t = 0$ for all values of $t$ that are multiples of $\Delta$, we have that $b_{N/2} = 0$ from (9.26b). Consequently, there are $N/2-1$ values for $b_k$, and hence a total of $N$ Fourier coefficients that can be computed. This conclusion is reasonable because the number of meaningful Fourier coefficients should be the same as the number of data points.

The `Analyze` class computes the Fourier coefficients $a_k$ and $b_k$ of a function $f(t)$ defined between $t = 0$ and $t = T$ at intervals of $\Delta$. To compute the coefficients we do the integrals in (9.26) numerically using the simple rectangular approximation (see Section 12.1):

$$a_k \approx \frac{2\Delta}{T}\sum_{i=0}^{N-1} f(t_i)\cos\omega_k t_i \tag{9.30a}$$

$$b_k \approx \frac{2\Delta}{T}\sum_{i=0}^{N-1} f(t_i)\sin\omega_k t_i, \tag{9.30b}$$

where the ratio $2\Delta/T = 2/N$.

Listing 9.5: The `Analyze` class calculates the Fourier coefficients of a function.

```java
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.numerics.Function;

public class Analyze {
   Function f;
   double period, delta;
   double omega0;
   int N;
   Analyze(Function f, int N, double delta) {
      this.f = f;
      this.delta = delta;
      this.N = N;
      period = N*delta;
      omega0 = 2*Math.PI/period;
   }

   double getSineCoefficient(int n) {
      double sum = 0;
      double t = 0;
      for(int i = 0;i<N;i++) {
         sum += f.evaluate(t)*Math.sin(n*omega0*t);
         t += delta;
      }
      return 2*delta*sum/period;
   }

   double getCosineCoefficient(int n) {
      double sum = 0;
      double t = 0;
      for(int i = 0;i<N;i++) {
         sum += f.evaluate(t)*Math.cos(n*omega0*t);
         t += delta;
      }
      return 2*delta*sum/period;
   }
}
```

The `AnalyzeApp` program reads a function and creates an `Analyze` object to plot the Fourier coefficients $a_k$ and $b_k$ versus $k$. Note how we use a *parser* to convert a string of characters into a function using a `ParsedFunction` object. Because typing mistakes are common when entering mathematical expressions, we return from the `calculate` method if a syntax error is detected.

Listing 9.6: A program that analyzes the Fourier components of a function.

```
package org.opensourcephysics.sip.ch09;
import java.awt.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class AnalyzeApp extends AbstractCalculation {
   PlotFrame frame = new PlotFrame("frequency", "coefficients", "Fourier analysis");

   public AnalyzeApp() {
      frame.setMarkerShape(0, Dataset.POST);
      frame.setMarkerColor(0, new Color(255, 0, 0, 128)); // semitransparent red
      frame.setMarkerShape(1, Dataset.POST);
      frame.setMarkerColor(1, new Color(0, 0, 255, 128)); // semitransparent blue
      frame.setXYColumnNames(0, "frequency", "cos");
      frame.setXYColumnNames(1, "frequency", "sin");
   }

   public void calculate() {
      double delta = control.getDouble("delta");
      int N = control.getInt("N");
      int numberOfCoefficients = control.getInt("number of coefficients");
      String fStr = control.getString("f(t)");
      Function f = null;
      try {
         f = new ParsedFunction(fStr, "t");
      } catch(ParserException ex) {
         control.println("Error parsing function string: "+fStr);
         return;
      }
      Analyze analyze = new Analyze(f, N, delta);
      double f0 = 1.0/(N*delta);
      for(int i = 0;i<=numberOfCoefficients;i++) {
         frame.append(0, i*f0, analyze.getCosineCoefficient(i));
         frame.append(1, i*f0, analyze.getSineCoefficient(i));
      }
      // Data tables can be displayed  by the user using
      // the tools menu but this statemment does so explicitly.
      frame.showDataTable(true);
   }

   public void reset() {
      control.setValue("f(t)", "sin(pi*t/10)");
      control.setValue("delta", 0.1);
      control.setValue("N", 200);
      control.setValue("number of coefficients", 10);
      calculate();
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new AnalyzeApp());
   }
}
```

In Problem 9.11 we compute the Fourier coefficients for several functions. We will see that if $f(t)$ is a sum of sinusoidal functions with different periods, it is essential that the period $T = N\Delta$ in the Fourier analysis program be an integer multiple of the periods of all the functions in the sum. If $T$ does not satisfy this condition, then the results for some of the Fourier coefficients will be spurious. In practice, the solution to this problem is to vary the sampling interval $\Delta$ and the total time over which the signal $f(t)$ is sampled. Fortunately, the results for the power spectrum (see Section 9.6) are less ambiguous than the values for the Fourier coefficients themselves.

**Problem 9.11.** Fourier analysis

a. Use the `AnalyzeApp` class with $f(t) = \sin \pi t/10$. Determine the first three nonzero Fourier coefficients by doing the integrals in (9.26) analytically before running the program. Choose the number of data points to be $N = 200$ and the sampling time $\Delta = 0.1$. Which Fourier components are nonzero? Repeat your analysis for $N = 400, \Delta = 0.1$; $N = 200, \Delta = 0.05$; $N = 205, \Delta = 0.1$; and $N = 500, \Delta = 0.1$, and other combinations of $N$ and $\Delta$. Explain your results by comparing the period of $f(t)$ with $N\Delta$, the assumed period. If the combination of $N$ and $\Delta$ are not chosen properly, do you find any spurious results for the coefficients?

b. Consider the functions $f_1(t) = \sin \pi t/10 + \sin \pi t/5$, $f_2(t) = \sin \pi t/10 + \cos \pi t/5$, and $f_3(t) = \sin \pi t/10 + \frac{1}{2} \cos \pi t/5$, and answer the same questions as in part (a) for each function. What combinations of $N$ and $\Delta$ give reasonable results for each function?

c. Consider a function that is not periodic, but goes to zero for $|t|$ large. For example, try $f(t) = t^4 e^{-t^2}$ and $f(t) = t^3 e^{-t^2}$. Interpret the difference between the Fourier coefficients of these two functions.

As shown in Appendix 9A, sine and cosine functions in a Fourier series can be combined into exponential functions with complex coefficients and complex exponents. We express $f(t)$ as

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \, e^{i\omega_k t}, \tag{9.31}$$

where

$$\omega_k = k\omega_0 \text{ and } \omega_0 = \frac{2\pi}{T}, \tag{9.32}$$

and use (9.24) to express the complex coefficients $c_k$ in terms of $a_k$ and $b_k$:

$$c_k = \frac{1}{2}(a_k - ib_k) \tag{9.33a}$$

$$c_0 = \frac{1}{2}a_0 \tag{9.33b}$$

$$c_{-k} = \frac{1}{2}(a_k + ib_k). \tag{9.33c}$$

The coefficients $c_k$ can be expressed in terms of $f(t)$ by using (9.33) and (9.26) and the fact that $e^{\pm i\omega_k t} = \cos \omega_k t \pm i \sin \omega_k t$. The result is

$$c_k = \frac{1}{T} \int_{T/2}^{T/2} f(t) \, e^{-i\omega_k t} \, dt. \tag{9.34}$$

As in (9.30), we can approximate the integral in (9.34) using the rectangular approximation. We write

$$g(\omega_k) \equiv c_k \frac{T}{\Delta} \approx \sum_{j=N/2}^{N/2} f(j\Delta) \, e^{-i\omega_k j\Delta} = \sum_{j=N/2}^{N/2} f(j\Delta) \, e^{-i2\pi kj/N}. \tag{9.35}$$

If we multiply (9.35) by $e^{i2\pi kj'/N}$, sum over $k$, and use the orthogonality condition

$$\sum_{k=N/2}^{N/2} e^{i2\pi kj/N} \, e^{-i2\pi kj'/N} = N\delta_{j,j'}, \tag{9.36}$$

we obtain the inverse Fourier transform

$$f(j\Delta) = \frac{1}{N}\sum_{k=N/2}^{N/2} g(\omega_k)\, e^{i2\pi kj/N} = \frac{1}{N}\sum_{k=N/2}^{N/2} g(\omega_k)\, e^{i\omega_k t_j}. \tag{9.37}$$

The frequencies $\omega_k$ for $k > N/2$ are greater than the Nyquist frequency $\omega_Q$. We can interpret the frequencies for $k > N/2$ as negative frequencies equal to $(k - N)\omega_0$ (see Problem 9.13). The occurrence of negative frequency components is a consequence of the use of the exponential functions rather than sines and cosines. Note that $f(t)$ is real if $g(-\omega_k) = g(\omega_k)$ because the $\sin\omega_k$ terms in (9.37) cancel due to symmetry.

The calculation of a single Fourier coefficient using (9.30) requires approximately $\mathcal{O}(N)$ multiplications. Because the complete Fourier transform contains $N$ complex coefficients, the calculation requires $\mathcal{O}(N^2)$ multiplications and may require hours to complete if the sample contains just a few megabytes of data. Because many of the calculations are redundant, it is possible to organize the calculation so that the computational time is order $N\log N$. Such an algorithm is called a *fast Fourier transform* (FFT) and is discussed in Appendix 9B. The improvement in speed is dramatic. A dataset containing $10^6$ points requires $\approx 6\times 10^6$ multiplications rather than $\approx 10^{12}$. Because we will use this algorithm to study diffraction and other phenomena and because coding this algorithm is nontrivial, we have provided an implementation of the FFT in the Open Source Physics numerics package. We can use this `FFT` class to transform between time and frequency or position and wavenumber. The `FFTApp` program shows how the `FFT` class is used.

Listing 9.7: The `FFTApp` program computes the fast Fourier transform of a function and displays the coefficients.

```java
package org.opensourcephysics.sip.ch09;
import java.text.DecimalFormat;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.numerics.FFT;

public class FFTApp extends AbstractCalculation {
   public void calculate() {
      DecimalFormat decimal = new DecimalFormat("0.0000"); // output format
      int N = 8;                                           // number of Fourier coefficients
      double[] z = new double[2*N];                        // array that will be transformed
      FFT fft = new FFT(N);                                // FFT implementation for N point
      int mode = control.getInt("mode");                   // mode or harmonic of e^(i*x)
      double
         x = 0, delta = 2*Math.PI/N;                       // signal will be sampled at f(x)
      for(int i = 0;i<N;i++) {
         z[2*i] = Math.cos(mode*x);    // real component of e^(i*mode*x)
         z[2*i+1] = Math.sin(mode*x);  // imaginary component of e^(i*mode*x)
         x += delta;                   // increase x
      }
      fft.transform(z); // transform data; data will be in wrap-around order
      for(int i = 0;i<N;i++) {
         System.out.println("index = "+i+"\t real = "+decimal.format(z[2*i])+"\t imag = "
                              +decimal.format(z[2*i+1]));
      }
   }

   public void reset() {
      control.setValue("mode", -1);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new FFTApp());
   }
}
```

The `FFT` class replaces the data in the input array with transformed values. (Make an array copy if you need to retain the original data.) Because the transformation assumes $N$ complex data points and Java does not support a primitive complex data type, the input array has length $2N$. The real part of the $n$th data point is located in array element $2n$ and the imaginary part is in element $2n+1$. The transformed data maintains this same ordering of real and imaginary parts. We test the `FFT` class in Problem 9.12.

**Problem 9.12.** The `FFT` class

a. The `FFTApp` initializes the array that will be transformed by evaluating the following complex function
$$f_n = f(n\Delta) = e^{in\Delta} = \cos n\Delta + i \sin n\Delta, \tag{9.38}$$
where $n$ is an integer and $\Delta = 2\pi/N$ is the interval between data points. We refer to $n$ as the `mode` variable in the program. What happens to the Fourier component if the phase of the complex exponential is shifted by $\alpha$? In other words, what happens if the data is initialized using $f_n = f(n\Delta) = e^{in\Delta+\alpha}$?

b. Modify and run `FFTApp` to show that the fast Fourier transform produces a single component only if the grid contains an integer number of wavelengths.

c. Change the number of grid points $N$ and show that the value of the nonzero Fourier coefficient is equal to $N$ if the input function is (9.38). Note that some other FFT implementations normalize the result by dividing by $N$.

d. Show that the original function can be recovered by invoking the `FFT.inverse` method.

**Problem 9.13.** Negative frequencies

a. Use (9.38) as the input function and show that `FFTApp` produces the same Fourier coefficients if $\omega = 2\pi/(N\Delta)$ or $\omega = -2\pi/(N\Delta)$. Repeat for $\omega = 4\pi/(N\Delta)$ and $\omega = -4\pi/(N\Delta)$.

b. Compute the Fourier coefficients using $f_n = f(n\Delta) = \cos n\Delta$, where $n = 0, 1, 2, \ldots, N-1$. Repeat using a sine function. Interpret your results using
$$\cos\theta = \frac{e^{i\theta} + e^{-i\theta}}{2} \tag{9.39a}$$
$$\sin\theta = \frac{e^{i\theta} - e^{-i\theta}}{2}. \tag{9.39b}$$

As shown in Problem 9.13, the Fourier coefficient indices for $n \geq N/2$ should be interpreted as negative frequencies. The transformed data still contains $N$ frequencies, and these frequencies are still separated by $2\pi/(N\Delta)$.

Because the frequencies switch sign at the array's midpoint index $N/2$, we refer to the transformed data as being in *wrap-around order*. The `toNaturalOrder` method can be used to sort and normalize the Fourier components in order of increasing frequency starting at $\omega_Q = -\pi/\Delta$ and continuing to $\omega_Q = \pi/\Delta((N-2)/N)$ if $N$ is even and continuing to $\omega_Q = \pi/\Delta$ if $N$ is odd.

**Exercise 9.14.** Natural order

Invoke the `toNaturalOrder` method after performing the FFT in the `FFTApp` program. Modify the print statement so that the natural frequency is shown and repeat Problem 9.13. If $N$ is even, the Fourier components have a frequency separation $\Delta\omega$ given by:

$$\Delta\omega = \frac{2\pi}{N\Delta}. \tag{9.40}$$

What is the frequency separation if $N$ is odd?

As we have seen, computing Fourier transformations is straightforward but requires a fair amount of bookkeeping. To simplify the process, we have defined the `FFTFrame` class in the frames package to perform a FFT and display the coefficients. This utility class accepts either data arrays or functions as input parameters in the `doFFT` method. The code shown in Listing 9.8 transforms an input array. We use the `FFTFrame` in Problem 9.12.

Listing 9.8: The `FFTCalculationApp` displays the coefficients of the function $e^{2\pi nx}$.

```java
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.FFTFrame;

public class FFTCalculationApp extends AbstractCalculation {
   FFTFrame frame = new FFTFrame("frequency", "amplitude", "FFT Frame Test");

   public void calculate() {
      double xmin = control.getDouble("xmin");
      double xmax = control.getDouble("xmax");
      int n = control.getInt("N");
      double
         xi = xmin, delta = (xmax-xmin)/n;
      double[] data = new double[2*n];
      int mode = control.getInt("mode");
      for(int i = 0;i<n;i++) {
         data[2*i] = Math.cos(mode*xi);
         data[2*i+1] = Math.sin(mode*xi);
         xi += delta;
      }
      frame.doFFT(data, xmin, xmax);
      frame.showDataTable(true);
   }
   public void reset() {
      control.setValue("mode", 1);
      control.setValue("xmin", 0);
      control.setValue("xmax", "2*pi");
      control.setValue("N", 32);
      calculate();
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new FFTCalculationApp());
   }
}
```

**Problem 9.15.** Spatial Fourier transforms and phase

So far we have considered only nonnegative values of $t$ for functions $f(t)$. Spatial Fourier transforms are of interest in many contexts and these transforms usually involve both positive and negative values of $x$.

a. Write a program using a `CalculationControl` that computes the real and imaginary parts of the Fourier transform $\phi(q)$ of a complex function $\psi(x) = f(x) + ig(x)$, where $f(x)$ and $g(x)$ are real and $x$ has both positive and negative values. Note that the wavenumber $q = 2\pi/L$ is analogous to the angular frequency $\omega = 2\pi/T$.

b. Compute the Fourier transform of the Gaussian function $\psi(x) = e^{-bx^2}$ in the interval $[-5, 5]$. Examine $\psi(x)$ and $\phi(q)$ for at least three values of $b$ such that the Gaussian is contained within the interval. Does $\phi(q)$ appear to be a Gaussian? Choose a reasonable criterion for the half-width of $\psi(x)$ and measure its value. Use the same criterion to measure the half-width of $\phi(q)$. How do these widths depend on $b$? How does the width of $\phi(q)$ change as the width of $\psi(x)$ increases?

c. Repeat part (b) with the function $\psi(x) = Ae^{-b(x-x_0)^2}$ for various values of $x_0$. What effect does shifting the peak have on $\phi(q)$?

d. Repeat part (b) with the function $\psi(x) = Ae^{-bx^2}e^{iq_0x}$ for various values of $q_0$. What effect does the phase oscillation have on $\phi(q)$?

## 9.4 Two-Dimensional Fourier Series

The extension of the ideas of Fourier analysis to two dimensions is simple and direct. We will use two-dimensional FFTs when we study diffraction in Section 9.9.

If we assume a function of two variables, $f(x, y)$, then a two-dimensional series is constructed using harmonics of both variables. The basis functions are the products of one dimensional basis functions $e^{ixq_x}e^{iyq_y}$, and the Fourier series is written as a sum of these harmonics:

$$f(x, y) = \sum_{n=-N/2}^{N/2} \sum_{m=-M/2}^{M/2} c_{n,m}\, e^{iq_n x} e^{iq_m y}, \tag{9.41}$$

where

$$q_n = \frac{2\pi n}{X} \quad \text{and} \quad q_m = \frac{2\pi m}{Y}. \tag{9.42}$$

The function $f(x, y)$ is assumed to be periodic in both $x$ and $y$ with periods $X$ and $Y$, respectively. The Fourier coefficients are again calculated by integrating the product of the function with a basis function

$$c_{n,m} = \int_{-X/2}^{X/2} \int_{-Y/2}^{Y/2} f(x, y) e^{i(q_n x + q_m y)} dx\, dy. \tag{9.43}$$

Because of the large number of coefficients $c_{n,m}$, the discrete two-dimensional Fourier transform is best implemented using the FFT algorithm. The `FFT2DCalculationApp` program shows how to compute a two-dimensional FFT using the `FFT2DFrame` utility class.

Listing 9.9: The `FFT2DCalculationApp` program computes the two-dimensional fast Fourier transform of a function and shows the resulting coefficients using a grid plot.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.FFT2DFrame;

public class FFT2DCalculationApp extends AbstractCalculation {
   FFT2DFrame frame = new FFT2DFrame("k_x", "k_y", "2D FFT");

   public void calculate() {
      int xMode = control.getInt("x mode"), yMode = control.getInt("y mode");
      double xmin = control.getDouble("xmin");
      double xmax = control.getDouble("xmax");
      int nx = control.getInt("Nx");
      double ymin = control.getDouble("ymin");
      double ymax = control.getDouble("ymax");
      int ny = control.getInt("Ny");
      double[] zdata = new double[2*nx*ny]; // data stored in row-major format
      double
         y = 0, yDelta = 2*Math.PI/ny;
      for(int iy = 0;iy<ny;iy++) { // loop over rows in array
         int offset = 2*iy*nx;      // offset to beginning of a row;  each row is nx long
         double
            x = 0, xDelta = 2*Math.PI/nx;
         for(int ix = 0;ix<nx;ix++) {
            // z function is e^(i*xmode*x)e^(i*ymode*y)
            zdata[offset+2*ix] =   //real part
               Math.cos(xMode*x)*Math.cos(yMode*y)-Math.sin(xMode*x)*Math.sin(yMode*y);
            zdata[offset+2*ix+1] = // imaginary part
               Math.sin(xMode*x)*Math.cos(yMode*y)+Math.cos(xMode*x)*Math.sin(yMode*y);
            x += xDelta;
         }
         y += yDelta;
      }
      frame.doFFT(zdata, nx, xmin, xmax, ymin, ymax);
   }
   public void reset() {
      control.setValue("x mode", 0);
      control.setValue("y mode", 1);
      control.setValue("xmin", 0);
      control.setValue("xmax", "2*pi");
      control.setValue("ymin", 0);
      control.setValue("ymax", "2*pi");
      control.setValue("Nx", 16);
      control.setValue("Ny", 16);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new FFT2DCalculationApp());
   }
}
```

The `FFT2DFrame` is based on FFT routines contributed to the GNU Scientific Library (GSL) by Brian Gough and adapted to Java by Bruce Miller at NIST. We initialize our data array to conform to the GSL API using a one-dimensional array such that rows follow sequentially. This ordering is known as row-major format. Because the input function is assumed to be complex, the array has dimension $2N_xN_y$, where $N_x$ and $N_y$ are the number of grid points in the $x$ and $y$ direction, respectively. The `FFT2DFrame` object transforms and displays the data when the `doFFT` method is invoked.

**Exercise 9.16.** Two-dimensional FFT

Write a program to transform a two-dimensional Gaussian using the `FFT2DFrame` class. Note that color is used to represent the complex phase. What happens if the Gaussian is not centered on the grid?

## 9.5 Fourier Integrals

Fourier analysis can be extended to approximate waveforms that do not repeat themselves by converting the Fourier sum over discrete frequency components to an integral. The *Fourier integral* transforms a continuous function of time $f(t)$ into a continuous function of frequency $g(\omega)$ as follows

$$g(\omega) = \int_{-\infty}^{\infty} f(t)e^{i\omega t}dt. \tag{9.44}$$

The inverse transformation reverses this process:

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} g(\omega)e^{i\omega t}d\omega. \tag{9.45}$$

Equations (9.44) and (9.45) are known as a Fourier transform pair.

Because we need to store functions such as $f(t)$ and $g(\omega)$ at a discrete number of points, Fourier integrals are usually approximated using Fourier series with a large number of terms and a

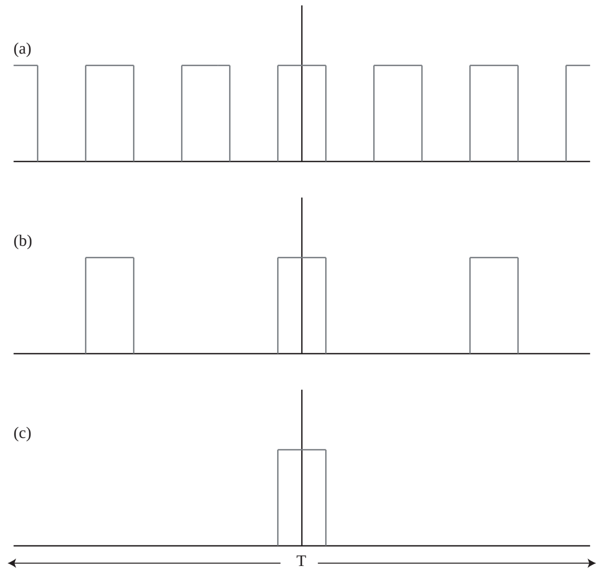

Figure 9.2: A series of pulses with increasing periods.

large sampling time. In other words, the time over which the signal is measured is large compared to the period of interest. Exercise 9.17 illustrates how a Fourier series for a pulse train approaches a continuous frequency spectrum as the period approaches infinity.

**Exercise 9.17.** Fourier Integral

Write a program to plot the frequency spectrum of the waveforms shown in Figure 9.2. Use the FFT algorithm. How does the frequency spectrum change as the waveform becomes less periodic? How does the finite time interval affect the result?

## 9.6 Power Spectrum

The power output of a periodic electrical signal $f(t)$ is proportional to the integral of the signal squared:

$$P = \frac{1}{T}\int_0^T |f(t)|^2 dt. \tag{9.46}$$

Another way to look at the power is to calculate the power $P$ associated with the Fourier components $\omega_k$ of a signal that has been sampled at regular intervals. If we substitute (9.31) into (9.46), rearrange terms, and apply the orthogonality condition (9.14), we obtain:

$$P = \sum_{j=-N/2}^{N/2} \sum_{k=-N/2}^{N/2} c_j^* c_k \frac{1}{T}\int_0^T e^{i(\omega_k - \omega_j)t} dt = \sum_{k=-N/2}^{N/2} |c_k|^2. \tag{9.47}$$

From (9.47) we conclude that the average power of $|f(t)|^2$ is the sum of the power in each frequency component $P(\omega_k) = |c_k|^2$. This result is one form of Parseval's theorem.

In most measurements the function $f(t)$ corresponds to an amplitude, and the power or intensity is proportional to the square of this amplitude, or for complex functions, the modulus squared. The *power spectrum* $P(\omega_k)$ is proportional to the power associated with a particular frequency component embedded in the quantity of interest. Well-defined peaks in $P(\omega_k)$ often correspond to normal mode frequencies.

What happens to the power associated with frequencies greater than the Nyquist frequency? To answer this question, consider two choices of the Nyquist frequency, $\omega_{Q_a}$ and $\omega_{Q_b} > \omega_{Q_a}$, and the corresponding sampling times, $\Delta_b < \Delta_a$. The calculation with $\Delta = \Delta_b$ is more accurate because the sampling time is smaller. Suppose that this calculation of the spectrum yields the result that $P(\omega > \omega_a) > 0$. What happens if we compute the power spectrum using $\Delta = \Delta_a$? The power associated with $\omega > \omega_a$ must be "folded" back into the $\omega < \omega_a$ frequency components. For example, the frequency component at $\omega + \omega_a$ is added to the true value at $\omega - \omega_a$ to produce an incorrect value at $\omega - \omega_a$ in the computed power spectrum. This phenomenon is called *aliasing* and leads to spurious results. Aliasing occurs in calculations of $P(\omega)$ if the latter does not vanish above the Nyquist frequency. To avoid aliasing, it is necessary to sample more frequently, or to remove the high frequency components from the signal before sampling the data.

Although the power spectrum can be computed by a simple modification of `AnalyzeApp`, it is a good idea to use the FFT for many of the following problems.

**Problem 9.18.** Aliasing

Sample the sinusoidal function, $\sin 2\pi t$, and display the resulting power spectrum using sampling frequencies above and below the Nyquist frequency. Start with a sampling time of $\Delta = 0.1$ and increase the time until $\Delta = 10.0$.

a. Is the power spectrum sharp? That is, is all the power located in a single frequency? Does your answer depend on the ratio of the period to the sampling time?

b. Explain the appearance of the power spectrum for $\Delta = 1.25$, $\Delta = 1.75$, and $\Delta = 2.5$.

c. What is the power spectrum if you sample at the Nyquist frequency or twice the Nyquist frequency?

**Problem 9.19.** Examples of power spectra

a. Create a data set with $N$ points corresponding to $f(t) = 0.3\cos(2\pi t/T) + r$, where $r$ is a uniform random number between 0 and 1 and $T = 4$. Plot $f(t)$ versus $t$ in time intervals of $\Delta = 4T/N$ for $N = 128$. Can you visually detect the periodicity? Compute the power spectrum using the same sampling interval $\Delta = 4T/N$. Does the frequency dependence of the power spectrum indicate that there are any special frequencies? Repeat with $T = 16$. Are high or low frequency signals easier to pick out from the random background?

b. Simulate a one-dimensional random walk, and compute the time series $x^2(t)$, where $x(t)$ is the distance from the origin of the walk after $t$ steps. Average $x^2(t)$ over several trials. Compute the power spectrum for a walk of $t \leq 256$. In this case $\Delta = 1$, the time between steps. Do you observe any special frequencies?

c. Let $f_n$ be the $n$th member of a random number sequence. As in part (b), $\Delta = 1$. Compute the power spectrum of the random number generator. Do you detect any periodicities? If so, is the random number generator acceptable?

**Problem 9.20.** Power spectrum of coupled oscillators

a. Modify your program developed in Problem 9.2 so that the power spectrum of the position of one of the $N$ particles is computed at the end of the simulation. Set $\Delta = 0.1$ so that the Nyquist frequency is $\omega_Q = \pi/\Delta \approx 31.4$. Choose the time of the simulation equal to $T = 25.6$ and let $k/m = 1$. Plot the power spectrum $P(\omega)$ at frequency intervals equal to $\Delta\omega = \omega_0 = 2\pi/T$. First choose $N = 2$ and choose the initial conditions so that the system is in a normal mode. What do you expect the power spectrum to look like? What do you find? Then choose $N = 10$ and choose initial conditions corresponding to various normal modes. Is the power spectrum the same for all particles?

b. Repeat part (a) for $N = 2$ and $N = 10$ with random initial particle displacements between $-0.5$ and $+0.5$ and zero initial velocities. Can you detect all the normal modes in the power spectrum? Repeat for a different set of random initial displacements.

c. Repeat part (a) for initial displacements corresponding to the equal sum of two normal modes. Does the power spectrum show two peaks? Are these peaks of equal height?

d. Recompute the power spectrum for $N = 10$ with $T = 6.4$. Is this time long enough? How can you tell?

**Problem 9.21.** Quasiperiodic power spectra

a. Write a program to compute the power spectrum of the circle map (6.62). Begin by exploring the power spectrum for $K = 0$. Plot $\ln P(\omega)$ versus $\omega$, where $P(\omega)$ is proportional to the modulus squared of the Fourier transform of $x_n$. Begin with 256 iterations. How does the power spectra differ for rational and irrational values of the parameter $\Omega$? How are the locations of the peaks in the power spectra related to the value of $\Omega$?

b. Set $K = 1/2$ and compute the power spectra for $0 < \Omega < 1$. Does the power spectra differ from the spectra found in part (a)?

c. Set $K = 1$ and compute the power spectra for $0 < \Omega < 1$. How does the power spectra compare to those found in parts (a) and (b)?

In Problem 9.20 we found that the peaks in the power spectrum yield information about the normal mode frequencies. In Problem 9.22 and 9.23 we compute the power spectra for a system of coupled oscillators with disorder. Disorder can be generated by having random masses or random spring constants (or both). We will see that one effect of disorder is that the normal modes are no longer simple sinusoidal functions. Instead, some of the modes are localized, meaning that only some of the particles move significantly while the others remain essentially at rest. This effect is known as *Anderson localization*. Typically, we find that modes above a certain frequency are *localized*, and those below this threshold frequency are *extended*. The threshold frequency is well defined for large systems. All states are localized in the limit of an infinite chain with any amount of disorder. The dependence of localization on disorder in systems of coupled oscillators in higher dimensions is more complicated.

**Problem 9.22.** Localization with a single defect

a. Modify your program developed in Problem 9.2 so that the mass of one oscillator is equal to one fourth that of the others. Set $N = 20$ and use fixed boundary conditions. Compute the power spectrum over a time $T = 51.2$ using random initial displacements between $-0.5$ and $+0.5$ and zero initial velocities. Sample the data at intervals of $\Delta = 0.1$. The normal mode frequencies correspond to the well-defined peaks in $P(\omega)$. Consider at least three different sets of random initial displacements to insure that you find all the normal mode frequencies.

b. Apply an external force $F_e = 0.3 \sin \omega t$ to each particle. (The steady state behavior occurs sooner if we apply an external force to each particle instead of just one particle.) Because the external force pumps energy into the system, it is necessary to add a damping force to prevent the oscillator displacements from becoming too large. Add a damping force equal to $-\gamma v_i$ to all the oscillators with $\gamma = 0.1$. Choose random initial displacements and zero initial velocities and use the frequencies found in part (a) as the driving frequencies $\omega$. Describe the motion of the particles. Is the system driven to a normal mode? Take a "snapshot" of the particle displacements after the system has run for a sufficiently long time so that the patterns repeat themselves. Are the particle displacements simple sinusoidal functions? Sketch the approximate normal mode patterns for each normal mode frequency. Which of the modes appear localized and which modes appear to be extended? What is the approximate cutoff frequency that separates the localized from the extended modes?

**Problem 9.23.** Localization in a disordered chain of oscillators

a. Modify your program so that the spring constants can be varied by the user. Set $N = 10$ and use fixed boundary conditions. Consider the following set of 11 spring constants: 0.704, 0.388, 0.707, 0.525, 0.754, 0.721, 0.006, 0.479, 0.470, 0.574, 0.904. To help you determine all the normal modes, we provide two of the normal mode frequencies: $\omega \approx 0.28$ and 1.15. Find the power spectrum using the procedure outlined in Problem 9.22a.

b. Apply an external force $F_e = 0.3 \sin \omega t$ to each particle, and find the normal modes as outlined in Problem 9.22b.

c. Repeat parts (a) and (b) for another set of random spring constants for $N = 40$. Discuss the nature of the localized modes in terms of the specific values of the spring constants. For example, is the edge of a localized mode at a spring that has a relatively large or small spring constant?

d. Repeat parts (a) and (b) for uniform spring constants, but random masses between 0.5 and 1.5. Is there a qualitative difference between the two types of disorder?

In 1955 Fermi, Pasta, and Ulam used the Maniac I computer at Los Alamos to study a chain of oscillators. Their surprising discovery might have been the first time a qualitatively new result, instead of a more precise number, was found from a simulation. To understand their results, we need to discuss an idea from statistical mechanics that was discussed in Project 8.23.

A fundamental assumption of statistical mechanics is that an isolated system of particles is quasi-ergodic, that is, the system will evolve through all configurations consistent with the conservation of energy. A system of linearly coupled oscillators is not quasi-ergodic, because if the system is initially in a normal mode, it stays in that normal mode forever. Before 1955 it was believed that if the interaction between the particles is weakly nonlinear (and the number of particles is sufficiently large), the system would be quasi-ergodic and evolve through the different normal modes of the linear system. In Problem 9.24 we will find, as did Fermi, Pasta, and Ulam, that the behavior of the system is much more complicated. The question of ergodicity in this system is known as the FPU problem.

**Problem 9.24.** Nonlinear oscillators

a. Modify your program so that cubic forces between the particles are added to the linear spring forces. That is, let the force on particle $i$ due to particle $j$ be

$$F_{ij} = -(u_i - u_j) - \alpha(u_i - u_j)^3, \tag{9.48}$$

where $\alpha$ is the amplitude of the nonlinear term. Choose the masses of the particles to be unity. Consider $N = 10$ and choose initial displacements corresponding to a normal mode of the linear ($\alpha = 0$) system. Compute the power spectrum over a time $T = 51.2$ with $\Delta = 0.1$ for $\alpha = 0$, 0.1, 0.2, and 0.3. For what value of $\alpha$ does the system become ergodic, that is, for what value of $\alpha$ are the heights of all the normal mode peaks approximately the same?

b. Repeat part (a) for the case where the displacements of the particles are initially random. Use the same set of random displacements for each value of $\alpha$.

c.* We now know that the number of oscillators is not as important as the magnitude of the nonlinear interaction. Repeat parts (a) and (b) for $N = 20$ and 40 and discuss the effect of increasing the number of particles.

## 9.7 Wave Motion

Our simulations of coupled oscillators have shown that the microscopic motion of the individual oscillators leads to macroscopic wave phenomena. To understand the transition between microscopic and macroscopic phenomena, we reconsider the oscillations of a linear chain of $N$ particles with equal spring constants $k$ and equal masses $m$. As we found in Section 9.1, the equations of motion of the particles can be written as (see (9.1))

$$\frac{d^2u_j(t)}{dt^2} = -\frac{k}{m}\big[2u_j(t) - u_{j+1}(t) - u_{j-1}(t)\big]. \qquad (j = 1, \ldots, N) \tag{9.49}$$

We consider the limits $N \to \infty$ and $a \to 0$ with the length of the chain $Na$ fixed. We will find that the discrete equations of motion (9.49) can be replaced by the continuous *wave equation*

$$\frac{\partial^2 u(x,t)}{\partial t^2} = c^2 \frac{\partial^2 u(x,t)}{\partial x^2}, \tag{9.50}$$

where $c$ has the dimension of velocity.

We obtain the wave equation (9.50) as follows. First we replace $u_j(t)$, where $j$ is a discrete variable, by the function $u(x,t)$, where $x$ is a continuous variable, and rewrite (9.49) in the form

$$\frac{\partial^2 u(x,t)}{\partial t^2} = \frac{ka^2}{m}\frac{1}{a^2}\big[u(x+a,t) - 2u(x,t) + u(x-a,t)\big]. \tag{9.51}$$

We have written the time derivative as a partial derivative because the function $u$ depends on two variables. If we use the Taylor series expansion

$$u(x \pm a) = u(x) \pm a\frac{du}{dx} + \frac{a^2}{2}\frac{d^2u}{dx^2} + \ldots, \tag{9.52}$$

it is easy to show that as $a \to 0$, the quantity

$$\frac{1}{a^2}\big[u(x+a,t) - 2u(x,t) + u(x-a,t)\big] \to \frac{\partial^2 u(x,t)}{\partial x^2}. \tag{9.53}$$

(We have written a spatial derivative as a partial derivative for the same reason as before.) The wave equation (9.50) is obtained by substituting (9.53) into (9.51) with $c^2 = ka^2/m$. If we introduce the linear mass density $\mu = M/a$ and the tension $T = ka$, we can express $c$ in terms of $\mu$ and $T$ and obtain the familiar result $c^2 = T/\mu$.

It is straightforward to show that any function of the form $f(x \pm ct)$ is a solution to (9.50). Among these many solutions to the wave equation are the familiar forms:

$$u(x,t) = A\cos\frac{2\pi}{\lambda}(x \pm ct) \tag{9.54a}$$

$$u(x,t) = A\sin\frac{2\pi}{\lambda}(x \pm ct). \tag{9.54b}$$

Because the wave equation is linear and hence satisfies a superposition principle, we can understand the behavior of a wave of arbitrary shape by representing its shape as a sum of sinusoidal waves.

One way to solve the wave equation (9.50) numerically is to retrace our steps back to the discrete equations (9.49) to find a discrete form of the wave equation that is convenient for numerical calculations. The conversion of a continuum equation to a physically motivated discrete form frequently leads to useful numerical algorithms. From (9.53) we see how to approximate the second derivative by a finite difference. If we replace $a$ by $\Delta x$ and take $\Delta t$ to be the time step, we can rewrite (9.49) by

$$\frac{1}{(\Delta t)^2}[u(x,t+\Delta t) - 2u(x,t) + u(x,t-\Delta t)] = \\ \frac{c^2}{(\Delta x)^2}[u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)]. \tag{9.55}$$

The quantity $\Delta x$ is the spatial interval. The result of solving (9.55) for $u(x, t + \Delta t)$ is

$$\begin{aligned} u(x, t + \Delta t) &= 2(1 - b)u(x, t) \\ &\quad + b[u(x + \Delta x, t) + u(x + \Delta x, t)] - u(x, t - \Delta t), \end{aligned} \tag{9.56}$$

where $b = (c\Delta t/\Delta x)^2$. Equation (9.56) expresses the displacements at time $t + \Delta t$ in terms of the displacements at the current time $t$ and at the previous time $t - \Delta t$.

**Problem 9.25.** Solution of the discrete wave equation

a. Write a program to compute the numerical solutions of the discrete wave equation (9.56). Three spatial arrays corresponding to $u(x)$ at times $t + \Delta t$, $t$, and $t - \Delta t$ are needed. Denote the displacement $u(j\Delta x)$ by the array element `u[j]` where $j = 0, \ldots, N + 1$. Use periodic boundary conditions so that $u_0 = u_N$ and $u_1 = u_{N+1}$. Draw lines between the displacements at neighboring values of $x$. Note that the initial conditions require the specification of `u` at $t = 0$ and at $t = -\Delta t$. Let the waveform at $t = 0$ and $t = -\Delta t$ be $u(x, t = 0) = \exp(-(x - 10)^2)$ and $u(x, t = -\Delta t) = \exp(-(x - 10 + c\Delta t)^2)$, respectively. What is the direction of motion implied by these initial conditions?

b. Our first task is to determine the optimum value of the parameter $b$. Let $\Delta x = 1$ and $N \geq 100$, and try the following combinations of $c$ and $\Delta t$: $c = 1, \Delta t = 0.1$; $c = 1, \Delta t = 0.5$; $c = 1, \Delta t = 1$; $c = 1, \Delta t = 1.5$; $c = 2, \Delta t = 0.5$; and $c = 2, \Delta t = 1$. Verify that the value $b = (c\Delta t)^2 = 1$ leads to the best results, that is, for this value of $b$, the initial form of the wave is preserved.

c. It is possible to show that the discrete form of the wave equation with $b = 1$ is exact up to numerical roundoff error (cf. DeVries). Hence, we can replace (9.56) by the simpler algorithm

$$u(x, t + \Delta t) = u(x + \Delta x, t) + u(x - \Delta x, t) - u(x, t - \Delta t). \tag{9.57}$$

That is, the solutions of (9.57) are equivalent to the solutions of the original partial differential equation (9.50). Try several different initial waveforms, and show that if the displacements have the form $f(x \pm ct)$, the waveform maintains its shape with time. For the remaining problems we will use (9.57) corresponding to $b = 1$. Unless otherwise specified, choose $c = 1$, $\Delta x = \Delta t = 1$, and $N \geq 100$ in the following problems.

**Problem 9.26.** Velocity of waves

a. Use the waveform given in Problem 9.25a and verify that the speed of the wave is unity by determining the distance traveled in a given amount of time. Because we have set $\Delta x = \Delta t = 1$ and $b = 1$, the speed $c = 1$. (A way of incorporating different values of $c$ is discussed in Problem 9.27c.)

b. Replace the waveform considered in part (a) by a sinusoidal wave that fits exactly, that is, choose $u(x, t) = \sin(qx - \omega t)$ such that $\sin q(N + 1) = 0$. Measure the period $T$ of the wave by measuring the time it takes for successive maxima to pass a given point. What is the wavelength $\lambda$ of the wave? Does it depends on the value of $q$? The frequency of the wave is given by $f = 1/T$. Verify that $\lambda f = c$.

**Problem 9.27.** Reflection of waves

a. Consider a wave of the form $u(x,t) = e^{-(x-10-ct)^2}$. Use fixed boundary conditions so that $u_0 = u_{N+1} = 0$. What happens to the reflected wave?

b. Modify your program so that free boundary conditions are incorporated: $u_0 = u_1$ and $u_N = u_{N+1}$. Compare the phase of the reflected wave to your result from part (a).

c. What happens to a pulse at the boundary between two media? Set $c = 1$ and $\Delta t = 1$ on the left side of your grid and $c = 2$ and $\Delta t = 0.5$ on the right side. These choices of $c$ and $\Delta t$ imply that $b = 1$ on both sides, but that the right side is updated twice as often as the left side. What happens to a pulse that begins on the left side and moves to the right? Is there both a reflected and transmitted wave at the boundary between the two media? What is their relative phase? Find a relation between the amplitude of the incident pulse and the amplitudes of the reflected and transmitted pulses. Repeat for a pulse starting from the right side.

**Problem 9.28.** Superposition of waves

a. Consider the propagation of the wave determined by $u(x, t = 0) = \sin(4\pi x/N)$. What must $u(x, -\Delta t)$ be so that the wave moves in the positive $x$ direction? Test your answer by doing a simulation. Use periodic boundary conditions. Repeat for a wave moving in the negative $x$ direction.

b. Simulate two waves moving in opposite directions each with the same spatial dependence given by $u(x, 0) = \sin(4\pi x/N)$. Describe the resultant wave pattern. Repeat the simulation for $u(x, 0) = \sin(8\pi x/N)$.

c. Assume that $u(x, 0) = \sin q_1 x + \sin q_2 x$, with $q_1 = 10\pi/N$ and $q_2 = 12\pi/N$. Describe the qualitative form of $u(x,t)$ for fixed $t$. What is the distance between modulations of the amplitude? Estimate the wavelength associated with the fine ripples of the amplitude. Estimate the wavelength of the envelope of the wave. Find a simple relation for these two wavelengths in terms of the wavelengths of the two sinusoidal terms. This phenomena is known as *beats*.

d. Consider the motion of the two Gaussian pulses moving in opposite directions, $u_1(x, 0) = e^{-(x-10)^2}$ and $u_2(x, 0) = e^{-(x-90)^2}$. Choose the array at $t = -\Delta t$ as in Problem 9.25. What happens to the two pulses when they overlap or partially overlap? Do they maintain their shape? While they are going through each other, is the displacement $u(x,t)$ given by the sum of the displacements of the individual pulses?

**Problem 9.29.** Standing waves

a. In Problem 9.28c we considered a *standing wave*, the continuum analog of a normal mode of a system of coupled oscillators. As is the case for normal modes, each point of the wave has the same time dependence. For fixed boundary conditions, the displacement is given by $u(x,t) = \sin qx \cos \omega t$, where $\omega = cq$ and the wavenumber $q$ is chosen so that $\sin qN = 0$. Choose an initial condition corresponding to a standing wave for $N = 100$. Describe the motion of the particles, and compare it with your observations of standing waves on a rope.

b. Establish a standing wave by displacing one end of a system periodically. The other end is fixed. Let $u(x,0) = u(x,-\Delta t) = 0$, and $u(x=0,t) = A\sin\omega t$ with $A = 0.1$. How long must the simulation run before you observe standing waves? How large is the standing wave amplitude?

We have seen that the wave equation can support pulses that propagate indefinitely without distortion. In addition, because the wave equation is linear, the sum of any two solutions also is a solution, and the principle of superposition is satisfied. As a consequence, we know that two pulses can pass through each other unchanged. We also have seen that similar phenomena exist in the discrete system of linearly coupled oscillators. What happens if we create a pulse in a system of nonlinear oscillators? As an introduction to nonlinear wave phenomena, we consider a system of $N$ coupled oscillators with the potential energy of interaction given by

$$V = \frac{1}{2}\sum_{j=1}^{N}\big[e^{-(u_j-u_{j-1})} - 1\big]^2. \tag{9.58}$$

This form of the interaction is known as the Morse potential. All parameters in the potential (such as the overall strength of the potential) have been set to unity. The force on the $j$th particle is

$$F_j = -\frac{\partial V}{\partial u_j} = Q_j(1-Q_j) - Q_{j+1}(1-Q_{j+1}), \tag{9.59a}$$

where

$$Q_j = e^{-(u_j-u_{j-1})}. \tag{9.59b}$$

In linear systems it is possible to set up a pulse of any shape and maintain the shape of the pulse indefinitely. In a nonlinear system there also exist solutions that maintain their shape, but we will find in Problem 9.30 that not all pulse shapes do so. The pulses that maintain their shape are called *solitons*.

**Problem 9.30.** Solitons

a. Modify the program developed in Problem 9.2 so that the force on particle $j$ is given by (9.59). Use periodic boundary conditions. Choose $N \geq 60$ and an initial pulse of the form $u(x,t) = 0.5\,e^{-(x-10)^2}$. You should find that the initial pulse splits into two pulses plus some noise. Describe the motion of the pulses (solitons). Do they maintain their shape, or is this shape modified as they move? Describe the motion of the particles far from the pulse. Are they stationary?

b. Save the displacements of the particles when the peak of one of the solitons is located near the center of your display. Is it possible to fit the shape of the soliton to a Gaussian? Continue the simulation, and after one of the solitons is relatively isolated, set `u(j) = 0` for all `j` far from this soliton. Does the soliton maintain its shape?

c. Repeat part (b) with a pulse given by $u(x,0) = 0$ everywhere except for $u(20,0) = u(21,0) = 1$. Do the resulting solitons have the same shape as in part (b)?

d. Begin with the same Gaussian pulse as in part (a), and run until the two solitons are well separated. Then change at random the values of `u(j)` for particles in the larger soliton by about 5%, and continue the simulation. Is the soliton destroyed? Increase the perturbation until the soliton is no longer discernible.

e. Begin with a single Gaussian pulse as in part (a). The two resultant solitons will eventually "collide." Do the solitons maintain their shape after the collision? The principle of superposition implies that the displacement of the particles is given by the sum of the displacements due to each pulse. Does the principle of superposition hold for solitons?

f. Compute the speeds, amplitudes, and width of the solitons produced from a single Gaussian pulse. Take the amplitude of a soliton to be the largest value of its displacement and the half-width to correspond to the value of $x$ at which the displacement is half its maximum value. Repeat these calculations for solitons of different amplitudes by choosing the initial amplitude of the Gaussian pulse to be 0.1, 0.3, 0.5, 0.7, and 0.9. Plot the soliton speed and width versus the corresponding soliton amplitude.

g. Change the boundary conditions to free boundary conditions and describe the behavior of the soliton as it reaches a boundary. Compare this behavior with that of a pulse in a system of linear oscillators.

h. Begin with an initial sinusoidal disturbance that would be a normal mode for a linear system. Does the sinusoidal mode maintain its shape? Compare the behavior of the nonlinear and linear systems.

## 9.8 Interference

Interference is one of the most fundamental characteristics of all wave phenomena. The term *interference* is used when there are a small number of sources and the term *diffraction* when the number of sources is large and can be treated as a continuum. Because it is relatively easy to observe interference and diffraction phenomena with light, we discuss these phenomena in this context.

Consider the field from one or more point sources lying in a plane. The electric field at position $\mathbf{r}$ associated with the light emitted from a monochromatic point source at $\mathbf{r}_1$ is a spherical wave radiating from that point. This wave can be thought of as the real part of a complex exponential

$$E(\mathbf{r},t) = \frac{A}{|\mathbf{r}-\mathbf{r}_1|} e^{i(q|\mathbf{r}-\mathbf{r}_1|-\omega t)}, \tag{9.60}$$

where $|\mathbf{r}-\mathbf{r}_1|$ is the distance between the source and the point of observation and $q$ is the wavenumber $2\pi/\lambda$. The superposition principle implies that the total electric field at $\mathbf{r}$ from $N$ point sources at $\mathbf{r}_i$ is

$$E(\mathbf{r},t) = e^{-i\omega t} \sum_{n=1}^{N} \frac{A_n}{|\mathbf{r}-\mathbf{r}_n|} e^{i(q|\mathbf{r}-\mathbf{r}_n|)} = e^{-i\omega t}\mathcal{E}(\mathbf{r}). \tag{9.61}$$

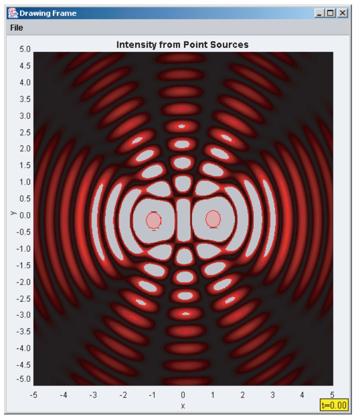

Figure 9.3: The computed energy density in the vicinity of two point sources.

The time evolution can be expressed as an oscillatory complex exponential $e^{-i\omega t}$ that multiplies a complex space part $\mathcal{E}(\mathbf{r})$. The spatial part $\mathcal{E}(\mathbf{r})$ is a *phasor* which contains both the maximum value of the electric field and the time within a cycle when the physical field reaches its maximum value. As the system evolves, the complex electric field $E(\mathbf{r},t)$ oscillates between purely real and purely imaginary values. Both the energy density (the energy per unit volume) and the light intensity (the energy passing through a unit area) are proportional to the square of the magnitude of the phasor. Because light fields oscillate at $\approx 6 \times 10^{14}$ Hz, typical optics experiments observe the time-average (rms value) of $\mathcal{E}$ and do not observe the phase angle.

Huygens's principle states that each point on a wavefront (a surface of constant phase) can be treated as the source of a new spherical wave or *Huygens's wavelet.* The wavefront at some later time is constructed by summing these wavelets. The `HuygensApp` program implements Huygens's principle by assuming superposition from an arbitrary number of point sources and displaying a two-dimensional animation of (9.61) using the tools described in Appendix 9C.

Sources are represented by circles and are added to the frame when a custom button invokes the `createSource` method.

```
public void createSource() {
   InteractiveShape ishape = InteractiveShape.createCircle(0, 0, 0.5);
   frame.addDrawable(ishape);
   initPhasors();
   frame.repaint();
}
```

Users can create as many sources as they wish. The program later retrieves a list of sources from the frame using the latter's `getDrawables` method.

The program uses $n \times n$ arrays to store the real and imaginary values. The code fragment from the `initPhasors` method shown in the following starts the process by obtaining a list of point sources in the frame. We then use an `Iterator` to access each source as we sum the vector components at each grid point.

```
ArrayList list=frame.getDrawables();    // gets list of point sources
Iterator it=list.iterator();           // creates an iterator for the list
// these two statements are combined in the final code
```

`List` and `Iterator` are interfaces that are implemented by the objects returned by `frame.getDrawables` and `list.iterator`, respectively. As the name implies, an iterator is a convenient way to access a list without explicitly counting its elements. The iterator's `getNext` method retrieves elements from the list and the `hasNext` method returns true if the end of the list has not been reached.

The `initPhasors` method in `HuygensApp` computes the phasors at every point by summing the phasors at each grid point. Note how the distance from the source to the observation point is computed by converting the grid's index values to world coordinates.

```
Iterator it = frame.getDrawables().iterator();  // source iterator
while(it.hasNext()) {
   InteractiveShape source = (InteractiveShape) it.next();
   double xs = source.getX(), ys = source.getY(); // world coordinates for source
   for(int ix = 0;ix<n;ix++) {
      double x = frame.indexToX(ix);
      double dx = (xs-x);  // source -> gridpoint x separation
      for(int iy = 0;iy<n;iy++) {
         double y = frame.indexToY(iy);
         double dy = (ys-y);  // charge -> gridpoint y separation
         double r = Math.sqrt(dx*dx+dy*dy);
         realArray[ix][iy] += (r==0) ? 0 : Math.cos(PI2*r)/r;  // real
         imagArray[ix][iy] += (r==0) ? 0 : Math.sin(PI2*r)/r;  // imaginary
      }
   }
}
```

To calculate the real and imaginary components of the phasor, the distance from the source to the grid point is determined in terms of the wavelength $\lambda$ and time is is determined in terms of the period $T$. For example, for green light one unit of distance is $\approx 5 \times 10^{-7}$ m and one unit of time is $\approx 1.6 \times 10^{-15}$ s.

The simulation is performed by multiplying the phasors by $e^{-i\omega t}$ in the `doStep` method. Multiplying each phasor by $e^{-i\omega t}$ mixes the phasor's real and imaginary components. We then obtain the physical field from (9.61) by taking the real part.

$$E(\mathbf{r},t) = \mathrm{Re}[e^{-i\omega t}\mathcal{E}(\mathbf{r})] = \mathrm{Re}[\mathcal{E}]\cos\omega t - \mathrm{Im}[\mathcal{E}]\sin\omega t. \tag{9.62}$$

Listing 9.10 shows the entire `HuygensApp` class. A custom button is used to create sources at the origin. Because the source is an `InteractiveShape`, it can be repositioned using the mouse. The program also implements the `InteractiveMouseHandler` interface to recalculate the phasors when the source is moved. (See Section 5.7 for a discussion of interactive handlers.)

Listing 9.10: The `HuygensApp` class simulates the energy density from one or more point sources.

```
package org.opensourcephysics.sip.ch09;
import java.util.*;
import java.awt.event.*;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.display2d.*;
import org.opensourcephysics.frames.*;

public class HuygensApp extends AbstractSimulation implements InteractiveMouseHandler {
   static final double PI2 = Math.PI*2;
   Scalar2DFrame frame = new Scalar2DFrame("x", "y", "Intensity from point sources");
   double time = 0;
   double[][] realPhasor, imagPhasor, amplitude;
   int n;    // grid points on a side
   double a; // side length

   public HuygensApp() {
      // interpolated plot looks best
      frame.convertToInterpolatedPlot();
      frame.setPaletteType(ColorMapper.RED);
      frame.setInteractiveMouseHandler(this);
   }

   public void initialize() {
      n = control.getInt("grid size");
      a = control.getDouble("length");
      frame.setPreferredMinMax(-a/2, a/2, -a/2, a/2);
      realPhasor = new double[n][n];
      imagPhasor = new double[n][n];
      amplitude = new double[n][n];
      frame.setAll(amplitude);
      initPhasors();
   }

   void initPhasors() {
      for(int ix = 0;ix<n;ix++) {
         for(int iy = 0;iy<n;iy++) {
            imagPhasor[ix][iy] = realPhasor[ix][iy] = 0; // zero the phasor
         }
      }
      // an iterator for the sources in the frame
      Iterator it = frame.getDrawables().iterator(); // source iterator
      int counter = 0;                               // counts the number of sources
      while(it.hasNext()) {
         InteractiveShape source = (InteractiveShape) it.next();
         counter++;
         double
            xs = source.getX(), ys = source.getY();
         for(int ix = 0;ix<n;ix++) {
            double x = frame.indexToX(ix);
            double dx = (xs-x);                                  //source->gridpoint
            for(int iy = 0;iy<n;iy++) {
               double y = frame.indexToY(iy);
```

```
          double dy = (ys-y);                                //charge->gridpoint
          double r = Math.sqrt(dx*dx+dy*dy);
          realPhasor[ix][iy] += (r==0) ? 0 : Math.cos(PI2*r)/r; // real
          imagPhasor[ix][iy] += (r==0) ? 0 : Math.sin(PI2*r)/r; // imaginary
        }
      }
    }
    double cos = Math.cos(-PI2*time);
    double sin = Math.sin(-PI2*time);
    for(int ix = 0;ix<n;ix++) {
      for(int iy = 0;iy<n;iy++) {
        // only the real part of the complex field is physical
        double re = cos*realPhasor[ix][iy]-sin*imagPhasor[ix][iy];
        amplitude[ix][iy] = re*re;
      }
    }
    frame.setZRange(false, 0, 0.2*counter); // scale the intensity
    frame.setAll(amplitude);
  }

  public void reset() {
    time = 0;
    control.setValue("grid size", 128);
    control.setValue("length", 10);
    frame.clearDrawables();
    frame.setMessage("t = "+decimalFormat.format(time));
    control.println("Source button creates a new source.");
    control.println("Drag sources after they are created.");
    initialize();
  }

  public void createSource() {
    InteractiveShape ishape = InteractiveShape.createCircle(0, 0, 0.5);
    frame.addDrawable(ishape);
    initPhasors();
    frame.repaint();
  }

  public void handleMouseAction(InteractivePanel panel, MouseEvent evt) {
    panel.handleMouseAction(panel, evt); // panel moves the source
    if(panel.getMouseAction()==InteractivePanel.MOUSE_DRAGGED) {
      initPhasors();
    }
  }

  protected void doStep() {
    time += 0.1;
    double cos = Math.cos(-PI2*time);
    double sin = Math.sin(-PI2*time);
    for(int ix = 0;ix<n;ix++) {
```

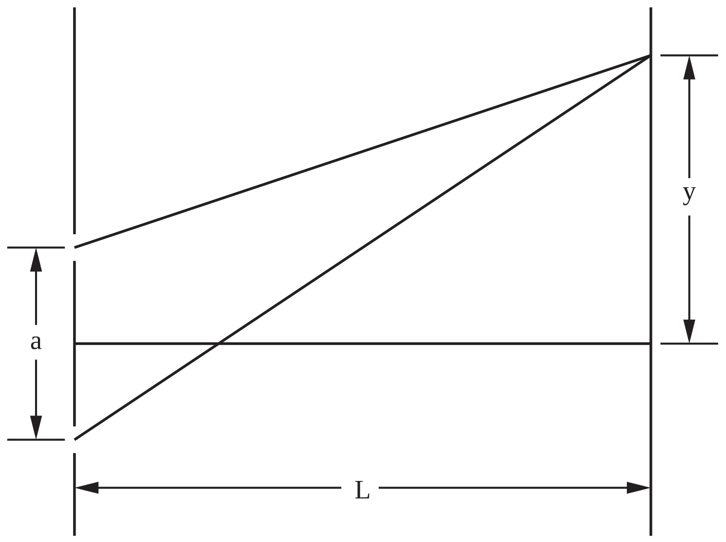

Figure 9.4: Young's double slit experiment. The figure defines the quantities $a$, $L$, and $y$ used in Problem 9.31.

```
          for(int iy = 0;iy<n;iy++) {
             double re = cos*realPhasor[ix][iy]-sin*imagPhasor[ix][iy]; //real part
             amplitude[ix][iy] = re*re;
          }
       }
       frame.setAll(amplitude);
       frame.setMessage("t="+decimalFormat.format(time));
   }

   public static void main(String[] args) {
       OSPControl control = SimulationControl.createApp(new HuygensApp());
       control.addButton("createSource", "Source");
   }
}
```

The classic example of interference is Young's double slit experiment (see Figure 9.4). Imagine two narrow parallel slits separated by a distance $a$ and illuminated by a light source that emits light of only one frequency (monochromatic light). If the light source is placed on the line bisecting the two slits and the slit opening is very narrow, the two slits become coherent light sources with equal phases. We first assume that the slits act as point sources, for example, pinholes. A screen that displays the intensity of the light from the two sources is placed a distance $L$ away. What do we see on the screen?

In the following problems we discuss writing programs to determine the intensity of light that is observed on a screen due to a variety of geometries. The wavelength of the light sources, the positions of the sources $\mathbf{r}_i$, and the observation points on the screen need to be specified. Your program should instantiate the necessary point sources and compute a plot showing the intensity on the obervation screen located to the right of the sources by summing the phasors. Although we suggest that you use Listing 9.10 as a guide, it is unrealistic to compute a two-dimensional grid that covers the entire region from the source to the screen. It is more efficient to plot the intensity on the screen, thereby reducing the computation to a single loop over the screen coordinate $y$.

**Problem 9.31.** Point source interference

a. Derive an analytical expression for $E$ from two and three point sources if the screen is far from the sources.

b. Compute and plot the intensity of light on a screen due to two small sources (a source that emits spherical waves). Compute the phasors using (9.61) and find the intensity by taking the magnitude of $\mathcal{E}$. Let $a$ be the distance between the sources and $y$ be the vertical position along the screen as measured from the central maximum. Set $L = 200\,\text{mm}$, $a = 0.1\,\text{mm}$, the wavelength of light $\lambda = 5000\,\text{Å}\,(1\,\text{Å}= 10^{-7}\,\text{mm})$, and consider $-5.0\,\text{mm} \leq \text{y} \leq 5.0\,\text{mm}$ (see Figure 9.4). Describe the interference pattern you observe. Identify the locations of the intensity maxima, and plot the intensity of the maxima as a function of $y$. Compare your result to the analytic expression for a two slit diffraction pattern.

c. Repeat part (b) for $L = 0.5\,\text{mm}$ and $1.0\,\text{mm} \leq \text{y} \leq 1.0\,\text{mm}$. Note that in this case $L$ is not much greater than $a$, and hence we cannot ignore the $r$ dependence of $|\mathbf{r} - \mathbf{r}_i|^{-1}$ in (9.61).

**Problem 9.32.** Diffraction grating

High resolution optical spectroscopy is done with multiple slits. In its simplest form, a diffraction grating consists of $N$ parallel slits. Compute the intensity of light for $N = 3$, 4, 5, and 10 slits with $\lambda = 5000\,\text{Å}$, slit separation $a = 0.01\,\text{mm}$, screen distance $L = 200\,\text{mm}$, and $-15\,\text{mm} \leq \text{y} \leq 15\,\text{mm}$. How does the intensity of the peaks and their separation vary with $N$?

In our analysis of the double slit and the diffraction grating, we assumed that each slit was a pinhole that emits spherical waves. In practice, real slits are much wider than the wavelength of visible light. In Problem 9.33 we consider the pattern of light produced when a plane wave is incident on an aperture such as a single slit. To do so, we use Huygens's principle and replace the slit by many coherent sources of spherical waves. This equivalence is not exact, but is applicable when the aperture width is large compared to the wavelength.

**Problem 9.33.** Single slit diffraction

a. Compute the time averaged intensity of light diffracted from a single slit of width 0.02 mm by replacing the slit by $N = 20$ point sources spaced 0.001 mm apart. Choose $\lambda = 5000\,\text{Å}$, $L = 200\,\text{mm}$, and consider $-30\,\text{mm} \leq \text{y} \leq 30\,\text{mm}$. What is the width of the central peak? How does the width of the central peak compare to the width of the slit? Do your results change if $N$ is increased?

b. Determine the position of the first minimum of the diffraction pattern as a function of the wavelength, slit width, and distance to the screen.

c. Compute the intensity pattern for $L = 1\,\text{mm}$ and $50\,\text{mm}$. Is the far field condition satisfied in this case? How do the patterns differ?

**Problem 9.34.** A more realistic double slit simulation

Reconsider the intensity distribution for double slit interference using slits of finite width. Modify your program to simulate two "thick" slits by replacing each slit by 20 point sources spaced $0.001\,\text{mm}$ apart. The centers of the thick slits are $a = 0.1\,\text{mm}$ apart. How does the intensity pattern change?

$^*$**Problem 9.35.** Diffraction pattern from a rectangular aperture

We can use a similar approach to determine the diffraction pattern due to a two-dimensional thin opaque mask with an aperture of finite width and height near the center. The simplest approach is to divide the aperture into little squares and to consider each square as a source of spherical waves. Similarly we can divide the viewing screen or photographic plate into small regions or cells and calculate the time averaged intensity at the center of each cell. The calculations are straightforward, but time consuming, because of the necessity of evaluating the cosine function many times. The less straightforward part of the problem is deciding how to plot the different values of the calculated intensity on the screen. One way is to plot "points" at random locations in each cell so that the number of points is proportional to the computed intensity at the center of the cell. Suggested parameters are $\lambda = 5000\,\text{Å}$ and $N = 200\,\text{mm}$ for a $1\,\text{mm} \times 3\,\text{mm}$ aperture.

If the number of sources $N$ becomes large, the summation becomes the Huygens-Fresnel integral. Optics texts discuss how different approximations can be used to evaluate (9.61). For example, if the sources are much closer to each other than they are to the screen (the *far field* condition), we obtain the condition for Fraunhofer diffraction. Otherwise, we obtain Fresnel diffraction.

## 9.9 Fraunhofer Diffraction

The contemporary approach to diffraction is based on Fourier analysis. Consider a plane wave incident on a one-dimensional aperture. Using Huygen's idea that every point within the aperture serves as the source of spherical secondary wavelets, we can approximate the aperture as a linear array of in-phase coherent oscillators. If the spatial extent of the array is small and the distance to the observing screen is large, the wavelet amplitudes arriving at the screen will be essentially equal for wavelets within the aperture and zero for wavelets on the opaque mask. The total field is thus given by the real part of the sum

$$E(\mathbf{r},t) = E_0 a_1 e^{i(qr_1-\omega t)} + E_0 a_2 e^{i(qr_2-\omega t)} + E_0 a_3 e^{i(qr_2-\omega t)} + \cdots + E_0 a_N e^{i(qr_N-\omega t)}, \tag{9.63}$$

where $a_i = 1$ within the aperture and $a_i = 0$ outside the aperture and the wavenumber $q$ and the angular frequency $\omega$ have their usual meaning. Equation (9.63) can be factored into a complex phasor and time-dependent phase shift

$$E(\mathbf{r},t) = e^{-i\omega t}\mathcal{E}(\mathbf{r}), \tag{9.64}$$

where

$$\mathcal{E}(\mathbf{r}) = E_0 e^{iqr_1}\big[1 + a_1 e^{iq(r_2-r_1)} + a_2 e^{iq(r_3-r_1)} + \cdots + a_N e^{iq(r_N-r_1)}\big]. \tag{9.65}$$

If the grid spacing $d$ is uniform, it follows that the phase difference between adjacent sources is $\delta = qd\sin\theta$, where $\theta$ is the angle between the aperture and a point on the screen. We substitute $\delta$ into (9.65) and observe that the field can be expressed as an overall phase times a Fourier sum:

$$\mathcal{E}(\mathbf{r}) = E_0 e^{ikqr_1}\big[1 + \sum_{n=1}^{N} a_n e^{in\delta}\big]. \tag{9.66}$$

Equation (9.66) shows that a Fraunhofer (far field) diffraction pattern can be obtained by dividing the aperture using a uniform grid and computing the complex Fourier transform. Because the intensity is proportional to the square of the electric field, the diffraction pattern is the modulus squared of the Fourier transform of the aperture's transmission function. Listing 9.11 uses the one-dimensional fast Fourier transform to compute the Fraunhofer diffraction pattern for a slit.

Listing 9.11: The `FraunhoferApp` program computes the Fraunhofer diffraction pattern from a slit.

```java
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.FFT;

public class FraunhoferApp {
   static final double PI2 = Math.PI*2;
   static final double LOG10 = Math.log(10);            // Math.log is natural log
   static final double ALPHA = Math.log(1.0e-3)/LOG10; // cutoff value

   public static void main(String[] args) {
      PlotFrame plot = new PlotFrame("x", "intensity", "Fraunhofer diffraction");
      int N = 512;
      FFT fft = new FFT(N);
      double[] cdata = new double[2*N];
      double a = 10; // aperture screen dimension
      double dx = (2*a)/N;
      double x = -a;
      for(int ix = 0;ix<N;ix++) {
         cdata[2*ix] = (Math.abs(x)<0.4) ? 1 : 0; // slit
         cdata[(2*ix)+1] = 0;
         x += dx;
      }
      fft.transform(cdata);
      fft.toNaturalOrder(cdata);
      double max = 0;
      // find the max intensity value
      for(int i = 0;i<N;i++) {
         double real = cdata[2*i];      // real
         double imag = cdata[(2*i)+1]; // imaginary
         max = Math.max(max, (real*real)+(imag*imag));
         plot.append(0, i, (real*real)+(imag*imag));
      }
      plot.setVisible(true);
      //create N by 30 raster plot to show an image
      int[][] data = new int[N][30];
      //compute pixel intensity
      for(int i = 0;i<N;i++) {
         double real = cdata[2*i];      // real
         double imag = cdata[(2*i)+1]; // imaginary
         // log scale increases visibility of weaker fringes
         double logIntensity = Math.log(((real*real)+(imag*imag))/max)/LOG10;
         int intensity = (logIntensity<=ALPHA) ? 0 : (int) (254*(1-(logIntensity/ALPHA)));
         for(int j = 0;j<30;j++) {
            data[i][j] = intensity;
         }
      }
      RasterFrame frame = new RasterFrame("Fraunhofer Diffraction (log scale)");
      frame.setBWPalette();
      frame.setAll(data); // send the fft data to the raster frame
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

To emphasize the weaker regions of the diffraction pattern, the program plots the logarithm of the intensity. First the intensity is normalized to its peak value, then the logarithm is taken and all values less than a cutoff are truncated. The resulting range is mapped linearly to $(0, 255)$ to set the gray scale.

**Problem 9.36.** Two-dimensional apertures

Modify `FraunhoferApp` to show the diffraction pattern from a double slit and compare the computed diffraction pattern to the analytic result.

The `Fraunhofer2DApp` program computes the Fraunhofer diffraction pattern for a circular aperture using a two-dimensional FFT. This program is used in Problem 9.37 but is not listed because it is too long and because it is similar to Listing 9.11.

**Problem 9.37.** Two-dimensional apertures

a. Compile and run the `Fraunhofer2DApp` program. Compute the diffraction pattern using aperture radii of $4\lambda$ and $0.25\lambda$ in a mask with dimension $a\lambda$. How does the radius influence the diffraction pattern? How does the rectangular grid influence the pattern? How can the effect of the rectangular grid be reduced?

b. Because a typical computer monitor displays only 256 gray scale values, `Fraunhofer2DApp` uses a logarithmic scale to enhance the visibility of the fringes. Add code to display a linear plot of intensity as a function of radius using a slice through the center of the pattern.

c. Compute the diffraction pattern for an annular ring with inner radius $1.8\lambda$ and outer radius $2.2\lambda$. Why is there a bright spot at the center of the diffraction pattern? What effect does the finite width of the annular ring produce?

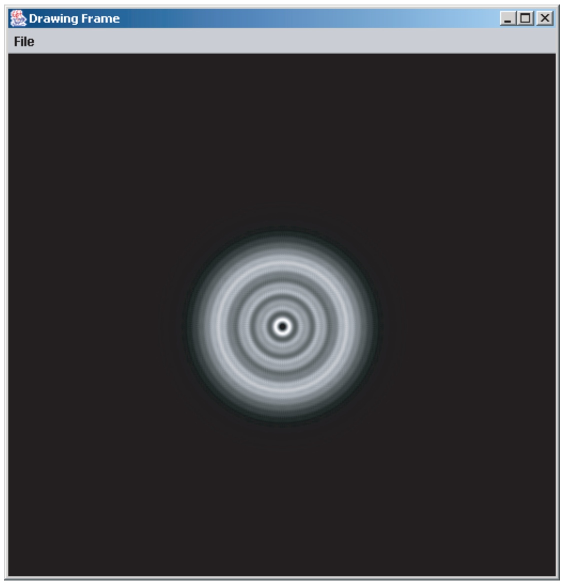

Figure 9.5: Computed Fresnel diffraction on a screen illuminated by a uniform plane wave and located $0.5 \times 10^6\lambda$ from a circular aperture of radius $2000\lambda$.

d. Compute the diffraction pattern for a rectangular aperture with width $2\lambda$ and height $6\lambda$. Describe the effect of the asymmetry of the slit.

**Problem 9.38.** Diffraction patterns due to multiple apertures

a. Compute the diffraction pattern due to a $5 \times 5$ array of rectangular slits. Each slit has a width of 0.5 and a height of 0.25 and is offset by (1,1) from neighboring slits using units such that $\lambda = 1$. The aperture array is centered within a mask ten units on a side. What is the effect of the asymmetry of the slit? What happens if the number of slits is decreased or increased?

b. Compute the diffraction pattern due to 25 randomly placed rectangular slits. Each slit should have a width of 0.5 and a height of 0.25 and be placed within a region ten units on a side. Do not be concerned if rectangles overlap.

c. Compare the results from (a) and (b). What effect does the random placement have on the pattern?

## 9.10 Fresnel Diffraction

Fourier analysis can be used to compute the Fresnel diffraction pattern by decomposing a wave incident on an aperture into a sum of plane waves and then propagating each plane wave from the aperture mask to the screen. A plane wave with wavenumber $(q_x, q_y, q_z)$ propagating in a homogenous environment can be written as

$$\mathcal{U} = \mathcal{U}_0 e^{i(q_x x + q_y y + q_z z)}. \tag{9.67}$$

where $\mathcal{U}_0$ is the amplitude of the field at the origin and $(q_x, q_y, q_z)$ is a vector of length $2\pi/\lambda$ in the direction of propagation. If we place a viewing screen perpendicular to the direction of the incoming light at a point $z_0$ along the $z$ axis, then the field on the screen is

$$\mathcal{U} = \mathcal{U}_0 e^{iq_z z_0} = \mathcal{U}_0 e^{iz_0(q^2 - q_x^2 - q_y^2)^{1/2}}, \tag{9.68}$$

where we have used the fact that $q_x^2 + q_y^2 + q_z^2 = q^2$.

We now place an aperture mask at the origin $z = 0$ in the $xy$-plane and illuminated it from the left by a plane wave. Because the aperture truncates the incident plane wave, we obtain a more complicated field $\mathcal{U}_0(q_x, q_y)$ that contains both $q_x$ and $q_y$ spatial components.

$$\mathcal{U}_0(q_x, q_y) = \iint_{\text{aperture}} e^{i(q_x x + q_y y)} dx\, dy. \tag{9.69}$$

The field that propagates from the origin contains the Fourier components of the aperture mask. In other words, because we have truncated the wave, we have a field composed of a mixture of plane waves with wavenumbers $(q_x, q_y, q_z)$. Each field component is multiplied by the $e^{iq_z z_0}$ phase factor in (9.68) as it propagates toward the viewing screen at $z_0$. The following steps summarize the algorithm:

1. Compute the Fourier transformation of the aperture (9.69) to obtain the field's components in the plane of the aperture;
2. Multiply each component by the propagation phase factor $e^{iz_0(q^2 - q_x^2 - q_y^2)^{1/2}}$;
3. Compute the inverse transformation to obtain the amplitude;
4. Compute the magnitude squared of the amplitude to obtain the intensity.

The Fresnel diffraction pattern algorithm is implemented in Listing 9.12. Note that the field includes evanescent waves if $q^2 - q_x^2 - q_y^2 < 0$.

Listing 9.12: The `FresnelApp` program computes the Fresnel diffraction pattern from a circular aperture.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.frames.RasterFrame;
import org.opensourcephysics.numerics.FFT2D;

public class FresnelApp {
   final static double PI2 = Math.PI*2;
   final static double PI4 = PI2*PI2;

   public static void main(String[] args) {
```

```
int N = 512;                           //power of 2 for optimum speed
double z = 0.5e+6;                     // distance from aperture to screen
FFT2D fft2d = new FFT2D(N, N);
double[] cdata = new double[2*N*N]; // complex data
double a = 6000;                       // aperture mask dimension
double
   dx = 2*a/N, dy = 2*a/N;
double x = -a;
for(int ix = 0;ix<N;ix++) {
   int offset = 2*ix*N;
   double y = -a;
   for(int iy = 0;iy<N;iy++) {
      double r2 = (x*x+y*y);
      cdata[offset+2*iy] = (r2<4e6) ? 1 : 0; // circular aperture
      cdata[offset+2*iy+1] = 0;
      y += dy;
   }
   x += dx;
}
fft2d.transform(cdata);
// get arrays containing the wavenumbers in wrapped order
double[] kx = fft2d.getWrappedOmegaX(-a, a);
double[] ky = fft2d.getWrappedOmegaY(-a, a);
for(int ix = 0;ix<N;ix++) {
   int offset = 2*ix*N; //offset to beginning of row
   for(int iy = 0;iy<N;iy++) {
      double radical = PI4-kx[ix]*kx[ix]-ky[iy]*ky[iy];
      if(radical>0) {
         double phase = z*Math.sqrt(radical);
         double real = Math.cos(phase);
         double imag = Math.sin(phase);
         double temp = cdata[offset+2*iy];
         cdata[offset+2*iy] = real*cdata[offset+2*iy]-imag*cdata[offset+2*iy+1];
         cdata[offset+2*iy+1] = real*cdata[offset+2*iy+1]+imag*temp;
      } else {          //evanescent waves decay exponentially
         double decay = Math.exp(-z*Math.sqrt(-radical));
         cdata[offset+2*iy] *= decay;
         cdata[offset+2*iy+1] *= decay;
      }
   }
}
fft2d.inverse(cdata);
double max = 0;
for(int i = 0;i<N*N;i++) {      // find max intensity
   double real = cdata[2*i];    // real
   double imag = cdata[2*i+1]; // imaginary
   max = Math.max(max, real*real+imag*imag);
}
// intensity is squared magnitude of the amplitude
int[] data = new int[N*N];
```

```
      for(int i = 0, N2 = N*N;i<N2;i++) {
         double real = cdata[2*i];    // real
         double imag = cdata[2*i+1]; // imaginary
         data[i] = (int) (255*(real*real+imag*imag)/max);
      }
      //raster for least memory and best speed
      RasterFrame frame = new RasterFrame("Fraunhofer Diffraction");
      frame.setBWPalette();
      frame.setAll(data, N, -0.5, 0.5, -0.5, 0.5);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

Because the algorithm in Listing 9.12 depends only on the linearity of the wave equation, it is exact and may be applied to many practical optics problems. Its main limitation occurs when $z_0$ is large because of rapid oscillations in the phase factor. In this case, the far field (Fraunhofer) approximation usually becomes applicable and should be used.

**Problem 9.39.** Fresnel diffraction

The diffraction pattern from a circular aperture is important because most lenses, mirrors, and optical instruments have cylindrical symmetry.

a. Compute the diffraction pattern due to a circular aperture with a radius of $1000\,\lambda$ at a screen distance of $10^5\,\lambda$. Is the center of the diffraction pattern dark or light? Reposition the screen to $2 \times 10^5\lambda$ and repeat the calculation. Does the intensity at the center of the shadow change? Use a $512 \times 512$ grid to sample a region of space $5000\,\lambda$ on a side.

b. Replace the screen by a circular disk. That is, use an aperture mask that is opaque if $r < 1000\lambda$. Is the center of the screen light or dark? Does the center change from bright to dark if the screen is repositioned?

**Problem 9.40.** Optical resolution

Consider a mask containing two circular openings of radius $500\lambda$ separated by $100\lambda$. Do a simulation to determine how far the screen can be placed from the aperture mask and still observe two distinct shadows.

## Appendix 9A: Complex Fourier Series

A function $f(t)$ with period $T$ can be expressed in terms of a trigonometric Fourier series

$$f(t) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty} \big(a_k \cos\omega_k t + b_k \sin\omega_k t\big), \tag{9.70}$$

where $\omega_k = k\omega_0$ and $\omega_0 = 2\partial/T$. To derive the exponential form of this series, we express the sine and cosine functions as

$$\sin x = \frac{e^{ix} - e^{-ix}}{2i} \tag{9.71a}$$

$$\cos x = \frac{e^{ix} + e^{-ix}}{2} \tag{9.71b}$$

We substitute (9.71) into (9.70) and obtain

$$f(t) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty}\big[e^{ik\omega_0 t}\frac{a_k - ib_k}{2} + e^{-ik\omega_0 t}\frac{a_k + ib_k}{2}\big]. \tag{9.72}$$

We use $1/i = -i$ and define new Fourier coefficients as follows:

$$c_0 \equiv \frac{1}{2}a_0 \tag{9.73a}$$

$$c_k \equiv \frac{a_k - ib_k}{2} \tag{9.73b}$$

$$c_{-k} \equiv \frac{a_{-k} - ib_{-k}}{2} = \frac{a_k + ib_k}{2}, \tag{9.73c}$$

where the right-hand side of (9.73c) follows from $a_k = a_{-k}$ and $b_k = -b_{-k}$. We substitute these coefficients into (9.72) and find

$$f(t) = c_0 + \sum_{k=1}^{\infty} c_k e^{ik\omega_0 t} + \sum_{k=1}^{\infty} c_{-k} e^{-ik\omega_0 t}, \tag{9.74}$$

or

$$f(t) = \sum_{k=0}^{\infty} c_k e^{ik\omega_0 t} + \sum_{k=1}^{\infty} c_{-k} e^{-ik\omega_0 t}. \tag{9.75}$$

Finally, we re-index the second sum from $-1$ to $-\infty$

$$f(t) = \sum_{k=0}^{\infty} c_k e^{ik\omega_0 t} + \sum_{k=-1}^{-\infty} c_k e^{ik\omega_0 t}, \tag{9.76}$$

and combine the summations to obtain the exponential form:

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{ik\omega_0 t}. \tag{9.77}$$

## Appendix 9B: Fast Fourier Transform

The fast Fourier transform (FFT) was discovered independently by many workers in a variety of contexts. There are several variations of the algorithm, and we describe a version due to Danielson and Lanczos. The goal is to compute the Fourier transform $g(\omega_k)$ given the data set $f(j\Delta) \equiv f_j$ of (9.35). For convenience we rewrite the relation:

$$g_k \equiv g(\omega_k) = \sum_{j=0}^{N-1} f(j\Delta)\, e^{-i2\pi kj/N}, \tag{9.78}$$

and introduce the complex number $W$ given by

$$W = e^{-i2\pi/N}. \tag{9.79}$$

The following algorithm works with any complex data set if $N$ is a power of two. Real data sets can be transformed by setting the array elements corresponding to the imaginary part equal to 0.

To understand the FFT algorithm, we consider the case $N = 8$, and rewrite (9.78) as

$$g_k = \sum_{j=0,2,4,6} f(j\Delta)\, e^{-i2\pi kj/N} + \sum_{j=1,3,5,7} f(j\Delta)\, e^{-i2\pi kj/N} \tag{9.80a}$$

$$= \sum_{j=0,1,2,3} f(2j\Delta) e^{-i2\pi k2j/N} + \sum_{j=0,1,2,3} f((2j+1)\Delta) e^{-i2\pi k(2j+1)/N} \tag{9.80b}$$

$$= \sum_{j=0,1,2,3} f(2j\Delta) e^{-i2\pi kj/(N/2)} + W^k \sum_{j=0,1,2,3} f((2j+1)\Delta)\, e^{-i2\pi kj/(N/2)} \tag{9.80c}$$

$$= g_k^{\rm e} + W^k g_k^{\rm o}, \tag{9.80d}$$

where $W^k = e^{-i2\pi k/N}$. The quantity $g^{\rm e}$ is the Fourier transform of length $N/2$ formed from the even components of the original $f(j\Delta)$; $g^{\rm o}$ is the Fourier transform of length $N/2$ formed from the odd components.

We can continue this decomposition if $N$ is a power of two. That is, we can decompose $g^{\rm e}$ into its $N/4$ even and $N/4$ odd components, $g^{\rm ee}$ and $g^{\rm eo}$, and decompose $g^{\rm o}$ into its $N/4$ even and $N/4$ odd components, $g^{\rm oe}$ and $g^{\rm oo}$. We find

$$g_k = g_k^{\rm ee} + W^{2k} g_k^{\rm eo} + W^k g_k^{\rm oe} + W^{3k} g_k^{\rm oo}. \tag{9.81}$$

One more decomposition leads to

$$\begin{aligned} g_k \;=\; & g_k^{\rm eee} + W^{4k} g_k^{\rm eeo} + W^{2k} g_k^{\rm eoe} + W^{6k} g_k^{\rm eoo} \\ & + W^k g_k^{\rm oee} + W^{5k} g_k^{\rm oeo} + W^{3k} g_k^{\rm ooe} + W^{7k} g_k^{\rm ooo}. \end{aligned} \tag{9.82}$$

At this stage each of the Fourier transforms in (9.82) uses only one data point. We see from (9.78) with $N = 1$ that the value of each of these Fourier transforms, $g_k^{\rm eee}$, $g_k^{\rm eeo}, \ldots$, is equal to the value of $f$ at the corresponding data point. Note that for $N = 8$, we have performed $3 = \log_2 N$ decompositions. In general, we would perform $\log_2 N$ decompositions.

There are two steps to the FFT. First, we reorder the components so that they appear in the order given in (9.82). This step makes the subsequent calculations easier to organize. To see how to do the reordering, we rewrite (9.82) using the values of $f$:

$$\begin{aligned} g_k \;=\; & f(0) + W^{4k} f(4\Delta) + W^{2k} f(2\Delta) + W^{6k} f(6\Delta) \\ & + W^k f(\Delta) + W^{5k} f(5\Delta) + W^{3k} f(3\Delta) + W^{7k} f(7\Delta). \end{aligned} \tag{9.83}$$

We use a trick to obtain the ordering in (9.83) from the original order $f(0\Delta)$, $f(1\Delta)$, ..., $f(7\Delta)$. Part of the trick is to refer to each $g$ in (9.82) by a string of "e" and "o" characters. We assign 0 to "e" and 1 to "o" so that each string represents the binary representation of a number. If we reverse the order of the representation, that is, set 110 to 011, we obtain the value of $f$ we want. For example, the fifth term in (9.82) contains $g^{\mathrm{oee}}$, corresponding to the binary number 100. The reverse of this number is 001, which equals 1 in decimal notation, and hence the fifth term in (9.83) contains the function $f(1\Delta)$. Convince yourself that this bit reversal procedure works for the other seven terms.

The first step in the FFT algorithm is to use this bit reversal procedure on the original array representing the data. In the next step this array is replaced by its Fourier transform. If you want to save your original data, it is necessary to first copy the data to another array before passing the array to a FFT implementation. The `SimpleFFT` class implements the Danielson-Lanczos algorithm using three loops. The outer loop runs over $\log_2 N$ steps. For each of these steps, $N$ calculations are performed in the two inner loops. As can be seen in Listing 9.13, in each pass through the innermost loop each element of the array `g` is updated once by the quantity `temp` formed from a power of $W$ multiplied by the current value of an appropriate element of `g`. The power of $W$ used in `temp` is changed after each pass through the innermost loop. The power of the FFT algorithm is that we do not separately multiply each $f(j\Delta)$ by the appropriate power of $W$. Instead, we first take pairs of $f(j\Delta)$ and multiply them by an appropriate power of $W$ to create new values for the array `g`. Then we repeat this process for pairs of the new array elements (each array element now contains four of the $f(j\Delta)$). We repeat this process until each array element contains a sum of all $N$ values of $f(j\Delta)$ with the correct powers of $W$ multiplying each term to form the Fourier transform.

Listing 9.13: A simple implementation of the FFT algorithm.

```java
package org.opensourcephysics.sip.ch09;
public class SimpleFFT {
   public static void transform(double[] real, double[] imag) {
      int N = real.length;
      int pow = 0;
      while(N/2>0) {
         if(N%2==0) { // N should be even
            pow++;
            N /= 2;
         } else {
            throw new IllegalArgumentException("Number of points in this FFT implementation r
         }
      }
      int N2 = N/2;
      int jj = N2;
      // rearrange input according to bit reversal
      for(int i = 1;i<N-1;i++) {
         if(i<jj) {
            double tempRe = real[jj];
            double tempIm = imag[jj];
            real[jj] = real[i];
            imag[jj] = imag[i];
            real[i] = tempRe;
            imag[i] = tempIm;
         }
         int k = N2;
         while(k<=jj) {
            jj = jj-k;
            k = k/2;
         }
         jj = jj+k;
      }
      jj = 1;
      for(int p = 1;p<=pow;p++) {
         int inc = 2*jj;
         double
            wp1 = 1, wp2 = 0;
         double theta = Math.PI/jj;
         double
            cos = Math.cos(theta), sin = -Math.sin(theta);
         for(int j = 0;j<jj;j++) {
            for(int i = j;i<N;i += inc) {
               // calculate the transform of 2^p
               int ip = i+jj;
               double tempRe = wp1*real[ip]-wp2*imag[ip];
               double tempIm = wp2*real[ip]+wp1*imag[ip];
               real[ip] = real[i]-tempRe;
               imag[ip] = imag[i]-tempIm;
               real[i] = real[i]+tempRe;
               imag[i] = imag[i]+tempIm;
            }
            double temp = wp1;
            wp1 = wp1*cos-wp2*sin;
            wp2 = temp*sin+wp2*cos;
         }
         jj = inc;
      }
   }
}
```

**Exercise 9.41.** Testing the FFT algorithm

1. Test the `SimpleFFT` class for $N = 8$ by going through the code by hand and showing that the class reproduces (9.83).

2. Display the Fourier coefficients of random real values of $f(j\Delta)$ for $N = 8$ using both `SimpleFFT` and the direct computation of the Fourier coefficients based on (9.34). Compare the two sets of data to insure that there are no errors in `SimpleFFT`. Repeat for a random collection of complex data points.

3. Modify the `SimpleFFT` class to compute the inverse Fourier transform defined by (9.37). The inverse Fourier transform of a Fourier transformed data set should be the original data set.

4. Compute the CPU time as a function of $N$ for $N = 16$, 64, 256, and 1024 for the `SimpleFFT` algorithm and the direct computation. You can use the `currentTimeMillis` method in `System` class to record the time.

   ```java
   int n = 10;
   long startTime = System.currentTimeMillis();
   for(int i=0; i<n; i++) { // average for better results
      fft.transform();
   }
   long endTime = System.currentTimeMillis();
   System.out.println("time/FFT = "+((endTime-startTime)/n));
   ```

   Verify that the dependence on $N$ is what you expect.

5. Compare the CPU time as a function of $N$ for the `SimpleFFT` class and the `FFT` class in the numerics package.

6. Compare the CPU time for the `FFT` class in the numerics package using two slightly different values of $N$ such that one value is a power of two and the other is not.

## Appendix 9C: Plotting Scalar Fields

Imagine a plate that is heated at an interior point and cooled along its edges. In principle, the temperature of this plate can be measured at every point. A scalar quantity, such as temperature, pressure, or light intensity, that is defined throughout a region of space is known as a *scalar field*. The Open Source Physics library contains a number of tools that help us visualize two-dimensional scalar fields. A more complete description of two-dimensional visualization tools is available in *Open Source Physics User's Guide*.

An image in which pixels are color coded can be used to visualize a scalar field. The frames package defines a `RasterFrame` class that makes this process easy and efficient if the scalar field can be represented by integers from 0 to 255. The following program shows how such a `RasterFrame` is used.

Listing 9.14: A scalar field visualization using a raster frame.

```java
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.frames.RasterFrame;

public class RasterFrameApp {
   public static void main(String[] args) {
      RasterFrame frame = new RasterFrame("x", "y", "Raster Frame");
      frame.setPreferredMinMax(-10, 10, -10, 10);
      // generate random data
      int nx = 256, ny = 256;
      int[][] data = new int[nx][ny];
      for(int i = 0;i<nx;i++) {
         for(int j = 0;j<ny;j++) {
            data[i][j] = (int) (255*Math.random());
         }
```

```
      }
      frame.setAll(data);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

After the scalar field's values are calculated, the raster's pixels are set using the `setBlock` method. Note that the $[0,0]$ array element maps to the lower left hand pixel. Because the image raster's mapping has been optimized for speed, the image cannot be resized. The on-screen image size, in pixels, always matches the array size. Listing 9.11 uses a raster frame to display a Fraunhofer diffraction pattern.

Although the `RasterFrame` makes it easy to work with integer-based data, it lacks the flexibility for more advanced visualizations. It is, for example, unsuitable if the array size is small because the image it too small or if the dynamic range of the scalar field is too large. The `Scalar2DFrame` class overcomes these limitations. Using a `Scalar2DFrame` allows us to view the data using different representations such as contour plots and three-dimensional surface plots. Listing 9.15 shows a `RasterFrame` being used to visualize the function $f(x, y) = xy$.

Listing 9.15: A scalar field test program.

```
package org.opensourcephysics.sip.ch09;
import org.opensourcephysics.frames.Scalar2DFrame;

public class Scalar2DFrameApp {
   final static int SIZE = 16; // array size

   public static void main(String[] args) {
      Scalar2DFrame frame = new Scalar2DFrame("x", "y", "Scalar Field");
      double[][] data = new double[16][16];
      frame.setAll(data, -10, 10, -10, 10);
      // generate sample data
      for(int i = 0;i<SIZE;i++) {
         double x = frame.indexToX(i);
         for(int j = 0;j<SIZE;j++) {
            double y = frame.indexToY(j);
            data[i][j] = x*y;
         }
      }
      frame.setAll(data);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
   }
}
```

**Exercise 9.42.** Scalar field visualization

Run the scalar field test program and describe the various types of visualizations available under the Tools menu. Which visualizations give respectable representations if the grid is small? What is the maximum grid size that can be used with each type of visualization and still give acceptable performance on your computer?

## References and Suggestions for Further Reading

David C. Champeney, *Fourier Transforms and Their Physical Applications*, Academic Press (1973).

James B. Cole, Rudolph A. Krutar, Susan K. Numrich, and Dennis B. Creamer, "Finite-difference time-domain simulations of wave propagation and scattering as a research and educational tool," Computers in Physics **9**, 235–239 (1995).

Frank S. Crawford, *Waves*, Berkeley Physics Course, Vol. 3, McGraw-Hill (1968). A delightful book on waves of all types. The home experiments are highly recommended. One observation of wave phenomena equals many computer demonstrations.

Paul DeVries, *A First Course in Computational Physics*, John Wiley & Sons (1994). Part of our discussion of the wave equation is based on Chapter 7. There also are good sections on the numerical solution of other partial differential equations, Fourier transforms, and the FFT.

N. A. Dodd, "Computer simulation of diffraction patterns," Phys. Educ. **18**, 294–299 (1983).

P. G. Drazin and R. S. Johnson, *Solitons: An Introduction*, Cambridge University Press (1989). This book focuses on analytical solutions to the Korteweg-de Vries equation which has soliton solutions.

Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics*, Vol. 1, Addison-Wesley (1963). Chapters relevant to wave phenomena include Chapters 28–30 and Chapter 33.

A. P. French, *Vibrations and Waves*, W. W. Norton & Co. (1971). An introductory level text that emphasizes mechanical systems.

Robert Guenther, *Modern Optics*, Vol. 1, Wiley (1990). Chapter 6 discusses Fourier analysis and Chapters 9–12 apply Fourier analysis to the study of Fraunhofer and Fresnel diffraction and holography.

Eugene Hecht, *Optics*, fourth edition, Addison-Wesley (2002). An intermediate level optics text that emphasizes wave concepts.

Akira Hirose and Karl E. Lonngren, *Introduction to Wave Phenomena*, John Wiley & Sons (1985). An intermediate level text that treats the general properties of waves in various contexts.

Amy Kolan, Barry Cipra, and Bill Titus, "Exploring localization in nonperiodic systems," Computers in Physics **9** (4), 387–395 (1995). An elementary discussion of how to solve the problem of a chain of coupled oscillators with disorder using transfer matrices.

J. F. James, *A Student's Guide to Fourier Transforms*, second edition, Cambridge University Press (2002).

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, second edition, Cambridge University Press (1992). See Chapter 12 for a discussion of the fast Fourier transform.

Iain G. Main, *Vibrations and Waves in Physics*, Cambridge University Press (1993). See Chapter 12 for a discussion of a chain of coupled oscillators.

Masud Mansuripur, *Classical Optics and its Applications*, Cambridge (2002). See Chapter 2 for a discussion of Fourier optics.

Timothy J. Rolfe, Stuart A. Rice, and John Dancz, "A numerical study of large amplitude motion on a chain of coupled nonlinear oscillators," J. Chem. Phys. **70**, 26–33 (1979). Problem 9.30 is based on this paper.

Garrison Sposito, *An Introduction to Classical Dynamics*, John Wiley & Sons (1976). A good discussion of the coupled harmonic oscillator problem is given in Chapter 6.

William J. Thompson, *Computing for Scientists and Engineers*, John Wiley & Sons (1992). See Chapters 9 and 10 for a discussion of Fourier transform methods.

Michael L. Williams and Humphrey J. Maris, "Numerical study of phonon localization in disordered systems," Phys. Rev. B **31**, 4508–4515 (1985). The authors consider the normal modes of a two-dimensional system of coupled oscillators with random masses. The idea of using mechanical resonance to extract the normal modes is the basis of a new numerical method for finding the eigenmodes of large lattices. See Kousuke Yukubo, Tsuneyoshi Nakayama, and Humphrey J. Maris, "Analysis of a new method for finding eigenmodes of very large lattice systems," J. Phys. Soc. Japan **60**, 3249 (1991).