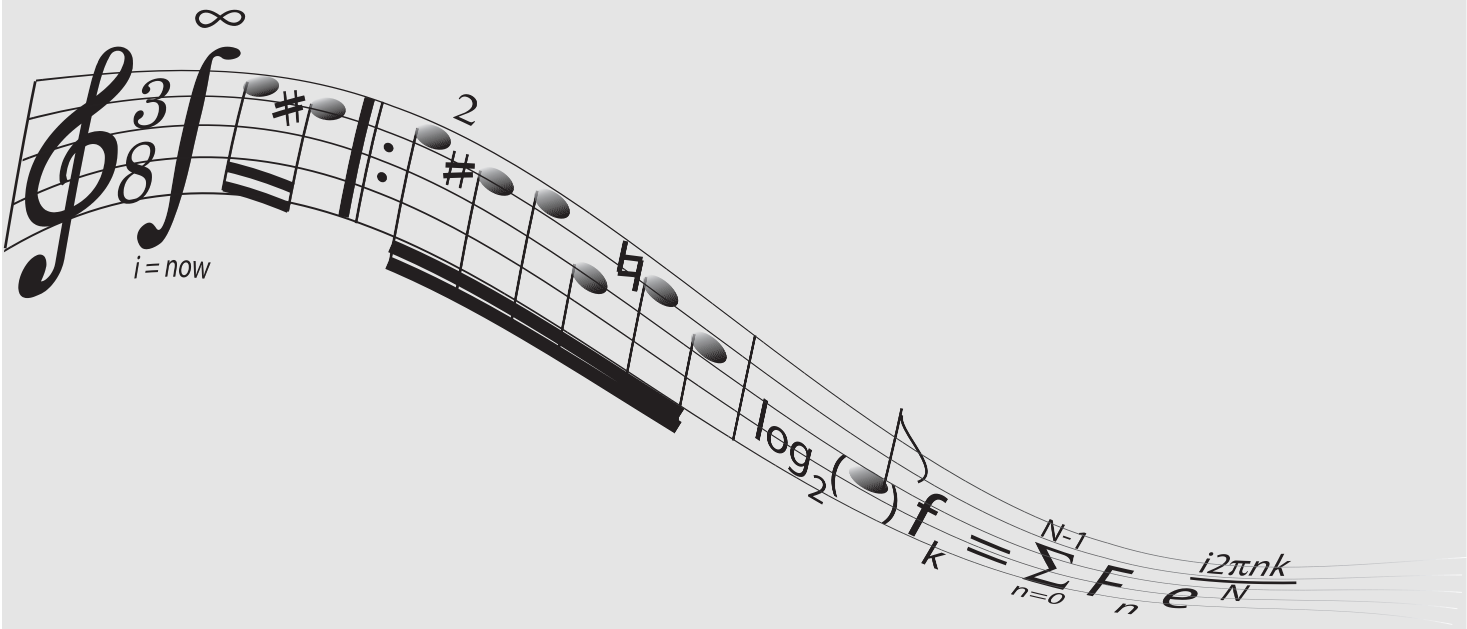
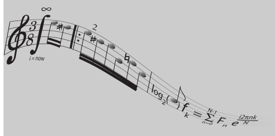

∞
3
8
i = now
2
log2( )fk = Σ N-1 n=o Fn e i2πnk/N

CHAPTER 4

# Digital Audio Representation



# CHAPTER 4

***Take care of the sense, and the sounds will take care of themselves.*** **—Lewis Carroll, Alice's Adventures in Wonderland**

OBJECTIVES FOR CHAPTER 4

- Understand how waveforms represent changing air pressure caused by sound.
- Be able to apply the Nyquist theorem to an understanding of digital audio aliasing.
- Given a sampling rate, be able to compute the Nyquist frequency.
- Given the frequency of an actual sound wave, be able to compute the Nyquist rate.
- Given a sampling rate and the frequency of an actual sound wave, be able to compute the frequency of the resulting aliased wave if aliasing occurs.
- Understand the relationship between quantization level (*i.e.*, sample size or bit depth) to dynamic range of an audio file.
- Given air pressure amplitude for a sound, be able to compute decibels, and vice versa.
- Given a bit depth for digital audio, be able to compute the signal-to-quantization noise ratio assuming linear quantization.
- Understand how dithering is done and be able to choose an appropriate audio dithering function.
- Understand how noise shaping is done.
- Understand the algorithm and mathematics for $\mu$-law encoding.
- Understand the application and implementation of the Fourier transform for digital audio processing.
- Be able to read and interpret a histogram of a waveform.
- Be able to compute the RMS (root-mean-square) power of a digital audio wave.
- Understand the information provided in a frequency or spectral analysis of an audio wave.
- Understand the difference between the formats of MIDI and digital audio sound files.
- Become familiar with basic terminology of MIDI and related areas in musical acoustics and musical notation.
- Be able to interpret a MIDI byte-stream, identifying status and data bytes.

## 4.1 INTRODUCTION

As we introduce each new medium in this book, we place the concepts in context. Why would you want or need to know the mathematical and scientific concepts covered in the next two chapters on digital audio? What work might you be doing that is based on these concepts?

You may find yourself working with digital audio in a variety of situations. You may want to be able to digitally record and edit music, combining instruments and voices. You may want to edit the sound track for a digital video. You may be doing game programming and want to create sound effects, voice, and musical accompaniment for the game. You may work in the theater designing the sound to be used in the performance of a play or dance.

Working with digital audio in these applications entails recording sound, choosing the appropriate sampling rate and sample size for a recording, knowing the microphones to use for your conditions, choosing a sound card and editing software for recording and editing, taking out the imperfections in recorded audio, processing with special effects, compressing, and selecting the right file type for storage.

In this chapter, we'll begin with the basic concepts underlying digital audio representation. In Chapter 5 we'll talk about how to apply these concepts in digital audio processing.

## 4.2 AUDIO WAVEFORMS

The notion of a sound wave is something you've probably become comfortable with because you encounter it in so many everyday contexts. But unless you've had to study it in a high school or college physics course, you might assume that you understand the sense in which sound is a "wave," when really you may never have thought about it very closely. So let's check your understanding, just to be sure.

A good way to understand sound waves is to picture how they act on microphones. There are many kinds of microphones, divided for our purposes into two main categories of electrodynamic and electrostatic types. ***Electrodynamic microphones*** (also called simply ***dynamic microphones***) operate by means of a moving coil or band (Figure 4.1). ***Electrostatic microphones*** (also called ***condenser*** or ***capacitor microphones***) are based on capacitors that require an external power supply. Both function according to the same principle, which is that changing air pressure produced by sound waves causes the parts inside the microphone to react, and this reaction can be recorded as a "wave." For illustration of the nature of a sound wave, let's use an electrodynamic microphone, since it's easy to picture how it works.

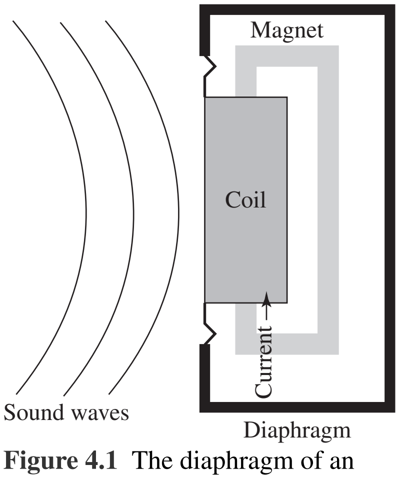


**Figure 4.1** The diaphragm of an electrodynamic microphone

Imagine that a single note is played on a piano, and a microphone is placed close to the piano to pick up the sound. A hammer inside the piano's mechanism strikes a string, vibrating the string. The string's vibrations "push" on the air molecules next to it, causing them alternately to move closer together and then farther apart in a regularly repeating pattern. When air molecules are squeezed together, air pressure rises. When they move apart, air pressure falls. These changes in air pressure are propagated through space. It's just like what happens when you splash in the water at the edge of a pool. The wave ripples through the pool as the water molecules move back and forth.

So how does this sound wave affect the microphone? The basic component of an electrodynamic microphone is a coil of wire wound in the microphone's diaphragm in the presence of a magnetic field. (The diaphragm is the place in a microphone that detects the air vibrations and responds.) If a wire moves in the environment of a magnetic field, a current is induced in the wire. This is what happens when the changes in air pressure reach the wire—the wire moves, and an electrical current proportional to the movement of the wire is created. The electrical current oscillates along with the vibrations of the piano string. The changes in current can be recorded as continuously changing voltages. If you draw a graph of how the voltages change, what you're drawing is the sound wave produced by the striking of the piano key. In essence, the changing voltages model how air pressure changes over time in response to some vibration, changes that are perceived by the human ear as sound.

Physically, the changing air pressure caused by sound is translated into changing voltages. Mathematically, the fluctuating pressure can be modeled as continuously changing numbers—a function where time is the input variable and amplitude (of air pressure or voltage) is the output. Graphing this function with time on the horizontal axis and amplitude on the vertical axis, we have the one-dimensional waveform commonly associated with sound. If the sound is a single-pitch tone with no overtones, the graph will be a single-frequency sinusoidal wave. The wave corresponding to the note A on the piano (440 Hz) is shown in Figure 4.2. Few sounds come to us as single-pitch tones. Even the single spoken word "boo" is a complex waveform, as shown in Figure 1.16.

You saw in Chapter 2 that an image can mathematically be formulated as the sum of its frequency components in two dimensions, horizontally and vertically over space. A complex

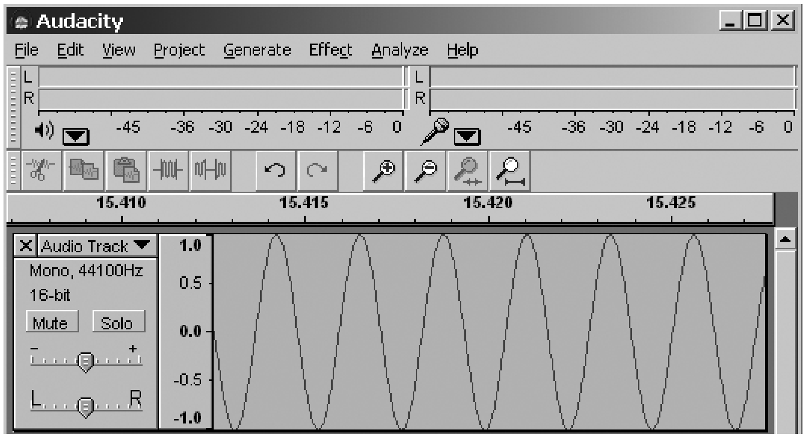


**Figure 4.2** Waveform view of the note A, 440 Hz (from Audacity)

sound wave can be formulated as a sum of its frequency components in one dimension, over time. Later in this chapter, you'll see how the transformation from the time domain to the frequency domain is done with the Fourier transform. For now, let's look at different views of a waveform that give you information about the sound in either the time or the frequency domain.

Figure 4.2 is a waveform view from an audio processing program. A waveform view shows the sound wave with time on the horizontal axis and amplitude on the vertical axis, as shown in the figure above. You may be able change the units on the axes. The vertical axis might be shown in decibels (specifically, dBFS, explained below), sample values, percentages, or normalized values. The horizontal axis might be shown as mm:ss:ddd (*i.e.*, minutes, seconds, and fractions of seconds to the thousandths), sample numbers, or different SMPTE (Society of Motion Picture and Television Engineers) formats.

## 4.3 PULSE CODE MODULATION AND AUDIO DIGITIZATION

Algorithms for sound digitization date back farther than you might imagine. ***Pulse code modulation*** (***PCM***) is a term that you'll encounter frequently as you read about digital audio. The term has been around for a long time—since it was first invented in 1937 by Alec Reeves, who at that time worked for International Telephone and Telegraph. PCM is based on the methods of sampling and quantization applied to sound. However, when Reeves devised this method in 1937, his focus was on the *transmission* of audio signals—the emphasis being on the word *modulation*. (See Chapters 1 and 6 for more on modulation techniques.) Reeves proposed an alternative to analog-based frequency and amplitude modulation: that signals be sampled at discrete points in time, each sample be quantized and encoded in binary, and the bits be transmitted as pulses representing 0s and 1s. It sounds familiar, doesn't it? When you realize that 1937 predates the advent of digital audio as we know it today, you realize how far ahead of his time Reeves was. The term PCM is still used in digital audio to refer to the sampling and quantization process. In fact, PCM files are files that are digitized but not compressed. When you want to save your files in "raw" version, you can save them as PCM files.

Supplement audio processing programs:

hands-on worksheet

**ASIDE:** A number of bit-saving alternatives to PCM have been devised, especially in the field of telephony, where compressing the signal is important for preserving bandwidth. Statistically based methods rely on the characteristic nature of voice signals. Because the amplitudes of sounds in human speech do not change dramatically from one moment to the next, it's possible to reduce the bit depth by recording only the difference between one sample and the succeeding one. This is called ***differential pulse code modulation*** (***DPCM***). The algorithm can be fine-tuned so that the quantization level varies in accordance with the dynamically changing characteristics of the speech pattern, yielding better compression rates. This is called ***adaptive differential pulse-code modulation*** (***ADPCM***).

When you create a new audio file in a digital audio processing program, you are asked to choose the sampling rate and bit depth. Audio sampling rates are measured in cycles per second, designated in units of Hertz. In the past, the most common choices were 8000 Hz mono for telephone quality voice, or 44.1 kHz two-channel stereo with 16 bits per channel for CD-quality sound. Digital audio tape (DAT) format uses a sampling rate of 48 kHz. Now higher sampling rates and bit depths have become more common (*e.g.*, sampling rates of 96 or 192 kHz for two-channel stereo DVD with 24 bits per channel). In general, your

**ASIDE:** In the context of digital audio, the bit depth is sometimes referred to as *resolution*, which is a little confusing because in digital imaging *resolution* relates to sampling rate.

sampling rate and bit depth should match that of other audio clips with which you might be mixing the current one. They should also be appropriate for the type of sound you're recording, the storage capacity of the medium on which the audio will be stored, and the sensitivity of the equipment on which it will be played. You may sometimes want to use a higher sampling rate than you will ultimately need so that less error is introduced if you add special effects. Sometimes 32-bit samples are used initially for greater accuracy during sound processing, after which the audio can be downsampled to 16 bits before compression. Most good audio processing programs do this automatically, behind the scenes. This is why it is recommended that you use dither on your output even if you're working with a 16-bit file and saving it as 16 bits. Even though you, the user, may not be doing any bit depth conversions, the software might be behind the scenes.

The concepts of sampling rate and bit depth that you find in digital imaging carry over to digital audio processing as well. Just as was true with digital imaging, the sampling rate for digital audio must obey the Nyquist theorem, meaning that it must be at least twice the frequency of the highest frequency component in the audio being captured. The bit depth puts a limit on the precision with which you can represent each sample, determining the signal-to-quantization-noise ratio and dynamic range. In the next sections, we'll examine these issues closely as they apply to digital audio representation.

## 4.4 SAMPLING RATE AND ALIASING

There are two related, but not synonymous, terms used with regard to the Nyquist theorem: Nyquist frequency and Nyquist rate. Given a sampling rate, the ***Nyquist frequency*** is the highest actual frequency component that can be sampled at the given rate without aliasing. Based on the Nyquist theorem, the Nyquist frequency is half the given sampling rate. For example, if you choose to take samples at a rate of 8000 Hz (*i.e.*, 8000 samples/s), then the Nyquist frequency is 4000 Hz. This means that if the sound you're digitizing has a frequency component greater than 4000 Hz, then that component will be aliased—that is, it will sound lower in pitch than it should.

Given an actual frequency to be sampled, the ***Nyquist rate*** is the lowest sampling rate that will permit accurate reconstruction of an analog digital signal. The Nyquist theorem tells us that the Nyquist rate is twice the frequency of the highest frequency component in the signal being sampled. For example, if the highest frequency component is 10,000 Hz, then the Nyquist rate is 20,000 Hz. The two terms are summarized in the key equations below. (Unfortunately, some sources incorrectly use Nyquist frequency and Nyquist rate interchangeably.)

### KEY EQUATION

Given $f_{\max}$, the frequency of the highest-frequency component in an audio signal to be sampled, then the ***Nyquist rate***, $f_{nr}$, is defined as

$$f_{nr} = 2f_{\max}$$

KEY EQUATION

Given a sampling frequency $f_{samp}$ to be used to sample an audio signal, then the ***Nyquist frequency***, $f_{nf}$, is defined as

$$f_{nf} = \frac{1}{2} f_{samp}$$

The main point is that when you're digitizing an analog audio wave and your sampling rate is below the Nyquist rate, ***audio aliasing*** will occur. That is, when the digitized sound is played, the frequency from the original sound will be translated to a different frequency, so the digitized sound doesn't sound exactly like the original. Let's look at how this happens, beginning with single-frequency waves, from which we can generalize to waves with more than one frequency component.

In essence, the reason a too-low sampling rate results in aliasing is that there aren't enough sample points from which to accurately interpolate the sinusoidal form of the original wave. If we take *more* than two samples per cycle on an analog wave, the wave can be precisely reconstructed from the samples, as shown in Figure 4.3. In this example, the wave being sampled has a frequency of 637 Hz, which is 637 cycles/s. This means that we need to sample it at a Nyquist rate of at least 1274 samples/s (*i.e.*, 1274 Hz). The black dots on the figure show where the samples are taken, at a rate of 1490 Hz, greater than the Nyquist rate. No aliasing occurs when the wave is reconstructed from the samples because there is sufficient information to reconstruct the wave's sinusoidal form.

**ASIDE:** The example where samples are taken exactly twice per cycle illustrates why there is some confusion in the literature over the definition of the Nyquist rate. The *Nyquist rate* is sometimes defined as "at least twice the actual frequency of the highest frequency component" and sometimes as "greater than twice the actual frequency of the highest frequency component." Sampling at exactly twice the frequency of the highest-frequency component can work but isn't guaranteed to work, as shown in the example.

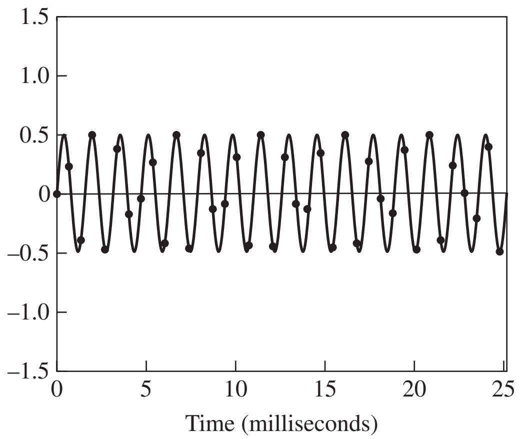


**Figure 4.3** Samples taken more than twice per cycle will provide sufficient information to reproduced the wave with no aliasing

Supplements on audio aliasing:

interactive tutorial

worksheet

mathematical modeling worksheet

If we have *exactly* two samples per cycle and the samples are taken at precisely the maximum and minimum values of the sine wave, once again the digitized wave can be reconstructed, as shown in Figure 4.4. However, if the samples are taken at locations other than peaks and troughs, the frequency may be correct but the amplitude incorrect. In fact, the amplitude can be 0 if samples are always taken at 0 points.

A wave sampled fewer than two times per cycle cannot be accurately reproduced. In Figure 4.5, we see the result of sampling a 637 Hz wave at 1000 Hz, resulting in an aliased wave of 363 Hz. The inadequate sampling rate skips over some of the cycles, making it appear that the frequency of the actual wave is lower than it really is. (The actual frequency is the sine wave drawn with the dashed line in the background.)

Figure 4.6 shows a 637 Hz wave sampled at 500 Hz. Again, the sampling rate is below the Nyquist rate. In this case, the aliased wave has a frequency of 137 Hz.

Figure 4.7 shows a 637 Hz wave sampled at 400 Hz, yielding an aliased wave at 163 Hz.

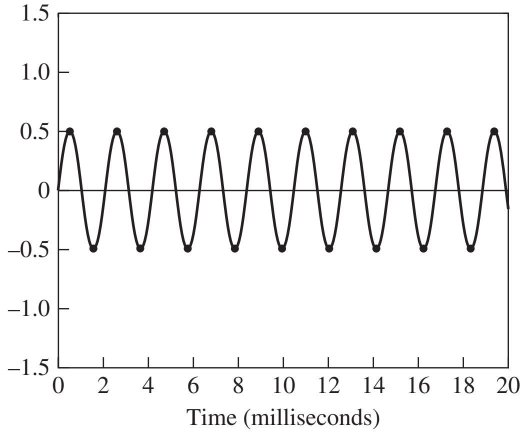


**Figure 4.4** Samples taken exactly twice per cycle *can* be sufficient for digitizing the original with no aliasing

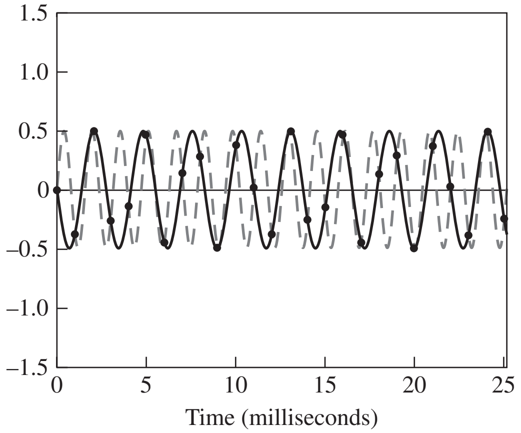


**Figure 4.5** A 637 Hz wave sampled at 1000 Hz aliases to 363 Hz

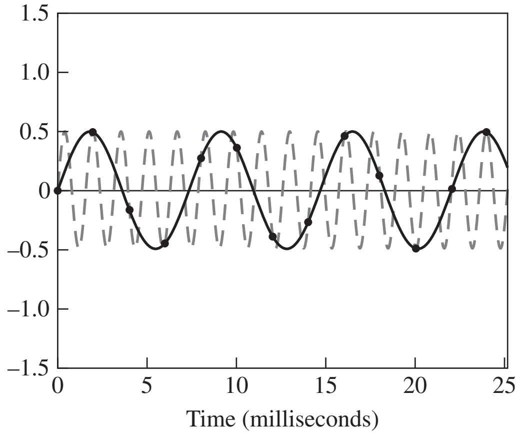


**Figure 4.6** A 637 Hz wave sampled at 500 Hz aliases to 137 Hz

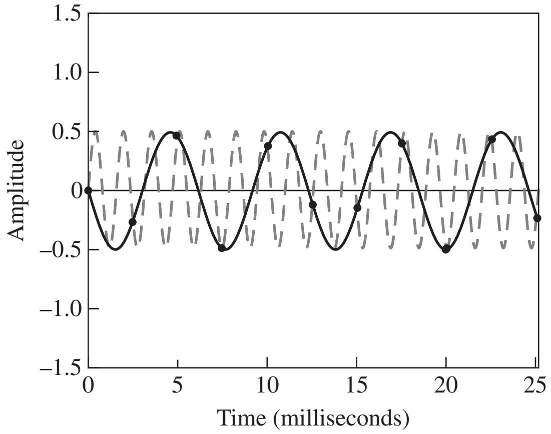


**Figure 4.7** A 637 Hz wave sampled at 400 Hz aliases to 163 Hz

Algorithm 4.1 shows how to compute the frequency of the aliased wave where aliasing occurs. Let's do an example of cases 2, 3, and 4.

### ALGORITHM 4.1

```
algorithm get_frequency
/*Input:  Frequency of the analog audio wave (a single tone)
          to be sampled, $f\_act$
          Sampling frequency, $f\_samp$
Output:   Frequency of the digitized audio wave, $f\_obs$*/
{
 $f\_nf = \frac{1}{2} f\_samp$
              /*$f\_nf$ is the Nyquist frequency*/
/*CASE 1*/
   if ($f\_act \le f\_nf$) then
     $f\_obs = f\_act$
/*CASE 2*/
   else if ($f\_nf < f\_act \le f\_samp$) then
     $f\_obs = f\_samp - f\_act$
   else {
     INT = $f\_act/f\_nf$ /* integer division */
     REM = $f\_act \bmod f\_nf$
/*CASE 3*/
     if (INT is even) then
       $f\_obs$ = REM
/*CASE 4*/
     else if (INT is odd) then
       $f\_obs = f\_nf -$ REM
  }
}
```

**Case 2:**
$f\_act = 637$ Hz; $f\_samp = 1000$ Hz; thus $f\_nf = 500$ Hz
$f\_nf < f\_act \leq f\_samp$
Therefore, $f\_obs = f\_samp - f\_act = 363$ Hz
(See Figure 4.5.)

**Case 3:**
$f\_act = 637$ Hz; $f\_samp = 500$ Hz; thus $f\_nf = 250$ Hz
$f\_act > f\_samp$
INT $= f\_act/f\_nf = 637/250 = 2$
(Note: / is integer division, which means throw away the remainder)
INT is even
REM $= f\_act \bmod f\_nf = 137$
(Note: *mod* saves only the remainder from the division)
Therefore, $f\_obs =$ REM $= 137$ Hz
(See Figure 4.6.)

**Case 4:**
$f\_act = 637$ Hz; $f\_samp = 400$ Hz; thus $f\_nf = 200$ Hz
$f\_act > f\_samp$
INT $= f\_act/f\_nf = 637/200 = 3$
INT is odd
REM $= f\_act \bmod f\_nf = 37$
Therefore, $f\_obs = f\_nf -$ REM $= 200 - 37 = 163$ Hz
(See Figure 4.7.)

It may be helpful to get a sense of how the algorithm works by looking at it as a function with one independent variable and graphing it. Assume that *f_samp* is given as a constant, *f_act* is the input to the function, and *f_obs* is the output. The function is graphed in Figure 4.8. A graph such as this can be drawn for any given sampling rate. Along the horizontal axis you have *f_act*, and along the vertical axis you have *f_obs*. The four cases of the algorithm always lie along the same parts of the graph: Case 1 is along the upward slope of the first triangle, case 2 on the downward slope of the first triangle, case 3 on the upward slope of any succeeding triangle, and case 4 on the downward slope of any succeeding triangle. All the triangles peak at $f\_obs = f\_nf$. The sampling rate is fixed for a particular graph, and it determines the Nyquist frequency. That is, $f\_nf = \frac{1}{2} f\_samp$ tells you the highest frequency that can be sampled at the given sampling rate without aliasing. Once you have fixed the sampling rate and corresponding Nyquist frequency and drawn the graph, you can consider any actual frequency *f_act* and predict what its corresponding observed frequency *f_obs* will be given the sampling rate *f_samp*.

A different graph can be drawn by fixing the actual frequency of a sound wave and graphing the observed frequency as a function of the sampling rate. As long as the sampling rate is below the Nyquist rate, the observed frequency will always be less than the actual frequency. This is shown in Figure 4.9.

Audio editing programs allow you to resample audio files. You sometimes may want to do this to lower the sampling rate and reduce the file size. As you have seen, using a lower

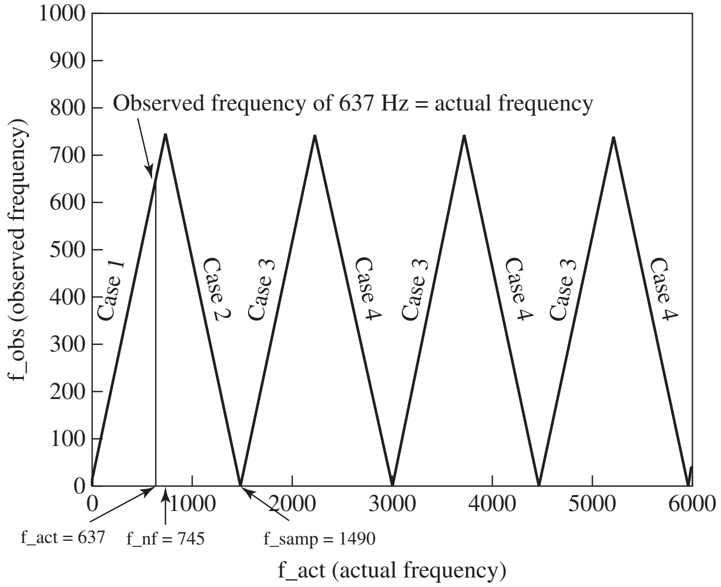


**Figure 4.8** The relationships among sampling rate, actual frequency, and observed frequency when $f_{samp} = 1490$

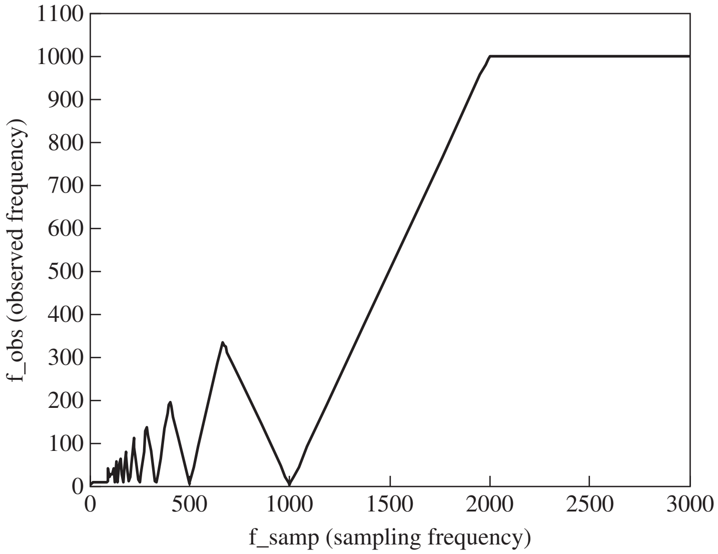


**Figure 4.9** Graph of aliasing function with fixed actual frequency of 1000 Hz

sampling rate may introduce aliased frequencies. However, audio editing programs have filters that will eliminate the high frequency components before resampling in order to avoid aliasing. You should look for these options when you work with your sound files. Filtering is also done in hardware analog-to-digital converters to avoid aliasing when a digital recording is made.

## 4.5 QUANTIZATION AND QUANTIZATION ERROR

### 4.5.1 Decibels and Dynamic Range

As you have seen in the previous section, sampling rate relates directly to the frequency of a wave. Quantization, on the other hand, relates more closely to the amplitude of a sound wave. ***Amplitude*** measures the intensity of the sound and is related to its perceived loudness. It can be measured with a variety of units, including voltages, newtons/m$^2$, or the unitless measure called decibels. To understand decibels it helps to consider first how amplitude can be measured in terms of air pressure.

In Chapter 1, we described how a vibrating object pushes molecules closer together, creating changes in air pressure. Since this movement is the basis of sound, it makes sense to measure the loudness of a sound in terms of air pressure changes. Atmospheric pressure is customarily measured in pascals (newtons/meter$^2$) (abbreviated Pa or N/m$^2$). The average atmospheric pressure at sea level is approximately 10$^5$ Pa. For sound waves, ***air pressure amplitude*** is defined as the average deviation from normal background atmospheric air pressure. For example, the ***threshold of human hearing*** (for a 1000 Hz sound wave) varies from the normal background atmospheric air pressure by $2 * 10^{-5}$ Pa, so this is its pressure amplitude.

Measuring sound in terms of pressure amplitude is intuitively easy to understand, but in practice decibels are a more common, and in many ways a more convenient, way to measure sound amplitude. Decibels can be used to measure many things in physics, optics, electronics, and signal processing. A decibel is not an absolute unit of measurement. A decibel is always based upon some agreed-upon reference point, and the reference point varies according to the phenomenon being measured. In networks, for example, decibels can be used to measure the attenuation of a signal across the transmission medium. The reference point is the strength of the original signal, and decibels describe how much of the signal is lost relative to its original strength. For sound, the reference point is the air pressure amplitude for the threshold of hearing. A decibel in the context of sound pressure level is called ***decibels-sound-pressure-level*** (***dB_SPL***).

KEY EQUATION

Let $E$ be the pressure amplitude of the sound being measured and $E_0$ be the sound pressure level of the threshold of hearing. Then ***decibels-sound-pressure-level***, (***dB_SPL***) is defined as

$$dB\_SPL = 20 \log_{10}\left(\frac{E}{E_0}\right)$$

Often, this is abbreviated simply as *dB*, but since decibels are always relative, it is helpful to indicate the reference point if the context is not clear. With this use of decibels, $E_0$, the threshold of hearing, is the point of comparison for the sound being measured.

Given a value for the air pressure amplitude, you can compute the amplitude of sound in decibels with the equation above. For example, what would be the amplitude of the audio threshold of pain, given as 30 Pa?

$$dB\_SPL = 20 \log_{10}\left(\frac{30 \text{ Pa}}{0.00002 \text{ Pa}}\right) = 20 \log_{10}(1500000) = 20 * 6.17 \approx 123$$

Thus, 30 N/m$^2$, the threshold of pain, is approximately equal to 123 decibels. (The threshold of pain varies with frequency and with individual perception.) You can also compute the pressure amplitude given the decibels. For example, what would be the pressure amplitude of normal conversation, given as 60 dB?

$$\begin{aligned}
60 &= 20\log_{10}\left(\frac{x}{0.00002\text{ Pa}}\right)\\
60 &= 20\log_{10}\left(\frac{50000x}{\text{Pa}}\right)\\
3 &= \log_{10}\left(\frac{50000x}{\text{Pa}}\right)\\
10^3 &= \frac{50000x}{\text{Pa}}\\
\frac{1000}{50000}\text{Pa} &= x\\
x &= 0.02\text{ Pa}
\end{aligned}$$

Thus, 60 dB is approximately equal to 0.02 Pa.

Decibels can also be used to describe sound intensity (as opposed to sound pressure amplitude). ***Decibels-sound-intensity-level*** (***dB_SIL***) is defined as $dB\_SPL = 10\log_{10}\left(\frac{I}{I_0}\right)$. $I_0$ is the intensity of sound at the threshold of hearing, given as $10^{-12}$ W/m$^2$. (W is watts.) It is sometimes more convenient to work with intensity decibels rather than pressure amplitude decibels, but essentially the two give the same information. The relationship between the two lies in the relationship between pressure (*potential* in volts) and intensity (*power* in watts). In this respect, *I* is proportional to the square of *E* (discussed in Chapter 1).

Decibels-sound-pressure-level are an appropriate unit for measuring sound because the values increase logarithmically rather than linearly. This is a better match for the way humans perceive sound. For example, a voice at normal conversation level could be 100 times the air pressure amplitude of a soft whisper, but to human perception it seems only about 16 times louder. Decibels are scaled to account for the nonlinear nature of human sound perception. Table 4.1 gives the decibels of some common sounds. The values in Table 4.1 vary with the frequency of the sound and with individual hearing ability.

**TABLE 4.1 Approximate Decibel Levels of Common Sounds**

| Sound | Decibels (dB_SPL) |
|---|---|
| Threshold of hearing | 0 |
| Rustling leaves | 20 |
| Conversation | 60–70 |
| Jackhammer | 100 (or more) |
| Threshold of pain | 130 |
| Damage to eardrum | 160 |

Experimentally, it has been determined that if you increase the amplitude of an audio recording by 10 dB, it will sound about twice as loud. (Of course, these perceived differences are subjective.) For most humans, a 3 dB change in amplitude is the smallest perceptible change.

While an insufficient sampling rate can lead to aliasing, an insufficient bit depth can create ***distortion***, also referred to as ***quantization noise***. In Chapter 1, we showed that signal-to-quantization-noise-ratio, SQNR, is defined as $SQNR = 20\log_{10}(2^n)$ where $n$ is the bit depth of a digital file. This can be applied to digital sound and related to the concept of dynamic range. Dynamic range is the ratio between the smallest nonzero value, which is 1, and the largest, which is $2^n$. For an $n$-bit file, the ratio, expressed in decibels, is then $20\log_{10}\left(\frac{2^n}{1}\right) = 20n\log_{10}(2)$. Thus, the definition is identical to the definition of SQNR, and this is why you see the terms SQNR and dynamic range sometimes used interchangeably.

We can simplify $20n\log_{10}(2)$ even further by taking $\log_{10}(2)$, which is about 0.30103, and multiplying by 20.

### KEY EQUATION

Let $n$ be the bit depth of a digital audio file. Then the ***dynamic range of the audio file***, $d$, in decibels, is defined as

$$d = 20n\log_{10}(2) \approx 6n$$

As a rule of thumb you can estimate that an $n$-bit digital audio file has a dynamic range (or, equivalently, a signal-to-noise-ratio) of $6n$ dB. For example, a 16-bit digital audio file has a dynamic range of about 96 dB, while an 8-bit digital audio file has a range of about 48 dB.

Be careful not to interpret this to mean that a 16-bit file allows louder amplitudes than an 8-bit file. Rather, dynamic range gives you a measure of the range of amplitudes that can be captured relative to the loss of fidelity compared to the original sound. Dynamic range is a relative measurement—the relative difference between the loudest and softest parts representable in a digital audio file, as a function of the bit depth.

There is a second way in which the term dynamic range is used. We've defined it as it applies to any file of a given bit depth. The term can also be applied to a particular audio piece, not related to bit depth. (In this usage, you don't even have to be talking about digital audio.) A particular piece of music can be said to have a wide dynamic range if there's a big difference between the loudest and softest parts of the piece. Symphonic classical music typically has a wide dynamic range. "Elevator music" is produced so that it doesn't have a wide dynamic range, and can lie in the background unobtrusively.

Let's return to the term decibels now. You'll find another variation of decibels when you use audio processing programs, where you may have the option of choosing the units for sample values. Units are shown on the vertical axes in the waveforms of Figure 4.10. On the left, we've chosen the *sample units* view. On the right, we've chosen the *decibels* view. However, the decibels being displayed are ***decibels-full-scale*** (***dBFS***) rather than the decibels-sound-pressure-level defined above.

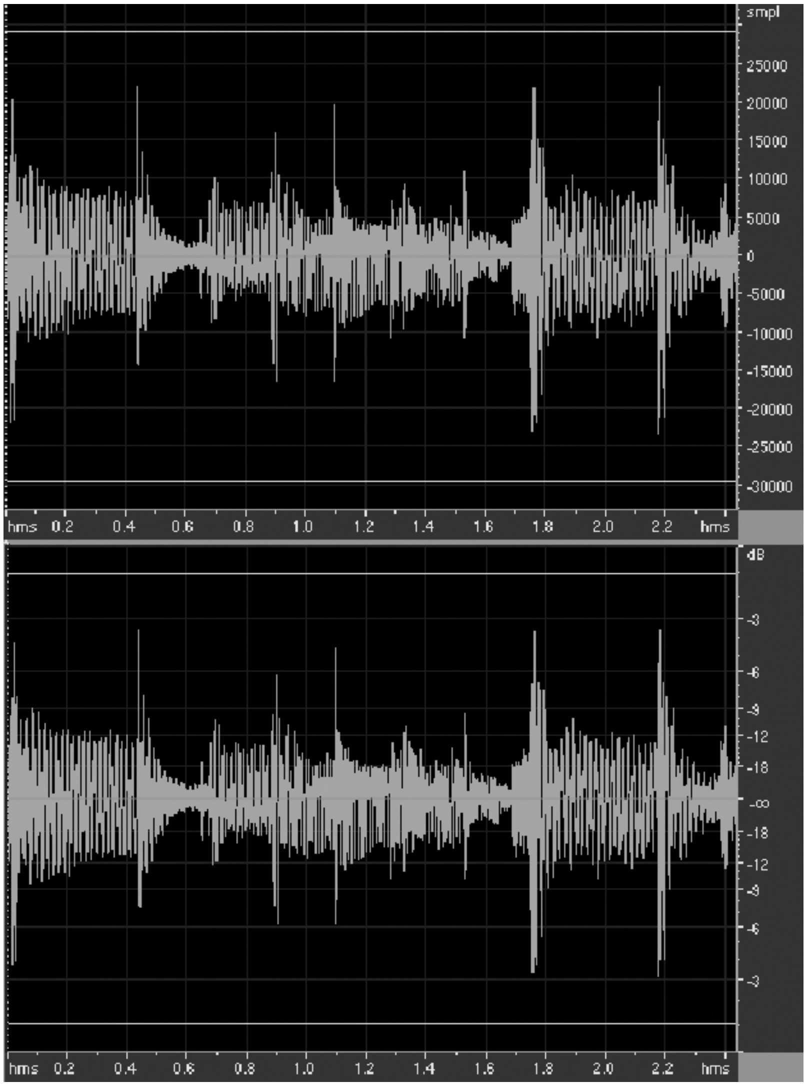


**Figure 4.10** Measuring amplitude in samples or decibels (from Audition)
Units are samples in top window, decibels in bottom window

The idea behind dBFS is that it makes sense to use the maximum possible amplitude as a fixed reference point and move down from there. There exists some maximum audio amplitude that can be generated by the system on which the audio processing program is being run. Because this maximum is a function of the system and not of a particular audio file, it is the same for all files and does not vary with bit depth. This maximum is given the value 0. When you look at a waveform with amplitude given in dBFS, the horizontal center of the waveform is $-\infty$ dBFS, and above and below this axis the values progress to the maximum of 0 dBFS. This is shown in the window on the right in Figure 4.10. The bit depth of each audio file determines how much lower you can go below the maximum amplitude before the sample value is reduced to 0.

This is the basis for the definition of dBFS, which measures amplitude values relative to the maximum possible value. For *n*-bit samples, dBFS is defined as follows:

KEY EQUATION

Let $x$ be an $n$-bit audio sample in the range of $-2^{n-1} \leq x \leq 2^{n-1} - 1$. Then $x$'s value expressed as ***decibels-full-scale***, **dBFS,** is

$$dBFS = 20 \log_{10}\left(\frac{|x|}{2^{n-1}}\right)$$

Try the definition of dBFS on a number of values, using $n = 16$. You'll find that a sample value of −32768 maps to 0, the maximum amplitude possible for the system; 10,000 maps to −10.3; 1 maps to −90.3; and 0.5 maps to −96.3296. These values are consistent with what you learned about dynamic range. A 16-bit audio file has a dynamic range of about 96 decibels. Any samples that map to decibel values that are more than 96 decibels below the maximum possible amplitude effectively are lost as silence.

## 4.5.2 Audio Dithering

Supplements on audio dithering:

interactive tutorial

mathematical modeling worksheet

***Audio dithering*** is a way to compensate for quantization error. Surprisingly, the way to do this is to add small random values to samples in order to mask quantization error. The rounding inherent in quantization causes a problem in that at low amplitudes, many values may be rounded down to 0. Since 0 is simply silence, this can cause noticeable breaks in the sound. If small random values between 0 and the least significant bit (on the scale of the new bit depth) are added to the signal before it is quantized, some of the samples that would have been lost will no longer fall to 0. Adding a little bit of noise to the signal is preferable to having discontinuities of silence.

Dithering is customarily performed by analog-to-digital converters before the quantization step. You'll also encounter the dithering option if you decide to reduce the bit depth of an audio file. If you know how audio dithering works, you'll know what effect it will have and you'll be able to make a more informed choice of a dithering function.

A good way to understand the effect that quantization error has on sound is to look at it graphically. Figure 4.11 shows a continuous waveform, the wave quantized to 16 quantization levels, and the quantization error wave. Note that the original wave plus the error wave equals the quantized wave. The error wave constitutes another sound component that can be heard as a distortion of the signal.

Although in the previous section we used the term *noise* with regard to quantization error, *distortion* is really more accurate. Notice that the error waveform is periodic; that is, it repeats in a regular pattern, and its period is related to the period of the original wave. This is the distinction that some sources make between distortion and noise. Noise is random, while distortion sounds meaningful even though it is not. For this reason, distortion can be more distracting in an audio signal than noise. The distortion wave moves in a pattern along with the original wave and thus, to human hearing, it seems to be a meaningful part of the sound. It is

**ASIDE:** The term signal-to-noise ratio is widely used with regard to quantization error, but this type of noise might more correctly be called distortion. However, not all sources make a distinction between distortion and noise.

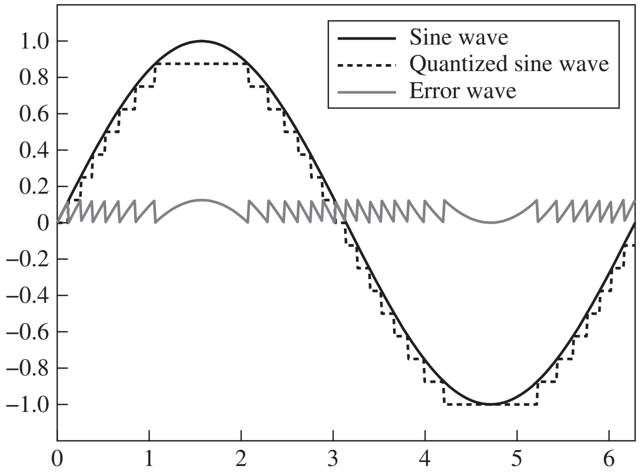


**Figure 4.11** Original wave (the sine wave), quantized sine wave, and quantization error wave

easier for the human brain to tune out noise because it seems to have no logical relationship to the dominant patterns of the sound.

It may be counterintuitive to understand that artificially introducing noise (the random kind) can actually have a helpful effect on sound that is quantized in an insufficient number of bits, but it's possible to add noise to a quantized audio wave in a way that reduces the effects of distortion. Imagine that we add between –1 and 1 unit, on the scale of the reduced bit depth, to each sample. The exact amount, *x*, to be added to the sample could be determined at random by a ***triangular probability density function*** **(*TPDF*)**. The function, shown in Figure 4.12, indicates that there is the greatest probability that 0 will be added to a sample. As *x* goes from 0 to 1 (and symmetrically as *x* goes from 0 to −1), the probability that *x* is the value to be added to a sample decreases. (A simple way to generate values for a triangular probability density function is to take the sum of two random numbers between −0.5 and 0.5.)

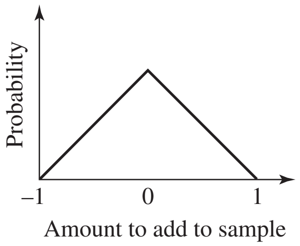


**Figure 4.12** Triangular probability function

Adding this random noise to the original wave eliminates the sharp stairstep effect in the quantized signal. Instead, the quantized wave jumps back and forth between two neighboring quantization levels. Another advantage is that with the addition of the small random value, there aren't as many neighboring low-amplitude values that are rounded to 0 when they are quantized. Neighboring low-amplitude values that become 0 in quantization create disturbing breaks in the audio. (You would be able to picture this better in an actual music audio clip than in the simple waveform pictured in Figure 4.11.)

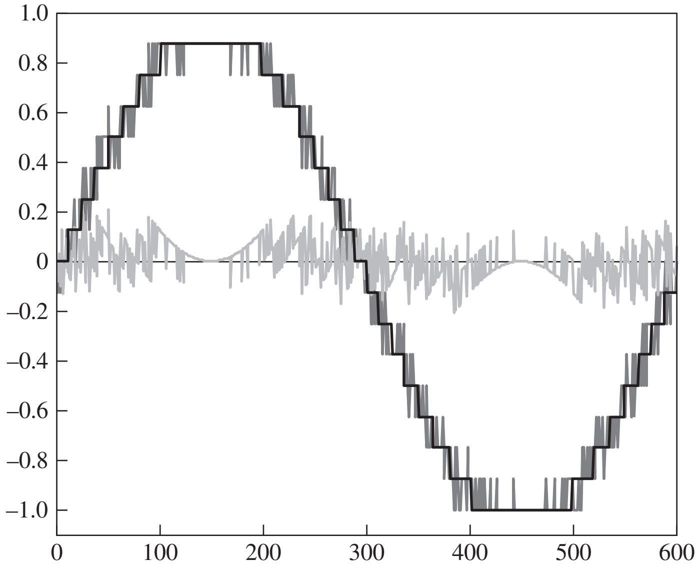


**Figure 4.13** Quantized sine wave, dithered quantized wave, and (along horizontal axis) error wave including dithering

Figure 4.13 shows dithering with the triangular dithering function, which produces random values between −1 and 1. (It's assumed in this figure that the signal is originally at a bit depth of 16 and is being reduced to a bit depth of four. A bit depth of four is unrealistically low but serves to illustrate the effect.) The dithered wave in the figure represents the original quantized wave to which the dither function has been added. The resulting dithered wave generally has fewer disturbing audio artifacts than the undithered wave does. By "artifacts," we mean areas where the sound breaks up or goes to silence.

Other dithering functions include the ***rectangular probability density function* (*RPDF*)**, the ***Gaussian PDF***, and ***colored dithering***. The RPDF generates random numbers such that all numbers in the selected range have the same probability of being generated. While the TPDF randomly generates numbers within a range of two units (*e.g.*, −1 to +1), the RPDF works better if numbers are chosen in a range of 1 unit (*e.g.*, 0 to 1). The Gaussian PDF is like the TPDF except that it weights the probabilities according to a Gaussian rather than a triangular shape. Gaussian dither creates noise that resembles common environmental noises, like tape hiss. Colored dithering produces noise that is primarily in higher frequencies rather than in the frequencies of human hearing. It's best to apply colored dithering only if no more audio processing will be done, since the noise it generates can be amplified by other effects applied afterwards. The TPDF is best to use when the audio file will be undergoing more processing.

Supplement on noise shaping:

mathematical modeling worksheet

## 4.5.3 Noise Shaping

***Noise shaping*** is another way to compensate for the quantization error. It is an important component in the design of analog-to-digital and digital-to-analog converters. You can also opt to use noise shaping in conjunction with dithering if you reduce the bit depth of an audio file. Noise shaping is *not* dithering, but it is often used along with dithering. The idea behind noise shaping is to redistribute the quantization error so that the noise is concentrated in the higher frequencies, where human hearing is less sensitive. Noise shaping algorithms, first developed by Cutler in the 1950s, work by computing the error that results from quantizing the *i*th sample and adding this error to the next sample, before that next sample is itself quantized. Say that you have an array $\boldsymbol{F}$ of audio samples. Consider the following definition of a first-order feedback loop for noise shaping:

## KEY EQUATIONS

Let $\boldsymbol{F\_in}$ be an array of $N$ digital audio samples that are to be quantized, dithered, and noise shaped, yielding $\boldsymbol{F\_out}$. For $0 \leq i \leq N - 1$, define the following:
$\boldsymbol{F\_in}_i$ is the *i*th sample value, not yet quantized.

$\boldsymbol{D}_i$ is a random dithering value added to the *i*th sample.

The assignment statement $\boldsymbol{F\_in}_i = \boldsymbol{F\_in}_i + \boldsymbol{D}_i + c\boldsymbol{E}_{i-1}$ dithers and noise shapes the sample. Subsequently, $\boldsymbol{F\_out}_i = \lfloor \boldsymbol{F\_in}_i \rfloor$ quantizes the sample.

$\boldsymbol{E}_i$ is the error resulting from quantizing the *i*th sample after dithering and noise shaping.

For $i = -1, \boldsymbol{E}_i = 0$. Otherwise, $\boldsymbol{E}_i = \boldsymbol{F\_in}_i - \boldsymbol{F\_out}_i$.

**Equation 4.1**

Let's try an example.

Assume that audio is being recorded in 8 bit samples. On the scale of 8 bits, sound amplitudes can take any value between −128 and 128. (These values do not become integers between −128 and 127 until after quantization.)

Say that $\boldsymbol{F\_in}_0 = 68.2, \boldsymbol{F\_in}_1 = 70.4, \boldsymbol{D}_0 = 0.9, \boldsymbol{D}_1 = -0.6$, and $c = 1$. Then

$$\boldsymbol{F\_in}_0 = \boldsymbol{F\_in}_0 + \boldsymbol{D}_0 + c\boldsymbol{E}_{-1} = 68.2 + 0.9 + 0 = 69.1$$

$$\boldsymbol{F\_out}_1 = \lfloor \boldsymbol{F\_in}_0 \rfloor = 69$$

$$\boldsymbol{E}_0 = \boldsymbol{F\_in}_0 - \boldsymbol{F\_out}_0 = 69.1 - 69 = 0.1$$

$$\boldsymbol{F\_in}_1 = \boldsymbol{F\_in}_1 + \boldsymbol{D}_1 + c\boldsymbol{E}_0 = 70.4 - 0.6 + 0.1 = 69.9$$

$$\boldsymbol{F\_out} = \lfloor \boldsymbol{F\_in}_0 \rfloor = 69$$

$$\boldsymbol{E}_1 = \boldsymbol{F\_in}_1 - \boldsymbol{F\_out}_1 = 69.9 - 69 = 0.9$$

To understand the benefit of noise shaping, think about the frequency spectrum of quantization noise—that is, the range of the frequency components—when noise shaping is *not* used. Quantization noise is part of the original audio signal in the sense that it shares the original signal's sampling rate, and thus its frequency components are spread out over the same range. By the Nyquist theorem, the highest valid frequency component is half the frequency of the sampling rate, and without noise shaping, this is where the frequency components of the quantization noise lie. If we filter out frequencies above the Nyquist frequency, we're not losing anything we care about in the sound. The more of the noise's frequency components we can move above the Nyquist frequency, the better. This is what noise shaping does. The idea is that if the error from dithering and quantizing $\boldsymbol{F}_i$ is larger than the previous one, then the error for $\boldsymbol{F}_{i+1}$ ought to be smaller for $\boldsymbol{F}_i$. Making the error wave go up and down rapidly has the effect of moving the error wave to a higher frequency. The feedback loop inherent in noise shaping has the effect of spreading out the noise's frequency components over a broader band so that more of them are at high levels and can be filtered out.

The term *shaping* is used because you can manipulate the "shape" of the noise by manipulating the noise shaping equations, adding more error terms and using different multipliers for the error terms. The general statement for an *n*th order noise shaper noise shaping equation becomes $\boldsymbol{F\_out}_i = \boldsymbol{F\_in}_i + \boldsymbol{D}_i + c_{i-1}\boldsymbol{E}_{i-1} + c_{i-2}\boldsymbol{E}_{i-2} + \cdots + c_{i-n}\boldsymbol{E}_{i-n}$. The coefficients of the error terms can be used to control the frequencies generated. Ninth-order noise shapers are not uncommon. The POW-r noise shaping algorithm, for example, uses a 9th order formula to requantize audio from 24 to 16 bits.

When you reduce the bit depth of an audio file in an audio processing program, you're given the options of using noise shaping along with dithering. Both operations reduce the negative effects of noise, but they do it in different ways. Recall from the previous section that some sources make a distinction between *distortion* and *noise*. Distortion is a certain kind of noise—noise that is correlated with the original signal, in that its frequency pattern follows the frequency pattern of the signal. Noise shaping doesn't do anything to dissociate the noise's frequency pattern from the signal, so it's important to use dithering in conjunction with noise shaping, not alone.

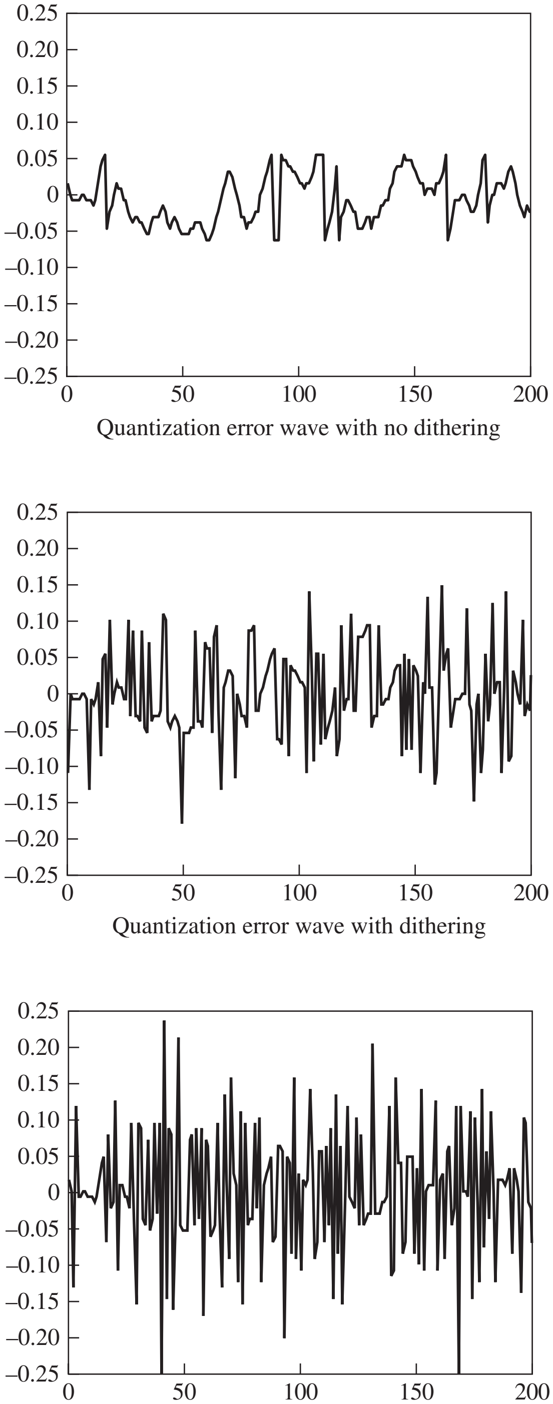
Figure 4.14 shows the effect of requantizing an audio file from 16 down to 4 bits. This is not something you are likely to do, since there's no reason to use so few bits, but it serves to illustrate the effect. Three versions of the quantization error are shown, in both waveform and spectral views. (To generate the error wave, you just subtract the requantized wave from the original one.) The first shows the error that is generated from requantization with no dithering. The second shows the error that results from requantization with dithering. The third shows the error that results from requantization with dithering and noise shaping. Dithering spreads out the error pretty evenly around the frequencies. Noise shaping moves the error to higher frequencies. A small clip of each wave is given on the right, and in this view as well you can see that the frequencies increase.

Dithering and noise shaping algorithms are provided for you in audio processing programs. You are given a choice of noise shaping algorithms to use along with dithering. These choices are associated with the sampling rate of your audio file. If your sampling rate is less than 32 kHz, noise shaping doesn't work very well. This is because the Nyquist frequency for this sampling rate is relatively low, so even when you spread the noise across a wider frequency band, some of it will probably still be in the range of human hearing.

### 4.5.4 Non-Linear Quantization

***Nonlinear encoding,*** or ***companding,*** is an encoding method that arose from the need for compression of telephone signals across low bandwidth lines. The word companding is derived from the fact that this encoding scheme requires compression and then expansion. In sketch, it works as follows:

- Take a digital signal with bit depth *n* and requantize it in *m* bits, $m < n$, using a nonlinear quantization method.



Quantization error wave with no dithering

Quantization error wave with dithering

Quantization error wave with dithering and noise shaping

**Figure 4.14** Quantization error

- Transmit the signal.
- Expand the signal to *n* bits at the receiving end. Some information will be lost, since quantization is lossy. However, nonlinear quantization lessens the error for low amplitude signals as compared to linear quantization.

The nonlinear encoding method is motivated by the observation that the human auditory system is perceptually nonuniform. Humans can perceive small differences between quiet sounds, but as sounds get louder our ability to perceive a difference in their amplitude diminishes. Also, quantization error generally has more impact on low amplitudes than on high ones. Think about why this is so. Say that your bit depth is eight. Then the quantization levels range for sound between −128 and 127. The sound amplitudes are scaled to this range. So that only integer values are used, values must be rounded to integers. If you consider the percent error for individual samples, you can easily see the relatively greater effect that a low bit depth has on low vs. high amplitude signals. A value of 0.499 rounds down to 0, for 100% error, while a value of 126.499 rounds down to 126, for about 0.4% error. In light of these observations, it makes sense to use *more* quantization levels for low amplitude signals and *fewer* quantization levels for high amplitudes. This is the idea behind nonlinear companding, the method used in $\mu$-law and A-law encoding for telephone transmissions.

Nonlinear companding schemes are widely used and have been standardized under the CCITT (Comité Consulatif Internationale de Télégraphique et Téléphonique) recommendations for telecommunications. In the United States and Japan, ***μ-law*** (also called ***mu-law***) ***encoding*** is the standard for compressing telephone transmissions, using a sampling rate of 8000 Hz and a bit depth of only eight bits, but achieving about 12 bits of dynamic range through the design of the compression algorithm. The equivalent standard for the rest of the world is called ***A-law encoding***. Let's look more closely at $\mu$-law to understand nonlinear companding in general. The encoding method is defined by the following function:

## KEY EQUATION

Let $x$ be a sample value normalized so that $-1 \le x < 1$. Let $sign(x) = -1$ if $x$ is negative and $sign(x) = 1$ otherwise. Then the ***μ-law function*** (also called ***mu-law***) is defined by

$$m(x) = sign(x)\left(\frac{\ln(1 + \mu|x|)}{\ln(1 + \mu)}\right)$$

$$= sign(x)\left(\frac{\ln(1 + 255|x|)}{5.5452}\right) \quad for \quad \mu = 255$$

The $\mu$-law function is graphed in Figure 4.15. You can see that it has a logarithmic shape. Its effect is to provide finer-grained quantization levels at low amplitudes compared to high.

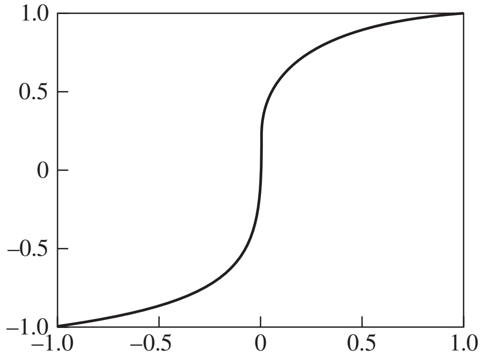


**Figure 4.15** Logarithmic function for nonlinear audio encoding

It may be easier to visualize what this function is doing if you think in terms of sample values rather than normalized values. Say that you begin with 16-bit audio samples with values ranging from −32,768 to 32,767. You're going to transmit the signal at a bit depth of 8, and expand it back to 16 bits at the receiving end.

To apply the $\mu$-law function, first normalize the input values by dividing by 32,768. Then apply the function to get $m(x)$. Then, compute $\lfloor 128m(x) \rfloor$ to scale the value to a bit depth of eight. Try this with an initial 16-bit sample of 16.

Supplements on $\mu$-law encoding:

interactive tutorial

mathematical modeling worksheet

programming exercise

Apply the $\mu$-law function: $m\left(\frac{16}{32{,}768}\right) \approx 0.02$

Scale to 8-bit samples: $\lfloor 128 * 0.02 \rfloor = 2$

Now let's try an initial value of 30,037.

Apply the $\mu$-law function: $m\left(\frac{30{,}037}{32{,}768}\right) = 0.9844$

Scale to 8-bit samples: $\lfloor 128 * 0.9844 \rfloor = 125$

The nonlinear companding values in Table 4.2 were computed by the method just shown. (Only positive values are shown in the table, but negatives work the same way.) You can see the benefit of nonlinear companding versus linear requantization in this table. Linear quantization using eight bits would create equal-sized quantization intervals—each of them containing 65,536/256 = 256 sample values ranging from −128 to 127. In nonlinear companding, the quantization intervals are smaller at lower amplitudes, resulting in fewer small-magnitude samples being mapped to the same value. The result is greater accuracy in the requantization at low amplitudes compared to the error you get with linear quantization.

After compression using the $\mu$-law function, samples can be transmitted in eight bits. At the user end, they are decompressed to 16 bits with the inverse function.

**TABLE 4.2 Comparison of Quantization Interval Size with Linear Requantization and Nonlinear Companding**

| Linear Requantization | | Nonlinear Companding | | |
|---|---|---|---|---|
| **Original 16-bit Sample Values** | **8-bit Sample Values After Linear Requantization (divide by 256 and round down to nearest integer)** | **Original 16-bit Sample Values** | **8-bit Sample Values After Non-linear Companding** | **Number of Values that are Mapped to the Same Value** |
| 0–255 | 0 | 0–5 | 0 | 6 |
| 256–511 | 1 | 6–11 | 1 | 6 |
| 512–767 | 2 | 12–17 | 2 | 6 |
| . . . | . . . | . . . | . . . | . . . |
| 32,000–32,255 | 125 | 28,759–30,037 | 125 | 1,279 |
| 32,256–32,511 | 126 | 30,038–31,373 | 126 | 1,336 |
| 32,512–32,767 | 127 | 31,374–32,767 | 127 | 1,394 |

## KEY EQUATION

Let $x$ be a $\mu$-law encoded sample normalized so that $-1 \leq x < 1$. Let $sign(x) = -1$ if $x$ is negative and $sign(x) = 1$ otherwise. Then the ***inverse* $\mu$*-law function*** is defined by

$$d(x) = sign(x)\left(\frac{(\mu + 1)^{|x|} - 1}{\mu}\right)$$

$$= sign(x)\left(\frac{256^{|x|} - 1}{255}\right) \quad for \quad \mu = 255$$

Continuing our example, let's see what happens with the sample that originally was 16. The $\mu$-law function, scaled to an 8-bit scale, yielded a value of 2. Reversing the process, we do the following:

Apply the inverse $\mu$-law function: $d\left(\frac{2}{128}\right) = 0.00035$

Scale to 16-bit samples: $\lceil 32{,}768 * 0.00035 \rceil = 11$

We can do the same computation for the sample that originally was 30,037 and yielded a value of 125 from the $\mu$-law function.

Apply the inverse $\mu$-law function: $d\left(\frac{125}{128}\right) = 0.8776$

Scale to 16-bit samples: $\lceil 32{,}768 * 0.8776 \rceil = 28{,}758$

An original sample of value 16 became 11 at the receiving end using $\mu$-law encoding, which is an error of about 31%. An original sample of 30,037 became 28,758 at the receiving end,

**TABLE 4.3 Comparison of Error with Linear Requantization Vs. Nonlinear Companding (representative values only)**

| | Linear Requantization | | | Nonlinear Companding | | |
|---|---|---|---|---|---|---|
| Original 16-bit Sample | 8-bit Sample After Compression | 16-bit Sample After Decompression | Percent Error | 8-bit Sample After Compression | 16-bit Sample After Decompression | Percent Error |
| 1–5 | 0 | 0 | avg. 100% | 0 | 0 | avg. 100% |
| 6–11 | 0 | 0 | avg. 100% | 1 | 6 | avg. 26% |
| 12–17 | 0 | 0 | avg. 100% | 2 | 12 | avg. 16% |
| 18–24 | 0 | 0 | avg. 100% | 3 | 18 | avg. 13% |
| 25–31 | 0 | 0 | avg. 100% | 4 | 25 | avg. 10% |
| 127 | 0 | 0 | 100% | 15 | 118 | 7% |
| 128 | 1 | 256 | 100% | 25 | 118 | 7.8% |
| 383 | 1 | 256 | 33% | 31 | 364 | 4.9% |
| 30,038 | 117 | 29,952 | 0.29% | 126 | 30,038 | 0% |
| 31,373 | 122 | 31,232 | 0.45% | 126 | 30,038 | 4.2% |

for an error of about 4%. There's still more error at low amplitudes than at high, but the situation is improved over the error you would get with linear quantization.

Think about how you might reduce the bit depth for transmission if you simply divided by 256 and rounded the values, as shown below. Say that we use $r(x) = round(x/256)$ for lowering the bit depth (compression) and $s(x) = 256x$ for expanding the bit depth back again (decompression).

Assuming that this is how you would do linear requantization, Table 4.3 shows the amount of error resulting from linear vs. nonlinear quantization at different amplitude levels. In general, nonlinear companding reduces the impact of error on low-amplitude samples, making it less than it would be with linear quantization.

With linear requantization, all 16-bit samples between 0 and 127 compress to 0 and decompress to 0. This is 100% error for the sample values between 1 and 127. In comparison, with nonlinear companding, 16-bit samples between, say, 12 and 17 compress to 2 and decompress to 12, for an average error of 16%.

Percent error for the two methods is graphed in Figure 4.16 and Figure 4.17. Note that the enlargement in Figure 4.16 covers samples only from 0 to 100, while the enlargement in Figure 4.17 shows samples from 0 to 2000.

Percent error has an inverse relation to signal-to-quantization noise ratio. A large SQNR is good; a large percent error is obviously not good. Nonlinear companding increases the SQNR—the dynamic range—compared to what it ordinarily would be for 8-bit samples. You recall that a digital audio file linearly quantized in $n$ bits has a dynamic range of approximately $6n$ decibels. Thus, a bit depth of eight ordinarily would give a dynamic range of 48 dB. $\mu$-law encoding, however, reduces the average error per sample, effectively yielding the equivalent of a 12-bit dynamic range, which is about 72 dB.

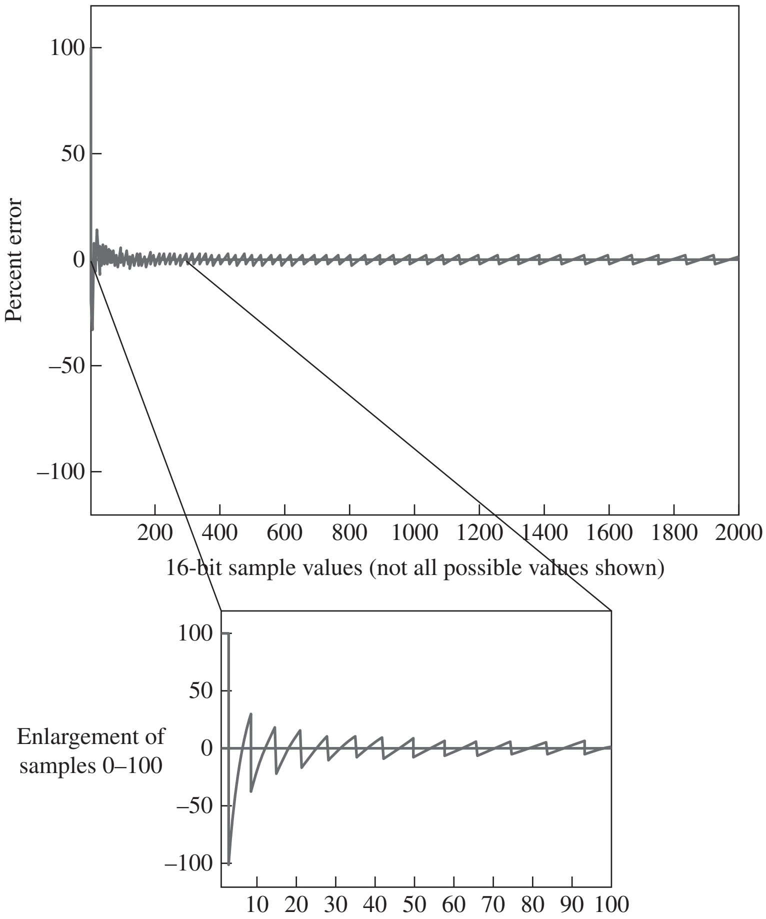


**Figure 4.16** Percent error with nonlinear companding

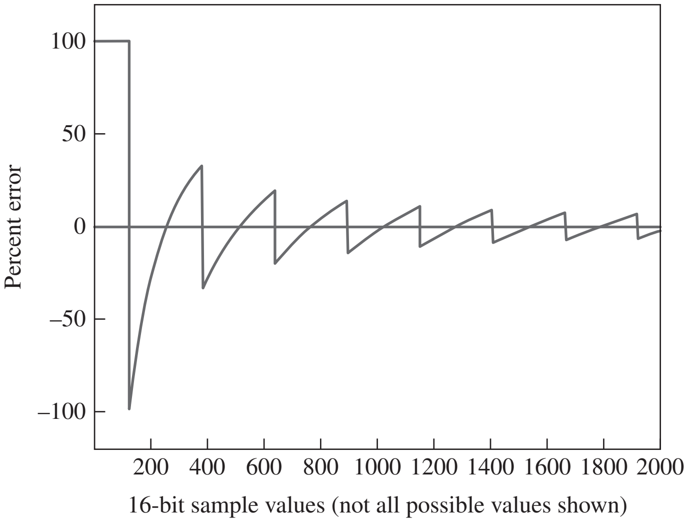


**Figure 4.17** Percent error with linear quantization

# 4.6 FREQUENCY ANALYSIS

## 4.6.1 Time and Frequency Domains

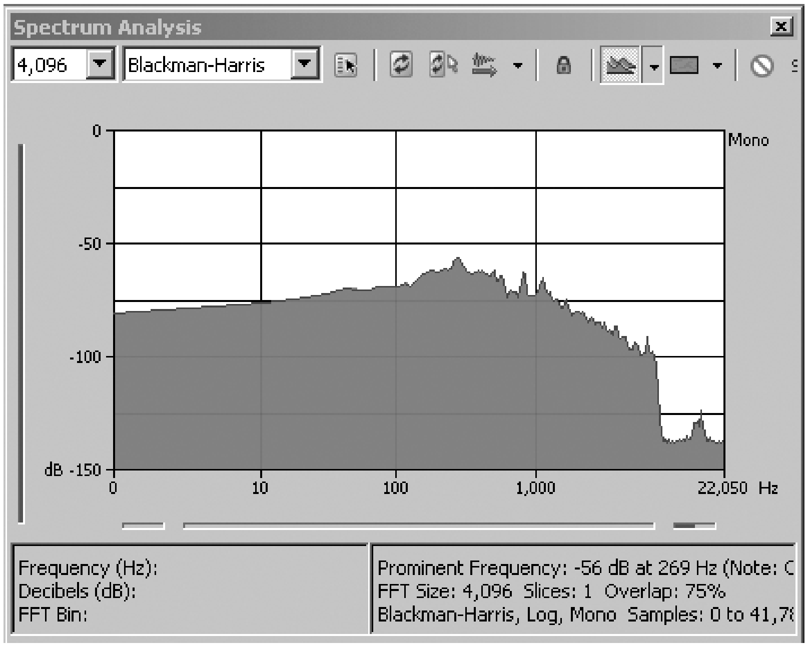
In Chapter 1, we introduced the idea that both sound and image data can be represented as waveforms—one-dimensional waveforms for sound and two-dimensional for images. We explained that any complex waveform is actually the sum of simple sinusoidal waves. This makes it possible to store sound and image data in two ways. Sound can be represented either over the time domain or the frequency domain. Similarly, images can be represented either over the spatial domain or the frequency domain. A transform is an operation that converts between one domain and the other. The transform that is used most frequently in digital audio processing is the Fourier transform. It's time to look more closely at the mathematics of this transform.

If you represent digital audio over the ***time domain***, you store the wave as a one-dimensional array of amplitudes—the discrete samples taken over time. This is probably the easiest way for you to think of audio data. If you think of it as a function, the input is time and the output is a sample value. But consider the alternative. A complex waveform is in fact equal to an infinite sum of simple sinusoidal waves, beginning with a ***fundamental frequency*** and going through frequencies that are integer multiples of the fundamental frequency. These integer multiples are called ***harmonic frequencies***. To capture the complex waveform, it is sufficient to know the amplitude and phase of each of the component frequencies. The amplitude is, in a sense, how much each frequency component contributes to the total complex wave. This is how the data is stored in the ***frequency domain***—as the amplitudes of frequency components. Sometimes we refer to these values as *coefficients*, because they represent the multiplier for each frequency in the summation. If you think of this as a function, the input is frequency and the output is the magnitude of the frequency component. It's useful to be able to separate the frequency components in order to analyze the nature of a sound wave and remove unwanted frequencies. Be sure you understand that the time domain and frequency domain are equivalent. They both fully capture the waveform. They just store the information about the waveform differently.

Audio processing programs provide information about the frequency components of an audio file. Two useful views of an audio file are the frequency analysis view and the spectral view.

In the ***frequency analysis view*** (also called the ***spectrum analysis***), frequency components are on the horizontal axis, and the amplitudes of the frequency components are on the vertical axis. The frequency analysis view is useful for seeing, in a glance, how much of each frequency you have in a segment of your audio file. (The magnitude of the frequency component will be defined formally below.) Notice that you can't take a frequency analysis at some instantaneous point in your audio file. Frequency implies a change in the amplitude of the wave over time, so some time must pass for there to be a frequency. You can select a portion of your audio file and ask for a frequency analysis on this portion. It's also possible to have the frequency analysis shown while an audio file is playing. In this case, the analysis is done over a window of time surrounding the playhead, and you can watch the frequency analysis graph bounce up and down as frequency components change over time. The frequency analysis of the word " boo" is shown in Figure 4.18. The file is in mono. For a stereo file, there would be two frequency graphs, one superimposed over the other.

The ***spectral view*** (also called the ***spectrum***) is another alternative for showing frequency components. In a spectral view, time is on the horizontal axis and the frequency components



Frequency (Hz):
Decibels (dB):
FFT Bin:

Prominent Frequency: -56 dB at 269 Hz (Note: C
FFT Size: 4,096 Slices: 1 Overlap: 75%
Blackman-Harris, Log, Mono Samples: 0 to 41,7

**Figure 4.18** Frequency analysis of the word “boo” (from Sound Forge)

are on the vertical axis, with the amplitude of the frequency components represented by color. Generally speaking, the brighter the color, the larger the amplitude for the frequency component. For example, blue could indicate the lowest amplitude for a frequency component. Increasingly high amplitudes could be represented by colors that move from blue to red to orange to yellow or white.

The spectral view computes an “instantaneous spectrum” of frequencies for time *t* by applying the Fourier transform to a window on the audio data that surrounds *t*. The window is then moved forward in time, and the transform is applied again. Showing the frequency components in one view as they change over time can give you an easily understood profile of your audio data. Figure 4.19 is a spectral view where the lowest amplitude is represented by blue, the medium by red, and the highest by yellow or white.

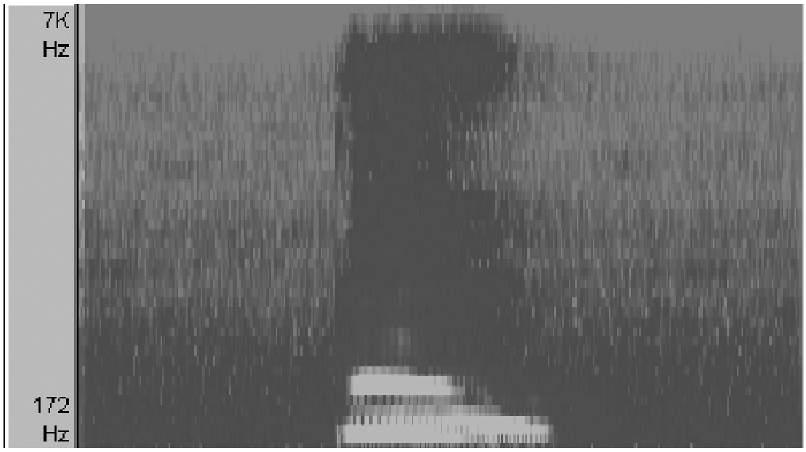


**Figure 4.19** Spectral view (from Audacity)

The term *energy* is sometimes applied to frequency and spectral analysis views. Informally defined in the context of a frequency analysis view, the energy is the area under the frequency curve. In Figure 4.18, the energy is concentrated in the low frequencies, which is the usual range of the human voice. In a spectral analysis view, the energy is concentrated in the brightest colors.

In the next section, we'll examine the mathematical operations that make frequency analysis possible.

### 4.6.2 The Fourier Series

If you continue to work in digital media, you're certain to encounter the Fourier transform. The goal of the discussion below is to make you comfortable with the mathematical concepts, relating them to the physical phenomenon of sound so that you can understand what the transforms mean and how they are applied. At the end of the mathematical discussion, we'll summarize the main points as they relate to your work in digital audio processing.

The observation that any complex sinusoidal waveform is in fact a sum of simple sinusoidals can be written in the form of a Fourier series. A ***Fourier series*** is a representation of a periodic function as an infinite sum of sinusoidals:

$$\boldsymbol{f}(t) = \sum_{n=-\infty}^{\infty} [\boldsymbol{a}_n \cos(n\omega t) + \boldsymbol{b}_n \sin(n\omega t)]$$

**Equation 4.2**

In the context of digital audio, $\boldsymbol{f}(t)$ represents a complex sound wave. Let $f$ be the fundamental frequency we described at the beginning of Section 4.6.1. (Note that $f$ and $\boldsymbol{f}(t)$ are two different things, the former being the fundamental frequency of the sound wave and the latter being the function for the complex waveform.) $\omega$ is the ***fundamental angular frequency*****,** where $\omega = 2\pi f$. As $n$ goes from $-\infty$ to $\infty$, $\omega n$ takes you through the harmonic frequencies related to $f$. For each of these, the coefficients $\boldsymbol{a}_n$ and $\boldsymbol{b}_n$ tell how much each of these component frequencies contributes to $\boldsymbol{f}(t)$.

**ASIDE:** To be more precise, a periodic function satisfying Dirichlet's conditions can be expressed as a Fourier series. Dirichlet's conditions are that the function be piecewise continuous, piecewise monotonic, and absolutely differentiable.

You should note that Equation 4.2 is true only for periodic functions. Also, $\boldsymbol{f}(t)$ is assumed to be continuous. There are different equations for Fourier analysis that vary according to whether the function is periodic or nonperiodic and discrete or continuous. We haven't yet shown you a form of the Fourier transform that is applicable to *digital* audio processing.

You sometimes see Equation 4.2 written in a different form, with the first term of the summation pulled out, as follows:

$$\boldsymbol{f}(t) = \boldsymbol{a}_0 + \sum_{n=-\infty}^{-1} [\boldsymbol{a}_n \cos(n\omega t) + \boldsymbol{b}_n \sin(n\omega t)] + \sum_{n=1}^{\infty} [\boldsymbol{a}_n \cos(n\omega t) + \boldsymbol{b}_n \sin(n\omega t)]$$

**Equation 4.3**

This separation of terms in Equation 4.3 is possible because $\sin(0) = 0$ and $\cos(0) = 1$. In this form, $\boldsymbol{a}_0$ is the ***DC component*****,** which gives the average amplitude value over one period. Given that $T = \frac{1}{f}$ is the period of the fundamental frequency, the DC component is given by

$$\boldsymbol{a}_0 = \frac{1}{T}\int_{-T/2}^{T/2} \boldsymbol{f}(t)\,dt$$

In an analogy with electrical currents (AC for alternating and DC for direct), you can picture the DC component graphically as a straight horizontal line at amplitude $\boldsymbol{a}_0$ (*i.e.*, it is the zero-frequency component). The integral in the equation is taking all the values between $-T/2$ and $T/2$, summing them, and dividing by $T$. In other words, it is the average amplitude of the complex waveform. Since the wave is assumed to be periodic, the average amplitude over one period $T$ is the same as the average over the entire wave. Each ***AC component*** is a pure sinusoidal wave, beginning with the one that has fundamental frequency $f$. The summation in Equation 4.3 creates harmonic components with frequencies that are integer multiples of $f$. The coefficients of each of these frequency components are given by the following equations:

$$\boldsymbol{a}_n = \frac{1}{T}\int_{-T/2}^{T/2} \boldsymbol{f}(t)\cos(n\omega t)\,dt$$

$$\boldsymbol{b}_n = \frac{1}{T}\int_{-T/2}^{T/2} \boldsymbol{f}(t)\sin(n\omega t)\,dt$$

$$for\ -\infty \le n \le \infty$$

Note that since $\cos(-x) = \cos(x)$, $a_{-n} = a_n$, and since $\sin(-x) = -\sin(x)$, $b_{-n} = -b_n$ and $b_0 = 0$. Rearranging Equation 4.3 and making these substitutions gives yet another form:

$$\begin{aligned}
\boldsymbol{f}(t) &= \boldsymbol{a}_0 + \sum_{n=1}^{\infty}[\boldsymbol{a}_{-n}\cos(-n\omega t) + \boldsymbol{b}_{-n}\sin(-n\omega t) + \boldsymbol{a}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] \\
&= \boldsymbol{a}_0 + \sum_{n=1}^{\infty}[\boldsymbol{a}_n\cos(-n\omega t) - \boldsymbol{b}_n\sin(-n\omega t) + \boldsymbol{a}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] \\
&= \boldsymbol{a}_0 + \sum_{n=1}^{\infty}[\boldsymbol{a}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t) + \boldsymbol{a}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] \\
&= \boldsymbol{a}_0 + \sum_{n=1}^{\infty}[2\boldsymbol{a}_n\cos(n\omega t) + 2\boldsymbol{b}_n\sin(n\omega t)] \\
&= \boldsymbol{a}_0 + 2\sum_{n=1}^{\infty}[\boldsymbol{a}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)]
\end{aligned}$$

**Equation 4.4**

On an intuitive level, it's fairly easy to understand how Equation 4.2, which is expressed in terms of sines and cosines, relates to a complex sound wave. However, this isn't the most common form for the Fourier series. There's another, equivalent way of expressing the Fourier series. It is as follows:

$$\boldsymbol{f}(t) = \sum_{n=-\infty}^{\infty} \boldsymbol{F}_n e^{in\omega t}$$

**Equation 4.5**

where $e$ is the ***base of the natural logarithm*** (~2.71828); $i$ is $\sqrt{-1}$; $\omega$ is defined as before; and $\boldsymbol{F}_n$ is a complex number. Recall that a complex number $c$ is defined as

$$c = a + bi$$

where $a$ is the real component and $b$ is the imaginary component.

Where does Equation 4.5 come from, and how can it be shown that it is equal to Equation 4.2? The equivalence is derivable from Euler's formula, which states that

$$e^{inx} = \cos(nx) + i\sin(nx)$$

Based on this identity

$$e^{in\omega t} = \cos(n\omega t) + i\sin(n\omega t)$$

**ASIDE:** One of the greatest mathematicians of all time, Swiss mathematician Leonhard Euler (1707–1783) made contributions to number theory, differential equations, complex numbers, Fermat's theorems, prime numbers, harmonics, optics, and mechanics, to name just some of the areas in which he excelled.

Equation 4.2 and Equation 4.5 really say the same thing. Let's see how they relate to each other. Recall that $\boldsymbol{F}_n$ is a complex number. The real and imaginary components of the complex number correspond to the coefficients of the cosine and sine terms, respectively, in Equation 4.2, as shown below.

$$\boldsymbol{F}_n = \boldsymbol{a}_n - i\boldsymbol{b}_n$$

**Equation 4.6**

We can do substitutions using Euler's identity and the identity given in Equation 4.6, and show that Equation 4.5 is equivalent to Equation 4.2 and Equation 4.4.

$$\begin{aligned}
\boldsymbol{f}(t) &= \sum_{n=-\infty}^{\infty} \boldsymbol{F}_n e^{in\omega t} \\
&= \sum_{n=-\infty}^{\infty} [\boldsymbol{F}_n(\cos(n\omega t) + i\sin(n\omega t))] && \textit{Step } 1 \\
&= \sum_{n=-\infty}^{\infty} [(\boldsymbol{a}_n - i\boldsymbol{b}_n)(\cos(n\omega t) + i\sin(n\omega t))] && \textit{Step } 2 \\
&= \sum_{n=-\infty}^{\infty} [\boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) - i^2\boldsymbol{b}_n\sin(n\omega t)] && \textit{Step } 3 \\
&= \sum_{n=-\infty}^{\infty} [\boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] && \textit{Step } 4 \\
&= \sum_{n=-\infty}^{-1} [\boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] + \boldsymbol{a}_0 - i\boldsymbol{b}_0 \\
&\quad + \sum_{n=1}^{\infty} [\boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] && \textit{Step } 5
\end{aligned}$$

$$= \sum_{n=-\infty}^{-1}[\boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] + \boldsymbol{a}_0$$

$$+ \sum_{n=1}^{\infty}[\boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] \qquad \textit{Step } 6$$

$$= \sum_{n=1}^{\infty}[\boldsymbol{a}_{-n}\cos(-n\omega t) + i\boldsymbol{a}_{-n}\sin(-n\omega t) - i\boldsymbol{b}_{-n}\cos(-n\omega t) + \boldsymbol{b}_{-n}\sin(-n\omega t)] + \boldsymbol{a}_0$$

$$+ \sum_{n=1}^{\infty}[\boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] \qquad \textit{Step } 7$$

$$= \boldsymbol{a}_0 + \sum_{n=1}^{\infty}\begin{bmatrix}\boldsymbol{a}_{-n}\cos(-n\omega t) + i\boldsymbol{a}_{-n}\sin(-n\omega t) - i\boldsymbol{b}_{-n}\cos(-n\omega t) + \boldsymbol{b}_{-n}\sin(-n\omega t) \\ + \boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)\end{bmatrix} \qquad \textit{Step } 8$$

$$= \boldsymbol{a}_0 + \sum_{n=1}^{\infty}\begin{bmatrix}\boldsymbol{a}_n\cos(n\omega t) - i\boldsymbol{a}_n\sin(n\omega t) + i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t) + \\ \boldsymbol{a}_n\cos(n\omega t) + i\boldsymbol{a}_n\sin(n\omega t) - i\boldsymbol{b}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)\end{bmatrix} \qquad \textit{Step } 9$$

$$= \boldsymbol{a}_0 + \sum_{n=1}^{\infty}2[\boldsymbol{a}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] \qquad \textit{Step } 10$$

$$= \boldsymbol{a}_0 + 2\sum_{n=1}^{\infty}[\boldsymbol{a}_n\cos(n\omega t) + \boldsymbol{b}_n\sin(n\omega t)] \qquad \textit{Step } 11$$

The derivation is justified as follows:

**Step 1:** Substitution based on Euler's identity.

**Step 2:** Substitution of Equation 4.6.

**Step 3:** Distribution of terms.

**Step 4:** $-i^2 = -(\sqrt{-1})^2 = 1.$

**Step 5:** Separation of summation into three parts.

**Step 6:** $b_0 = 0.$

**Step 7:** Change first summation to $\sum_{n=1}^{\infty}$ rather than $\sum_{n=-\infty}^{-1}$ and reverse the sign of $n$ on all terms in the summation.

**Step 8:** Since both summations are now over the same range, combine them.

**Step 9:** $\boldsymbol{a}_n = \boldsymbol{a}_{-n}$, $-\boldsymbol{b}_n = \boldsymbol{b}_{-n}$, $\cos(-n) = \cos(n)$, and $\sin(-n) = -\sin(n)$.

**Step 10:** Combine terms, and we now have Equation 4.4.

The point is that Equation 4.2 and Equation 4.4 and Equation 4.5 give the same information. They all tell you that a continuous complex waveform is equal to an infinite sum of simple cosine and sine functions. $\boldsymbol{a}_n$ and $\boldsymbol{b}_n$ tell "how much" of each frequency component contributes to the total waveform. $\boldsymbol{a}_n$ and $\boldsymbol{b}_n$ are explicit in Equation 4.2. They are implicit in Equation 4.5, derivable from Equation 4.6.

### 4.6.3 The Discrete Fourier Transform

Recall that $f(t)$ is assumed to be a continuous function that can be graphed as a continuous waveform. Now we need to move to the domain of digital audio, where an audio file is an array of discrete samples. The discrete equivalent of Equation 4.2 is the defined as follows:

## KEY EQUATION

Let $\boldsymbol{f}_k$ be a discrete integer function representing a digitized audio signal in the time domain. Let $\boldsymbol{F}_n$ be a discrete, complex number function representing a digital audio signal in the frequency domain. $i = \sqrt{-1}$ and $\omega = 2\pi f$. Then the ***inverse discrete Fourier transform*** is defined by

$$\begin{aligned}\boldsymbol{f}_k &= \sum_{n=0}^{N-1}\left[\boldsymbol{a}_n \cos\left(\frac{2\pi nk}{N}\right) + \boldsymbol{b}_n \sin\left(\frac{2\pi nk}{N}\right)\right] \\ &= \sum_{n=0}^{N-1}\boldsymbol{F}_n e^{\frac{i2\pi nk}{N}}\end{aligned}$$

**Equation 4.7**

(Notice that that we've changed to subscript notation, $\boldsymbol{f}_k$, to emphasize that this is an array of discrete sample values rather than a continuous function.) Since $\omega = 2\pi f$, then the ***fundamental frequency*** $f$ is $\frac{1}{N}$. As it applies to sound, Equation 4.7 states that a digital audio waveform consisting of $N$ samples is equal to a sum of $N$ cosine and sine frequency components. If you know the amplitude for each component ($\boldsymbol{a}_n$ and $\boldsymbol{b}_n$ for $0 \leq n \leq N - 1$), you can reconstruct the wave. The DC component, $\boldsymbol{a}_0$, is defined by $\boldsymbol{a}_0 = \frac{1}{N}\sum_{k=0}^{N-1}\boldsymbol{f}_k$, giving an average amplitude. The AC components $\boldsymbol{a}_n$ and $\boldsymbol{b}_n$, for $1 \leq n \leq N$, are $\boldsymbol{a}_n = \frac{1}{N}\sum_{k=0}^{N-1}\boldsymbol{f}_k \cos\left(\frac{2\pi nk}{N}\right)$ and $\boldsymbol{b}_n = \frac{1}{N}\sum_{k=0}^{N-1}\boldsymbol{f}_k \sin\left(\frac{2\pi nk}{N}\right)$.

**ASIDE:** The fundamental frequency is defined here in terms of cycles per number of samples. You might expect that it would be defined as cycles per unit time. You can assume that time units are implicit in the number of samples.

If we view the inverse discrete Fourier transform as an effective procedure, then it begins with a digital audio file in the frequency domain and transforms it to the time domain. But what if you have an array of audio samples over the time domain and want to derive the frequency components? For this you need the discrete Fourier transform.

The ***discrete Fourier transform*** (**DFT**) operates on an array of $N$ audio samples, returning cosine and sine coefficients that represent the audio data in the frequency domain.

## KEY EQUATION

Let $\boldsymbol{F}_n$ be a discrete, complex number function representing a digital audio signal in the frequency domain. Let $\boldsymbol{f}_k$ be a discrete integer function representing a digitized audio signal in the time domain. $i = \sqrt{-1}$ and $\omega = 2\pi f$. Then the ***discrete Fourier transform*** is defined by

$$\boldsymbol{F}_n = \frac{1}{N}\sum_{k=0}^{N-1}\boldsymbol{f}_k \cos\left(\frac{2\pi nk}{N}\right) - i\boldsymbol{f}_k \sin\left(\frac{2\pi nk}{N}\right) = \frac{1}{N}\sum_{k=0}^{N-1}\boldsymbol{f}_k e^{\frac{-i2\pi nk}{N}}$$

$$\text{for } 0 \leq n \leq N - 1$$

**Equation 4.8**

Supplements on Fourier transform:

interactive tutorial

programming exercise

Each $\boldsymbol{F}_n$ is a complex number with real and imaginary parts that correspond to the cosine and sine frequency components. That is, $\boldsymbol{F}_n = \boldsymbol{a}_n - i\boldsymbol{b}_n$. $\boldsymbol{f}_k$ is the $k$th sample in the array of discrete audio samples.

The two forms of Equation 4.8 are equivalent. The first may be a more intuitive way of thinking about complex waveforms—as a sum of simple sinusoidals—but the second is more concise and turns out to be more convenient for computing the Fourier transform, and thus it's the form you see most often in the literature.

Another thing that might confuse you is the seemingly-interchangeable use of either positive or negative exponents along with the imaginary number $i$. For example, Equation 4.8 might have $i2\pi nk$ rather than $-i2\pi nk$ as the power of $e$. The choice of a positive or negative exponent is arbitrary, with one caveat: If the exponent is negative in the forward transform, the sign must be positive in the inverse, and vice versa. And there's one last thing to add to the confusion. Engineers use $j$ instead of $i$ to represent $\sqrt{-1}$.

The purpose of applying the discrete Fourier transform to digital audio data is to separate the frequency components, analyze the nature of an audio clip, and edit it by possibly filtering out or altering some frequencies. The Fourier transform is widely used in both audio and (in the 2-D case) image processing, and it is implemented in a wide variety of mathematical programs and multimedia editing tools.

### 4.6.4 A Comparison of the DFT and DCT

At this point, you may be wondering what the difference is between the discrete Fourier transform and the discrete cosine transform described in Chapter 2. We'll first look at the mathematical differences and then consider how these relate to applications.

Mathematically, the one-dimensional DCT is in fact the DFT applied to $N$ audio samples to which (implicitly) another $N$ samples—a symmetric copy—are appended. By "symmetric copy" we mean that in addition to sample values $[f_0, f_1, \ldots, f_{N-1}]$, the DCT algorithm operates as if there were another $N$ sample values, $[-f_{N-1}, -f_{N-2}, \ldots, -f_0]$. Now think about the sine term in the discrete cosine transform given in Equation 4.8. The sine function is an odd function, which means that $\sin(-x) = -\sin(x)$. The cosine function is an even function, which means that $\cos(-x) = \cos(x)$. Since the DCT algorithm assumes that the data includes the negative of all the given sample values, it is in effect canceling the sine term, since for every term there is also the negative of the term. This leaves only the cosine terms—which give us the one-dimensional discrete cosine transform. (The two-dimensional DCT that we showed in Chapter 2 is an extension of this for image processing.)

When you implement the DCT, you don't really append the extra samples to the data. The algorithm just "pretends" that they are there. This is how the algorithm assumes that the data it is given represents a *periodic* discrete function—one that repeats in a pattern. Mathematically, the statements we make in Equation 4.3 and Equation 4.7—which describe the nature of our function defining the audio data, and the fact that it is decomposable into frequency components—are true only if the function is periodic. We have to assume in what sense the audio function is periodic. In fact, it is probably something quite complex with no single repeated pattern. For the Fourier transform, we assume it is periodic by assuming that the entire set of data points keeps repeating—the whole audio file constitutes one period; that is, sample values $[f_0, f_1, \ldots, f_{N-1}]$ keep repeating. For the cosine transform, we assume that the data points have appended to them a "mirror image" of themselves, and then this total pattern repeats.

What are the practical consequences of these simplifying assumptions? In the case of the DFT, one important consequence is that for an audio file of *N* samples, the DFT yields no more than $N/2$ valid frequency components. The emphasis is on the word *valid*. You saw in Chapter 2 that the DCT yields *N* frequency components for *N* samples. The reason the DFT yields only $N/2$ frequency components is based on the Nyquist theorem. Let's do the math. Our variables are defined as follows:

Supplements on DCT and DFT:

interactive tutorial

worksheet

$N =$ number of samples
$T =$ total sampling time
$s = N/T =$ sampling rate

By the Nyquist theorem, if we sample at a frequency of *s*, then we can validly sample only frequencies up to $s/2$. This implies that when we perform the DFT, there's no point in detecting frequency components above $s/2$, since they couldn't have been sampled by this sampling rate to begin with. The DFT actually yields *N* output values, but we don't want to use all *N* of them; some of the higher ones must be discarded.

Let's look more closely at how the DFT chooses the component frequencies to measure. It does so by dividing the total time of the audio data being transformed, *T*, into *N* equal parts. The frequency components correspond to $k/T$ for $0 \leq k \leq N - 1$. $1/T$ is the fundamental frequency (using units of time rather than samples). This makes sense, because *T* is assumed to be the period of the wave. $2/T$ is the second harmonic frequency, and so forth. For what value of *k*, with $0 \leq k \leq N - 1$, do we find $k/T = s/2$? This would correspond to the highest valid frequency component. We can easily solve this as follows:

$$\frac{k}{T} = \frac{s}{2}$$

$$k = \frac{Ts}{2} = \frac{T(N/T)}{2} = \frac{N}{2}$$

When we reach $N/2$, we have reached the limits of the usable frequency components from the output.

Here's an example: Say that you have an audio clip that is a pure tone at 440 Hz. The sampling rate is 1000 Hz. You are going to perform a Fourier transform on 1024 samples. Thus $T = 1024/1000 = 1.024$ sec. The frequency components that are measured by the transform are separated by $1/T = 0.9765625$ Hz. There are $N/2 = 512$ valid frequency components, the last one being $0.9765625 * 512 = 500$ Hz. This is as it should be, since the sampling rate is $2 * 500$ Hz $= 1000$ Hz, so we are not violating the Nyquist limit.

The DCT works differently. Because it takes *N* sample points and assumes it has *N* more, it implicitly has *2N* samples per period, so *N* frequency components are given as output.

You may think now that the DCT is inherently superior to the DFT because it doesn't trouble you with complex numbers, and it yields twice the number of frequency components. But it isn't as simple as that. Because of their mathematical properties, the DCT works better for some applications, the DFT for others.

Discarding the DCT's sine component may make you a little suspicious anyway. If it occurs to you that the DFT must contain more information, you're right. Thinking about the relationship between the ***trigonometric form*** of the DFT (the one containing sines and cosines) with the ***exponential form*** (the one containing *e* to a power) will help you to understand the additional information contained in the DFT but not in the DCT.

$\boldsymbol{F_n}$ in the exponential form of the Fourier transform is a complex number, having a real and an imaginary part. The cosine term in the trigonometric form relates to the real-number part of $\boldsymbol{F}_n$, and the sine term relates to the imaginary part. Specifically, $\boldsymbol{F}_n = \boldsymbol{a}_n - i\boldsymbol{b}_n$. (The DCT, in comparison, has only a real-number part.) Earlier in this chapter, we said that in the spectral view of a waveform, color represents the amplitudes of the frequency components. We can now be more precise about this. $\boldsymbol{a}_n$ and $\boldsymbol{b}_n$ can be recombined to yield the magnitude and phase of the *n*th frequency component.

KEY EQUATION

Let the equation for the inverse discrete Fourier transform be as given in Equation 4.7. Then the ***magnitude of the* n*th frequency component***, $\boldsymbol{A}_n$, is given by

$$\boldsymbol{A}_n = \sqrt{\boldsymbol{a}_n^{\,2} + \boldsymbol{b}_n^{\,2}} \quad \text{for} \quad 0 \leq n \leq N-1$$

KEY EQUATION

Let the equation for the inverse discrete Fourier transform be as given in Equation 4.7. Then the ***phase of the* n*th frequency component***, $\phi_n$, is given by

$$\phi_n = -\tan^{-1}\left(\frac{\boldsymbol{b}_n}{\boldsymbol{a}_n}\right) 0 \leq n \leq N-1$$

This gives us yet one more way to describe a complex waveform—the magnitude/phase form of the inverse DFT, as a sum of cosine waves offset by their phase.

KEY EQUATION

Let the equation for the inverse discrete Fourier transform be as given in Equation 4.7. Then the ***magnitude/phase form of the inverse DFT*** is given by

$$\boldsymbol{f}_k = \sum_{n=0}^{N-1} \boldsymbol{A}_n \cos(2\pi nk + \phi_n)$$

(Some sources use a sine function rather than a cosine in this magnitude/phase form, but that's fine since a sine function is just a phase-offset cosine.)

Now you can see that with the sine term that is part of the DFT, we have information about the phase of the signal. Is phase important? In the realm of sound, can we hear phase differences? Some sources will tell you that the human ear doesn't really detect phase differences, so it's information that in many situations can be discarded. Ohm's phase law, formulated in the 1800s, states that the phase of a waveform has no effect on how humans perceive the sound waveform. Let's be precise about what this means. That is,

**ASIDE:** German physicist George Simon Ohm (1789–1854) is best known for his research in electrical current and his precise formulation of the relationship between potential and current in electrical conduction. The unit of electrical resistance was named in his honor.

a pure-tone sound wave could be sent to you at one moment. Then later, the same tone could be sent, but with the phase shifted. You would hear no difference in these two sounds. However, if both of these tones were sent to you at the same time—with the phase of one shifted—you would be able to hear the destructive interference that results from summing the two out-of-phase tones. In environments where sounds are reverberating a great deal anyway, phase differences can be masked. But in more controlled environments, phase distortion is audible to discriminating ears. Stereo speakers that are out-of-phase can be quite annoying to audiophiles, for example.

And what about phase in images? Perhaps surprisingly, it is easier experimentally to show the importance of phase in images than in sound. Experiments show that if you take two images—call them A and B—that are decomposed into their magnitude and phase components, and you replace B's phase component with A's while leaving the magnitude component unchanged, image B will then look more like A than what it originally looked like. That is, the phase component dominates in our perception. This is one of the things that makes the two-dimensional Fourier transform very useful in image editing and restoration.

The Fourier transform also can be applied to fast implementations of convolutions. In Chapter 3, we described a convolution as a kind of two-dimensional spatial filter, altering pixels that are represented in the spatial domain. Convolutions can also be done in one dimension on audio data that are represented in the time domain. An alternative to performing the convolution in the time/space domain is to transform the data to the frequency domain by means of the Fourier transform. Then the operations can be done more efficiently by simple multiplications. In short, the Fourier transform can be the basis for a very efficient implementation of convolution.

For digital audio, the Fourier transform is primarily used to analyze an audio signal. Figure 4.2 showed how it is used to create a frequency spectrum of an audio file based on the magnitudes of the frequency components. From the frequency spectrum, you can identify which frequencies are present and which are not, and you can detect areas in the file that may need to be corrected.

On the other hand, the main application of the DCT is image compression. The DCT is one of the main steps in JPEG compression. It is a good choice in this context because the DCT concentrates energy into the coefficients corresponding to the low-frequency components. This is where we want to store the most detailed information as compression is performed. As you saw in Chapter 3, it's possible to discard more of the high-frequency information because the human eye can't detect it very well anyway.

The bottom line is that both the DCT and the DFT are useful. Fourier analysis in general, under which both the DCT and the DFT are subsumed, has wide applications that extend beyond digital imaging and digital audio into cryptography, optics, number theory, statistics, oceanography, signal processing, and other areas.

### 4.6.5 The Fast Fourier Transform

The usefulness of the discrete Fourier transform was extended greatly when a fast version was invented by Cooley and Tukey in 1965. This implementation, called the ***fast Fourier transform* (*FFT*)**, reduces the computational complexity from $O(N^2)$ to $O(N\log_2(N))$. For those of you who haven't yet studied computational complexity, this means that you can transform much larger audio or image sample files through the FFT than through the DFT and still finish the computation in a reasonable amount of time. The time it takes to do the computation of the DFT grows at the same rate that $N^2$ grows, while the time for the FFT grows at the same rate that $N \log_2(N)$ grows, where $N$ is the number of samples. This is a big savings in computational time in light of the fact that CD quality stereo, for example, has 44,100 samples in every second, giving you a very large $N$ in just a few minutes. Let's look at the time savings more closely.

The ratio of the time it takes to compute the DFT versus the time it takes to compute the FFT is given by $\frac{N^2}{N \log_2(N)} = \frac{N}{\log_2(N)}$. You can see how fast this ratio grows by trying increasingly large values of $N$.

For $N = 256$, you get $\frac{256}{\log_2(256)} = \frac{256}{8} = 32$.

For $N = 1024$, you get $\frac{1024}{\log_2(1024)} = \frac{1024}{10} \approx 102$.

For $N = 65{,}536$, you get $\frac{65{,}536}{\log_2(65{,}536)} = \frac{65{,}536}{16} = 4096$.

It takes the DFT more than 100 times longer to compute a Fourier transform on a block of 1024 samples than it takes the FFT. The larger the $N$, the more important it is that you use the FFT. Otherwise, it simply takes too long to do the computation. The FFT is more efficient than the DFT because redundant or unnecessary computations are eliminated. For example, there's no need to perform a multiplication with a term that contains sin(0) or cos(0). We know that $\sin(0) = 0$, so a term containing sin(0) can be discarded, and $\cos(0) = 1$, so a term containing cos(0) can just be added or subtracted without a multiplication. Many repetitive computations can also be identified so that they are done only one time.

The FFT has some details in its implementation that you need to know about because they are considerations when you apply the Fourier transform in digital audio processing programs. The first thing to point out is that you can't perform any kind of Fourier transform—either the DFT or the FFT—on a single sample. You need a block of samples. Imagine looking at the waveform of an audio file you're working on in a digital audio processing program, picking one point in the waveform, and wondering what the frequency components are at precisely that point. The problem is that frequency arises from the changing values in the waveform, but you don't have any changing values if you're only looking at one point.

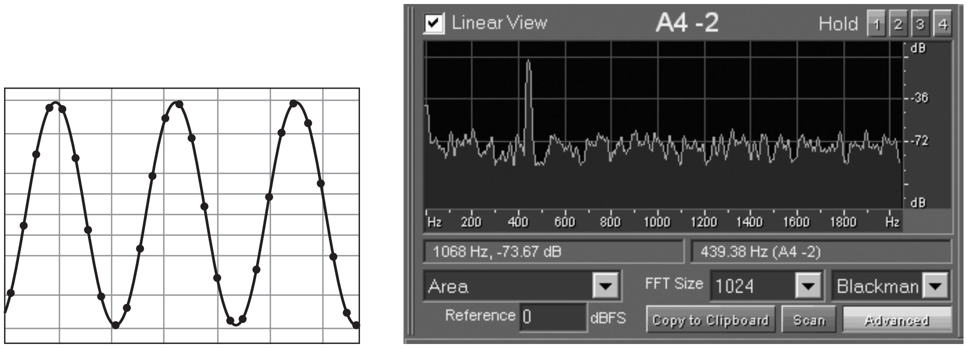
Another way of thinking about this is in terms of the frequencies (*i.e.*, pitches) that you can hear. Say that you were asked to listen to a pure tone that lasted less than a sixteenth of a second. Do you think you'd be able to identify what pitch it was? Probably not. At best it would sound like a click.

The point is that for the FFT or DFT to divide sound into frequency components, it needs a block of samples—not a just one sample—to work with. So why not just take the whole audio file—the entire waveform—and perform the transform on the whole thing? You can't do this with the FFT. Because of the way it weeds out redundant and unnecessary calculations, the FFT algorithm has to operate on blocks of samples where the number of samples is a power of 2. The DFT can work with a sample set of any block size. Because it is so fast, the FFT is the form of the Fourier transform used in digital audio programs. When you use it, you can specify the window size, which is the number of samples per block processed by the FFT. The number might vary from, say, 64 to 65,536. When you choose a point in the audio file and ask for a frequency analysis, the FFT determines the frequency components in an area of the specified window size, around the point you selected. The frequency analysis view you'll get will look something like Figure 4.18.

Some audio processing tools require that you choose an FFT window size for their required frequency analysis. For example, noise reduction generally begins with a frequency analysis of a portion of audio that ought to be silent but that contains background noise. The size of the FFT window is significant here because adjusting its size is a tradeoff between frequency and time resolution. You have seen that for an FFT window of size $N$, $N/2$ frequency components are produced. Thus, the larger the FFT size, the greater the frequency resolution. However, the larger the FFT size, the smaller the time resolution. Think about why this is so. A larger FFT window covers a longer span of time. A noise profiler divides the selected portion of audio—the portion that ought to be silent—and repeatedly does an FFT analysis of blocks of size $N$ within this window. The average of the frequency components yielded from each block becomes the frequency profile for the selected portion of audio. A larger FFT size looks at larger blocks of time, losing some of the audio detail that occurs over time (*i.e.*, sudden changes in amplitude. If the FFT size is too large, the result is time slurring, a situation where the FFT of the audio selection doesn't capture sufficient detail over time).

The frequency analysis in Figure 4.20 was done on a simple waveform representing a pure tone at 440 Hz. Since there is only one frequency in the wave, it doesn't matter what point you select for the transform. You'll always get the same frequency analysis because the frequency doesn't change over time. But you would expect that there would be just one frequency component, a single spike in the graph at 440 Hz. Why is this not the case?

The problem has to do with the assumption that the FFT makes about the periodicity of the signal. When the FFT operates on a window of $N$ samples, it assumes that this constitutes one period of a periodic signal. Even the simplest case—a pure sinusoidal wave—reveals the problem with this assumption. Assume that the FFT is operating on 1024 samples of a 440 Hz wave sampled at 8000 samples per second. The window size of 1024 samples would cover about 56.32 cycles of the wave—a non-integer multiple. This means that the end of the window would break the wave in the middle of a cycle. For the operation of the FFT, the file is assumed to consist of repeated periods of this



☑ Linear View

A4 -2

1068 Hz, -73.67 dB

439.38 Hz (A4 -2)

Area

FFT Size 1024

Blackman

Reference 0 dBFS

**Figure 4.20** 440 Hz wave on left and its frequency analysis on the right (from Audition)

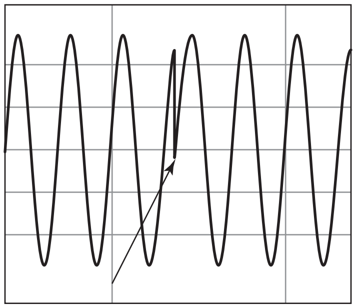


**Figure 4.21** Discontinuity caused by an FFT window that does not cover an integral number of cycles

shape. Figure 4.21 shows what it looks like when you put two of these periods end-to-end. They are out-of-phase, ending about a third of the way through a cycle at the end of the first period, and jumping to the beginning of the cycle at the beginning of the second period. Thus, the wave the FFT assumes it is analyzing doesn't correctly represent the true frequency of the audio clip. The FFT interprets the discontinuity as additional frequency components. This phenomenon is called ***spectral leakage***. As the FFT moves across the entire waveform, each time performing its operations on a window of size 1024, it repeatedly detects the spurious frequencies, which is why you see them in the frequency analysis view of Figure 4.20.

Supplements on windowing functions:

interactive tutorial

worksheet

mathematical modeling worksheet

If you could juggle your numbers so that your window covered an integral number of cycles, or if you were lucky enough that the numbers came out evenly on their own, you wouldn't get spectral leakage. For example, if your sampling rate is 1024 Hz, the frequency of the wave is 512 Hz, and you sample for one second, you'll have a block of 1024 samples that cover exactly two cycles, so there would be no spectral leakage. Of course, things don't work this way in the real world. Your sound file has a length and frequencies dictated by the nature of the sound. So the problem of spectral leakage has to be dealt with in some other way.

Spectral leakage is handled in audio processing programs by the application of a windowing function. The purpose of a windowing function is to reduce the effect of the phase discontinuities that result from the assumption that the block on which the FFT is applied is one period in a periodic wave. When you apply the FFT for frequency analysis in an audio processing program, you're asked to specify which windowing function to apply. Common choices are ***triangular***, ***Hanning***, ***Hamming***, and ***Blackman windowing functions***. In Figure 4.20, you can see that the Blackman function is being used, with an FFT window size of 1024.

The purpose of a windowing function is to reduce the amplitude of the sound wave at the beginning and end of the FFT window. The phase discontinuities occur at the ends of the window, and these phase discontinuities introduce spurious frequencies into the frequency analysis. If the amplitude of the wave is smaller at the beginning and end of the window, then the spurious frequencies will be smaller in magnitude as well.

Shaping a wave in this way is done by multiplying the wave by another periodic function. When you multiply one sinusoidal wave by another, it's as if you putting the first one in an envelope that is the shape of the second. Thus, each different windowing function is a different sinusoidal multiplier that shapes the original wave in a slightly different way. Four windowing functions are given in Table 4.4 and are graphed in Figure 4.22. Other functions exist, including Blackman-Harris, Welch, and Flat-Top.

**TABLE 4.4 Windowing Function for FFT**

| | |
|---|---|
| $u(t) = \begin{Bmatrix} \dfrac{2t}{T} & for \quad 0 \leq \mathrm{t} < \dfrac{\mathrm{T}}{2} \\ 2 - \dfrac{2t}{T} & for \quad \dfrac{T}{2} \leq t \leq T \end{Bmatrix}$<br>Triangular windowing function | $u(t) = \dfrac{1}{2}\left[1 - \cos\left(\dfrac{2\pi t}{T}\right)\right] \quad for \quad 0 \leq t \leq T$<br>Hanning windowing function |
| $u(t) = 0.54 - 0.46\cos\left(\dfrac{2\pi t}{T}\right) \quad for \quad 0 \leq t \leq T$<br>Hamming windowing function | $u(t) = 0.42 - 0.5\cos\left(\dfrac{2\pi t}{T}\right) + 0.08\cos\left(\dfrac{4\pi t}{T}\right)$<br>for $0 \leq t \leq T$<br>Blackman windowing function |

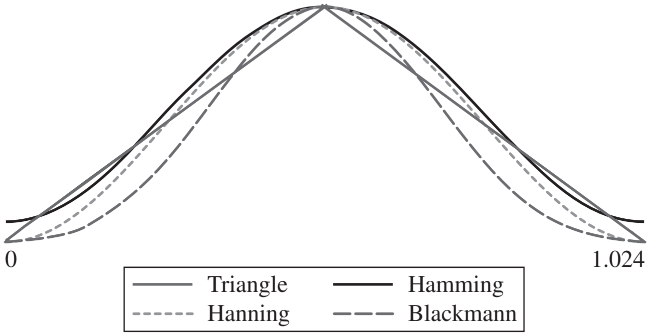


**Figure 4.22** Graphs of windowing functions for FFT

Figure 4.23 shows how application of the Hanning windowing function changes the shape of a single-frequency sound wave. The frequency of the wave is 440 Hz. The sampling frequency is 1000 Hz. We want 1024 samples. Thus, the period is $T = 1024/1000 = 1.024$ seconds. Notice that applying the windowing function does not alter the frequency of the sound wave—only the amplitude at the ends.

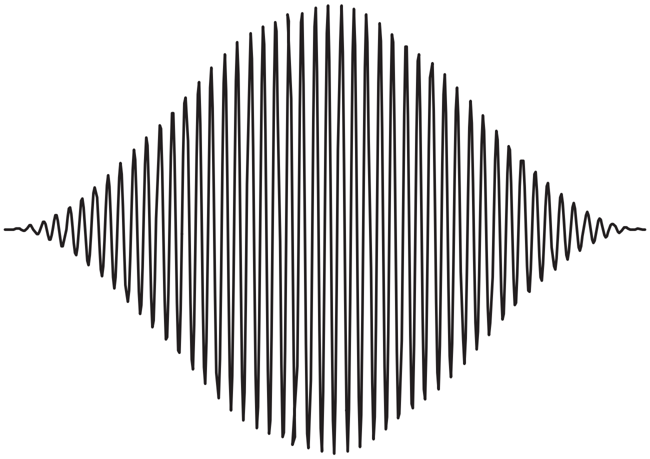


**Figure 4.23** Hanning window function applied to a simple sinusoidal wave

From the previous discussion, you should now understand that when you do frequency analysis in an audio processing program, there are two settings that affect your results: the window size and the windowing function applied. The size of the window involves a tradeoff between frequency and time resolution. Remember that you're not measuring an instantaneous frequency; instead, you're measuring the frequency over a period of time *T*. Unless you have a perfectly constant sound wave, the frequency changes over that period of time. Applying the FFT to the window gives you the average frequency components over period *T*. A bigger window averages over a bigger span of time, and thus provides a less detailed picture of the changes that occur over period *T* than would be captured in a smaller window. In short, increasing the window size increases frequency resolution in that it measures more frequency components, but it decreases time resolution in that the frequency components are derived from averages that span a longer period of time.

Loss of time resolution can be partially compensated for by the way in which FFT is implemented. Rather than apply the first FFT to samples 0 through $N - 1$ and then slide over to sample $N$ to $2N - 1$ for the next application of the transform, it's possible to slide the window over by smaller amounts; that is, overlapping windows can be analyzed.

So how do you know what window size to use? Consider first using the default value in your audio processing program. From there, you can experiment with different window sizes to see what information each one gives you. Larger window sizes take more processing time, so this is another factor in your choice. It's possible to play a sound file and watch the frequency analysis view, which changes dynamically to show you how the frequency components change over time. To be able to do this in real time, however, you have to set your window size relatively low—say, 1024 samples or less.

Windowing functions differ in the types of signals for which they are best suited; the accuracy of their amplitude measurements, even for low-level components; their frequency resolution—that is, their ability to differentiate between neighboring frequency components; and their ability to reduce spectral leakage. Hanning and Blackman give good results for most applications. If detailed frequency analysis is important to your work, you should investigate the literature on each windowing function and experiment with them to see what results they yield.

## 4.6.6 Key Points Regarding the Fourier Transform

So that you don't lose the forest in the trees, we end this section with a summary of the key points related to the Fourier transform.

- The Fourier transform is applicable in both audio and image processing. It is performed in one dimension for audio processing and two dimensions for image processing.
- The discrete Fourier transform (DFT) has two equivalent forms,

$$\boldsymbol{F}_n = \frac{1}{N}\sum_{k=0}^{N-1} \boldsymbol{f_k} \cos\left(\frac{2\pi nk}{N}\right) - i\boldsymbol{f_k} \sin\left(\frac{2\pi nk}{N}\right)$$

and

$$\boldsymbol{F}_n = \frac{1}{N}\sum_{k=0}^{N-1} \boldsymbol{f_k} e^{\frac{-i2\pi nk}{N}} \quad \text{where} \quad 0 \le n \le N - 1$$

Either form yields a complex number $\boldsymbol{F}_n = \boldsymbol{a}_n - i\boldsymbol{b}_n$, where $\boldsymbol{a}_n$ and $\boldsymbol{b}_n$ are the coefficients for the cosine and sine frequency components of the waveform being analyzed.

- The Fourier transform can be thought of in pairs: the forward and inverse transforms. The forward transform goes from the time to the frequency domain. The inverse transform goes from the frequency domain to the time domain. The transform is invertible without loss of information (down to rounding errors). The two forms of the inverse discrete Fourier transform corresponding to the forward transform shown above are

$$\boldsymbol{f_k} = \sum_{n=0}^{N-1}\left[\boldsymbol{a}_n \cos\left(\frac{2\pi nk}{N}\right) + \boldsymbol{b}_n \sin\left(\frac{2\pi nk}{N}\right)\right]$$

  and

$$\boldsymbol{f_k} = \sum_{n=0}^{N-1}\boldsymbol{F}_n e^{\frac{i2\pi nk}{N}}$$

- One difference between the discrete Fourier and the discrete cosine transform is that the DFT contains phase information while the DCT does not. You can express an audio wave in terms of the magnitude $\boldsymbol{A}_n$ of its frequency components offset by a phase $\phi_n$. This is captured in the magnitude/phase form of the inverse discrete Fourier transform:

$$\boldsymbol{f_k} = \sum_{n=0}^{N-1}\boldsymbol{A}_n \cos(2\pi nk + \phi_n)$$

- The fast Fourier transform (FFT) is a fast implementation of the discrete Fourier transform. Both versions operate on a window of $N$ samples. The FFT requires that $N$ be a power of 2, while the DCT does not. (However, there do exist variants of the FFT that don't have this requirement.) The FFT is the version of the Fourier transform that is generally used in audio processing programs, because it is fast.
- For both the DFT and the FFT, $N/2$ frequency components are given as output, with frequencies $k/T$ for $1 \leq k \leq N/2$. This is another difference between the DCT and the DFT or FFT. The DCT yields $N$ frequency components for $N$ samples, while the DFT and FFT yield $N/2$ frequency components for $N$ samples.
- Increasing the window size for the FFT increases the frequency resolution but decreases the time resolution.
- Both the DCT and the FFT yield frequency components that aren't actually part of the audio signal. This is because they have to make assumptions about the periodicity of the sample blocks they analyze. The DCT assumes that the block and its mirror image constitute one period of the entire sound wave. The FFT assume that the block alone constitutes one period. Windowing functions are used to help eliminate the spurious frequencies that are output from the transforms.
- Examples of windowing functions are Hanning, Hamming, Blackman, and Blackman–Harris.

Supplement on RMS:

mathematical modeling worksheet

## 4.7 STATISTICAL ANALYSIS OF AN AUDIO FILE

In addition to frequency and spectral views, which analyze sound data in the frequency domain, audio processing programs sometimes offer a statistical analysis of your audio files, which analyze sample values in the time domain. The statistical analysis may include the minimum and maximum possible sample values; the peak amplitude in the file; the number of clipped samples; the DC offset; the total, minimum, maximum, and average root-mean-square (RMS) amplitude; and a histogram.

Let's consider what is meant by the DC offset. You have seen that no matter how complex a sound wave is, you can decompose it into frequency components. Each frequency component is a pure sinusoidal wave that is centered on the horizontal axis. However, analog to digital conversion is not a perfect process, and it can happen that the frequency components of the sampled waveform are not perfectly centered at 0. The amount of deviation is called the ***DC offset***. You generally won't hear if your audio file has a DC offset, but it may affect certain audio processing steps, particularly those based on finding places where the waveform crosses 0, called ***zero-crossings***. Your audio processing program probably will have a feature for adjusting the DC offset.

The ***root-mean-square amplitude*** (also referred to as ***RMS power*** or ***RMS level***, depending on your audio processing program) is a measure of the average amplitude in the sound wave over a given period—either over the entire sound wave or over a portion of it that you've selected. It is computed as follows:

## KEY EQUATION

Let $N$ be the number of samples in an audio signal. $x_i$ is the amplitude of the $i$th sample. Then the ***root-mean-square amplitude***, $r$, is defined as

$$r = \sqrt{\frac{1}{N}\sum_{i=1}^{N} {x_i}^2}$$

**Equation 4.9**

**ASIDE:** You may also see the RMS equation in the form $r = \frac{\max}{\sqrt{2}}$ where *max* is the maximum sample value. This derives from $\sqrt{\frac{1}{T_2 - T_1}\int_{T_1}^{T_2}(f(x))^2 dx}$, which is the continuous version of Equation 4.9. When applied to a sine for $k$ full cycles, you get $\sqrt{\frac{1}{2k\pi}\int_0^{2k\pi}(\max * \sin(x))^2} = \frac{\max}{\sqrt{2}}$.

***Minimum*** or ***maximum RMS power*** makes sense only if you have defined a window size, which you should be able to do in the statistics view. If the window size is 50 milliseconds, for example, then the minimum RMS power is the minimum RMS amplitude for any 50-millisecond period in the selected waveform. The ***average RMS amplitude*** (sometimes referred to as ***average RMS power***) is the average RMS amplitude for all windows of the specified size.

An ***audio histogram*** shows how many samples there are at each amplitude level in the audio selection. An example of an audio histogram is given in Figure 4.24.

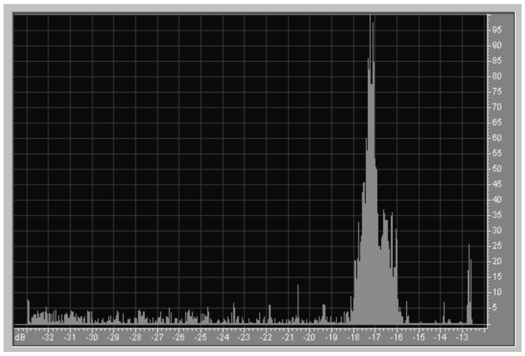


**Figure 4.24** Audio histogram (from Audition)

# 4.8 MIDI

## 4.8.1 MIDI Vs. Sampled Digital Audio

Supplement on MIDI:

hands-on worksheet

Thus far, we've been considering digital audio that is created from sampling analog sound waves and quantizing the sample values. There's another way to store sound in digital form, however. It's called ***MIDI***, which stands for ***Musical Instrument Digital Interface***. It may be more precise to say that MIDI stores "sound events" or "human performances of sound" rather than sound itself. Written in a scripting language, a MIDI file contains messages that indicate the notes, instruments, and duration of notes to be played. In MIDI terminology, each message describes an ***event*** (*i.e.*, the change of note, key, tempo, etc.) as a musical piece is played.

With sampled digital audio, an audio file contains a vector of samples—perhaps millions of them—that capture the waveform of the sound. These are reconstructed into an analog waveform when the audio is played. In comparison, MIDI messages tell what note and instrument to play, and they are translated into sound by a ***synthesizer***. If a message says the equivalent of "Play the note middle C for ½ second, and make it sound like a piano," then the computer or MIDI output device either retrieves "piano middle C" from a memory bank of stored sounds, or it creates the sound from mathematical calculations, a process called ***FM synthesis***. We'll look at these synthesis methods more closely in a moment.

The difference between sampled digital audio and MIDI is analogous to the difference between bitmapped graphics and vector graphics. Bitmaps store color values at discrete points in space, while vector graphics store symbolic descriptions of shapes—for example, an encoding of "draw a red square." Analogously, sampled digital audio files store sound wave amplitudes at discrete points in time, while MIDI files store symbolic descriptions of musical notes and how they are to be played. Just as vector graphic files are generally much smaller than bitmaps, MIDI files are generally much smaller than sampled digital audio. It's much more concise to encode the instruction "Play a piano note middle C for ½ second" than to store half a second of sound recording the piano's middle C. At 44.1 kHz in stereo, with two bytes per sample per channel, just a second requires 176,400 bytes when the audio is stored as samples. The MIDI message, in comparison, would require just a few bytes.

Because MIDI stores information in terms of notes and instruments played, it's easier to deal with this information in discrete units and edit it. For example, it's possible to change individual notes, or even to change a whole piece so that it's played with a different instrument. This would be very difficult with sampled digital audio. Think about it. With sampled audio, how would you determine exactly where one musical note ends and another begins? How would you identify what instrument is being played, just on the basis of an audio waveform? How would you change a waveform representing a tuba to a waveform representing a piccolo playing the same piece of music? No easy task.

**ASIDE:** Actually, "*wav* to MIDI converters" *do* exist. They do a fairly good job of converting an audio file into MIDI messages if the audio contains only one instrument. However, it is nearly impossible to separate the frequencies of multiple instruments played simultaneously.

A disadvantage of MIDI is that it can sound more artificial or mechanical than sampled digital audio. Imagine recording someone playing the flute. Sampled digital audio will capture the sound just as it is played, with all the subtle changes in pitch, tone, resonance, and timing that may be characteristic of the musician's style. The digitally recorded audio retains that human touch. MIDI audio, on the other hand, uses synthesized sounds to recreate each note played on the flute. Without any added audio "color," the same MIDI note played from a flute will always sound exactly the same. However, this disadvantage to MIDI is not as bad as you might imagine. Additional information can be sent in MIDI messages to describe how a note should be bent (as in pitch bend) or modulated. Most MIDI input devices can detect how hard a note is played, so the dynamics of a piece can be better reproduced. And MIDI input devices also allow you to play a piece and capture the exact timing of the performance (or near-exact, down to the maximum timing quantization level of the input and storage devices). So while it is true that it is more difficult to capture the personality of an individual's performance with synthesized sound than with sampled sound, MIDI music still has great expressive capabilities.

A main advantage to MIDI audio is the ease with which you can create and edit a piece of music. With a MIDI keyboard connected to your computer, you can play a piece, record it on your computer in MIDI format, and then edit it with a MIDI-editing program. With such a program, you can change the musical instrument used to play the piece with a click of the mouse. You can edit individual notes, change the key in which the piece is played, and fix errors note by note. And because the MIDI format is an industry standard, you can also easily transport your music from one MIDI device or computer to another.

### 4.8.2 The MIDI Standard

MIDI is a standard or protocol agreed upon by the makers of musical instruments, computers, and computer software. The protocol defines how MIDI messages are constructed, how they are transmitted, how they are stored, and what they mean. The hardware part of the protocol specifies how connections are made between two MIDI devices, including how MIDI ports convert data to electrical voltages, and how MIDI cables transmit the voltages. The software part of the protocol specifies the format and meaning of MIDI messages, and how they should be stored.

The first formal definition of the MIDI protocol was released in 1983 as the MIDI 1.0 Detailed Specification. This protocol evolved from a collaboration of industry representatives from Roland Corporation, Sequential Circuits, Oberheim Electronics, Yamaha, Korg, Kawai, and other companies associated with music. The ***MIDI Manufacturers Association*** (***MMA***) was subsequently created to oversee changes and enhancements to the MIDI standard. An important part of the MIDI Detailed Specification is the ***General MIDI*** standard, **GM-1**, adopted by the MMA in 1991. GM standardizes how musical instruments are assigned to ***patch numbers***, as they are called. Before the adoption of GM-1, you could create a musical piece on a certain keyboard or MIDI input device, defining the instruments as you wanted them. For example, your keyboard might have stored flute as patch number 74. But you had no assurance that patch number 74 would be interpreted as a flute on some other MIDI device. With GM-1, 128 standard patch numbers were adopted, as listed in Table 4.5. The patches are organized into 16 family types with eight instruments in each family. (FX stands for "special effects.") General MIDI was updated in GM-2 in 1999 and revised again in 2003. ***GM-2*** increases the number of sounds defined in the standard and also includes character information (*e.g.*, karaoke lyrics). GM-2 is backward compatible with GM-1.

### 4.8.3 How MIDI Files Are Created, Edited, and Played

The first thing you probably want to know is how to create a MIDI file, so let's begin by looking at the features of MIDI hardware and software that make this possible.

Hardware devices that generate MIDI messages are called ***MIDI controllers***. MIDI controllers take a variety of forms. A musical instrument like an electronic piano keyboard, a

**TABLE 4.5** General MIDI Mapping of Instruments to Patch Numbers

| Patch/Instrument | Patch/Instrument | Patch/Instrument | Patch/Instrument |
|---|---|---|---|
| **Piano** | **Bass** | **Reed** | **Synth. FX** |
| 1. Acoust. Grand Piano | 33. Acoustic Bass | 65. Soprano Sax | 97. FX 1 (rain) |
| 2. Bright Acoust. Piano | 34. Electric Bass (finger) | 66. Alto Sax | 98. FX 2 (soundtrack) |
| 3. Electric Grand Piano | 35. Electric Bass (pick) | 67. Tenor Sax | 99. FX 3 (crystal) |
| 4. Honky Tonk Piano | 36. Fretless Bass | 68. Baritone Sax | 100. FX 4 (atmosphere) |
| 5. Electric Piano 1 | 37. Slap Bass 1 | 69. Oboe | 101. FX 5 (brightness) |
| 6. Electric Piano 2 | 38. Slap Bass 2 | 70. English Horn | 102. FX 6 (goblins) |
| 7. Harpsichord | 39. Synth. Bass 1 | 71. Bassoon | 103. FX 7 (echoes) |
| 8. Clavichord | 40. Synth. Bass 2 | 72. Clarinet | 104. FX 8 (sci-fi) |
| **Chromatic Percussion** | **Strings** | **Pipe** | **Ethnic** |
| 9. Celesta | 41. Violin | 73. Piccolo | 105. Sitar |
| 10. Glockenspiel | 42. Viola | 74. Flute | 106. Banjo |
| 11. Music Box | 43. Cello | 75. Recorder | 107. Samisen |
| 12. Vibraphone | 44. Contrabass | 76. Pan Flute | 108. Koto |
| 13. Marimba | 45. Tremolo Strings | 77. Blown Bottle | 109. Kalimba |
| 14. Xylophone | 46. Pizzicato Strings | 78. Shakuhachi | 110. Bagpipe |
| 15. Tubular Bells | 47. Orchestral Harp | 79. Whistle | 111. Fiddle |
| 16. Dulcimer | 48. Timpani | 80. Ocarina | 112. Shanai |
| **Organ** | **Ensemble** | **Synth. Lead** | **Percussive** |
| 17. Drawbar Organ | 49. String Ensemble 1 | 81. Lead 1 (square) | 113. Tinkle Bell |
| 18. Percussive Organ | 50. String Ensemble 2 | 82. Lead 2 (sawtooth) | 114. Agogo |
| 19. Rock Organ | 51. Synth. Strings 1 | 83. Lead 3 (calliope) | 115. Steel Drums |
| 20. Church Organ | 52. Synth. Strings 2 | 84. Lead 4 (chiff) | 116. Woodblock |
| 21. Reed Organ | 53. Choir Aahs | 85. Lead 5 (charang) | 117. Tailo Drum |
| 22. Accordian | 54. Voice Oohs | 86. Lead 6 (voice) | 118. Melodic Tom |
| 23. Harmonica | 55. Synth. Voice | 87. Lead 7 (fifths) | 119. Synth. Drum |
| 24. Tango Accordian | 56. Orchestra Hit | 88. Lead 8 (bass + lead) | 120. Reverse Cymbal |
| **GUITAR** | **BRASS** | **SYNTH. PAD** | **SOUND FX** |
| 25. Acoustic Guitar (nylon) | 57. Trumpet | 89. Pad 1 (new age) | 121. Guitar Fret Noise |
| 26. Acoustic Guitar (steel) | 58. Trombone | 90. Pad 2 (warm) | 122. Breath Noise |
| 27. Electric Guitar (jazz) | 59. Tuba | 91. Pad 3 (polysynth) | 123. Seashore |
| 28. Electric Guitar (clean) | 60. Muted Trumpet | 92. Pad 4 (choir) | 124. Bird Tweet |
| 29. Electric Guitar (muted) | 61. French Horn | 93. Pad 5 (bowed) | 125. Telephone Ring |
| 30. Overdriven Guitar | 62. Brass Section | 94. Pad 6 (metallic) | 126. Helicopter |
| 31. Distortion Guitar | 63. Synth, Brass 1 | 95. Pad 7 (halo) | 127. Applause |
| 32. Guitar Harmonics | 64. Synth. Brass 2 | 96. Pad 8 (sweep) | 128. Gunshot |

saxophone, a guitar, or a trumpet can serve as a MIDI controller if it is designed for MIDI. Devices that read MIDI messages and turn them into audio signals that can be played through an output device are called ***MIDI synthesizers***. Some MIDI keyboards can serve as both controllers and synthesizers, which means that they can both generate MIDI messages and also serve as the sound output device (through which you hear the sound played). Other MIDI keyboards are silent; that is, they are used only to generate MIDI messages without creating any audible sound. Many computer sound cards are equipped to synthesize MIDI audio, and if your sound card doesn't have this capability, the operating system can provide a MIDI software synthesizer. You can also buy an external sound card (also called a sound interface) if you want to upgrade. A sound studio can have a wide range of MIDI components and setups that link samplers, effects processors, instruments, mixing consoles, and more. However, we'll restrict our discussion in this chapter to a common setup for the average user: an electronic keyboard connected to a personal computer.

A MIDI keyboard looks like a piano keyboard with extra controls. The number of keys, controls, and sensitivity features varies with the keyboard. You can find keyboards with (for example) 25, 32, 49, 61, 76, or 88 keys. (A standard piano has 88 keys.) Some keyboards can detect the velocity with which you strike a key and add this information to the MIDI message. Some can detect how hard you hold down a key after it is pressed. These features make the instrument more sensitive to musical dynamics.

The type of MIDI cable you need depends on the connection type your computer uses. Older computers sometimes use a 15-pin MIDI/joystick connection at the computer side. More commonly, a MIDI cable connects to the USB port of the computer, connecting at the MIDI keyboard with two 5-pin circular connectors, one for the *in* and one for the *out* port. A through-port is also available on some MIDI devices, (to pass data directly through to another MIDI device). A standard MIDI connection passes data serially at a rate of 31.25 kb/s. High-speed serial ports make it possible to use multiport MIDI interfaces so that a computer can address multiple MIDI devices at the same time.

A ***MIDI sequencer*** is a hardware device or software application program that allows you to receive, store, and edit MIDI data. Stand-alone hardware sequencers exist for storing and editing MIDI files. By "stand-alone," we mean that these devices work independently from a personal computer. However, we're assuming that you're more likely to be working with a computer, and in this case your MIDI sequencer will be an application program running on your computer (for example, Cakewalk Music Creator or Cubase). A sequencer captures the MIDI messages generated by your controller and stores them in General MIDI file format. Many sequencers allow you to view your MIDI file in a variety of formats—a staff view showing musical notation, a piano roll view, or an event list, for example, as shown in Figure 4.25–Figure 4.27.

Closely related to MIDI sequencers are musical notation programs. In musical notation programs, the emphasis is on creating music files in musical notation, but MIDI capability is nearly always included. With both sequencers and musical notation programs, it is also possible to create a MIDI file by inputting musical notes one note at a time. You can do this using a mouse to click on a picture of a keyboard on your computer screen, or you may be able to edit MIDI messages directly with an event editor. This isn't as convenient as playing an actual musical instrument and recording what you play as MIDI, but it works for small pieces.

A third way to capture a MIDI file is simply to read one in from another source. MIDI files with no copyright restrictions are widely available on the web or can be purchased as collections.

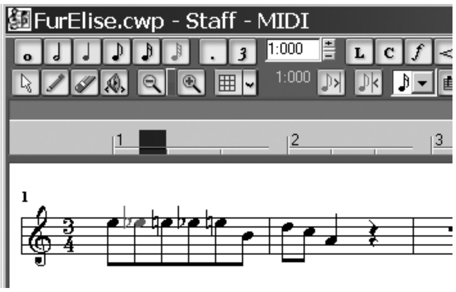


**Figure 4.25** Staff view (from Cakewalk Music Creator)

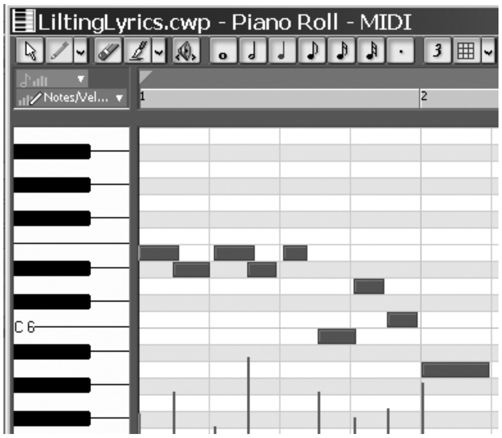
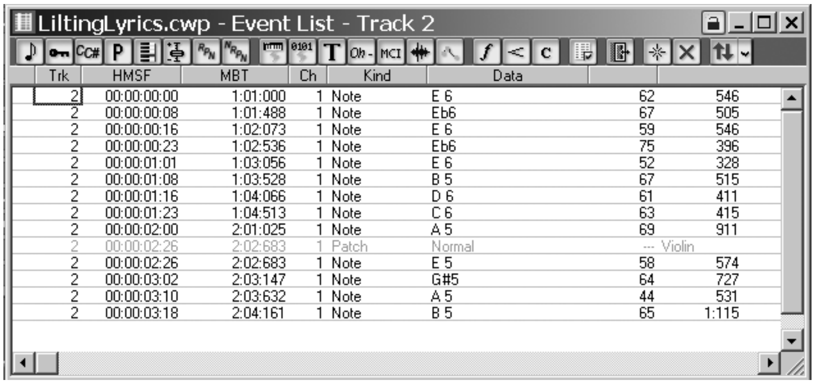


**Figure 4.26** Piano roll view (from Cakewalk Music Creator)


LiltingLyrics.cwp - Event List - Track 2

| Trk | HMSF | MBT | Ch | Kind | Data | | |
|---|---|---|---|---|---|---|---|
| 2 | 00:00:00:00 | 1:01:000 | 1 | Note | E 6 | 62 | 546 |
| 2 | 00:00:00:08 | 1:01:488 | 1 | Note | Eb6 | 67 | 505 |
| 2 | 00:00:00:16 | 1:02:073 | 1 | Note | E 6 | 59 | 546 |
| 2 | 00:00:00:23 | 1:02:536 | 1 | Note | Eb6 | 75 | 396 |
| 2 | 00:00:01:01 | 1:03:056 | 1 | Note | E 6 | 52 | 328 |
| 2 | 00:00:01:08 | 1:03:528 | 1 | Note | B 5 | 67 | 515 |
| 2 | 00:00:01:16 | 1:04:066 | 1 | Note | D 6 | 61 | 411 |
| 2 | 00:00:01:23 | 1:04:513 | 1 | Note | C 6 | 63 | 415 |
| 2 | 00:00:02:00 | 2:01:025 | 1 | Note | A 5 | 69 | 911 |
| 2 | 00:00:02:26 | 2:02:683 | 1 | Patch | Normal | --- Violin | |
| 2 | 00:00:02:26 | 2:02:683 | 1 | Note | E 5 | 58 | 574 |
| 2 | 00:00:03:02 | 2:03:147 | 1 | Note | G#5 | 64 | 727 |
| 2 | 00:00:03:10 | 2:03:632 | 1 | Note | A 5 | 44 | 531 |
| 2 | 00:00:03:18 | 2:04:161 | 1 | Note | B 5 | 65 | 1:115 |

**Figure 4.27** Event view (from Cakewalk Music Creator)

It's important to understand the difference between MIDI sequencers and standard digital audio processing programs—those which work with sampled digital audio. (*e.g.*, Audition, Logic, Audacity, or Sound Forge). Digital audio processing programs may have only limited MIDI capability. For example, they may allow you to import a MIDI file, but then the intention is that you convert the MIDI file into sampled digital audio. Doing the conversion in this direction—from MIDI to, say, WAV or PCM, isn't too hard. The MIDI file can be synthesized to digital audio, played by the sound card, recorded back through the sound card, and saved to a new file. On the other hand, taking a sampled digital audio file and converting it to MIDI would be very difficult. As described above, this would entail identifying where a note begins and ends and what instrument is playing the note, simply on the basis of the sound samples. That's nontrivial.

The beauty of MIDI is that it's so easy to create and edit MIDI files, especially for musicians. Once you've created a MIDI file, you can use your sequencing software to edit it note by note, measure by measure. This is usually done at a high level of abstraction. Musicians are accustomed to looking at musical notation in the form of sheet music, and this is the level of abstraction at which they can work in sequencing software. Easy mouse clicks or menu selections allow you to change the tempo, transpose the key, switch instruments, add or delete individual notes or measures, and correct errors. Complex and interesting musical effects can be created in this way.

### 4.8.4 MIDI for Nonmusicians

MIDI is not only for musicians. MIDI is a protocol for message passing, and applications of MIDI are not restricted to music. A system can be designed to read and interpret MIDI messages in whatever way is desired. For example, MIDI messages can be used to control complex lights, special effects, and multimedia in a theater. Since its inception, new applications have continually been added to the MIDI standard. For example, MIDI Show Control is a command and control language originally designed for theaters and later used in theme parks rides. MIDI Machine Control is used in recording studios to synchronize and remotely control recording equipment. The MIDI format has also been applied to polyphonic ring tones for mobile phones.

Once you understand how MIDI messages are constructed and communicated, you may want to experiment with catching a MIDI message and using it to control an activity. Although MIDI messages are usually intended to describe musical notes, they don't really have to be interpreted that way. If you intercept a MIDI message, you can interpret it to mean whatever you want. So if you're a computer programmer, you can find interesting and creative ways to use a MIDI device as a controller for something that might have nothing to do with music.

If you would like to experiment with MIDI's music, drum, and sound effects capability, there's still a lot you can do even without musical training. You can pick out a tune by ear on a MIDI keyboard and try out the keyboard's auto-accompaniment feature if one is available. You can look at your file in the event or piano roll view and try your hand at editing it by changing notes, changing instruments, adding more notes, changing the timing, and so forth. You can be quite creative without ever reading a note of music. As you work with MIDI, you'll learn something about music terminology and theory. In fact, one of the most common applications of MIDI is in music education. With MIDI, you can train your ear to recognize notes, keys, and intervals between notes, or you can play a piece of music and get feedback on your accuracy. If you're interested in the musical features of MIDI but don't have much background in that area, the next section will help you get started.

### 4.8.5 Musical Acoustics and Notation

If you're a musician and can read music, then you'll already be familiar with most of the terminology in this section, but you may find it helpful to place your knowledge in the context of digital media. If you're not a musician, don't be intimidated. You don't have to know how to read or compose music in order to work with MIDI, but it's good to be familiar with basic concepts and terminology.

In Western music notation, musical sounds—called ***tones***—are characterized by their pitch, timbre, and loudness. With the addition of onset and duration, a musical sound is called a ***note***. The pitch of a note is how high or low it sounds to the human ear. For musical notes that are simple sinusoidal waves, the higher the frequency of the wave, the higher the pitch. The range of human hearing is from about 20 Hz to about 20,000 Hz. Actually, if you test the frequency limits of your own hearing, you'll probably not be able to hear frequencies as high as 20,000 Hz. As you get older, you lose your ability to hear very high frequency sounds.

Different cultures have developed different terminology with regard to music—that is, different ways of arranging and labeling frequencies within the range of human hearing. Nearly all cultures, however, base their musical terminology on the following observation: If the frequency of one note is $2^n$ times of the frequency of another, where $n$ is an integer, the two notes sound "the same" to the human ear, except that the first is higher-pitched than the second. For example, a 400 Hz note sounds like a 200 Hz note, only higher. This leads to the following definition:

### KEY EQUATION

Let $g$ be the frequency of a musical note. Let $h$ be the frequency of a musical tone $n$ ***octaves*** higher than $g$. Then

$$h = 2^n g$$

The note-by-note scale at which the frequencies between two one-octave-apart notes are divided varies from culture to culture. In Western culture, 440 Hz is taken as a starting reference point and is called the note A. The octave between one A and the next higher A is then divided into twelve notes. (The word octave comes from the fact that there are eight whole notes in it: A, B, C, D, E, F, G, and again A.) The twelve notes are called A, A sharp, B, C, C sharp, D, D sharp, E, F, F sharp, G, and G sharp. A sharp can also be called B flat; C sharp can be called D flat; B sharp can be called E flat; F sharp can be called G flat; and G sharp can be called A flat. Sharps are symbolized with #, and flats are symbolized with ♭. Only the letters A through G are used to label notes. After G, the next note is A again.

The way these notes look on a piano keyboard is shown in Figure 4.28. There are black keys and white keys on the keyboard. The sharps are black keys. To show an octave, we could have started with any note, as long as we end on the same note at the end of the octave ("the same" in the sense that it is twice the frequency of the first). This is how notes and their corresponding frequencies are divided in Western music. Other cultures may have more or fewer notes per octave.

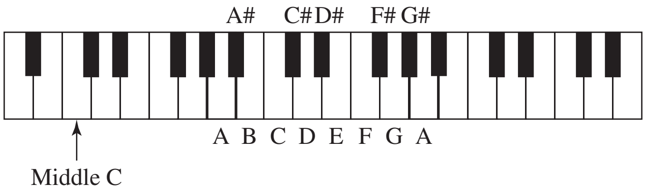


**Figure 4.28** An octave

Given that every twelve notes you double the frequency, if you know the frequency $f1$ of a certain note, you can compute the frequency $f2$ of the succeeding note. Based on the definition of an octave, start with $f1$ and multiply each succeeding frequency by some $x$, doing this 12 times to get to $2f1$. This $x$ will be the multiplier we use to get each successive frequency. Thus, we can derive $x$ from

$$\begin{aligned} 2f1 &= ((((((((((((f1x)x)x)x)x)x)x)x)x)x)x)x) = f1x^{12} \\ 2 &= x^{12} \\ x &= \sqrt[12]{2} \approx 1.05946309436 \end{aligned}$$

That is, the frequency relationship between any two successive notes $f1$ and $f2$, where $f2$ immediately follows $f1$ on the keyboard is

$$f2 = 1.05946309436f1$$

If A has frequency 440 Hz, then A# has frequency 440 Hz * 1.05946309436 $\approx$ 466.16 Hz; B has frequency 466.16 Hz * 1.05946309436 $\approx$ 493.88 Hz; and so forth.

Musical notation is written on a ***musical staff***—a set of lines and spaces, as pictured in Figure 4.25. The staff has a ***key signature*** at the beginning of the piece, telling which notes are supposed to be played as sharps and flats. (This is another meaning for the term ***key***, as opposed to a physical key on the keyboard.) It's good to know this use of the term key as you work with MIDI. With a MIDI sequencer, it is easy to transpose a piece of music to a higher or lower key. If you transpose a piece to a higher key, each note is higher, but all the notes retain their frequency relationship to each other, so the piece sounds exactly the same, only at a higher pitch. The notes will be moved up on the staff if you look at the music in staff view after transposing it.

The ***timbre*** of a musical sound is its "tone color." Think about two different instruments playing sounds of exactly the same basic pitch—say, a violin and a flute. Even though you can recognize that they're playing the same note, you can also hear a difference. The sounds, when produced by two different instruments, have a different timbre. The timbre of a musical sound is a function of its ***overtones***. A sound produced by a musical instrument is not a single-frequency wave. The shape and dynamics of the instrument produce vibrations at other, related frequencies. The lowest frequency of a given sound produced by a particular instrument is its fundamental frequency. It is the fundamental frequency that tells us the basic note that is being played. Then there are other frequencies combined in the sound. Usually, these are integer multiples of the fundamental frequency, referred to as ***harmonics***. The fundamental frequency is also called the ***first harmonic***. A frequency that is two times the fundamental frequency is called the ***second harmonic***, a frequency that is three times the fundamental frequency is the ***third harmonic***, and so forth. The harmonics above the fundamental frequency are the sound's overtones. Theoretically, all harmonics up to infinity can be produced, but in fact only some of the harmonics for a note are present at any moment in time, and they change over time, adding even more complexity to the waveform. The intensity of the different-level harmonics is determined by how the instrument is played—how a guitar string is plucked, a horn is blown, or a piano key is struck, for example. Each instrument produces its own characteristic overtones, and thus its own recognizable timbre.

There are instruments that create inharmonic overtones as well—drums, for example. These inharmonic overtones make the fundamental frequency unrecognizable, and the sound is thus more like noise than like a musical note.

***Resonance*** affects the perceived sound of an instrument, as well. When an object at rest is put in the presence of a second, vibrating object, the first object can be set into sympathetic vibration at the same frequency. This is resonance. The shape of a musical instrument—like the tubular shape of a horn or the body of a guitar—can cause the frequencies produced by the instrument to resonate, and in this way some of the harmonics are strengthened while others may be lessened. Resonance thus has an effect on an instrument's timbre.

The perceived loudness of a musical sound is a function of the air pressure amplitude. The period covered by a single musical note is called the sound's ***amplitude envelope***. The ***attack*** covers the moment when the sound is first played and reaches its maximum amplitude—for example, when a piano key is struck, a guitar is plucked, or a trumpet is blown. The relatively quick drop in amplitude after the initial attack is called the ***decay***. Following this decay, the ***sustain*** period is the span of time during which the sound continues vibrating at a fairly even level. If the sound is stopped before it fades away naturally, the moment when it is stopped is called its ***release***. Each instrument has its own typical sound envelope, and each individual sound produced by an instrument has a particular envelope. For example, a drum beat typically has a sharply peaking attack. A piano note has a fairly sharp attack as well, sharper if it is struck hard and quickly. A flute generally has a less sharply peaked attack because it must be blown to create a sound, and the blowing of a flute is a slower, smoother action than the striking of a key. A piano has a slow, steadily fading sustain, as compared to an organ, where the sound amplitude fades less during the sustain period. The general form of an amplitude envelope is shown in Figure 4.29.

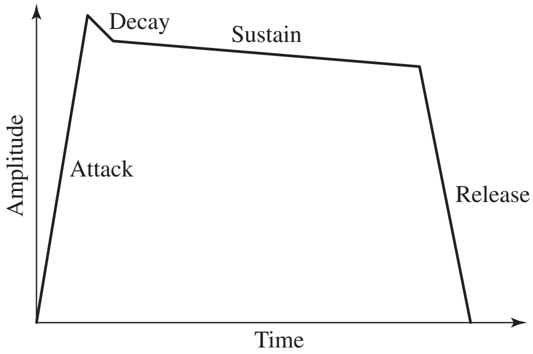


**Figure 4.29** Amplitude envelope

## 4.8.6 Features of MIDI Sequencers and Keyboards

MIDI controllers and sequencers have certain standard features and a variety of additional options, depending on the sophistication of the equipment. One of the most basic features is the ability to set patches. A ***patch*** number specifies the instrument to be played. In descriptions of MIDI messages, the patch number is referred to as the ***program number***. When an instrument is selected on a MIDI keyboard, it is sometimes referred to as a ***voice***. A ***bank*** is a database of patches, each database composed of its own samples.

A MIDI file can be played into any output device that has the ability to synthesize sound from MIDI data. If your computer has a MIDI-equipped sound card (most do), it can become the output device for playing MIDI. If you have a MIDI keyboard connected to your computer, you can play a MIDI file through the keyboard's synthesizer (if it has one) and hear the sound through the speakers of the keyboard. When you're working with a MIDI sequencer, you select which MIDI inputs and outputs you want to use.

The controls, features, and number of keys on a MIDI keyboard vary from one model to the next. A ***polyphonic*** keyboard is able to play more than one note at a time. For example, a MIDI keyboard may have 32-note maximum polyphony, which means that it can play 32 notes at the same time. A ***multitimbral*** MIDI output device can play different instruments at the same time. An output device must be polyphonic in order to be multitimbral, but the maximum number of notes and instruments it can play simultaneously doesn't have to be the same. You may be able to set a ***split-point*** for your keyboard—a division between two keys on the keyboard such that options can be set differently above and below that point. For example, you could designate that the keys above the split-point to sound like one instrument, and the keys below the split-point to sound like a different instrument. With a split-point set, you could play one part with your right hand and it could sound like a piano, while the part you play with your left hand sounds like a bass guitar.

A ***touch-sensitive*** keyboard can detect the velocity with which you strike a key, encode this in a MIDI message, and control the loudness of notes accordingly. In addition to sensing velocity, some keyboards can also sense how hard a key is held down after it is pressed, called ***aftertouch***. A key that is held down relatively hard after it is pressed can be a signal that a note should swell in volume as it is sustained—a realistic effect for horns and brass. ***Monophonic aftertouch*** assigns the same aftertouch value to all notes that are played at the same time. ***Polyphonic aftertouch*** can make one note in a chord grow louder while other notes played at the same time don't change in volume.

**ASIDE:** To be more precise, a velocity message can be interpreted however you like in a MIDI track. For example, when you play the MIDI file via a MIDI sampler, you can set the sampler control to interpret a velocity message as a change in timbre.

Two important MIDI terms that are sometimes confused are channel and track. A MIDI ***channel*** is a path of data communication between two MIDI devices. A ***track*** is an area in memory where MIDI data is stored, with a corresponding area on the sequencer's timeline where the MIDI notes can be viewed. In a MIDI sequencer's user interface, the track view shows you each track separately. You always have to record to a specific track. You can also record on more than one track simultaneously; you can designate tracks as either MIDI or audio; and you can mute tracks and listen to them separately on playback. Tracks are shown in Figure 4.30.

There is a close relationship between channels and tracks, but they are not the same. The following recording scenarios should help you to understand the difference between a track and a channel. (Your ability to re-enact these scenarios depends on the features of your equipment.)

When you play something on your MIDI keyboard, you record it to a certain track. Then you may want to record something else—something that will be played at the same time as the first clip. You can rewind to the beginning and record the second clip on the same track with the first, selecting the option to blend the two recordings rather than have the second overwrite the first. As an alternative, you can record the second clip onto a different track. What's the advantage of recording on a different track? The advantage is that the two clips are stored in separate areas in memory, so you can edit them separately. You can mute one track and play the other alone. You can change the patch on the two tracks separately, so that they play different instruments. You can delete either one without affecting the other.

So how does a channel relate to this? We said above that you can change the patch on the two tracks separately so that they play different instruments. But you can do this only if the two tracks are able to play on two different channels. The channel is the path along which the MIDI messages are passed from the place where they are stored, on your computer, to

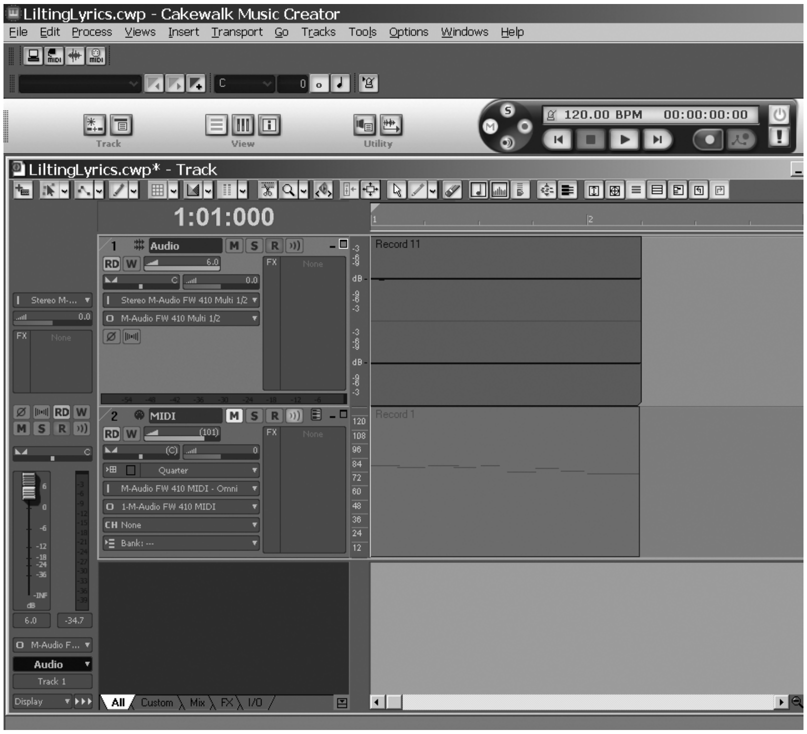

LiltingLyrics.cwp - Cakewalk Music Creator
File Edit Process Views Insert Transport Go Tracks Tools Options Windows Help
Track
View
Utility
120.00 BPM 00:00:00:00
LiltingLyrics.cwp* - Track
1:01:000
1 Audio
Stereo M-Audio FW 410 Multi 1/2
M-Audio FW 410 Multi 1/2
Record 11
2 MIDI
Quarter
M-Audio FW 410 MIDI - Omni
1-M-Audio FW 410 MIDI
CH None
Bank: ---
Record 1
Audio
Track 1
Display
All Custom Mix FX I/O

Figure 4.30 Two tracks (from Cakewalk Music Creator)

the place where they are played, on your keyboard or some other synthesizer. The GM standard calls for 16 channels on a MIDI device. In your MIDI sequencing software, you can designate that track 1 plays on channel 1 and track 2 plays on channel 2, and then if tracks 1 and 2 are set to play different instruments, you'll actually hear two different instruments played. But if both tracks are set to channel 1, then you'll hear only one of the two instruments playing both tracks. Figure 4.30 shows two tracks, the first set to channel 1 to play as a grand piano, and the second set to channel 2 to play as an alto sax. Since they are playing on separate channels, they can play different instruments.

In the MIDI standard, channel 10 is designated to carry drum and percussion sounds. If you set a track to record from channel 10, then each key you play at the keyboard will be recorded as some drum or percussion instrument as defined by the Percussion Key Map shown in Table 4.6. For example, if you record middle C, C#, and D on channel 10, you'll hear the hi bongo, low bongo, and mute hi conga in succession when you play it back. (Middle C is note 60.)

A ***sustain pedal*** can be connected to some keyboards so that you can delay the release of a note. This pedal works just like the sustain pedal on a piano. Other types of pedals (*e.g.*, sostenato pedal or soft pedal) can also be added.

A keyboard may also have a ***pitch bend wheel***. The pitch bend wheel can be slid up or down to move the pitch of a note up or down as it is being played. The wheel slides back to its original position when you let it go. If you do this when your instrument is set to

**ASIDE:** The sustain pedal causes a *sustain* MIDI message to be sent and recorded, but that message could be interpreted later in whatever way you want. For example, you could specify by means of your sampler that a *sustain* message should trigger reverb.

**TABLE 4.6 Percussion Key Map**

| Key #/Percussion Sound | Key #/Percussion Sound | Key #/Percussion Sound |
|---|---|---|
| 35. Acoustic bass drum | 51. Ride cymbal 1 | 67. High agogo |
| 36. Bass drum 1 | 52. Chinese cymbal | 68. Low agogo |
| 37. Side stick | 53. Ride bell | 69. Cabasa |
| 38. Acoustic snare | 54. Tambourine | 70. Maracas |
| 39. Hand clap | 55. Splash cymbal | 71. Short whistle |
| 40. Electric snare | 56. Cowbell | 72. Long whistle |
| 41. Low floor tom | 57. Crash cymbal 2 | 73. Short guiro |
| 42. Closed hi-hat | 58. Vibraslap | 74. Long guiro |
| 43. High floor tom | 59. Ride cymbal 2 | 75. Claves |
| 44. Pedal hi-hat | 60. Hi bongo | 76. Hi wood block |
| 45. Low tom | 61. Low bongo | 77. Low wood block |
| 46. Open hi-hat | 62. Mute hi conga | 78. Mute cuica |
| 47. Low-mid tom | 63. Open hi conga | 79. Open cuica |
| 48. Hi-mid tom | 64. Low conga | 80. Mute triangle |
| 49. Crash cymbal 1 | 65. High timbal | 81. Open triangle |
| 50. High tom | 66. Low timbal | |

some kind of guitar, it sounds like you're manipulating the guitar string while you're playing the note. The wheel has an analogous effect with other instruments.

A ***metronome*** is an audible timing device that ticks out the beats as you play. You can usually turn the metronome on from both the sequencer and the controller. By keeping pace with the metronome, you can maintain a consistent beat as you play. The sound of the metronome does not have to be recorded with the music.

A wide variety of changes and effects can be achieved with MIDI sequencing software. Some of the important features are ***transposition*** (changing the key signature), ***timing quantization*** (moving notes to more evenly spaced timing intervals), tempo change, and digital signal processing (DSP) effects like flange, reverb, delay, and echo. With many sequencers, it is also possible to have digital audio—for example, vocals or sound effects—on one track and MIDI on another. The tracks can be mixed down to a digital audio file if this is your final intent.

## 4.8.7 MIDI Behind the Scenes

### 4.8.7.1 Types and Formats of MIDI Messages

A MIDI message is a packet of data that encodes an event. MIDI events describe how music is to be played. There are two main types of MIDI messages: channel messages and system messages. ***Channel messages***, as the name implies, always contain information relevant to channels. ***Channel voice messages*** are the most common, indicating when a note begins (Note On), when a note ends (Note Off), what the note is, how hard it is pressed (Velocity), how hard it is held down (Aftertouch), what instrument is played (Program Change), what channels are activated, and so forth. ***Channel mode messages*** tell the MIDI

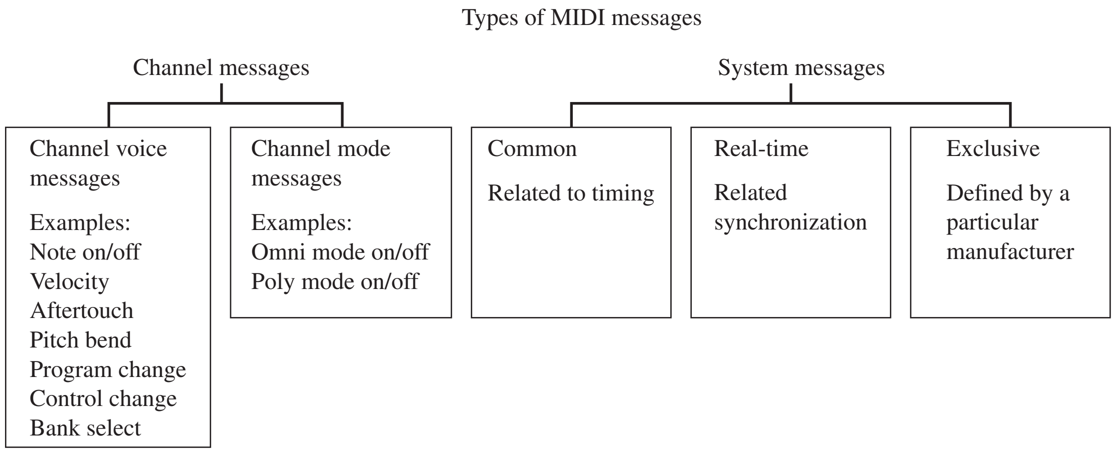


**Figure 4.31** Types of MIDI messages

receiving device—the device that plays the sound—what channels to listen to and how to interpret what it hears. For example, the Omni On message allows the receiver to listen to messages on any channel. The Mono On message indicates that the receiver is to play only one note at a time. ***System messages*** contain information that is not specific to any particular channel—for example, messages about timing, synchronization, and setup information. The general classification of MIDI messages is shown in Figure 4.31.

MIDI messages are transmitted in 10-bit bytes. Each byte begins with a start bit of 0 and ends with a stop bit of 1. The start and stop bits mark the beginnings and endings of bytes at the serial port. It's important to know that they're there when you compute the data rate for MIDI messages, but you don't need to take the start and stop bits into account if you capture and read MIDI messages through the port. The information part of the data is contained in a standard eight-bit byte, and you read only eight bits at a time for a byte. (For example, if you write a C program to capture MIDI messages, you can read the bytes into variables of type *unsigned char*, which are eight bits long.)

For each message, one ***status byte*** and zero or more ***data bytes*** are sent. The status byte tells what type of message is being communicated. Status bytes can be distinguished from data bytes by their most significant bit (MSB). The MSB of a status byte is 1, while the MSB of a data byte is 0. This implies that the value of a status byte always lies between 128 and 255, while the value of a data byte always lies between 0 and 127.

We'll look closely only at channel voice messages, since these are the most common. The complete specification of MIDI messages can be found at the MMA website. Channel voice messages tell what note is played and how it is played, requiring both a status byte and data bytes. The four least significant bits of each channel voice message tell the channel on which the note is to be transmitted. (Four bits can encode values 0 through 15, but at the user level the channels are referred to as 1 through 16, so the mapping is offset by one.) The four most significant bits of a channel voice message tell what action is to be taken—for example, begin playing a note, release the note, make the note swell; modulate, pan, turn up the volume, or create some other effect; change the instrument; or bend the pitch. Seven channel voice messages are described in Table 4.7. The x before numbers in the second column indicate that they are given in hexadecimal. The equivalent binary numbers are in the third column. From this table, consider what the three-byte message x91 x3C x64

**TABLE 4.7 Channel Voice Messages**

| Message | Status Byte in Hex (n is channel) | Status Byte in Binary | Information in Data Bytes |
|---|---|---|---|
| Note On | x9n | 1001 ---- | Note being played and velocity with which the key is struck (two bytes) |
| Note Off | x8n | 1000 ---- | Note being released and velocity with which key is released (two bytes) |
| Aftertouch for one key | xAn | 1010 ---- | Note, pressure (two bytes) |
| Aftertouch for entire channel | xDn | 1101 ---- | Pressure (one byte) |
| Program change | xCn | 1100 ---- | Patch number (one byte) |
| Control change | xBn | 1011 ---- | Type of control (*e.g.,* modulation, pan, etc.) and control change (two bytes) |
| Pitch bend | xEn | 1110 ---- | The range of frequencies through which the pitch is bent (two bytes) |

would mean. (This would be 10010001 00111100 01100100 in binary.) Hexadecimal is base 16, so x3C = 3 * 16 + 12 = 60 and x64 = 6 * 16 + 4 = 100. The first byte, x91, indicates "Note on, channel 1" The second byte indicates that the note to be played is note 60, which is middle C. The third byte indicates that the note should be played with a velocity of 100.

In addition to the messages shown in Table 4.7, it is also possible to signal the selection of a new bank of patches using a sequence of control change messages followed by appropriate data bytes.

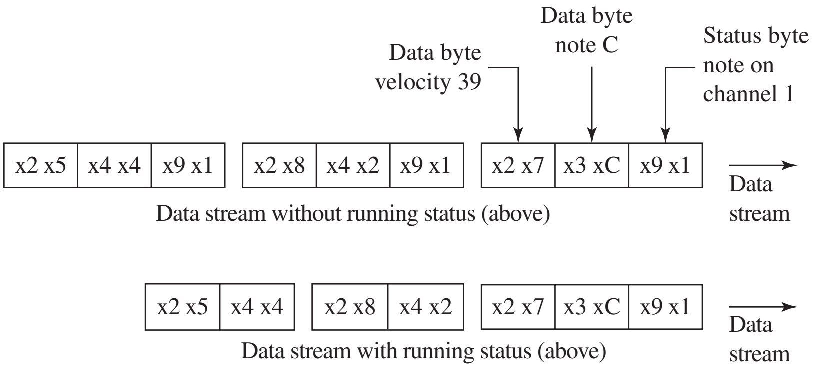
### 4.8.7.2 Transmission of MIDI Messages

Supplement on MIDI:

programming exercise

MIDI messages are transmitted serially in 10-bit bytes at a rate of 31.25 kb/s. Each message is a data packet that begins with 0 for the start bit and ends with 1 for the stop bit. In between, the data is transmitted from the least to the most significant bit. Each message has a status byte and zero or more data bytes. Usually, there are two or three data bytes, but there can be even more. Think about what this implies. A Note On message requires three 10-bit bytes. That's 30 bits. What if you have a piece of piano music that has a single note being played by the right hand and a three-note chord being played by the left hand—all simultaneously? Now you have four notes at 30 bits each, or 120 bits. At a rate of 31.25 kb/s, it takes approximately 0.004 seconds to transmit the notes—about 0.001 seconds per note (and we haven't even accounted for the Note Off messages). If the time between notes isn't too long, it won't be detected by the human ear, so the notes sound like they're being played at the same time. But we haven't considered other messages that might be pertinent to these notes, like pitch bend or control change. Rolling the pitch bend wheel on the keyboard generates tens, or sometimes even hundreds of messages. With the relatively slow serial bit rate, the number of messages can start to clog the transmissions. When this happens, there is a audible lag between notes.

***Running status*** is a technique for reducing the amount of MIDI data that needs to be sent by eliminating redundancy. The idea is simple. Once a status byte is communicated, then it doesn't have to be repeated as long as it still applies to the data bytes that follow. For example, if you have a three-note chord requiring that three Note On messages be sent in a



Notes: Start and stop bits not shown.
x indicates hexadecimal representation

**Figure 4.32** Running status

row, all to be transmitted on the same channel, then the status byte for Note On is given only once, followed by the notes and velocities for the three notes to be played. A comparison of the bit streams is given in Figure 4.32.

### 4.8.7.3 Synthesized Sound

The device that reads and plays a MIDI file—a sound card or a MIDI keyboard, for example—must be able to synthesize the sounds that are described in the messages. Two methods for synthesizing sound are ***frequency modulation synthesis*** (FM synthesis) and ***wavetable synthesis***.

FM synthesis is done by performing mathematical operations on sounds that begin as simple sinusoidal frequencies. The operations begin with a carrier frequency which is altered with a modulating frequency. Assume that the original frequency can be modeled by a sinusoidal function. This frequency can be altered in complex and interesting ways by making another sinusoidal function an argument to the first—*e.g.*, taking a sine of a sine. The outer function then can be multiplied by an amplitude envelope function that models the kind of amplitude envelope shown in Figure 4.29. Additional functions can be applied to change how the sound is modulated over time, thereby creating overtones. FM synthesis can capture the timbre of real instruments fairly well, and it can also be used to create interesting new sounds not modeled after any particular musical instrument. Sound cards can use FM synthesis as a relatively inexpensive way to produce MIDI.

Wavetable synthesis is based on stored sound samples of real instruments. The software or hardware device in which the sounds are stored is called a ***sampler***. (See Figure 4.33.) Wavetable synthesis is more expensive than FM synthesis in the amount of storage it requires, but it also reproduces the timbre of instruments more faithfully. Because of its greater fidelity, wavetable synthesis is generally preferred over FM synthesis, and it is made affordable by methods for decreasing the number and size of samples that must be stored.

Let's consider the number of samples that would have to be stored for an instrument, and how this number could be reduced. Is one sample enough to represent an instrument? Do you need a sample for each note played by an instrument? The answer actually lies somewhere in between. Although it's possible to take a single sample and shift its pitch with mathematical operations, if the pitch is shifted too far the sample begins to lose the characteristic timbre of the instrument. An instrument's timbre results in part from the

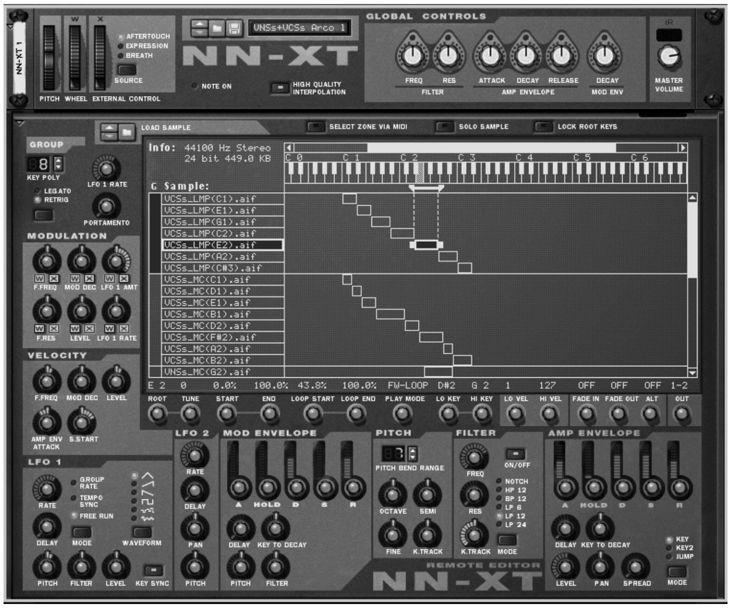


**Figure 4.33** Software digital sampler (from Reason)

overtones of notes, appearing at frequencies higher than the fundamental. Shifting the pitch can cause these overtones to be aliased, inserting false frequencies into the sample. Thus, numerous samples of each instrument are stored—but it isn't necessary to store a sample for each note. Instead, the range of notes is divided into regions, called ***key splits***. One sample is taken per key split, and then all other notes within the split are created by pitch-shifting this sample.

> **ASIDE:** Some people prefer the term ***key groups*** rather than *key splits* to refer to groups of samples because key splitting has another meaning in the context of assigning patches to keys on a keyboard. In that context, key splitting refers to the process of splitting up a keyboard so that different instruments are associated with the playing of different regions.

Other ways have been devised to reduce the storage requirements for sampled sound. The size of the samples themselves can be reduced. Specifically, only a small representative sample of a note's sustain section has to be recorded. When the sample is played, a loop (possibly with some fading) is created over this section to make it the appropriate length. Digital filtering techniques can be applied to improve the accuracy of pitch shifting such that key splits can be larger. Finally, samples can also be compressed in ways that preserve their dynamic range. With these combined techniques, wavetable synthesis has become affordable enough to be widely adopted.

## EXERCISES AND PROGRAMS

1. Convert 160 dB_SPL (damage-to-eardrum level) to air pressure amplitude in Pa. Show your work.
2. Convert 20 Pa (approximately the air pressure level of very loud music) to dB_SPL. Show your work.
3. What is the dBFS equivalent of a 16-bit sample value of 5000?

4. If the frequency of a note B is about 494 Hz, what is the frequency of the next note E up the scale from this B?
5. If the frequency of a note A is about 440 Hz, what's the frequency of an A two octaves below the 440 Hz A?
6. What does this MIDI message xC2 x39 say?
7. What is the savings in bytes for a sequence of MIDI messages that has six notes played simultaneously, using running status instead of a Note On message for each note?
8. The fast Fourier transform assumes that a wave is periodic. What does it assume to be the length of the period when the FFT uses a window of 4096 samples on a file with a sampling rate of 44.1 kHz? What's the fundamental frequency? What's the third harmonic frequency?
9. Audio aliasing interactive tutorial, worksheet, and mathematical modeling exercise, online
10. Audio dithering interactive tutorial, worksheet, and mathematical modeling exercise, online
11. Noise shaping mathematical modeling exercise, online
12. $\mu$-law encoding interactive tutorial, programming exercise, and mathematical modeling exercise, online
13. Fourier transform interactive tutorial, worksheet, and programming exercise
14. Comparison of Fourier and discrete cosine transforms interactive tutorial and worksheet
15. Windowing functions interactive tutorial, worksheet, and mathematical modeling exercise
16. Root-mean-square amplitude mathematical modeling worksheet

## APPLICATIONS

1. By generating single-frequency tones in an audio processing program, identify the highest and lowest frequencies you're able to hear.
2. Generate a simple frequency tone of *x* Hz. Generate another at 2*x* Hz. Mix them into one file. Play them. What do you expect to hear? What do you hear? Make another wave that is 3*x* the frequency of the first. What do you expect to hear when you play it together with the first wave? Play the mixed waves to confirm your prediction. Make another wave that is *not* an integer multiple of the first. What do you expect to hear when you play it together with the first wave? Play the mixed waves to confirm your prediction.

**Examine the features of your audio processing program and try the exercises below with features that are available. You should be able to find sample WAV files on the web to experiment with.**

3. Can you generate a single-frequency tone in your audio processing program at an arbitrary sampling rate? If so, create a file with a sampling rate of 1000 Hz and try to generate a tone that is 600 Hz. What happens? Why?

4. Can you get a frequency analysis and/or a spectral view of sound waves? If so, look at some sound waves and analyze the information given to you in these views. Select different portions of the file and look at the frequency analysis. Play the file and watch the frequency analysis change over time. Can you change the window size and/or windowing function for the frequency analysis? If so, try different sound files, window sizes, and windowing functions and observe the results.
5. Change an audio file to a raw format. Examine the values in the file.
6. Can you reduce the bit depth of an audio file in your audio processing program? If so, experiment with bit depth reduction. Reduce the bit depth of a file, and then listen to it. At what bit depth do you get noticeable quantization distortion? Find places where values are reduced to 0 and listen to how these sections sound.
7. Try reducing an audio file from 16 to 8 bits per sample, with no dithering. Then for comparison, take the original audio file and save it in an 8-bit $\mu$-law encoded format. Compare the two versions by listening to them. Can you hear the difference in quality? (Try a piece of music that has a wide dynamic range, and listen with good earphones.)
8. Does your audio processing program give you statistics regarding an audio file? If so, look at the RMS amplitude and histograms of an audio file. Interpret them as they relate to the audio file.
9. Working with audio, hands-on worksheet, online
10. Working with audio and MIDI in Chuck, worksheet, online
11. Working with audio and MIDI in MAX/MSP, worksheet, online
12. Working with MIDI, hands-on worksheet, online
13. Capturing and interpreting MIDI signals, programming exercise, online

***Additional exercises or applications may be found at the book or author's websites.***

## REFERENCES

### Print Publications

Adobe Creative Team. *Adobe Audition 2.0 Classroom in a Book*. Berkeley, CA: Adobe Press, 2006.

Cutler, C. C. 1960. Transmission Systems Employing Quantization. U.S. Patent No. 2,927,962.

Huntington, John. *Control Systems for Live Entertainment*, 2nd ed. Oxford: Focal Press, 2000.

Ifeachor, Emmanuel C., and Barrie W. Jervis. *Digital Signal Processing: A Practical Approach*. Addison-Wesley Publishing, 1993.

Kientzle, Tim. *A Programmer's Guide to Sound.* Reading, MA; Addison-Wesley Developers Press, 1998.

Kirk, Ross, and Andy Hunt. *Digital Sound Processing for Music and Multimedia.* Oxford: Focal Press, 1999.

Lehrman, Paul D., and Tim Tully. *MIDI for the Professional*, New York: Amsco Publications, 1993.

Loy, Gareth. *Musimathics: The Mathematical Foundations of Music. Vols. I and II.* Cambridge, MA: The MIT Press, 2006.

Messick, Paul. *Maximum MIDI: Music Applications in C++*. Greenwich: Manning Publications, 1998.

Penfold, R. A. *Electronic Music and MIDI Projects*. Kent, UK: PC Publishing, 1994.

Petelin, Roman, and Hury Petelin. *Cool Edit Pro 2 In Use*. Wayne, PA: A-List Publishing, 2003.

Petelin, Roman, and Hury Petelin. *Adobe Audition: Soundtracks for Digital Video.* Wayne, PA: A-List Publishing, 2004.

Pohlmann, Ken C. *Principles of Digital Audio*, 4th ed. New York: McGraw-Hill, 2000.

Phillips, Dave. *Linux Music and Sound.* San Francisco, CA: No Starch Press, 2000.

Roads, Curtis. *The Computer Music Tutorial*. Cambridge, MA: The MIT Press, 1996.

Roberts, Lawrence G. February 1962. "Picture Coding Using Pseudo-Random Noise." *IEEE Transactions on Information Theory* 8, 2: 145–154.

Rothstein, Joseph. *MIDI: A Comprehensive Introduction*, 2nd ed. Madison, WI: A-R Editions, 1995.

Schuchman, L. December 1964. "Dither Signals and Their Effect on Quantization Noise." *IEEE Transactions on Communications* 12, 4: 162–165.

Smith, Julius O., III. *Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications*. 2nd ed. Seattle: Book Surge Publishing, 2007.

Smith, Steven W. *Digital Signal Processing: A Practical Guide for Engineers and Scientists.* Burlington, MA: Elsevier Science, 2003.

Tranter, Jeff. *Linux Multimedia Guide*. Cambridge, MA: O'Reilly, 1996.

Winkler, Todd. *Composing Interactive Music: Techniques and Ideas Using MAX*. Cambridge, MA: The MIT Press, 1998.

## Websites

MIDI Manufacturers Association, MMA.
http://www.midi.org/

ChucK: Strongly-timed, Concurrent, and On-the-fly Audio Programming Language.
http://chuck.cs.princeton.edu/

**See references in previous chapters for additional sources on multimedia and digital signal processing applicable to this chapter.**