## Chapter 2 Sound Waves

### 2.1 Concepts

#### 2.1.1 Sound Waves, Sine Waves, and Harmonic Motion

Working with digital sound begins with an understanding of sound as a physical phenomenon. The sounds we hear are the result of vibrations of objects – for example, the human vocal chords, or the metal strings and wooden body of a guitar. In general, without the influence of a specific sound vibration, air molecules move around randomly. A vibrating object pushes against the randomly-moving air molecules in the vicinity of the vibrating object, causing them first to...
crowd together and then to move apart. The alternate crowding together and moving apart of these molecules in turn affects the surrounding air pressure. The air pressure around the vibrating object rises and falls in a regular pattern, and this fluctuation of air pressure, propagated outward, is what we hear as sound.

Sound is often referred to as a wave, but we need to be careful with the commonly-used term “sound wave,” as it can lead to a misconception about the nature of sound as a physical phenomenon. On the one hand, there’s the physical wave of energy passed through a medium as sound travels from its source to a listener. (We’ll assume for simplicity that the sound is traveling through air, although it can travel through other media.) Related to this is the graphical view of sound, a plot of air pressure amplitude at a particular position in space as it changes over time. For single-frequency sounds, this graph takes the shape of a “wave,” as shown in Figure 2.1. More precisely, a single frequency sound can be expressed as a sine function and graphed as a sine wave (as we’ll describe in more detail later). Let’s see how these two things are related.

First, consider a very simple vibrating object – a tuning fork. When the tuning fork is struck, it begins to move back and forth. As the prong of the tuning fork vibrates outward (in Figure 2.2), it pushes the air molecules right next to it, which results in a rise in air pressure corresponding to a local increase in air density. This is called compression. Now, consider what happens when the prong vibrates inward. The air molecules have more room to spread out again, so the air pressure beside the tuning fork falls. The spreading out of the molecules is called decompression or rarefaction. A wave of rising and falling air pressure is transmitted to the listener’s ear. This is the physical phenomenon of sound, the actual sound wave.
Assume that a tuning fork creates a single frequency wave. Such a sound wave can be graphed as a sine wave, as illustrated in Figure 2.1. An incorrect understanding of this graph would be to picture air molecules going up and down as they travel across space from the place in which the sound originates to the place in which it is heard. This would be as if a particular molecule starts out where the sound originates and ends up in the listener’s ear. This is not what is being pictured in a graph of a sound wave. It is the energy, not the air molecules themselves, that is being transmitted from the source of a sound to the listener’s ear. If the wave in Figure 2.1 is intended to depict a single frequency sound wave, then the graph has time on the x-axis (the horizontal axis) and air pressure amplitude on the y-axis. As described above, the air pressure rises and falls. For a single frequency sound wave, the rate at which it does this is regular and continuous, taking the shape of a sine wave.

Thus, the graph of a sound wave is a simple sine wave only if the sound has only one frequency component in it – that is, just one pitch. Most sounds are composed of multiple frequency components – multiple pitches. A sound with multiple frequency components also can be represented as a graph which plots amplitude over time; it’s just a graph with a more complicated shape. For simplicity, we sometimes use the term “sound wave” rather than “graph of a sound wave” for such graphs, assuming that you understand the difference between the physical phenomenon and the graph representing it.

The regular pattern of compression and rarefaction described above is an example of harmonic motion, also called harmonic oscillation. Another example of harmonic motion is a spring dangling vertically. If you pull on the bottom of the spring, it will bounce up and down in a regular pattern. Its position – that is, its displacement from its natural resting position – can be
graphed over time in the same way that a sound wave’s air pressure amplitude can be graphed over time. The spring’s position increases as the spring stretches downward, and it goes to negative values as it bounces upwards. The speed of the spring’s motion slows down as it reaches its maximum extension, and then it speeds up again as it bounces upwards. This slowing down and speeding up as the spring bounces up and down can be modeled by the curve of a sine wave. In the ideal model, with no friction, the bouncing would go on forever. In reality, however, friction causes a damping effect such that the spring eventually comes to rest. We’ll discuss damping more in a later chapter.

Now consider how sound travels from one location to another. The first molecules bump into the molecules beside them, and they bump into the next ones, and so forth as time goes on. It’s something like a chain reaction of cars bumping into one another in a pile-up wreck. They don’t all hit each other simultaneously. The first hits the second, the second hits the third, and so on. In the case of sound waves, this passing along of the change in air pressure is called sound wave propagation. The movement of the air molecules is different from the chain reaction pile up of cars, however, in that the molecules vibrate back and forth. When the molecules vibrate in the direction opposite of their original direction, the drop in air pressure amplitude is propagated through space in the same way that the increase was propagated.

Be careful not to confuse the speed at which a sound wave propagates and the rate at which the air pressure amplitude changes from highest to lowest. The speed at which the sound is transmitted from the source of the sound to the listener of the sound is the speed of sound. The rate at which the air pressure changes at a given point in space – i.e., vibrates back and forth – is the frequency of the sound wave. You may understand this better through the following analogy. Imagine that you’re watching someone turn a flashlight on and off, repeatedly, at a certain fixed rate in order to communicate a sequence of numbers to you in binary code. The image of this person is transmitted to your eyes at the speed of light, analogous to the speed of sound. The rate at which the person is turning the flashlight on and off is the frequency of the communication, analogous to the frequency of a sound wave.

The above description of a sound wave implies that there must be a medium through which the changing pressure propagates. We’ve described sound traveling through air, but sound also can travel through liquids and solids. The speed at which the change in pressure propagates is the speed of sound. The speed of sound is different depending upon the medium in which sound is transmitted. It also varies by temperature and density. The speed of sound in air is approximately 1130 ft/s (or 344 m/s). Table 2.1 shows the approximate speed in other media.

<table>
<thead>
<tr>
<th>Medium</th>
<th>Speed of sound in m/s</th>
<th>Speed of sound in ft/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>air (20° C, which is 68° F)</td>
<td>344</td>
<td>1130</td>
</tr>
<tr>
<td>water (just above 0° C, which is 32° F)</td>
<td>1410</td>
<td>4626</td>
</tr>
<tr>
<td>steel</td>
<td>5100</td>
<td>16,700</td>
</tr>
<tr>
<td>lead</td>
<td>1210</td>
<td>3970</td>
</tr>
<tr>
<td>glass</td>
<td>approximately 4000 (depending on type of glass)</td>
<td>approximately 13,200</td>
</tr>
</tbody>
</table>

Table 2.1 The speed of sound in various media
For clarity, we’ve thus far simplified the picture of how sound propagates. Figure 2.2 makes it look as though there’s a single line of sound going straight out from the tuning fork and arriving at the listener’s ear. In fact, sound radiates out from a source at all angles in a sphere. Figure 2.3 shows a top-view image of a real sound radiation pattern, generated by software that uses sound dispersion data, measured from an actual loudspeaker, to predict how sound will propagate in a given three-dimensional space. In this case, we’re looking at the horizontal dispersion of the loudspeaker. Colors are used to indicate the amplitude of sound, going highest to lowest from red to yellow to green to blue. The figure shows that the amplitude of the sound is highest in front of the loudspeaker and lowest behind it. The simplification in Figure 2.2 suffices to give you a basic concept of sound as it emanates from a source and arrives at your ear. Later, when we begin to talk about acoustics, we’ll consider a more complete picture of sound waves.

![Figure 2.3 Loudspeaker viewed from top with sound waves radiating at multiple angles](image)

Sound waves are passed through the ear canal to the eardrum, causing vibrations which pass to little hairs in the inner ear. These hairs are connected to the auditory nerve, which sends the signal onto the brain. The rate of a sound vibration – its frequency – is perceived as its pitch by the brain. The graph of a sound wave represents the changes in air pressure over time resulting from a vibrating source. To understand this better, let’s look more closely at the concept of frequency and other properties of sine waves.

### 2.1.2 Properties of Sine Waves

We assume that you have some familiarity with sine waves from trigonometry, but even if you don’t, you should be able to understand some basic concepts of this explanation.

A sine wave is a graph of a sine function $y = \sin(x)$. In the graph, the x-axis is the horizontal axis, and the y-axis is the vertical axis. A graph or phenomenon that takes the shape of a sine wave – oscillating up and down in a regular, continuous manner – is called a **sinusoid**.
In order to have the proper terminology to discuss sound waves and the corresponding sine functions, we need to take a little side trip into mathematics. We’ll first give the sine function as it applies to sound, and then we’ll explain the related terminology.

A single-frequency sound wave with frequency $f$, maximum amplitude $A$, and phase $\theta$ is represented by the sine function

$$y = Asin(2\pi fx + \theta)$$

where $x$ is time and $y$ is the amplitude of the sound wave at moment $x$.

Equation 2.1

Single-frequency sound waves are sinusoidal waves. Although pure single-frequency sound waves do not occur naturally, they can be created artificially by computer means. Naturally occurring sound waves are combinations of frequency components, as we’ll discuss later in this chapter.

The graph of a sound wave is repeated Figure 2.4 with some of its parts labeled. The amplitude of a wave is its $y$ value at some moment in time given by $x$. If we’re talking about a pure sine wave, then the wave’s amplitude, $A$, is the highest $y$ value of the wave. We call this highest value the crest of the wave. The lowest value of the wave is called the trough. When we speak of the amplitude of the sine wave related to sound, we’re referring essentially to the change in air pressure caused by the vibrations that created the sound. This air pressure, which changes over time, can be measured in Newtons/meter$^2$ or, more customarily, in decibels (abbreviated dB), a logarithmic unit explained in detail in Chapter 4. Amplitude is related to perceived loudness. The higher the amplitude of a sound wave, the louder it seems to the human ear.

In order to define frequency, we must first define a cycle. A cycle of a sine wave is a section of the wave from some starting position to the first moment at which it returns to that same position after having gone through its maximum and minimum amplitudes. Usually, we choose the starting position to be at some position where the wave crosses the x-axis, or zero crossing, so the cycle would be from that position to the next zero crossing where the wave starts to repeat, as shown in Figure 2.4.
The frequency of a wave, \( f \), is the number of cycles per unit time, customarily the number of cycles per second. A unit that is used in speaking of sound frequency is Hertz, defined as 1 cycle/second, and abbreviated Hz. In Figure 2.4, the time units on the x-axis are seconds. Thus, the frequency of the wave is 6 cycles/0.0181 seconds = 331 Hz. Henceforth, we’ll use the abbreviation s for seconds and ms for milliseconds.

Frequency is related to pitch in human perception. A single frequency sound is perceived as a single pitch. For example, a sound wave of 440 Hz sounds like the note A on a piano (just above middle C). Humans hear in a frequency range of approximately 20 Hz to 20,000 Hz. The frequency ranges of most musical instruments fall between about 50 Hz and 5000 Hz. The range of frequencies that an individual can hear varies with age and other individual factors.

The period of a wave, \( T \), is the time it takes for the wave to complete one cycle, measured in s/cycle. Frequency and period have an inverse relationship, given below.

\[
\text{Let the frequency of a sine wave be } f \text{ and the period of a sine wave be } T. \text{ Then}
\]
\[
f = \frac{1}{T}
\]
\[
\text{and}
\]
\[
T = \frac{1}{f}
\]

Equation 2.2

The period of the wave in Figure 2.4 is about three milliseconds per cycle. A 440 Hz wave (which has a frequency of 440 cycles/s) has a period of 1 s/440 cycles, which is about 0.00227 s/cycle. There are contexts in which it is more convenient to speak of period only in units of time, and in these contexts the "per cycle" can be omitted as long as units are handled consistently for a particular computation. With this in mind, a 440 Hz wave would simply be said to have a period of 2.27 milliseconds.

The phase of a wave, \( \theta \), is its offset from some specified starting position at \( x = 0 \). The sine of 0 is 0, so the blue graph in Figure 2.5 represents a sine function with no phase offset. However, consider a second sine wave with exactly the same frequency and amplitude, but displaced in the positive or negative direction on the x-axis relative to the first, as shown in Figure 2.5. The extent to which two waves have a phase offset relative to each other can be measured in degrees. If one sine wave is offset a full cycle from another, it has a 360 degree offset (denoted 360°); if it is offset a half cycle, it has a 180° offset; if it is offset a
quarter cycle, it has a 90° offset, and so forth. In Figure 2.5, the red wave has a 90° offset from the blue. Equivalently, you could say it has a 270° offset, depending on whether you assume it is offset in the positive or negative x direction.

![Figure 2.5 Two sine waves with the same frequency and amplitude but different phases](image)

**Wavelength,** $\lambda$, is the distance that a single frequency wave propagates in space as it completes one cycle. Another way to say this is that wavelength is the distance between a place where the air pressure is at its maximum and a neighboring place where it is at its maximum. Distance is not represented on the graph of a sound wave, so we cannot directly observe the wavelength on such a graph. Instead, we have to consider the relationship between the speed of sound and a particular sound wave’s period. Assume that the speed of sound is 1130 ft/s. If a 440 Hz wave takes 2.27 milliseconds to complete a cycle, then the position of maximum air pressure travels $1\ cycle \times \left(\frac{0.00227 \ s}{cycle}\right) \times \left(\frac{1130 \ ft}{s}\right)$ in one wavelength, which is 2.57 ft. This relationship is given more generally in the equation below.

<table>
<thead>
<tr>
<th>Let the frequency of a sine wave representing a sound be $f$, the period be $T$, the wavelength be $\lambda$, and the speed of sound be $c$. Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda = cT$</td>
</tr>
<tr>
<td>or equivalently</td>
</tr>
<tr>
<td>$\lambda = c / f$</td>
</tr>
</tbody>
</table>

*Equation 2.3*

![Figure 2.6 Wavelength](image)

### 2.1.3 Longitudinal and Transverse Waves

Sound waves are **longitudinal waves**. In a longitudinal wave, the displacement of the medium is parallel to the direction in which the wave propagates. For sound waves in air, air molecules are oscillating back and forth and propagating their energy in the same direction as their motion. You can picture a more concrete example if you remember the slinky toy of your childhood. If you and a friend lay a slinky along the floor and pull and push it back and forth, you create a
longitudinal wave. The coils that make up the slinky are moving back and forth horizontally, in the same direction in which the wave propagates. The bouncing of a spring that is dangled vertically amounts to the same thing – a longitudinal wave.

In contrast, in a transverse wave, the displacement of the medium is perpendicular to the direction in which the wave propagates. A jump rope shaken up and down is an example of a transverse wave. We call the quick shake that you give to the jump rope an impulse – like imparting a “bump” to the rope that propagates to the opposite end. The rope moves up and down, but the wave propagates from side to side, from one end of the rope to another. (You could also use your slinky to create a transverse wave, flipping it up and down rather than pushing and pulling it horizontally.)

2.1.4 Resonance

2.1.4.1 Resonance as Harmonic Frequencies

Have you ever heard someone use the expression, “That resonates with me”? A more informal version of this might be “That rings my bell.” What they mean by these expressions is that an object or event stirs something essential in their nature. This is a metaphoric use of the concept of resonance.

Resonance is an object’s tendency to vibrate or oscillate at a certain frequency that is basic to its nature. These vibrations can be excited in the presence of a stimulating force – like the ringing of a bell – or even in the presence of a frequency that sets it off – like glass shattering when just the right high-pitched note is sung. Musical instruments have natural resonant frequencies. When they are plucked, blown into, or struck, they vibrate at these resonant frequencies and resist others.

Resonance results from an object’s shape, material, tension, and other physical properties. An object with resonance – for example, a musical instrument – vibrates at natural resonant frequencies consisting of a fundamental frequency and the related harmonic
frequencies, all of which give an instrument its characteristic sound. The fundamental and harmonic frequencies are also referred to as the **partials**, since together they make up the full sound of the resonating object. The harmonic frequencies beyond the fundamental are called **overtones**. These terms can be slightly confusing. The fundamental frequency is the **first harmonic** because this frequency is one times itself. The frequency that is twice the fundamental is called the **second harmonic** or, equivalently, the first overtone. The frequency that is three times the fundamental is called the **third harmonic** or second overtone, and so forth. The number of harmonic frequencies depends upon the properties of the vibrating object.

One simple way to understand the sense in which a frequency might be natural to an object is to picture pushing a child on a swing. If you push a swing when it is at the top of its arc, you’re pushing it at its resonant frequency, and you’ll get the best effect with your push. Imagine trying to push the swing at any other point in the arc. You would simply be fighting against the natural flow. Another way to illustrate resonance is by means of a simple transverse wave, as we’ll show in the next section.

### 2.1.4.2 Resonance of a Transverse Wave

We can observe resonance in the example of a simple transverse wave that results from sending an impulse along a rope that is fixed at both ends. Imagine that you’re jerking the rope upward to create an impulse. The widest upward bump you could create in the rope would be the entire length of the rope. Since a wave consists of an upward movement followed by a downward movement, this impulse would represent half the total wavelength of the wave you’re transmitting. The full wavelength, twice the length of the rope, is conceptualized in Figure 2.9. This is the **fundamental wavelength** of the fixed-end transverse wave. The fundamental wavelength (along with the speed at which the wave is propagated down the rope) defines the fundamental frequency at which the shaken rope resonates.

If \( L \) is the length of a rope fixed at both ends, then \( \lambda \) is the fundamental wavelength of the rope, given by

\[
\lambda = 2L
\]

**Equation 2.4**

![Figure 2.9 Full wavelength of impulse sent through fixed-end rope](image)

Now imagine that you and a friend are holding a rope between you and shaking it up and down. It’s possible to get the rope into a state of vibration where there are stationary points and other points between them where the rope vibrates up and down, as shown in Figure 2.10. This
is called a **standing wave**. In order to get the rope into this state, you have to shake the rope at a resonant frequency. A rope can vibrate at more than one resonant frequency, each one giving rise to a specific **mode** – i.e., a pattern or shape of vibration. At its fundamental frequency, the whole rope is vibrating up and down (mode 1). Shaking at twice that rate excites the next resonant frequency of the rope, where one half of the rope is vibrating up while the other is vibrating down (mode 2). This is the second harmonic (first overtone) of the vibrating rope. In the third harmonic, the “up and down” vibrating areas constitute one third of the rope’s length each.

![Figure 2.10 Vibrating a rope at a resonant frequency](image)

This phenomenon of a standing wave and resonant frequencies also manifests itself in a musical instrument. Suppose that instead of a rope, we have a guitar string fixed at both ends. Unlike the rope that is shaken at different rates of speed, guitar strings are plucked. This pluck, like an impulse, excites multiple resonant frequencies of the string at the same time, including the fundamental and any harmonics. The fundamental frequency of the guitar string results from the length of the string, the tension with which it is held between two fixed points, and the physical material of the string.

The harmonic modes of a string are depicted in Figure 2.11. The top picture in the figure illustrates the string vibrating according to its fundamental frequency. The wavelength $\lambda$ of the fundamental frequency is two times the length of the string $L$.

The second picture from the top in Figure 2.11 shows the second harmonic frequency of the string. Here, the wavelength is equal to the length of the string, and is twice the frequency of the fundamental. In the third harmonic frequency, the wavelength is $2/3$ times the length of the string, and three times the frequency of the fundamental. In the fourth harmonic frequency, the wavelength is $1/2$ times the length of the string, and four times the frequency of the fundamental. More harmonic frequencies could exist beyond this depending on the type of string.
Like a rope held at both ends, a guitar string fixed at both ends creates a standing wave as it vibrates according to its resonant frequencies. In a standing wave, there exist points in the wave that don’t move. These are called the nodes, as pictured in Figure 2.11. The antinodes are the high and low points between which the string vibrates. This is hard to illustrate in a still image, but you should imagine the wave as if it’s anchored at the nodes and swinging back and forth between the nodes with high and low points at the antinodes.

It’s important to note that this figure illustrates the physical movement of the string, not a graph of a sine wave representing the string’s sound. The string’s vibration is in the form of a transverse wave, where the string moves up and down while the tensile energy of the string propagates perpendicular to the vibration. Sound is a longitudinal wave.

The speed of the wave’s propagation through the string is a function of the tension force on the string, the mass of the string, and the string’s length. If you have two strings of the same length and mass and one is stretched more tightly than another, it will have a higher wave propagation speed and thus a higher frequency. The frequency arises from the properties of the string, including its fundamental wavelength, $2L$, and the extent to which it is stretched.
What is most significant is that you can hear the string as it vibrates at its resonant frequencies. These vibrations are transmitted to a resonant chamber, like a box, which in turn excites the neighboring air molecules. The excitation is propagated through the air as a transfer of energy in a longitudinal sound wave. The frequencies at which the string vibrates are translated into air pressure changes occurring with the same frequencies, and this creates the sound of the instrument. Figure 2.12 shows an example harmonic spectrum of a plucked guitar string. It is clear to see the resonant frequencies of the string, starting with the fundamental and increasing in integer multiples (twice the fundamental, three times the fundamental, etc.). It is interesting to note that not all the harmonics resonate with the same energy. Typically, the magnitude of the harmonics decreases as the frequency increases, where the fundamental is the most dominant. Also keep in mind that the harmonic spectrum and strength of the individual harmonics can vary somewhat depending on how the resonator is excited. How hard a string is plucked, or whether it is bowed, or struck with a wooden stick or soft mallet, can have an effect on the way the object resonates and sounds.

![Figure 2.12 Example harmonic spectrum of a plucked guitar string](image)

### 2.1.4.3 Resonance of a Longitudinal Wave

Not all musical instruments are made from strings. Many are constructed from cylindrical spaces of various types, like those found in clarinets, trombones, and trumpets. Let’s think of these cylindrical spaces in the abstract as a pipe.

A significant difference between the type of wave created from blowing air into a pipe and a wave created by plucking a string is that the wave in the pipe is longitudinal while the wave on the string is transverse. When air is blown into the end of a pipe, air pressure changes are propagated through the pipe to the opposite end. The direction in which the air molecules vibrate is parallel to the direction in which the wave propagates.

Consider first a pipe that is open at both ends. Imagine that a sudden pulse of air is sent through one of the open ends of the pipe. The air is at atmospheric pressure at both open ends of the pipe. As the air is blown into the end, the air pressure rises, reaching its maximum at the middle and falling to its minimum again at the other open end. This is shown in top part of Figure 2.13. The figure shows that the resulting fundamental wavelength of sound produced in the pipe is twice the length of the pipe (similar to the guitar string fixed at both ends).
The situation is different if the pipe is closed at the end opposite to the one into which it is blown. In this case, air pressure rises to its maximum at the closed end. The bottom part of Figure 2.13 shows that in this situation, the closed end corresponds to the crest of the fundamental wavelength. Thus, the fundamental wavelength is four times the length of the pipe.

Because the wave in the pipe is traveling through air, it is simply a sound wave, and thus we know its speed – approximately 1130 ft/s. With this information, we can calculate the fundamental frequency of both closed and open pipes, given their length.

Let $L$ be the length of an open pipe, and let $c$ be the speed of sound. Then the fundamental frequency of the pipe is

$$f = \frac{c}{2L}$$

Equation 2.5

Let $L$ be the length of a closed pipe, and let $c$ be the speed of sound. Then the fundamental frequency of the pipe is

$$f = \frac{c}{4L}$$

Equation 2.6

This explanation is intended to shed light on why each instrument has a characteristic sound, called its timbre. The timbre of an instrument is the sound that results from its fundamental frequency and the harmonic frequencies it produces, all of which are integer.
multiples of the fundamental. All the resonant frequencies of an instrument can be present simultaneously. They make up the frequency components of the sound emitted by the instrument. The components may be excited at a lower energy and fade out at different rates, however. Other frequencies contribute to the sound of an instrument as well, like the squeak of fingers moving across frets, the sound of a bow pulled across a string, or the frequencies produced by the resonant chamber of a guitar’s body. Instruments are also characterized by the way their amplitude changes over time when they are plucked, bowed, or blown into. The changes of amplitude are called the **amplitude envelope**, as we’ll discuss in a later section.

Resonance is one of the phenomena that gives musical instruments their characteristic sounds. Guitar strings alone do not make a very audible sound when plucked. However, when a guitar string is attached to a large wooden box with a shape and size that is proportional to the wavelengths of the frequencies generated by the string, the box resonates with the sound of the string in a way that makes it audible to a listener several feet away. Drumheads likewise do not make a very audible sound when hit with a stick. Attach the drumhead to a large box with a size and shape proportional to the diameter of the membrane, however, and the box resonates with the sound of that drumhead so it can be heard. Even wind instruments benefit from resonance. The wooden reed of a clarinet vibrating against a mouthpiece makes a fairly steady and quiet sound, but when that mouthpiece is attached to a tube, a frequency will resonate with a wavelength proportional to the length of the tube. Punching some holes in the tube that can be left open or covered in various combinations effectively changes the length of the tube and allows other frequencies to resonate.

### 2.1.5 Digitizing Sound Waves

In this chapter, we have been describing sound as continuous changes of air pressure amplitude. In this sense, sound is an **analog** phenomenon – a physical phenomenon that could be represented as continuously changing voltages. Computers require that we use a **discrete** representation of sound. In particular, when sound is captured as data in a computer, it is represented as a list of numbers. Capturing sound in a form that can be handled by a computer is a process called **analog-to-digital conversion**, whereby the amplitude of a sound wave is measured at evenly-spaced intervals in time – typically 44,100 times per second, or even more. Details of analog-to-digital conversion are covered in Chapter 5. For now, it suffices to think of digitized sound as a list of numbers. Once a computer has captured sound as a list of numbers, a whole host of mathematical operations can be performed on the sound to change its loudness, pitch, frequency balance, and so forth. We'll begin to see how this works in the following sections.

### 2.2 Applications

#### 2.2.1 Acoustics

In each chapter, we begin with basic concepts in Section 1 and give applications of those concepts in Section 2. One main area where you can apply your understanding of sound waves is in the area of acoustics. "Acoustics" is a large topic, and thus we have devoted a whole chapter to it. Please refer to Chapter 4 for more on this topic.
2.2.2 Sound Synthesis

Naturally occurring sound waves almost always contain more than one frequency. The frequencies combined into one sound are called the sound’s **frequency components**. A sound that has multiple frequency components is a **complex sound wave**. All the frequency components taken together constitute a sound’s **frequency spectrum**. This is analogous to the way light is composed of a spectrum of colors. The frequency components of a sound are experienced by the listener as multiple pitches combined into one sound.

To understand frequency components of sound and how they might be manipulated, we can begin by synthesizing our own digital sound. **Synthesis** is a process of combining multiple elements to form something new. In **sound synthesis**, individual sound waves become one when their amplitude and frequency components interact and combine digitally, electrically, or acoustically. The most fundamental example of sound synthesis is when two sound waves travel through the same air space at the same time. Their amplitudes at each moment in time sum into a composite wave that contains the frequencies of both. Mathematically, this is a simple process of addition.

We can experiment with sound synthesis and understand it better by creating three single-frequency sounds using an audio editing program like Audacity or Adobe Audition. Using the “Generate Tone” feature in Audition, we’ve created three separate sound waves – the first at 262 Hz (middle C on a piano keyboard), the second at 330 Hz (the note E), and the third at 393 Hz (the note G). They’re shown in Figure 2.14, each on a separate track. The three waves can be **mixed down** in the editing software – that is, combined into a single sound wave that has all three frequency components. The mixed down wave is shown on the bottom track.

![Figure 2.14 Three waves mixed down into a wave with three frequency components](image)

In a digital audio editing program like Audition, a sound wave is stored as a list of numbers, corresponding to the amplitude of the sound at each point in time. Thus, for the three audio tones generated, we have three lists of numbers. The mix-down procedure simply adds the corresponding values of the three waves at each point in time, as shown in Figure 2.15. Keep in mind that negative amplitudes (rarefactions) and positive amplitudes (compressions) can cancel each other out.
We’re able to hear multiple sounds simultaneously in our environment because sound waves can be added. Another interesting consequence of the addition of sound waves results from the fact that waves have phases. Consider two sound waves that have exactly the same frequency and amplitude, but the second wave arrives exactly one half cycle after the first—that is, $180^\circ$ out-of-phase, as shown in Figure 2.16. This could happen because the second sound wave is coming from a more distant loudspeaker than the first. The different arrival times result in phase-cancellations as the two waves are summed when they reach the listener's ear. In this case, the amplitudes are exactly opposite each other, so they sum to 0.

2.2.3 Sound Analysis

We showed in the previous section how we can add frequency components to create a complex sound wave. The reverse of the sound synthesis process is sound analysis, which is the determination of the frequency components in a complex sound wave. In the 1800s, Joseph Fourier developed the mathematics that forms the basis of frequency analysis. He proved that
any periodic sinusoidal function, regardless of its complexity, can be formulated as a sum of frequency components. These frequency components consist of a fundamental frequency and the harmonic frequencies related to this fundamental. Fourier's theorem says that no matter how complex a sound is, it's possible to break it down into its component frequencies – that is, to determine the different frequencies that are in that sound, and how much of each frequency component there is.

Fourier analysis begins with the fundamental frequency of the sound – the frequency of the longest repeated pattern of the sound. Then all the remaining frequency components that can be yielded by Fourier analysis – i.e., the harmonic frequencies – are integer multiples of the fundamental frequency. By “integer multiple” we mean that if the fundamental frequency is \( f_0 \), then each harmonic frequency \( f_n \) is equal to \( (n + 1)f_0 \) for some non-negative integer \( n \).

The Fourier transform is a mathematical operation used in digital filters and frequency analysis software to determine the frequency components of a sound. Figure 2.17 shows Adobe Audition’s waveform view and a frequency analysis view for a sound with frequency components at 262 Hz, 330 Hz, and 393 Hz. The frequency analysis view is to the left of the waveform view. The graph in the frequency analysis view is called a frequency response graph or simply a frequency response. The waveform view has time on the x-axis and amplitude on the y-axis. The frequency analysis view has frequency on the x-axis and the magnitude of the frequency component on the y-axis. (See Figure 2.18.) In the frequency analysis view in Figure 2.17, we zoomed in on the portion of the x-axis between about 100 and 500 Hz to show that there are three spikes there, at approximately the positions of the three frequency components. You might expect that there would be three perfect vertical lines at 262, 330, and 393 Hz, but digitized sound is not a perfectly accurate representation of sound. Still, the Fourier transform is accurate enough to be the basis for filters and special effects with sounds.

Aside: "Frequency response" has a number of related usages in the realm of sound. It can refer to a graph showing the relative magnitudes of audible frequencies in a given sound. With regard to an audio filter, the frequency response shows how a filter boosts or attenuates the frequencies in the sound to which it is applied. With regard to loudspeakers, the frequency response is the way in which the loudspeakers boost or attenuate the audible frequencies. With regard to a microphone, the frequency response is the microphone's sensitivity to frequencies over the audible spectrum.
In the example just discussed, the frequencies that are combined in the composite sound never change. This is because of the way we constructed the sound, with three single frequency waves that are held for one second. This sound, overall, is periodic because the pattern created from adding these three component frequencies is repeated over time, as you can see in the bottom of Figure 2.14.

Natural sounds, however, generally change in their frequency components as time passes. Consider something as simple as the word “information.” When you say “information,” your voice produces numerous frequency components, and these change over time. Figure 2.19 shows a recording and frequency analysis of the spoken word “information.” You can see in the frequency analysis view on the left that there are a few high frequency components due to the “f” sound and “sh” in the syllable “tion.”

When you look at the frequency analysis view, don’t be confused into thinking that the x-axis is time. The position of the “hump” on the right part of the graph indicates that there are frequencies around 15,000 Hz, but the frequency analysis graph doesn’t tell you where in time these high frequency components occurred.

A one-note song would not be very interesting. In music and other sounds, pitches – i.e., frequencies – change as time passes. Natural sounds are not periodic in the way that a one-chord sound is. The frequency components in the first second of such sounds are different from the
frequency components in the next second. The upshot of this fact is that for complex non-periodic sounds, you have to analyze frequencies over a specified time period, called a window. When you ask your sound analysis software to provide a frequency analysis, you have to set the window size. The window size in Adobe Audition’s frequency analysis view is called “FFT size.” In the examples above, the window size is set to 65536, indicating that the analysis is done over a span of 65,536 audio samples. The meaning of this window size is explained in more detail in Chapter 7. What is important to know at this point is that there’s a tradeoff between choosing a large window and a small one. A larger window gives higher resolution across the frequency spectrum – breaking down the spectrum into smaller bands – but the disadvantage is that it “blurs” its analysis of the constantly changing frequencies across a larger span of time. A smaller window focuses on what the frequency components are in a more precise, short frame of time, but it doesn’t yield as many frequency bands in its analysis.

2.2.4 Frequency Components of Non-Sinusoidal Waves

In Section 2.1.3, we categorized waves by the relationship between the direction of the medium’s movement and the direction of the wave’s propagation. Another useful way to categorize waves is by their shape – square, sawtooth, and triangle, for example. These waves are easily described in mathematical terms and can be constructed artificially by adding certain harmonic frequency components in the right proportions. You may encounter square, sawtooth, and triangle waves in your work with software synthesizers. Although these waves are non-sinusoidal – i.e., they don’t take the shape of a perfect sine wave – they still can be manipulated and played as sound waves, and they’re useful in simulating the sounds of musical instruments.

A square wave rises and falls regularly between two levels (Figure 2.20, left). A sawtooth wave rises and falls at an angle, like the teeth of a saw (Figure 2.20, center). A triangle wave rises and falls in a slope in the shape of a triangle (Figure 2.20, right). Square waves create a hollow sound that can be adapted to resemble wind instruments. Sawtooth waves can be the basis for the synthesis of violin sounds. A triangle wave sounds very similar to a perfect sine wave, but with more body and depth, making it suitable for simulating a flute or trumpet. The suitability of these waves to simulate particular instruments varies according to the ways in which they are modulated and combined.

Non-sinusoidal waves can be generated by computer-based tools – for example, Reason or Logic, which have built-in synthesizers for simulating musical instruments. Mathematically, non-sinusoidal waveforms are constructed by adding or subtracting harmonic frequencies in various patterns. A perfect square wave, for
example, is formed by adding all the odd-numbered harmonics of a given fundamental frequency, with the amplitudes of these harmonics diminishing as their frequencies increase. The odd-numbered harmonics are those with frequency \( nf \) where \( f \) is the fundamental frequency and \( n \) is a positive odd integer. A sawtooth wave is formed by adding all harmonic frequencies related to a fundamental, with the amplitude of each frequency component diminishing as the frequency increases. If you would like to look at the mathematics of non-sinusoidal waves more closely, see Section 2.3.2.

### 2.2.5 Frequency, Impulse, and Phase Response Graphs

Section 2.2.3 introduces frequency response graphs, showing one taken from Adobe Audition. In fact, there are three interrelated graphs that are often used in sound analysis. Since these are used in this and later chapters, this is a good time to introduce you to these types of graphs. The three types of graphs are impulse response, frequency response, and phase response.

Impulse, frequency, and phase response graphs are simply different ways of storing and graphing the same set of data related to an instance of sound. Each type of graph represents the information in a different mathematical domain. The domains and ranges of the three types of sound graphs are given in Table 2.2.

<table>
<thead>
<tr>
<th>graph type</th>
<th>domain (x-axis)</th>
<th>range (y-axis)</th>
</tr>
</thead>
<tbody>
<tr>
<td>impulse response</td>
<td>time</td>
<td>amplitude of sound at each moment in time</td>
</tr>
<tr>
<td>frequency response</td>
<td>frequency</td>
<td>magnitude of frequency across the audible spectrum of sound</td>
</tr>
<tr>
<td>phase response</td>
<td>phase</td>
<td>phase of frequency across the audible spectrum of sound</td>
</tr>
</tbody>
</table>

Table 2.2 Domains and ranges of impulse, frequency, and phase response graphs

Let’s look at an example of these three graphs, each associated with the same instance of sound. The graphs in the figures below were generated by sound analysis software called Fuzzmeasure Pro, which we’ll use in Section 2 as we talk about how frequencies are analyzed in practice.
The impulse response graph shows the amplitude of the sound wave over time. The data used to draw this graph are produced by a microphone (and associated digitization hardware and software), which samples the amplitude of sound at evenly-spaced intervals of time. The details of this sound sampling process are discussed in detail in Chapter 5. For now, all you need to understand is that when sound is captured and put into a form that can be handled by a computer, it is nothing more than a list of numbers, each number representing the amplitude of sound at a moment in time.

Related to each impulse response graph are two other graphs – a frequency response graph that shows “how much” of each frequency is present in the instance of sound, and a phase
response graph that shows the phase that each frequency component is in. Each of these two graphs covers the audible spectrum. In Section 3, you’ll be introduced to the mathematical process – the Fourier transform – that converts sound data from the time domain to the frequency and phase domain. Applying a Fourier transform to impulse response data – i.e., time represented in the time domain – yields both frequency and phase information from which you can generate a frequency response graph and a phase response graph. The frequency response graph has the magnitude of the frequency on the y-axis on whatever scale is chosen for the graph. The phase response graph has phases ranging from -180° to 180° on the y-axis.

The main points to understand are these:

- A graph is a visualization of data.
- For any given instance of sound, you can analyze the data in terms of time, frequency, or phase, and you can graph the corresponding data.
- These different ways of representing sound – as amplitude of sound over time or as frequency and phase over the audible spectrum – contain essentially the same information.
- The Fourier transform can be used to transform the sound data from one domain of representation to another. The Fourier transform is the basis for processes applied at the user-level in sound measuring and editing software.
- When you work with sound, you look at it and edit it in whatever domain or representation is most appropriate for your purposes at the time. You’ll see this later in examples concerning frequency analysis of live performance spaces, room modes, precedence effect, and so forth.

2.2.6 Ear Testing and Training

If you plan to work in sound, it’s important to know the acuity of your own ears in three areas – the range of frequencies that you’re able to hear, the differences in frequencies that you can detect, and the sensitivity of your hearing to relative time and direction of sounds. A good place to begin is to have your hearing tested by an audiologist to discover the natural frequency response of your ears. If you want to do your own test, you can use a sine wave generator in Logic, Audition, or similar software to step through the range of audible sound frequencies and determine the lowest and highest ones you can hear. The range of human hearing is about 20 Hz to 20,000 Hz, but this varies with individuals and changes as an individual ages.

Not only can you test your ears for their current sensitivity; you also can train your ears to get better at identifying frequency and time differences in sound. Training your ears to recognize frequencies can be done by having someone boost frequency bands, one at a time, in a full-range noise or music signal while you guess which frequency is being boosted. In time, you’ll start “guessing” correctly. Training your ears to recognize time or direction differences requires that someone create two sound waves with location or time offsets and then ask you to discriminate between the two. The ability to identify frequencies and hear subtle differences is
very valuable when working with sound. The learning supplements for this section give sample exercises and worksheets related to ear training.

2.3 Science, Mathematics, and Algorithms

2.3.1 Modeling Sound in Max

Max (distributed by Cycling ‘74 and formerly called Max/MSP/Jitter) is a software environment for manipulating digital audio and MIDI data. The interface for Max is at a high level of abstraction, allowing you to patch together sound objects by dragging icons for them to a window and linking them with patcher cords, similar to the way audio hardware is connected. If you aren’t able to use Max, which is a commercial product, you can try substituting the freeware program Pure Data. We introduce you briefly to Pure Data in Section 2.3.2. In future chapters, we’ll limit our examples to Max because of its highly developed functionality, but PD is a viable free alternative that you might want to try.

Max can be used in two ways. First, it’s an excellent environment for experimenting with sound simply to understand it better. As you synthesize and analyze sound through built-in Max objects and functions, you develop a better understanding of the event-driven nature of MIDI versus the streaming data nature of digital audio, and you see more closely how these kinds of data are manipulated through transforms, effects processors, and the like. Secondly, Max is actually used in the sound industry. When higher level audio processing programs like Logic, Pro Tools, Reason, or Sonar don’t meet the specific needs of audio engineers, they can create specially-designed systems using Max.

Max is actually a combination of two components that can be made to work together. Max allows you to work with MIDI objects, and MSP is designed for digital audio objects. Since we won’t go into depth in MIDI until Chapter 6, we’ll just look at MSP for now.

Let’s try a simple example of MSP to get you started. Figure 2.38 shows an MSP patcher for adding three pure sine waves. A patcher – whether it has Max objects, MSP objects, or both – is a visual representation of a program that operates on sound. The objects are connected by patch cords. These cords run between the inlets and outlets of objects.

One of the first things you’ll probably want to do in MSP is create simple sine waves and listen to them. To be able to work on a patcher, you have to be in edit mode, as opposed to lock mode. You enter edit mode by clicking CTRL-E (or Apple-E) or by clicking the lock icon on the task bar. Once in edit mode, you can click on the + icon on the task bar, which gives you a menu of objects. (We refer you to the Max Help for details on inserting various objects.) The cycle~ object creates a sine wave of whatever frequency you specify.

Notice that MSP objects are distinguished from Max objects by the tilde at the end of their names. This reminds you that MSP objects send audio data streaming continuously through them, while Max objects typically are triggered by and trigger discrete events.

The speaker icon in Figure 2.24 represents the ezdac~ object, which stands for “easy digital to analog conversion,” and sends sound to the sound card to be played. A number object sends the desired frequency to each cycle~ object, which in turn sends the sine wave to a scope~ object (the sine wave graphs) for display. The three frequencies are summed with two consecutive +~ objects, and the resulting complex waveform is displayed.
To understand these objects and functions in more detail, you can right click on them to get an Inspector window, which shows the parameters of the objects. Figure 2.25 shows the inspector for the meter~ object, which looks like an LED and is to the left of the Audio On/Off switch in Figure 2.24. You can change the parameters in the Inspector as you wish. You also can right click on an object, go to the Help menu (Figure 2.26), and access an example patcher that helps you with the selected object. Figure 2.27 shows the Help patcher for the ezdac~ object. The patchers in the Help can be run as programs, opened in Edit mode, and copied and pasted into your own patchers.

Figure 2.24 MSP patcher for adding three sine waves

Figure 2.25 meter~ Inspector in Max
When you create a patcher, you may want to define how it looks to the user in what is called **presentation mode**, a view in which you can hide some of the implementation details of the patcher to make its essential functionality clearer. The example patcher’s presentation mode is shown in Figure 2.28.
2.3.2 Modeling Sound Waves in Pure Data (PD)

Pure Data (aka PD) is a free alternative to Max developed by one of the originators of Max, Miller Puckette, and his collaborators. Like Max, it is a graphical programming environment that links digital audio and MIDI.

The Max program to add sine waves is implemented in PD in Figure 2.29. You can see that the interface is very similar to Max’s, although PD has fewer GUI components and no presentation mode.
2.3.3  Modeling Sound in MATLAB

It's easy to model and manipulate sound waves in MATLAB, a mathematical modeling program. If you learn just a few of MATLAB’s built-in functions, you can create sine waves that represent sounds of different frequencies, add them, plot the graphs, and listen to the resulting sounds. Working with sound in MATLAB helps you to understand the mathematics involved in digital audio processing. In this section, we'll introduce you to the basic functions that you can use for your work in digital sound. This will get you started with MATLAB, and you can explore further on your own. If you aren't able to use MATLAB, which is a commercial product, you can try substituting the freeware program Octave. We introduce you briefly to Octave in Section 2.3.5. In future chapters, we'll limit our examples to MATLAB because it is widely used and has an extensive Signal Processing Toolbox that is extremely useful in sound processing. We suggest Octave as a free alternative that can accomplish some, but not all, of the examples in remaining chapters.

Before we begin working with MATLAB, let’s review the basic sine functions used to represent sound. In the equation \( y = A \sin(2\pi f x + \theta) \), frequency \( f \) is assumed to be measured in Hertz. An equivalent form of the sine function, and one that is often used, is expressed in terms of angular frequency, \( \omega \), measured in units of radians/s rather than Hertz. Since there are \( 2\pi \)
radians in a cycle, and Hz is cycles/s, the relationship between frequency in Hertz and **angular frequency** in radians/s is as follows:

$$\omega = 2\pi f$$  
**Equation 2.7**

We can now give an alternative form for the sine function.

$$y = A\sin(\omega t + \theta)$$  
**Equation 2.8**

In our examples below, we show the frequency in Hertz, but you should be aware of these two equivalent forms of the sine function. MATLAB’s sine function expects angular frequency in Hertz, so $f$ must be multiplied by $2\pi$.

Now let’s look at how we can model sounds with sine functions in MATLAB. Middle C on a piano keyboard has a frequency of approximately 262 Hz. To create a sine wave in MATLAB at this frequency and plot the graph, we can use the `fplot` function as follows:

```matlab
fplot('sin(262*2*pi*t)', [0, 0.05, -1.5, 1.5]);
```

The graph in Figure 2.30 pops open when you type in the above command and hit Enter. Notice that the function you want to graph is enclosed in single quotes. Also, notice that the constant $\pi$ is represented as `pi` in MATLAB. The portion in square brackets indicates the limits of the horizontal and vertical axes. The horizontal axis goes from 0 to 0.05, and the vertical axis goes from $-1.5$ to $1.5$.

![Figure 2.30 262 Hz sine wave](image)

If we want to change the amplitude of our sine wave, we can insert a value for $A$. If $A > 1$, we may have to alter the range of the vertical axis to accommodate the higher amplitude, as in
After multiplying by $A=2$ in the statement above, the top of the sine wave goes to 2 rather than 1.

To change the phase of the sine wave, we add a value $\theta$. Phase is essentially a relationship between two sine waves with the same frequency. When we add $\theta$ to the sine wave, we are creating a sine wave with a phase offset of $\theta$ compared to a sine wave with phase offset of 0. We can show this by graphing both sine waves on the same graph. To do so, we graph the first function with the command

```matlab
fplot('2*sin(262*2*pi*t)', [0, 0.05, -2.5, 2.5]);
```

We then type

```matlab
hold on
```

This will cause all future graphs to be drawn on the currently open figure. Thus, if we type

```matlab
fplot('2*sin(262*2*pi*t+pi)', [0, 0.05, -2.5, 2.5]);
```

we have two phase-offset graphs on the same plot. In Figure 2.31, the 0-phase-offset sine wave is in red and the 180° phase offset sine wave is in blue.

![Figure 2.31 Two sine waves, one offset 180° from the other](image)

Notice that the offset is given in units of radians rather than degrees, 180° being equal to $\pi$ radians.

To change the frequency, we change $\omega$. For example, changing $\omega$ to $440*2*\pi$ gives us a graph of the note A above middle C on a keyboard.

```matlab
fplot('sin(440*2*pi*t)', [0, 0.05, -1.5, 1.5]);
```

The above command gives this graph:
Then with

```matlab
fplot('sin(262*2*pi*t)', [0, 0.05, -1.5, 1.5], 'red');
hold on
```

we get this figure:

The 262 Hz sine wave in the graph is red to differentiate it from the blue 440 Hz wave. The last parameter in the \texttt{fplot} function causes the graph to be plotted in red. Changing the color or line width also can be done by choosing Edit/Figure Properties on the figure, selecting the sine wave, and changing its properties.

We also can add sine waves to create more complex waves, as we did using Adobe Audition in Section 2.2.2. This is a simple matter of adding the functions and graphing the sum, as shown below.

```matlab
figure
fplot('sin(262*2*pi*t)+sin(440*2*pi*t)', [0, 0.05, -2.5, 2.5]);
```

First, we type \texttt{figure} to open a new empty figure (so that our new graph is not overlaid on the currently open figure). We then graph the sum of the sine waves for the notes C and A. The result is this:
We've used the \textit{fplot} function in these examples. This function makes it appear as if the graph of the sine function is continuous. Of course, MATLAB can't really graph a continuous list of numbers, which would be infinite in length. The name MATLAB, in fact, is an abbreviation of "matrix laboratory." MATLAB works with arrays and matrices. In Chapter 5, we'll explain how sound is digitized such that a sound file is just a array of numbers. The \textit{plot} function is the best one to use in MATLAB to graph these values. Here's how this works.

First, you have to declare get an array of values to use as input to a sine function. Let's say that you want one second of digital audio at a sampling rate of 44,100 Hz (i.e., samples/s) (a standard sampling rate). Let's set the values of variables for sampling rate $sr$ and number of seconds $s$, just to remind you for future reference of the relationship between the two.

\begin{verbatim}
sr = 44100;
s = 1;
\end{verbatim}

Now, to give yourself an array of time values across which you can evaluate the sine function, you do the following:

\begin{verbatim}
t = linspace(0, s, sr*s);
\end{verbatim}

This creates an array of $sr$ values, $\left[0, \frac{1}{sr}, \frac{2}{sr}, \frac{3}{sr}, \ldots, s \right]$. Note that when you don't put a semi-colon after a command, the result of the command is displayed on the screen. Thus, without a semi-colon above, you'd see the 44,100 values scroll in front of you.

To evaluate the sine function across these values, you type

\begin{verbatim}
y = sin(2*pi*262*t);
\end{verbatim}

One statement in MATLAB can cause an operation to be done on every element of an array. In this example, $y = sin(2*pi*262*t)$ takes the sine on each element of array $t$ and stores the result in array $y$. To plot the sine wave, you type

\begin{verbatim}
plot(t,y);
\end{verbatim}

Time is on the x-axis, between 0 and 1 second. Amplitude of the sound wave is on the vertical axis, scaled to values between $-1$ and $1$. The graph is too dense for you to see the wave properly. There are three ways you can zoom in. One is by choosing Axes Properties from the
graph's Edit menu and then resetting the range of the horizontal axis. The second way is to type an *axis* command like the following:

```matlab
axis([0 0.1 -2 2]);
```

This displays only the first 1/10 of a second on the horizontal axis, with a range of –2 to 2 on the vertical axis so you can see the shape of the wave better.

You can also ask for a plot of a subset of the points, as follows:

```matlab
plot(t(1:1000),y(1:1000));
```

The above command plots only the first 1000 points from the sine function. Notice that the length of the two arrays must be the same for the *plot* function, and that numbers representing array indices must be positive integers. In general, if you have an array *t* of values and want to look at only the *i*th to the *j*th values, use *t(i:j)*.

An advantage of generating an array of sample values from the sine function is that with that array, you actually can hear the sound. When you send the array to the *wavplay* or *sound* function, you can verify that you've generated one second's worth of the frequency you wanted, middle C. You do this with

```matlab
wavplay(y, sr);
```

or

```matlab
sound(y, sr);
```

The first parameter is an array of sound samples. The second parameter is the sampling rate, which lets the system know how many samples to play per second.

MATLAB has other built-in functions for generating waves of special shapes. We'll go back to using *fplot* for these. For example, we can generate square, sawtooth, and triangular waves with the three commands given below:

```matlab
fplot('square(t)',[0,10*pi,-1.5,1.5]);
```

```
```

```
```

Figure 2.35 Square wave

```matlab
fplot('sawtooth(t)',[0,10*pi]);
```
Figure 2.36 Sawtooth wave

fplot('2*pulstran(t,[0:10],''tripuls'')-1',[0,10]);

Figure 2.37 Triangle wave

(Notice that the tripuls parameter is surrounded by two single quotes on each side.)

This section is intended only to introduce you to the basics of MATLAB for sound manipulation, and we leave it to you to investigate the above commands further. MATLAB has an extensive Help feature which gives you information on the built-in functions.

Each of the functions above can be created “from scratch” if you understand the nature of the non-sinusoidal waves. The ideal square wave is constructed from an infinite sum of odd-numbered harmonics of diminishing amplitude. More precisely, if \( f \) is the fundamental frequency of the non-sinusoidal wave to be created, then a square wave is constructed by the following infinite summation:

\[
\sum_{n=0}^{\infty} \frac{1}{(2n+1)} \sin(2\pi(2n+1)ft)
\]

Let \( f \) be a fundamental frequency. Then a square wave created from this fundamental frequency is defined by the infinite summation

\[
\sum_{n=0}^{\infty} \frac{1}{(2n+1)} \sin(2\pi(2n+1)ft)
\]

Equation 2.9

Of course, we can’t do an infinite summation in MATLAB, but we can observe how the graph of the function becomes increasingly square as we add more terms in the summation. To create the first four terms and plot the resulting sum, we can do

\[
f1 = 'sin(2*pi*262*t) + sin(2*pi*262*3*t)/3 + sin(2*pi*262*5*t)/5 + sin(2*pi*262*7*t)/7';
\]

fplot(f1, [0 0.01 -1 1]);

This gives the wave in Figure 2.38.
You can see that it is beginning to get square but has many ripples on the top. Adding four more terms gives further refinement to the square wave, as illustrated in Figure 2.39:

The sawtooth wave is an infinite sum of all harmonic frequencies with diminishing amplitudes, as in the following equation:

\[
\text{Let } f \text{ be a fundamental frequency. Then a sawtooth wave created from this fundamental frequency is defined by the infinite summation}
\]

\[
\frac{2}{\pi} \sum_{n=1}^{\infty} \frac{\sin(2\pi ft)}{n}
\]

\[\text{Equation 2.10}\]

\[
\frac{2}{\pi} \text{ is a scaling factor to ensure that the result of the summation is in the range of } -1 \text{ to } 1.
\]

The sawtooth wave can be plotted by the following MATLAB command:
The triangle wave is an infinite sum of odd-numbered harmonics that alternate in their signs, as follows:

\[
\frac{8}{\pi^2} \sum_{n=0}^{\infty} \left( \frac{\sin(2\pi(4n+1)ft)}{(4n+1)^2} - \frac{\sin(2\pi(4n+3)ft)}{(4n+3)^2} \right)
\]

Equation 2.11

\( \frac{8}{\pi^2} \) is a scaling factor to ensure that the result of the summation is in the range of \(-1\) to 1.

We leave the creation of the triangle wave as a MATLAB exercise.

If you actually want to hear one of these waves, you can generate the array of audio samples with

```mex
s = 1;
sr = 44100;
t = linspace(0, s, sr*s);
y = sawtooth(262*2*pi*t);
```

and then play the wave with

```mex
wavplay(y, sr);
```

It's informative to create and listen to square, sawtooth, and triangle waves of various amplitudes and frequencies. This gives you some insight into how these waves can be used in sound synthesizers to mimic the sounds of various instruments. We'll cover this in more detail in Chapter 6.

In this chapter, all our MATLAB examples are done by means of expressions that are evaluated directly from MATLAB's command line. Another way to approach the problems is to write programs in MATLAB's scripting language. We leave it to the reader to explore MATLAB script programming, and we'll have examples in later chapters. Chapter 5.

### 2.3.4 Reading and Writing WAV Files in MATLAB

In the previous sections, we generated sine waves to generate sound data and manipulate it in various ways. This is useful for understanding basic concepts regarding sound. However, in practice you have real-world sounds that have been captured and stored in digital form. Let's look now at how we can read audio files in MATLAB and perform operations on them.

We've borrowed a short WAV file from Adobe Audition's demo files, reducing it to mono rather than stereo. MATLAB's `wavread` function imports WAV files, as follows:

```mex
y = wavread('HornsE04Mono.wav');
```
This reads an array of audio samples into \( y \), assuming that the file is in the current folder of MATLAB. (You can set this through the Current Folder window at the top of MATLAB.) If you want to know the sampling rate and bit depth (the number of bits per sample) of the audio file, you can get this information with

\[
[y, sr, b] = \text{wavread}('HornsE04Mono.wav');
\]

\( sr \) now contains the sampling rate and \( b \) contains the bit depth. The Workspace window shows you the values in these variables.

You can play the sound with

\[
\text{sound}(y, sr);
\]

Once you've read in a WAV file and have it stored in an array, you can easily do mathematical operations on it. For example, you can make it quieter by multiplying by a number less than 1, as in

\[
y = y \times 0.5;
\]

You can also write out the new form of the sound file, as in

\[
\text{wavwrite}(y, 'HornsNew.wav');
\]

### 2.3.5 Modeling Sound in Octave

Octave is a freeware, open-source version of MATLAB distributed by GNU. It has many but not all of the functions of MATLAB. There are versions of Octave for Linux, UNIX, Windows, and Mac OS X.

If you try to do the above exercise in Octave, most of the functions are the same. The \texttt{fplot} function is the same in Octave as in MATLAB except that for colors, you must put a digit
from 0 to 7 in single quotes rather than use the name of the color. The \texttt{linspace} function is the same in Octave as in MATLAB. To play the sound, you need to use the \texttt{playsound} function rather than \texttt{wavplay}. You also can use a \texttt{wavwrite} function (which exists in both MATLAB and Octave) to write the audio data to an external file. Then you can play the sound with your favorite media player.

There is no \texttt{square} or \texttt{sawtooth} function in Octave. To create your own sawtooth, square, or triangle wave in Octave, you can use the Octave programs below. You might want to consider why the mathematical shortcuts in these programs produced the desired waveforms.

\begin{verbatim}
function saw = sawtooth(freq, samplerate)
    x = [0:samplerate];
    wavelength = samplerate/freq;
    saw = 2*mod(x, wavelength)/wavelength-1;
end
\end{verbatim}

\textbf{Program 2.1 Sawtooth wave in Octave}

\begin{verbatim}
function sqwave = squarewave(freq, samplerate)
    x = [0:samplerate];
    wavelength = samplerate/freq;
    saw = 2*mod(x, wavelength)/wavelength-1; %start with sawtooth wave
    sawzeros = (saw == zeros(size(saw))); %elminates division by zero in next step
    sqwave = -abs(saw)./(saw+sawzeros); %./ for element-by-element division
end
\end{verbatim}

\textbf{Program 2.2 Square wave in Octave}

\begin{verbatim}
function triwave = trianglewave(freq, samplerate)
    x = [0:samplerate];
    wavelength = samplerate/freq;
    saw = 2*mod(x, wavelength)/wavelength-1; %start with sawtooth wave
    tri = 2*abs(saw)-1;
end
\end{verbatim}

\textbf{Program 2.3 Triangle wave in Octave}

\subsection*{2.3.6 Transforming from One Domain to Another}

In Section 2.2.3, we showed how sound can be represented graphically in two ways. In the waveform view, time is on the horizontal axis and amplitude of the sound wave is on the vertical axis. In the frequency analysis view, frequency is on the horizontal axis and the magnitude of the frequency component is on the vertical axis. The waveform view represents sound in the \textbf{time domain}. The frequency analysis view represents sound in the \textbf{frequency domain}. (See Figure 2.18 and Figure 2.19.) Whether sound is represented in the time or the frequency domain, it's just a list of numbers. The information is essentially the same – it's just that the way we look at it is different.
The Fourier transform is a mathematical process that can transform audio data from the time to the frequency domain. Sometimes it's more convenient to represent sound data one way as opposed to another because it's easier to manipulate it in a certain domain. For example, in the time domain we can easily change the amplitude of the sound by multiplying each amplitude by a number. On the other hand, it may be easier to eliminate certain frequencies or change the relative magnitudes of frequencies if we have the data represented in the frequency domain.

2.3.7 The Discrete Fourier Transfer and its Inverse

The discrete Fourier transform is a version of the Fourier transform that can be applied to digitized sound. The algorithm is given in Algorithm 2.1.

```
/*Input:
f, an array of digitized audio samples
N, the number of samples in the array
Note: i = √−1
Output:
F, an array of complex numbers which give the amplitudes of the frequency components of the sound given by f */
for (n = 0 to N – 1)
    Fn = 1/N(∑k=0N-1 fk cos(2πnk/N) – ifk sin(2πnk/N))
```

Algorithm 2.1 Discrete Fourier transform

Each time through the loop, the n\textsuperscript{th} frequency components is computed, \(F_n\). Each \(F_n\) is a complex number with a cosine and sine term, the sine term having the factor \(i\) in it.

We assume that you're familiar with complex numbers, but if not, a short introduction should be enough so that you can work with the Fourier algorithm.

A complex number takes the form \(a + bi\), where \(i = √−1\). Thus,

\[
\cos\left(\frac{2πnk}{N}\right) - if_k \sin\left(\frac{2πnk}{N}\right)
\]

is a complex number. In this case, \(a\) is replaced with \(\cos\left(\frac{2πnk}{N}\right)\) and \(b\) with \(-f_k\sin\left(\frac{2πnk}{N}\right)\).

Handling the complex numbers in an implementation of the Fourier transform is not difficult. Although \(i\) is an imaginary number, \(√−1\), and you might wonder how you’re supposed to do computation with it, you really don’t have to do anything with it at all except assume it’s there. The summation in the formula can be replaced by a loop that goes from \(0\) through \(N-1\). Each time through that loop, you add another term from the summation into an accumulating total. You can do this separately for the cosine and sine parts, setting aside \(i\). This is explained in more detail in the exercise associated with this section. Also, in object-oriented programming languages, you may have a Complex number class to do complex number calculations for you.

The result of the Fourier transform is a list of complex numbers \(F\), each of the form \(a + bi\), where the magnitude of the frequency component is equal to \(\sqrt{a^2 + b^2}\).

The inverse Fourier transform transforms audio data from the frequency domain to the time domain. The inverse discrete Fourier transform is given in Algorithm 6.2.
/* Input:
  \( F \), an array of complex numbers representing audio data in the frequency domain, the elements represented by the coefficients of their real and imaginary parts, \( a \) and \( b \) respectively
  \( N \), the number of samples in the array
Note: \( i = \sqrt{-1} \)
Output: \( f \), an array of audio samples in the time domain*/
for (\( n = 0 \) to \( N - 1 \))
\[ f_k = \sum_{n=0}^{N-1} \left( a_n \cos \frac{2\pi nk}{N} + b_n \sin \frac{2\pi nk}{N} \right) \]

Algorithm 2.2 Inverse discrete Fourier transform

2.3.8 The Fast Fourier Transform (FFT)
If you know how to program, it's not difficult to write your own discrete Fourier transform and its inverse through a literal implementation of the equations above. We include this as an exercise in this section. However, the "literal" implementation of the transform is computationally expensive. The equation in Algorithm 2.1 has to be applied \( N \) times, where \( N \) is the number of audio samples. The equation itself has a summation that goes over \( N \) elements. Thus, the discrete Fourier transform takes on the order of \( N^2 \) operations.

The fast Fourier transform (FFT) is a more efficient implementation of the Fourier transform that does on the order of \( N \cdot \log_2 N \) operations. The algorithm is made more efficient by eliminating duplicate mathematical operations. The FFT is the version of the Fourier transform that you'll often see in audio software and applications. For example, Adobe Audition uses the FFT to generate its frequency analysis view, as shown in Figure 2.41.
Figure 2.41 Frequency analysis view (left) and waveform view (right) in Adobe Audition, showing audio data in the frequency domain and time domain, respectively.

### 2.3.9 Applying the Fourier Transform in MATLAB

Generally when you work with digital audio, you don't have to implement your own FFT. Efficient implementations already exist in many programming language libraries. For example, MATLAB has FFT and inverse FFT functions, `fft` and `ifft`, respectively. We can use these to experiment and generate graphs of sound data in the frequency domain.

First, let's use sine functions to generate arrays of numbers that simulate single-pitch sounds. We'll make three one-second long sounds using the standard sampling rate for CD quality audio, 44,100 samples per second. First, we a generate an array of $sr\times s$ numbers across which we can evaluate sine functions, putting this array in the variable $t$.

```matlab
sr = 44100; %sr is sampling rate
s = 1; %s is number of seconds
t = linspace(0, s, sr*s);
```

Now we use the array $t$ as input to sine functions at three different frequencies and phases, creating the note A at three different octaves (110 Hz, 220 Hz, and 440 Hz).

```matlab
x = cos(2*pi*110*t);
y = cos(2*pi*220*t + pi/3);
z = cos(2*pi*440*t + pi/6);
```
$x$, $y$, and $z$ are arrays of numbers that can be used as audio samples. $\pi/3$ and $\pi/6$ represent phase shifts for the 220 Hz and 440 Hz waves, to make our phase response graph more interesting. The figures can be displayed with the following:

```matlab
defigure;
plot(t,x);
axis([0 0.05 -1.5 1.5]);
title('x');
defigure;
plot(t,y);
axis([0 0.05 -1.5 1.5]);
title('y');
defigure;
plot(t,z);
axis([0 0.05 -1.5 1.5]);
title('z');
```

We look at only the first 0.05 seconds of the waveforms in order to see their shape better. You can see the phase shifts in the figures below. The second and third waves don't start at 0 on the vertical axis.

![Figure 2.42 110 Hz, no phase offset](image1)

![Figure 2.43 220 Hz, $\pi/3$ phase offset](image2)
Now we add the three sine waves to create a composite wave that has three frequency components at three different phases.

\[ a = \frac{x + y + z}{3}; \]

Notice that we divide the summed sound waves by three so that the sound doesn’t clip. You can graph the three-component sound wave with the following:

```matlab
figure;
plot(t, a);
axis([0 0.05 -1.5 1.5]);
title('a = x + y + z');
```

This is a graph of the sound wave in the time domain. You could call it an impulse response graph, although when you’re looking at a sound file like this, you usually just think of it as “sound data in the time domain.” The term “impulse response” is used more commonly for time domain filters, as we’ll see in Chapter 7.

You might want to play the sound to be sure you have what you think you have. The `sound` function requires that you tell it the number of samples it should play per second, which for our simulation is 44,100.
sound(a, sr);

When you play the sound file and listen carefully, you can hear that it has three tones.

MATLAB's Fourier transform (fft) returns an array of double complex values (double-precision complex numbers) that represent the magnitudes and phases of the frequency components.

fftdata = fft(a);

In MATLAB's workspace window, fftdata values are labeled as type double, giving the impression that they are real numbers, but this is not the case. In fact, the Fourier transform produces complex numbers, which you can verify by trying to plot them in MATLAB. The magnitudes of the complex numbers are given in the Min and Max fields, which is computed by the abs function. For a complex number $a + bi$, the magnitude is computed as $\sqrt{a^2 + b^2}$.

Figure 2.46 Workspace in MATLAB showing values and types of variables currently in memory

To plot the results of the fft function such that the values represent the magnitudes of the frequency components, we first apply the abs function to fftdata.

fftmag = abs(fftdata);

Let's plot the frequency components to be sure we have what we think we have.

For a sampling rate of $sr$ on an array of sample values of size $N$, the Fourier transform returns the magnitudes of $N/2$ frequency components evenly spaced between 0 and $sr/2$ Hz. (We'll explain this completely in Chapter 5.) Thus, we want to display frequencies between 0 and $sr/2$ on the horizontal axis, and only the first $sr/2$ values from the fftmag vector.
figure;
freqs = [0: (sr/2)-1];
plot(freqs, fftmag(1:sr/2));

When you do this, you'll see that all the frequency components are way over on the left side of the graph. Since we know our frequency components should be 110 Hz, 220 Hz, and 440 Hz, we might as well look at only the first, say, 600 frequency components so that we can see the results better. One way to zoom in on the frequency response graph is to use the zoom tool in the graph window, or you can reset the axis properties in the command window, as follows.

axis([0 600 0 8000]);

This yields the frequency response graph for our composite wave, which shows the three frequency components.

![Figure 2.47 Frequency response graph for a 3-component wave](image)

To get the phase response graph, we need to extract the phase information from the `fftdata`. This is done with the `angle` function. We leave that as an exercise.

Let's try the Fourier transform on a more complex sound wave – a sound file that we read in.

```matlab
y = wavread('HornsE04Mono.wav');
```

As before, you can get the Fourier transform with the `fft` function.

```matlab
fftdata = fft(y);
```

You can then get the magnitudes of the frequency components and generate a frequency response graph from this.

```matlab
fftmag = abs(fftdata);
```
Let's zoom in on frequencies up to 5000 Hz.

The graph below is generated.

![Figure 2.48 Frequency response for HornsE04Mono.wav](image)

The inverse Fourier transform gives us back our original sound data in the time domain.

```
ynew = ifft(fftdata);
```

If you compare $y$ with $ynew$, you'll see that the inverse Fourier transform has recaptured the original sound data.

### 2.3.10 Windowing the FFT

When we applied the Fourier transform in MATLAB in Section 2.3.9, we didn't specify a window size. Thus, we were applying the FFT to the entire piece of audio. If you listen to the WAV file HornsE04Mono.wav, a three second clip, you'll first hear some low tubas and them some higher trumpets. Our graph of the FFT shows frequency components up to and beyond 5000 Hz, which reflects the sounds in the three seconds. What if we do the FFT on just the first second (44100 samples) of this WAV file, as follows? The resulting frequency components are shown in Figure 2.49.

```
y = wavread('HornsE04Mono.wav');
sr = 44100;
freqs = [0:(sr/2)-1];
ybegin = y(1:44100);
```
fftdata2 = fft(ybegin);
fftdata2 = fftdata2(1:22050);
plot(freqs, abs(fftdata2));
axis([0 5000 0 4500]);

Figure 2.49 Frequency components of first second of HornsE04Mono.wav

What we've done is focus on one short window of time in applying the FFT. An **FFT window** is a contiguous segment of audio samples on which the transform is applied. If you consider the nature of sound and music, you'll understand why applying the transform to relatively small windows makes sense. In many of our examples in this book, we generate segments of sound that consist of one or more frequency components that do not change over time, like a single pitch note or a single chord being played without change. These sounds are good for experimenting with the mathematics of digital audio, but they aren't representative of the music or sounds in our environment, in which the frequencies change constantly. The WAV file HornsE04Mono.wav serves as a good example. The clip is only three seconds long, but the first second is very different in frequencies (the pitches of tubas) from the last two seconds (the pitches of trumpets). When we do the FFT on the entire three seconds, we get a kind of "blurred" view of the frequency components, because the music actually changes over the three second period. It makes more sense to look at small segments of time. This is the purpose of the FFT window.

Figure 2.50 shows an example of how FFT window sizes are used in audio processing programs. Notice the drop down menu, which gives you a choice of FFT sizes ranging from 32 to 65536 samples. The FFT window size is typically a multiple of 2. If your sampling rate is 44,100 samples per second, then a window size of 32 samples is about 0.0007 s, and a window size of 65536 is about 1.486 s.

There's a tradeoff in the choice of window size. A small window focuses on the frequencies present in the sound over a short period of time. However, as mentioned earlier, the number of frequency components yielded by an FFT of size \( N \) is \( N/2 \). Thus, for a window size of, say, 128, only 64 frequency bands are output, these bands spread over the frequencies from 0 Hz to \( sr/2 \) Hz where \( sr \) is the sampling rate. (See Chapter 5.) For a window size of 65536, 37768 frequency bands are output, which seems like a good thing, except that with the large window size, the FFT is not isolating a short moment of time. A window size of around 2048
usually gives good results. If you set the size to 2048 and play the piece of music loaded into Audition, you'll see the frequencies in the frequency analysis view bounce up and down, reflecting the changing frequencies in the music as time pass.

![Image of Adobe Audition](image.png)

Figure 2.50 Choice of FFT window size in Adobe Audition

### 2.3.11 Windowing Functions to Eliminate Spectral Leakage

In addition to choosing the FFT window size, audio processing programs often let you choose from a number of windowing functions. The purpose of an FFT windowing function is to smooth out the discontinuities that result from applying the FFT to segments (i.e., windows) of audio data. A simplifying assumption for the FFT is that each windowed segment of audio data contains an integral number of cycles, this cycle repeating throughout the audio. This, of course, is not generally the case. If it were the case – that is, if the window ended exactly where the cycle ended – then the end of the cycle would be at exactly the same amplitude as the beginning. The beginning and end would “match up.” The actual discontinuity between the end of a window and its beginning is interpreted by the FFT as a jump from one level to another, as shown in Figure 2.51. (In this figure, we've cut and pasted a portion from the beginning of the window to its end to show that the ends don't match up.)
Figure 2.51 Discontinuity between the end of a window and its beginning

In the output of the FFT, the discontinuity between the ends and the beginnings of the windows manifests itself as frequency components that don't really exist in audio – called **spurious frequencies**, or **spectral leakage**. You can see the spectral leakage Figure 2.41. Although the audio signal actually contains only one frequency at 880 Hz, the frequency analysis view indicates that there is a small amount of other frequencies across the audible spectrum.

In order to smooth over this discontinuity and thereby reduce the amount of spectral leakage, the windowing functions effectively taper the ends of the segments to 0 so that they connect from beginning to end. The drop-down menu to the left of the FFT size menu in Audition is where you choose the windowing function. In Figure 2.50, the Hanning function is chosen. Four commonly-used windowing functions are given in the table below.

<table>
<thead>
<tr>
<th>Windowing Function</th>
<th>Equation</th>
</tr>
</thead>
</table>
| Triangular windowing function | \( u(t) = \begin{cases} 
\frac{2t}{T} & \text{for } 0 \leq t < \frac{T}{2} \\
2 - \frac{2t}{T} & \text{for } \frac{T}{2} \leq t \leq T 
\end{cases} \) |
| Hanning windowing function | \( u(t) = \frac{1}{2} \left[ 1 - \cos \left( \frac{2\pi t}{T} \right) \right] \) for \( 0 \leq t \leq T \) |
| Hamming windowing function | \( u(t) = 0.54 - 0.46 \cos \left( \frac{2\pi t}{T} \right) \) for \( 0 \leq t \leq T \) |
| Blackman windowing function | \( u(t) = 0.42 - 0.5 \cos \left( \frac{2\pi t}{T} \right) + 0.08 \cos \left( \frac{4\pi t}{T} \right) \) for \( 0 \leq t \leq T \) |

\( t \) is time.
\( T \) is length of period. If \( w \) is window size and \( sr \) is sampling rate, then \( T = \frac{w}{sr} \).

Figure 2.52 Windowing functions

Windowing functions are easy to apply. The segment of audio data being transformed is simply multiplied by the windowing function before the transform is applied. In MATLAB, you can accomplish this with vector multiplication, as shown in the commands below.

```matlab
y = wavread('HornsE04Mono.wav');
sr = 44100; % sampling rate
w = 2048; % window size
T = w/sr; % period
% t is an array of times at which the sine function is evaluated
t = linspace(0, 1, 44100);
twindow = t(1:2048); % first 2048 elements of t
% Create the values for the hanning function, stored in vector called hamming
hamming = 0.54 - 0.46 * cos((2 * pi * twindow)/T);
```
plot(hamming);
title('Hamming');

The Hamming function is shown in

![Figure 2.53 Hamming windowing function](image)

```matlab
yshort = y(1:2048);  %first 2048 samples from sound file
%Multiply the audio values in the window by the Hamming function values,
% using element by element multiplication with .*.
% first convert hamming from a column vector to a row vector
ywindowed = hamming .* yshort;
figure;
plot(yshort);
title('First 2048 samples of audio data');
figure;
plot(ywindowed);
title('First 2048 samples of audio data, tapered by windowing function');
```

Before the Hamming function is applied, the first 2048 samples of audio data look like this:

![Figure 2.54 Audio data](image)

After the Hamming function is applied, the audio data look like this:
Notice that the ends of the segment are tapered toward 0.

Figure 2.56 compares the FFT results with no windowing function vs. with the Hamming windowing function applied. The windowing function eliminates some of the high frequency components that are caused by spectral leakage.

```matlab
gfigure
plot(abs(fft(yshort)));
axis([0 300 0 60]);
hold on;
plot(abs(fft(ywindowed)),'r');
```

Figure 2.56 Comparing FFT results with and without windowing function
2.3.12 Modeling Sound in C++ under Linux

If you want to work at an even lower level of abstraction, a good environment for experimentation is the Linux operating system using “from scratch” programs written in C++. In our first example C++ sound program, we show you how to create sound waves of a given frequency, add frequency components to get a complex wave, and play the sounds via the sound device. This program is another implementation of the exercises in Max and MATLAB in Sections 2.3.1 and 2.3.3. The C++ program is given in Program 2.4.

```cpp
// This program uses the OSS library.
#include <sys/ioctl.h> // for ioctl()
#include <math.h> // sin(), floor(), and pow()
#include <stdio.h> // perror
#include <fcntl.h> // open, O_WRONLY
#include <linux/soundcard.h> // SOUND_PCM*
#include <iostream>
#include <unistd.h>
using namespace std;

#define TYPE char
#define LENGTH 1 // number of seconds per frequency
#define RATE 44100 // sampling rate
#define SIZE sizeof(TYPE) // size of sample, in bytes
#define CHANNELS 1 // number of audio channels
#define PI 3.14159
#define NUM_FREQS 3 // total number of frequencies
#define BUFFSIZE (int) (NUM_FREQS*LENGTH*RATE*SIZE*CHANNELS) // bytes sent to audio device
#define ARRSIZE (int) (NUM_FREQS*LENGTH*RATE*CHANNELS) // total number of samples
#define SAMPLE_MAX (pow(2,SIZE*8 - 1) - 1)

void writeToSoundDevice(TYPE buf[], int deviceID) {
    int status;
    status = write(deviceID, buf, BUFFSIZE);
    if (status != BUFFSIZE)
        perror("Wrote wrong number of bytes\n");
    status = ioctl(deviceID, SOUND_PCM_SYNC, 0);
    if (status == -1)
        perror("SOUND_PCM_SYNC failed\n");
}

int main() {
    int deviceID, arg, status, f, t, a, i;
    TYPE buf[ARRSIZE];
    deviceID = open("/dev/dsp", O_WRONLY, 0);
    if (deviceID < 0)
        perror("Opening /dev/dsp failed\n");
    // working
    arg = SIZE * 8;
    status = ioctl(deviceID, SOUND_PCM_WRITE_BITS, &arg);
    if (status == -1)
        perror("Unable to set sample size\n");
    arg = CHANNELS;
    status = ioctl(deviceID, SOUND_PCM_WRITE_CHANNELS, &arg);
}```
if (status == -1)
    perror("Unable to set number of channels\n");
arg = RATE;
status = ioctl(deviceID, SOUND_PCM_WRITE_RATE, &arg);
if (status == -1)
    perror("Unable to set sampling rate\n");
a = 1;
for (i = 0; i < NUM_FREQS; ++i) {
    switch (i) {
        case 0:
            f = 262;
            break;
        case 1:
            f = 330;
            break;
        case 2:
            f = 392;
            break;
    }
    for (t = 0; t < ARRAYSIZE/NUM_FREQS; ++t) {
        buf[t + ((ARRAYSIZE / NUM_FREQS) * i)] = floor(a * 
            sin(2*PI*f*t/RATE));
    }
}
writeToSoundDevice(buf, deviceID);

Program 2.4 Adding sine waves and sending sound to sound device in C++

To be able to compile and run a program such as this, you need to install a sound library in your Linux environment. At the time of the writing of this chapter, the two standard low-level sound libraries for Linux are the OSS (Open Sound System) and ALSA (Advanced Linux Sound Architecture). A sound library provides a software interface that allows your program to access the sound devices, sending and receiving sound data. ALSA is the newer of the two libraries and is preferred by most users. At a slightly higher level of abstraction are PulseAudio and Jack, applications which direct multiple sound streams from their inputs to their outputs. Ultimately, PulseAudio and Jack use lower level libraries to communicate directly with the sound cards.

Program 2.4 uses the OSS library. In a program such as this, the sound device is opened, read from, and written to in a way similar to how files are handled. The sample program shows how you open /dev/dsp, an interface to the sound card device, to ask this device to receive audio data. The variable deviceID serves as an ID of the sound device and is used as a parameter indicating the size of data to expect, the number of channels, and the data rate. We’ve set a size of eight bits (one byte) per audio sample, one channel, and a data rate of 44,100 samples per second. The significance of these numbers will be clearer when we talk about digitization in Chapter 5. The buffer size is a product of the sample size, data rate, and length of the recording (in this case, three seconds), yielding a buffer of 44,100 * 3 bytes.

The sound wave is created by taking the sine of the appropriate frequency (262 Hz, for example) at 44,100 evenly-spaced intervals for one second of audio data. The value returned from the sine function is between −1 and 1. However, the sound card expects a value that is
stored in two bytes (i.e., 16 bits), ranges from \(-32768\) to \(32767\). To put the value into this range, we multiply by 32767 and, with the \textit{floor} function, round down.

The three frequencies are created and concatenated into one array of audio values. The \textit{write} function has the device ID, the name of the buffer for storing the sound data, and the size of the buffer as its parameters. This function sends the sound data to the sound card to be played. The three frequencies together produce a harmonious chord in the key of C. In Chapter 3, we’ll explore what makes these frequencies harmonious.

The program requires some header files for definitions of constants like \texttt{O_WRONLY} (restricting access to the sound device to writing) and \texttt{SOUND_PCM_WRITE_BITS}. After you install the sound libraries, you’ll need to locate the appropriate header files and adjust the \texttt{#include} statement accordingly. You’ll also need to check the way your compiler handles the math and sound libraries. You may need to include the option \texttt{–lm} on the compile line to include the math library, or the \texttt{–lasound} option for the ALSA library.

This program introduces you to the notion that sound must be converted to a numeric format that is communicable to a computer. The solution to the programming assignment given as a learning supplement has an explanation of the variables and constants in this program. A full understanding of the program requires that you know something about sampling and quantization, the two main steps in analog-to-digital conversion, a topic that we’ll examine in depth in Chapter 5.

### 2.3.13 Modeling Sound in Java

The Java environment allows the programmer to take advantage of Java libraries for sound and to benefit from object-oriented programming features like encapsulation, inheritance, and interfaces. In this chapter, we are going to use the package \texttt{javax.sound.sampled}. This package provides functionality to capture, mix, and play sounds with classes such as \texttt{SourceDataLine}, \texttt{AudioFormat}, \texttt{AudioSystem}, and \texttt{LineUnavailableException}.

Program 2.5 uses a \texttt{SourceDataLine} object. This is the object to which we write audio data. Before doing that, we must set up the data line object with a specified audio format object. (See line 30.) The \texttt{AudioFormat} class specifies a certain arrangement of data in the sound stream, including the sampling rate, sample size in bits, and number of channels. A \texttt{SourceDataLine} object is created with the specified format, which in the example is 44,100 samples per second, eight bits per sample, and one channel for mono. With this setting, the line gets the required system resource and becomes operational. After the \texttt{SourceDataLine} is opened, data is written to the mixer using a buffer that contains data generated by a sine function. Notice that we don’t directly access the Sound Device because we are using a \texttt{SourceDataLine} object to deliver data bytes to the mixer. The mixer mixes the samples and finally delivers the samples to an audio output device on a sound card.

```java
1 import javax.sound.sampled.AudioFormat;
2 import javax.sound.sampled.AudioSystem;
3 import javax.sound.sampled.SourceDataLine;
4 import javax.sound.sampled.LineUnavailableException;
5
6 public class ExampleTone1{
7    8   public static void main(String[] args){
```
```java
try {
    ExampleTone1.createTone(262, 100);
} catch (LineUnavailableException lue) {
    System.out.println(lue);
}
```

```java
/** parameters are frequency in Hertz and volume */
public static void createTone(int Hertz, int volume)
    throws LineUnavailableException {
/** Exception is thrown when line cannot be opened */
    float rate = 44100;
    byte[] buf;
    AudioFormat audioF;
    buf = new byte[1];
    audioF = new AudioFormat(rate, 8, 1, true, false);
    //sampleRate, sampleSizeInBits,channels,signed,bigEndian
    SourceDataLine sourceDL = AudioSystem.getSourceDataLine(audioF);
    sourceDL = AudioSystem.getSourceDataLine(audioF);
    sourceDL.open(audioF);
    sourceDL.start();
    for(int i=0; i<rate; i++){
        double angle = (i/rate)*Hertz*2.0*Math.PI;
        buf[0] = (byte)(Math.sin(angle)*volume);
        sourceDL.write(buf, 0, 1);
    }
    sourceDL.drain();
    sourceDL.stop();
    sourceDL.close();
}
```

Program 2.5 A simple sound generating program in Java

This program illustrates a simple way of generating a sound by using a sine wave and the javax.sound.sampled library. If we change the values of the createTone procedure parameters, which are 262 Hz for frequency and 100 for volume, we can produce a different tone. The second parameter, volume, is used to change the amplitude of the sound. Notice that the sine function results is multiplied by the volume parameter in line 40.

Although the purpose of this section of the book is not to demonstrate how Java graphics classes are used, it may be helpful to use some basic plot features in Java to generate sine wave drawings. An advantage of Java is that it facilitates your control of windows and containers. We inherit this functionality from the JPanel class, which is a container where we are going to paint the sine wave generated. Program 2.6 is a variation of Program 2.5. It produces a Java Window by using the procedure paintComponent. This sine wave generated again has a frequency of 262 Hz and a volume of 100.
```java
import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.SourceDataLine;
import javax.sound.sampled.LineUnavailableException;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class ExampleTone2 extends JPanel{

    static double[] sines;
    static int vol;

    public static void main(String[] args){
        try {
            ExampleTone2.createTone(262, 100);
        } catch (LineUnavailableException lue) {
            System.out.println(lue);
        }
    }

    //Frame object for drawing
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.add(new ExampleTone2());
    frame.setSize(800,300);
    frame.setLocation(200,200);
    frame.setVisible(true);
}

public static void createTone(int Hertz, int volume) throws LineUnavailableException {
    float rate = 44100;
    byte[] buf;
    buf = new byte[1];
    sines = new double[(int)rate];
    vol=volume;

    AudioFormat audioF;
    audioF = new AudioFormat(rate,8,1,true,false);

    SourceDataLine sourceDL = AudioSystem.getSourceDataLine(audioF);
    sourceDL = AudioSystem.getSourceDataLine(audioF);
    sourceDL.open(audioF);
    sourceDL.start();

    for(int i=0; i<rate; i++){
        double angle = (i/rate)*Hertz*2.0*Math.PI;
        buf[0]=(byte)(Math.sin(angle)*vol);
        sourceDL.write(buf,0,1);
        sines[i]=(double)(Math.sin(angle)*vol);
    }
}
```
sourceDL.drain();
sourceDL.stop();
sourceDL.close();
}

protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
    int pointsToDraw=4000;
    double max=sines[0];
    for(int i=1;i<pointsToDraw;i++) if (max<sines[i]) max=sines[i];
    int border=10;
    int w = getWidth();
    int h = (2*border+(int)max);
    double xInc = 0.5;
    //Draw x and y axes
    g2.draw(new Line2D.Double(border, border, border, 2*(max+border)));
    g2.draw(new Line2D.Double(border, (h-sines[0]), w-border, (h-
                          sines[0])));
    g2.setPaint(Color.red);
    for(int i = 0; i < pointsToDraw; i++) {
        double x = border + i*xInc;
        double y = (h-sines[i]);
        g2.fill(new Ellipse2D.Double(x-2, y-2, 2, 2));
    }
}

Program 2.6 Visualizing the sound waves in a Java program

If we increase the value of the frequency in line 18 to 400 Hz, we can notice how the number of cycles increases, as shown in Figure 2.57. On the other hand, by increasing the volume, we obtain a higher amplitude for each frequency.
We can also create square, triangle, and sawtooth waves in Java by modifying the for loop in lines 38 to 42. For example, to create a square wave, we may change the for loop to something like the following:

```java
for (int i=0; i<rate; i++) {
    double angle1 = i/frequency*hz*1.0*2.0*Math.PI;
    double angle2 = i/frequency*hz*3.0*2.0*Math.PI;
    double angle3 = i/frequency*hz*5.0*2.0*Math.PI;
    double angle4 = i/frequency*hz*7.0*2.0*Math.PI;
    buf[0]=(byte)(Math.sin(angle1)*vol+Math.sin(angle2)*vol/3+Math.sin(angle3)*vol/5+Math.sin(angle4)*vol/7);
    sdl.write(buf,0,1);
    sines[i]=(double)(Math.sin(angle1)*vol+Math.sin(angle2)*vol/3+Math.sin(angle3)*vol/5+Math.sin(angle4)*vol/7);
}
```

This for loop produces the sine wave shown in Figure 2.58. This graph doesn't look like a perfect square wave, but the more harmonic frequencies we add, the closer we get to a square wave. (Note that you can create these waveforms more exactly by adapting the Octave programs above to Java.)
Figure 2.58 Creating a square wave in Java

2.4 References

In addition to references cited in previous chapters:


