Basics of Digital Audio

6

Audio information is crucial for multimedia presentations and, in a sense, is the simplest type of multimedia data. However, some important differences between audio and image information cannot be ignored. For example, while it is customary and useful to occasionally drop a video frame from a video stream, to facilitate viewing speed, we simply cannot do the same with sound information or all sense will be lost from that dimension. We introduce basic concepts for sound in multimedia in this chapter and examine the arcane details of compression of sound information in Chaps. 13 and 14. The digitization of sound necessarily implies sampling and quantization of signals, so we introduce these topics here.

We begin with a discussion of just what makes up sound information, then we go on to examine the use of MIDI as an enabling technology to capture, store, and play back musical notes. We go on to look at some details of audio quantization, and give some introductory information on how digital audio is dealt with for storage or transmission. This entails a first discussion of how subtraction of signals from predicted values yield numbers that are close to zero, and hence easier to deal with.

6.1 Digitization of Sound

6.1.1 What is Sound?

Sound is a wave phenomenon like light, but it is macroscopic and involves molecules of air being compressed and expanded under the action of some physical device. For example, a speaker in an audio system vibrates back and forth and produces a longitudinal pressure wave that we perceive as sound. (As an example, we get a longitudinal wave by vibrating a Slinky along its length; in contrast, we get a transverse wave by waving the Slinky back and forth perpendicular to its length).

Without air there is no sound—for example, in space. Since sound is a pressure wave, it takes on continuous values, as opposed to digitized ones with a finite range.

Z.-N. Li et al., *Fundamentals of Multimedia*,
Texts in Computer Science, DOI: 10.1007/978-3-319-05290-8_6,

Nevertheless, if we wish to use a digital version of sound waves, we must form digitized representations of audio information.

Although such pressure waves are longitudinal, they still have ordinary wave properties and behaviors, such as reflection (bouncing), refraction (change of angle when entering a medium with a different density), and diffraction (bending around an obstacle). This makes the design of "surround sound" possible.

Since sound consists of measurable pressures at any 3D point, we can detect it by measuring the pressure level at a location, using a transducer to convert pressure to voltage levels.

In general, any signal can be decomposed into a sum of sinusoids, if we are willing to use enough sinusoids. Figure 6.1 shows how weighted sinusoids can build up quite a complex signal. Whereas frequency is an absolute measure, pitch is a perceptual, subjective quality of sound. Generally, pitch is relative. Pitch and frequency are linked by setting the note A above middle C to exactly 440 Hz. An *octave* above that note corresponds to doubling the frequency and takes us to another A note. Thus, with the middle A on a piano ("A4" or "A440") set to 440 Hz, the next A up is 880 Hz, one octave above; and so on.

Given a sound with fundamental frequency f, we define *harmonics* as any musical tones whose frequencies are integral multiples of the fundamental frequency, i.e., $2f, 3f, 4f, \ldots$, etc. For example, if the fundamental frequency (also known as *first harmonic*) is $f = 100$ Hz, the frequency of the second harmonic is 200 Hz. For the third harmonic it is 300 Hz, and so on. The harmonics can be linearly combined to form a new signal. Because all the harmonics are periodic at the fundamental frequency, their linear combinations are also periodic at the fundamental frequency. Figure 6.1 shows the appearance of the combination of these harmonics.

Now, if we allow any (integer and noninteger) multiples higher than the fundamental frequency, we allow *overtones* and have a more complex and interesting resulting sound. Together, the fundamental frequency and overtones are referred to as *partials*. The harmonics we discussed above can also be referred to as *harmonic partials*.

6.1.2 Digitization

Figure 6.2 shows the "one-dimensional" nature of sound. Values change over time in *amplitude*: the pressure increases or decreases with time [1]. Since there is only one independent variable, time, we call this a 1D signal—as opposed to images, with data that depends on two variables, x, and y, or video, which depends on 3 variables, x, y, t. The amplitude value is a continuous quantity. Since we are interested in working with such data in computer storage, we must *digitize* the *analog signals* (i.e., continuous-valued voltages) produced by microphones. Digitization means conversion to a stream of numbers—preferably *integers* for efficiency.

Since the graph in Fig. 6.2 is two-dimensional, to fully digitize the signal shown we have to *sample* in each dimension—in time and in amplitude. Sampling means measuring the quantity we are interested in, usually at evenly spaced intervals.

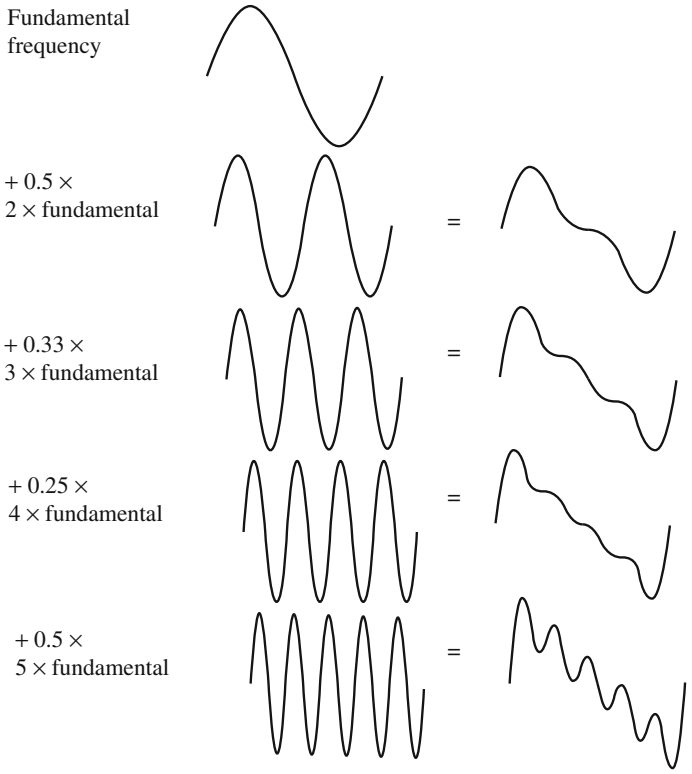

Fig. 6.1 Building up a complex signal by superposing sinusoids

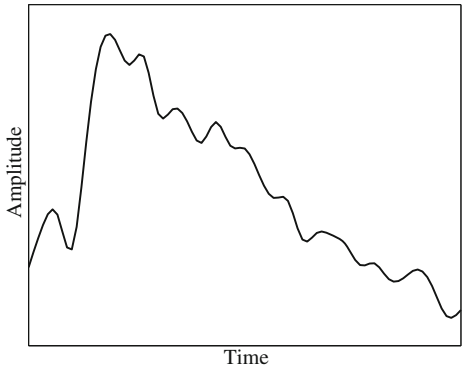

Fig. 6.2 An analog signal: continuous measurement of pressure wave

The first kind of sampling—using measurements only at evenly spaced *time* intervals—is simply called *sampling* (surprisingly), and the rate at which it is performed is called the *sampling frequency*. Figure 6.3a shows this type of digitization.

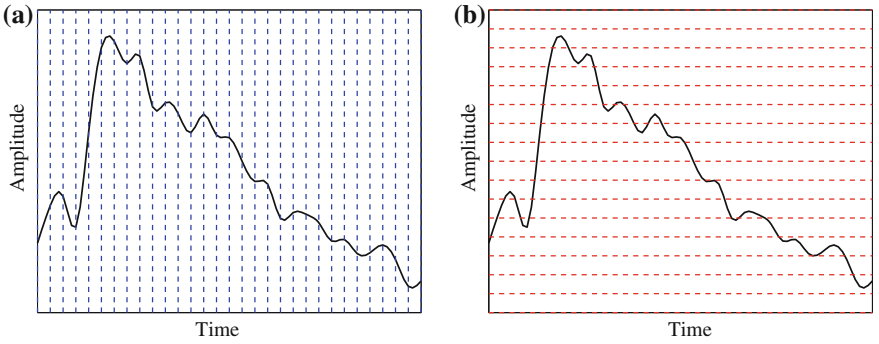

Fig. 6.3 Sampling and quantization: **a** sampling the analog signal in the time dimension; **b** quantization is sampling the analog signal in the amplitude dimension

For audio, typical sampling rates are from 8 kHz (8,000 samples per second) to 48 kHz. The human ear can hear from about 20 Hz (a very deep rumble) to as much as 20 kHz; above this level, we enter the range of ultrasound. The human voice can reach approximately 4 kHz and we need to bound our sampling rate from below by at least double this frequency (see the discussion of the Nyquist sampling rate, below). Thus, we arrive at the useful range about 8 to 40 or so kHz.

Sampling in the amplitude or voltage dimension is called *quantization*, shown in Fig. 6.3b. While we have discussed only uniform sampling, with equally spaced sampling intervals, nonuniform sampling is possible. This is not used for sampling in time but is used for quantization (see the μ-law rule, below). Typical uniform quantization rates are 8-bit and 16-bit; 8-bit quantization divides the vertical axis into 256 levels, and 16-bit divides it into 65,536 levels.

To decide how to digitize audio data, we need to answer the following questions:

1. What is the sampling rate?
2. How finely is the data to be quantized, and is the quantization uniform?
3. How is audio data formatted (i.e., what is the file format)?

6.1.3 Nyquist Theorem

As we know now, each sound is just made from sinusoids. As a simple illustration, Fig. 6.4a shows a single sinusoid: it is a single, pure, frequency (only electronic instruments can create such boring sounds).

Now if the sampling rate just equals the actual frequency, we can see from Fig. 6.4b that a false signal is detected: it is simply a constant, with zero frequency. If, on the other hand, we sample at 1.5 times the frequency, Fig. 6.4c shows that we obtain an incorrect (*alias*) frequency that is lower than the correct one—it is half the correct one (the wavelength, from peak to peak, is double that of the actual signal). In computer

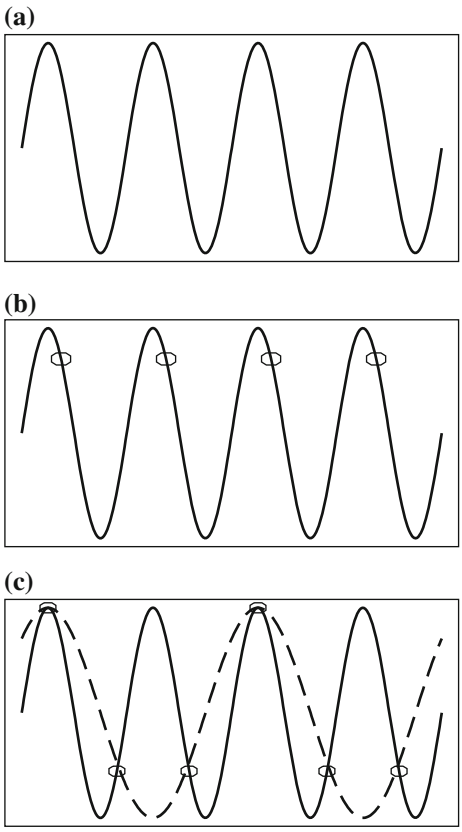

Fig. 6.4 Aliasing: **a** a single frequency; **b** sampling at exactly the frequency produces a constant; **c** sampling at 1.5 times per cycle produces an *alias* frequency that is perceived

graphics, much effort is aimed at masking such alias effects by various methods of anti-aliasing. An alias is any artifact that does not belong to the original signal. Thus, for correct sampling we must use a sampling rate equal to at least twice the maximum frequency content in the signal. This is called the *Nyquist rate*.

The Nyquist Theorem is named after Harry Nyquist, a famous mathematician who worked at Bell Labs. More generally, if a signal is *band-limited*—that is, if it has a lower limit f_1 and an upper limit f_2 of frequency components in the signal—then we need a sampling rate of at least $2(f_2 - f_1)$.

Suppose we have a *fixed* sampling rate. Since it would be impossible to recover frequencies higher than half the sampling rate in any event, most systems have an *anti-aliasing filter* that restricts the frequency content of the sampler's input to a range at or below half the sampling frequency. Confusingly, the frequency equal to half the Nyquist rate is called the *Nyquist frequency*. Then for our fixed sampling rate, the Nyquist frequency is half the sampling rate. The highest possible signal frequency component has frequency equal to that of the sampling itself.

Note that the true frequency and its alias are located symmetrically on the frequency axis with respect to the Nyquist frequency pertaining to the sampling rate

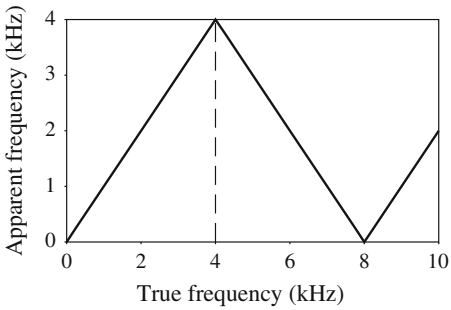

Fig. 6.5 Folding of sinusoid frequency sampled at 8,000 Hz. The folding frequency, shown dashed, is 4,000 Hz

used. For this reason, the Nyquist frequency associated with the sampling frequency is often called the "folding" frequency. That is to say, if the sampling frequency is less than twice the true frequency, and is greater than the true frequency, then the alias frequency equals the sampling frequency minus the true frequency. For example, if the true frequency is 5.5 kHz and the sampling frequency is 8 kHz, then the alias frequency is 2.5 kHz:

$$f_{\text{alias}} = f_{\text{sampling}} - f_{\text{true}}, \quad \text{for} \quad f_{\text{true}} < f_{\text{sampling}} < 2 \times f_{\text{true}}. \tag{6.1}$$

As well, a frequency at double any frequency could also fit sample points. In fact, adding any positive or negative multiple of the sampling frequency to the true frequency always gives another possible alias frequency, in that such an alias gives the same set of samples when sampled at the sampling frequency.

So, if again the sampling frequency is less than twice the true frequency and is less than the true frequency, then the alias frequency equals n times the sampling frequency minus the true frequency, where the n is the lowest integer that makes n times the sampling frequency larger than the true frequency. For example, when the true frequency is between 1.0 and 1.5 times the sampling frequency, the alias frequency equals the true frequency minus the sampling frequency.

In general, the apparent frequency of a sinusoid is the lowest frequency of a sinusoid that has exactly the same samples as the input sinusoid. Figure 6.5 shows the relationship of the apparent frequency to the input (true) frequency.

6.1.4 Signal-to-Noise Ratio (SNR)

In any analog system, random fluctuations produce *noise* added to the signal, and the measured voltage is thus incorrect. The ratio of the power of the correct signal to the noise is called the *signal-to-noise ratio* (*SNR*). Therefore, the SNR is a measure of the quality of the signal.

The SNR is usually measured in *decibels* (*dB*), where 1 dB is a tenth of a *bel*. The SNR value, in units of dB, is defined in terms of base-10 logarithms of squared

Table 6.1 Magnitudes of common sounds, in decibels

Threshold of hearing	0
Rustle of leaves	10
Very quiet room	20
Average room	40
Conversation	60
Busy street	70
Loud radio	80
Train through station	90
Riveter	100
Threshold of discomfort	120
Threshold of pain	140
Damage to eardrum	160

voltages:

$$\text{SNR} = 10 \log_{10} \frac{V_{\text{signal}}^2}{V_{\text{noise}}^2} = 20 \log_{10} \frac{V_{\text{signal}}}{V_{\text{noise}}} \tag{6.2}$$

The power in a signal is proportional to the square of the voltage. For example, if the signal voltage V_{signal} is 10 times the noise, the SNR is $20 \times \log_{10}(10) = 20$ dB.

In terms of power, if the squeaking we hear from ten violins playing is ten times the squeaking we hear from one violin playing, then the ratio of power is given in terms of decibels as 10 dB, or, in other words, 1 Bel. Notice that decibels are always defined in terms of a ratio. The term "decibels" as applied to sounds in our environment usually is in comparison to a just-audible sound with frequency 1 kHz. The levels of sound we hear around us are described in terms of decibels, as a ratio to the quietest sound we are capable of hearing. Table 6.1 shows approximate levels for these sounds.

6.1.5 Signal-to-Quantization-Noise Ratio (SQNR)

For digital signals, we must take into account the fact that only quantized values are stored. For a digital audio signal, the precision of each sample is determined by the number of bits per sample, typically 8 or 16.

Aside from any noise that may have been present in the original analog signal, additional error results from quantization. That is, if voltages are in the range of 0 to 1 but we have only 8 bits in which to store values, we effectively force all continuous values of voltage into only 256 different values.

Inevitably, this introduces a roundoff error. Although it is not really "noise," it is called *quantization noise* (or *quantization error*). The association with the concept of noise is that such errors will essentially occur randomly from sample to sample.

The quality of the quantization is characterized by the *signal-to-quantization-noise ratio (SQNR)*. Quantization noise is defined as the difference between the value of the analog signal, for the particular sampling time, and the nearest quantization interval value. At most, this error can be as much as half of the interval.

For a quantization accuracy of N bits per sample, the range of the digital signal is -2^{N-1} to $2^{N-1}-1$. Thus, if the actual analog signal is in the range from $-V_{\max}$ to $+V_{\max}$, each quantization level represents a voltage of $2V_{\max}/2^N$, or $V_{\max}/2^{N-1}$. SQNR can be simply expressed in terms of the peak signal, which is mapped to the level V_{signal} of about 2^{N-1}, and the SQNR has as denominator the maximum $V_{\text{quan\_noise}}$ of 1/2. The ratio of the two is a simple definition of the SQNR[1]:

$$\begin{aligned} SQNR &= 20\log_{10}\frac{V_{\text{signal}}}{V_{\text{quan\_noise}}} = 20\log_{10}\frac{2^{N-1}}{\frac{1}{2}} \\ &= 20\times N\times\log 2 = 6.02N(\text{dB}) \end{aligned} \tag{6.3}$$

In other words, each bit adds about 6 dB of resolution, so 16 bits provide a maximum SQNR of 96 dB.

We have examined the worst case. If, on the other hand, we assume that the input signal is sinusoidal, that quantization error is statistically independent, and that its magnitude is uniformly distributed between 0 and half the interval, we can show [2] that the expression for the SQNR becomes

$$\text{SQNR} = 6.02N + 1.76(\text{d}B) \tag{6.4}$$

Since larger is better, this shows that a more realistic approximation gives a better characterization number for the quality of a system.

We can simulate quantizing samples, e.g., drawing values from a sinusoidal probability function, and verify Eq. (6.4). Defining the SQNR in terms of the RMS (root-mean-square) value of the signal, versus the RMS value of the quantization noise, the following MATLAB fragment does indeed comply with Eq. (6.4):

```
% sqnr_sinusoid.m
%
% Simulation to verify SQNR for sinusoidal
%  probability density function.
b = 8; % 8-bit quantized signals
q = 1/10000; % make sampled signal with interval size 1/10001
seq = [0 : q : 1];
x = sin(2*pi*seq); % analog signal --> 10001 samples
% Now quantize:
x8bit = round(2^(b-1)*x) / 2^(b-1); % in [-128,128]/128=[-1,+1]
quanterror = x - x8bit;
%
SQNR = 20*log10( sqrt(mean(x.^2))/sqrt(mean(quanterror.^2)) ) %
% 50.0189dB
SQNRtheory = 6.02*b + 1.76 % 1.76=20*log10(sqrt(3/2))
% 49.9200dB
```

[1] This ratio is actually the *peak* signal-to-quantization-noise ratio, or PSQNR.

The more careful equation, Eq. (6.4), can actually be proved analytically if wish to do so: if the error obeys a uniform-random probability distribution in the range $[-0.5, 0.5]$, then its RMS (Root-Mean-Square) value is $\sqrt{\int_{-0.5}^{0.5} x^2 \mathrm{d}x} = 1/\sqrt{12}$. Now assume the signal itself is a sinusoid, $\sin(2\pi x) = \sin(\theta)$. This has to multiplied by a scalar value D, giving the range over which the sinusoid travels, i.e. the max minus the min: $D = [2^{N-1} - 1) - (-2^{N-1})] \simeq 2^N$. The sine curve is multiplied by the factor $D/2$.

Then the RMS value of the signal is $\sqrt{1/(2\pi) \int_0^{2\pi} (\frac{D}{2} \sin\theta)^2 \mathrm{d}\theta} = D/(2\sqrt{2})$. Forming the ratio of the RMS signal over the RMS quantization noise, we get $20\log_{10}(\sqrt{12}D/(2\sqrt{2})) = 20\log_{10}(D\sqrt{3/2}) = 20\log_{10}(D) + 20\log_{10}(\sqrt{3/2}) = 20\log_{10}(2^N) + 20\log_{10}(\sqrt{3/2})$, which just gives Eq. (6.4).

Typical digital audio sample precision is either 8 bits per sample, equivalent to about telephone quality, or 16 bits, for CD quality. In fact, 12 bits or so would likely do fine for adequate sound reproduction.

6.1.6 Linear and Nonlinear Quantization

We mentioned above that samples are typically stored as uniformly quantized values. This is called *linear format*. However, with a limited number of bits available, it may be more sensible to try to take into account the properties of human perception and set up nonuniform quantization levels that pay more attention to the frequency range over which humans hear best.

Remember that here we are quantizing magnitude, or amplitude—how loud the signal is. In Chap. 4, we discussed an interesting feature of many human perception subsystems (as it were)—Weber's Law —which states that the more there is, proportionately more must be added to discern a difference. Stated formally, Weber's Law says that equally perceived differences have values proportional to absolute levels:

$$\Delta\text{Response} \propto \Delta\text{Stimulus}/\text{Stimulus} \tag{6.5}$$

This means that, for example, if we can feel an increase in weight from 10 to 11 pounds, then if instead we start at 20 pounds, it would take 22 pounds for us to feel an increase in weight.

Inserting a constant of proportionality k, we have a differential equation that states

$$dr = k(1/s)\, ds \tag{6.6}$$

with response r and stimulus s. Integrating, we arrive at a solution

$$r = k \ln s + C \tag{6.7}$$

with constant of integration C. Stated differently, the solution is

$$r = k \ln(s/s_0) \tag{6.8}$$

where s_0 is the lowest level of stimulus that causes a response ($r = 0$ when $s = s_0$).

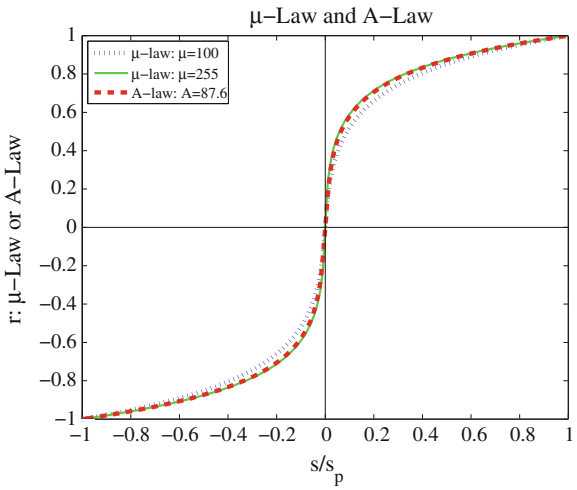

Fig. 6.6 Nonlinear transform for audio signals

Thus, nonuniform quantization schemes that take advantage of this perceptual characteristic make use of logarithms. The idea is that in a log plot derived from Eq. (6.8), if we simply take uniform steps along the s axis, we are not mirroring the nonlinear response along the r axis.

Instead, we would like to take uniform steps along the r axis. Thus, nonlinear quantization works by first transforming an analog signal from the raw s space into the theoretical r space, then uniformly quantizing the resulting values. The result is that for steps near the low end of the signal, quantization steps are effectively more concentrated on the s axis, whereas for large values of s, one quantization step in r encompasses a wide range of s values.

Such a law for audio is called *μ-law* encoding, or *u-law*, since it's easier to write. A very similar rule, called *A-law*, is used in telephony in Europe.

The equations for these similar encoding methods are as follows:

μ-law:

$$r = \frac{\text{sign}(s)}{\ln(1+\mu)} \ln\left\{1 + \mu\left|\frac{s}{s_\text{p}}\right|\right\}, \qquad \left|\frac{s}{s_\text{p}}\right| \le 1 \tag{6.9}$$

A-law:

$$r = \begin{cases} \frac{A}{1+\ln A}\left(\frac{s}{s_\text{p}}\right), & \left|\frac{s}{s_\text{p}}\right| \le \frac{1}{A} \\ \frac{\text{sign}(s)}{1+\ln A}\left[1 + \ln A\left|\frac{s}{s_\text{p}}\right|\right], & \frac{1}{A} \le \left|\frac{s}{s_\text{p}}\right| \le 1 \end{cases} \tag{6.10}$$

$$\text{where } \text{sign}(s) = \begin{cases} 1 & \text{if } s > 0, \\ -1 & \text{otherwise} \end{cases}$$

Figure 6.6 depicts these curves. The parameter of the μ-law encoder is usually set to $\mu = 100$ or $\mu = 255$, while the parameter for the A-law encoder is usually set to $A = 87.6$.

Here, s_p is the *peak signal value* and s is the current signal value. So far, this simply means that we wish to deal with s/s_p, in the range -1 to 1.

The idea of using this type of law is that if s/s_p is first transformed to values r as above and then r is quantized uniformly before transmitting or storing the signal, most of the available bits will be used to store information where changes in the signal are most apparent to a human listener, because of our perceptual nonuniformity.

To see this, consider a small change in $|s/s_p|$ near the value 1.0, where the curve in Fig. 6.6 is flattest. Clearly, the change in s has to be much larger in the flat area than near the origin to be registered by a change in the quantized r value. And it is at the quiet, low end of our hearing that we can best discern small changes in s. The μ-law transform concentrates the available information at that end.

First we carry out the μ-law transformation, then we quantize the resulting value, which is a nonlinear transform away from the input. The logarithmic steps represent low-amplitude, quiet signals with more accuracy than loud, high-amplitude ones. What this means for signals that are then encoded as a fixed number of bits is that for low-amplitude, quiet signals, the amount of noise—the error in representing the signal—is a smaller number than for high-amplitude signals. Therefore, the μ-law transform effectively makes the signal-to-noise ratio more uniform across the range of input signals.

This technique is based on human perception—a simple form of "perceptual coder." Interestingly, we have in effect also made use of the statistics of sounds we are likely to hear, which are generally in the low-volume range. In effect, we are asking for most bits to be assigned where most sounds occur—where the probability density is highest. So this type of coder is also one that is driven by statistics.

In summary, a logarithmic transform, called a "compressor" in the parlance of telephony, is applied to the analog signal before it is sampled and converted to digital (by an analog-to-digital, or AD, converter). The amount of compression increases as the amplitude of the input signal increases. The AD converter carries out a uniform quantization on the "compressed" signal. After transmission, since we need analog to hear sound, the signal is converted back, using a digital-to-analog (DA) converter, then passed through an "expander" circuit that reverses the logarithm. The overall transformation is called *companding*. Nowadays, companding can also be carried out in the digital domain.

The μ-law in audio is used to develop a nonuniform quantization rule for sound. In general, we would like to put the available bits where the most perceptual acuity (sensitivity to small changes) is. Ideally, bit allocation occurs by examining a curve of stimulus versus response for humans. Then we try to allocate bit levels to intervals for which a small change in stimulus produces a large change in response.

That is, the idea of companding reflects a less specific idea used in assigning bits to signals: put the bits where they are most needed to deliver finer resolution where the result can be perceived. This idea militates against simply using uniform quantization schemes, instead favoring nonuniform schemes for quantization. The μ-law (or A-law) for audio is an application of this idea.

Savings in bits can be gained by transmitting a smaller bit-depth for the signal, if this is indeed possible without introducing too much error. Once telephony signals

became digital, it was found that the original continuous-domain μ-law transform could be used with a substantial reduction of bits during transmission and still produce reasonable-sounding speech upon expansion at the receiver end. The μ-law often starts with a bit-depth of 16 bits, but transmits using 8 bits, and then expands back to 16 bits at the receiver.

Suppose we use the μ-law Eq. (6.9) with $\mu = 255$. Here the signal s is normalized into the range $[-1, 1]$. If the input is in -2^{15} to $(+2^{15} - 1)$, we divide by 2^{15} to normalize. Then the μ-law is applied to turn s into r; this is followed by reducing the bit-depth down to 8-bit samples, using $\hat{r} = \text{sign}(s) * \text{floor}(128 * r)$.

Now the 8-bit signal $\hat{r}$ is transmitted.

Then, at the receiver side, we normalize $\hat{r}$ by dividing by 2^7, and then apply the inverse function to Eq. (6.9), which is as follows:

$$\hat{s} = \text{sign}(s)\left(\frac{(\mu+1)^{|\hat{r}|} - 1}{\mu}\right) \tag{6.11}$$

Finally, we expand back up to 16 bits: $\tilde{s} = ceil(2^{15} * \hat{s})$. Below we show a MATLAB function for these operations.

```
function x_out = mu_law_8bitsf(x)
% signal x  is 16-bit
mu=255;
xnormd = x/2^15;
y=sign(x)*( (log(1+mu*abs(xnormd)))/log(1+mu) );

y8bit = floor(128*y);
 % TRANSMIT
y8bitnormd = y8bit/2^7;
x_hat = sign(x)*( ( (mu+1)^abs(y8bitnormd)-1)/mu);

% scale to 16 bits:
x_out = ceil(2^15*x_hat);
```

For the 2^{16} input values, shown as a solid line in Fig.6.7, the companded output values are shown as the staircase steps, in a thicker line. Indeed, we see that the companding puts the most accuracy at the quiet end nearest zero.

6.1.7 Audio Filtering

Prior to sampling and AD conversion, the audio signal is also usually *filtered* to remove unwanted frequencies. The frequencies kept depend on the application. For speech, typically from 50 Hz to 10 kHz is retained. Other frequencies are blocked

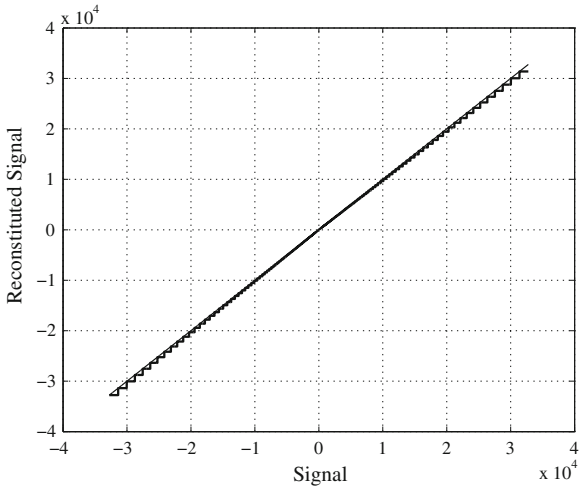

Fig. 6.7 Nonlinear quantization by companding

by a *bandpass filter*, also called a *band-limiting* filter, which screens out lower and higher frequencies.

An audio music signal will typically contain from about 20 Hz up to 20 kHz. (Twenty Hz is the low rumble produced by an upset elephant. Twenty kHz is about the highest squeak we can hear.) So the bandpass filter for music will screen out frequencies outside this range.

At the DA converter end, even though we have removed high frequencies that are likely just noise in any event, they reappear in the output. The reason is that because of sampling and then quantization, we have effectively replaced a perhaps smooth input signal by a series of step functions. In theory, such a discontinuous signal contains all possible frequencies. Therefore, at the decoder side, a *low-pass* filter is used after the DA circuit, making use of the same cutoff as at the high-frequency end of the coder's bandpass filter.

6.1.8 Audio Quality Versus Data Rate

The uncompressed data rate increases as more bits are used for quantization. Stereo information, as opposed to mono, doubles the *bitrate* (in bits per second) needed to transmit a digital audio signal. Table 6.2 shows how audio quality is related to bitrate and bandwidth.

The term *bandwidth*, derived from analog devices in Signal Processing, refers to the part of the response or transfer function of a device that is approximately constant, or flat, with the x-axis being the frequency and the y-axis equal to the transfer function. *Half-power bandwidth* (*HPBW*) refers to the bandwidth between points when the power falls to half the maximum power. Since $10 \log_{10}(0.5) \approx -3.0$, the term -3 dB bandwidth is also used to refer to the HPBW.

Table 6.2 Bitrate and bandwidth in sample audio applications

Quality	Sampling rate (kHz)	Bits per sample	Mono/ Stereo	Bitrate (if uncompressed) (kB/s)	Signal bandwidth (Hz)
Telephone	8	8	Mono	8	200–3,400
AM radio	11.025	8	Mono	11.0	100–5,500
FM radio	22.05	16	Stereo	88.2	20–11,000
CD	44.1	16	Stereo	176.4	5–20,000
DVD audio	192 (max)	24 (max)	Up to 6 channels	1,200.0 (max)	0–96,000 (max)

So for analog devices, the bandwidth was expressed in the frequency unit, called *Hertz* (Hz), which is cycles per second (for example, heartbeats per second). For digital devices, on the other hand, the amount of data that can be transmitted in a fixed bandwidth is usually expressed in bitrate, i.e., bits per second (bps) or bytes per amount of time.

In contrast, in Computer Networking, the term *bandwidth* refers to the data rate (bps) that the network or tranmission link can deliver. We will examine this issue in detail in later chapters on multimedia networks.

Telephony uses μ-law (which may be written "u-law") encoding, or A-law in Europe. The other formats use linear quantization. Using the μ-law rule shown in Eq. (6.9), the dynamic range—the ratio of highest to lowest nonzero value, expressed in dB for the value 2^n for an n-bit system, or simply stated as the number of bits—of digital telephone signals is effectively improved from 8 bits to 12 or 13.

The standard sampling frequencies used in audio are 5.0125 kHz, 11.025 kHz, 22.05 kHz, and 44.1 kHz, with some exceptions, and these frequencies are supported by most sound cards.

Sometimes it is useful to remember the kinds of data rates in Table 6.2 in terms of bytes per minute. For example, the uncompressed digital audio signal for CD-quality stereo sound is 10.6 megabytes per minute—roughly 10 megabytes—per minute.

6.1.9 Synthetic Sounds

Digitized sound must still be converted to analog, for us to hear it. There are two fundamentally different approaches to handling stored sampled audio. The first is termed *FM*, for *frequency modulation*. The second is called *Wave Table*, or just *Wave*, sound.

In the first approach, a carrier sinusoid is changed by adding another term involving a second, modulating frequency. A more interesting sound is created by changing the argument of the main cosine term, putting the second cosine inside the argument itself—then we have a cosine of a cosine. A time-varying amplitude "envelope" function multiplies the whole signal, and another time-varying function multiplies

the inner cosine, to account for overtones. Adding a couple of extra constants, the resulting function is complex indeed.

For example, Fig. 6.8a shows the function $\cos(2\pi t)$, and Fig. 6.8b is another sinusoid at twice the frequency. A cosine of a cosine is the more interesting function Fig. 6.8c, and finally, with carrier frequency 2 and modulating frequency 4, we have the much more interesting curve Fig. 6.8d. Obviously, once we consider a more complex signal, such as the following [3],

$$x(t) = A(t)\cos[\omega_c \pi t + I(t)\cos(\omega_m \pi t + \phi_m) + \phi_c] \tag{6.12}$$

we can create a most complicated signal.

This FM synthesis equation states that we make a signal using a basic carrier frequency ω_c and also use an additional, modulating frequency ω_m. In Fig. 6.8d, these values were $\omega_c = 2$ and $\omega_m = 4$. The *phase* constants ϕ_m and ϕ_c create time-shifts for a more interesting sound. The time-dependent function $A(t)$ is called the *envelope*—it specifies overall loudness over time and is used to fade in and fade out the sound. A guitar string has an *attack* period, then a *decay* period, a *sustain* period, and finally a *release* period. This is shown below in Fig. 6.10.

Finally, the time-dependent function $I(t)$ is used to produce a feeling of *harmonics* ("overtones") by changing the amount of modulation frequency heard. When $I(t)$ is small, we hear mainly low frequencies, and when $I(t)$ is larger, we hear higher frequencies as well. FM synthesis is used in low-end sound cards, but is also provided in many sound cards to provide backward compatibility.

A more accurate way of generating sounds from digital signals is called *wave table synthesis*. In this technique, digital samples are stored sounds from real instruments. Since wave tables are stored in memory on the sound card, they can be manipulated by software so that sounds can be combined, edited, and enhanced. Sound reproduction is a good deal better with wave tables than with FM synthesis. To save memory space, a variety of special techniques, such as sample looping, pitch shifting, mathematical interpolation, and polyphonic digital filtering, can be applied [4,5].

For example, it is useful to be able to change the key—suppose a song is a bit too high for your voice. A wave table can be mathematically shifted so that it produces lower pitched sounds. However, this kind of extrapolation can be used only just so far without sounding wrong. Wave tables often include sampling at various notes of the instrument, so that a key change need not be stretched too far. Wave table synthesis is more expensive than FM synthesis, partly because the data storage needed is much larger. On the other hand, storage has become much less expensive, and it is possible to compress wave table data; but nonetheless there are clearly simple tricks that one can accomplish easily using the compact formulation of FM synthesis, whereas making changes from a particular wave table is a good deal more complex. Nevertheless with the advent of cheap storage, wave data has become generally used, including in ring tones.

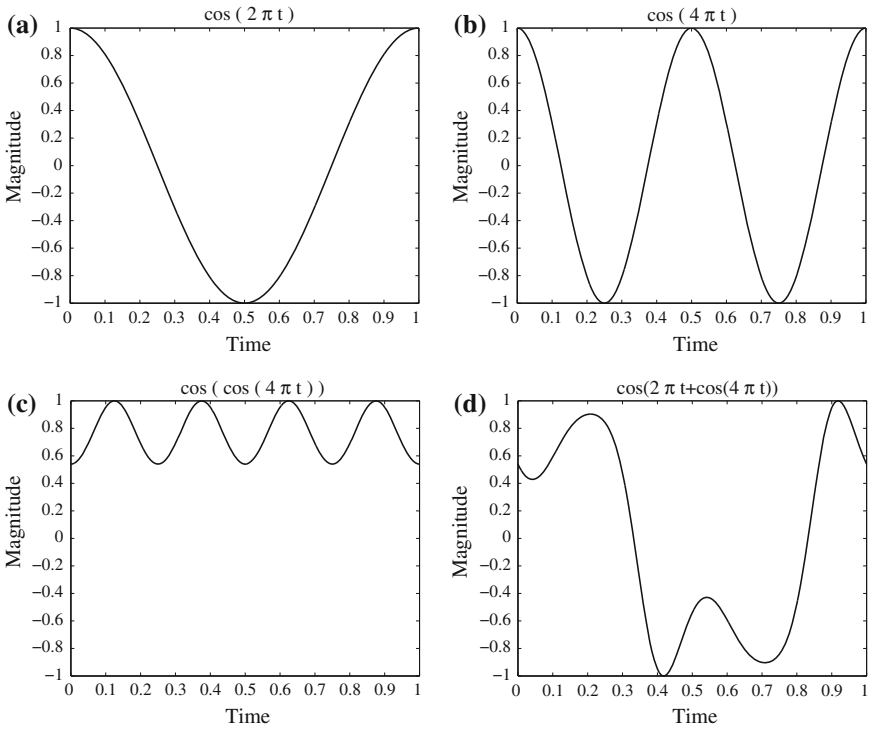

Fig. 6.8 Frequency modulation: **a** a single frequency; **b** twice the frequency; **c** usually, FM is carried out using a sinusoid argument to a sinusoid; **d** a more complex form arises from a carrier frequency $2\pi t$ and a modulating frequency $4\pi t$ cosine inside the sinusoid

6.2 MIDI: Musical Instrument Digital Interface

Wave table files provide an accurate rendering of real instrument sounds but are quite large. For simple music, we might be satisfied with FM synthesis versions of audio signals that could easily be generated by a sound card. Essentially, every computer is equipped with a sound card; a sound card is capable of manipulating and outputting sounds through speakers connected to the board, recording sound input from a microphone or line-in connection to the computer, and manipulating sound stored in memory.

If we are willing to be satisfied with the sound card's defaults for many of the sounds we wish to include in a multimedia project, we can use a simple scripting language and hardware setup called MIDI.

6.2.1 MIDI Overview

MIDI, which dates from the early 1980s, is an acronym that stands for *Musical Instrument Digital Interface*. It forms a protocol adopted by the electronic music industry that enables computers, synthesizers, keyboards, and other musical devices to communicate with each other. A synthesizer produces synthetic music and is included on sound cards, using one of the two methods discussed above. The MIDI standard is supported by most synthesizers, so sounds created on one can be played and manipulated on another and sound reasonably close. Computers must have a special MIDI interface, but this is incorporated into most sound cards. The sound card must also have both DA and AD converters.

MIDI is a scripting language—it codes "events" that stand for the production of certain sounds. Therefore, MIDI files are generally very small. For example, a MIDI event might include values for the pitch of a single note, its volume, and what instrument sound to play.

Role of MIDI. MIDI makes music notes (among other capabilities), so is useful for inventing, editing, and exchanging musical ideas that can be encapsulated as notes. This is quite a different idea than sampling, where the specifics of actual sounds are captured. Instead, MIDI is aimed at music, which can then be altered as the "user" wishes. Since MIDI is intimately related to music composition (music notation) programs, MIDI is a very useful vehicle for music education.

One strong capability of MIDI-based musical communication is the availability of a single MIDI instrument to control other MIDI instruments, allowing a master-slave relationship: the other MIDI instruments must play the same music, in part, as the master instrument, thus allowing interesting music. MIDI instruments may include excellent, or poor, musical capabilities. For example, suppose a keyboard mimics a traditional instrument well, but generates poor actual sound with a built-in synthesizer. Then "daisy-chaining" to a different synthesizer, one that generates excellent sound, may make an overall good combination. Since MIDI comes with a built-in timecode, the master's clock can be used to synchronize all the slave timecodes, making for more exact synchronization.

A so-called "sequencer-sampler" can be used to reorder and manipulate sets of digital-audio samples *and/or* sequences of MIDI. In a Digital Audio Workstation, running ProTools for example, multitrack recording is possible, either sequentially or concurrently. For example, one could simultaneously record 8 vocal tracks and 8 instrument tracks.

MIDI Concepts

- Music is organized into *tracks* in a sequencer. Each track can be turned on or off on recording or playing back. Usually, a particular instrument is associated with a MIDI *channel*. MIDI channels are used to separate messages. There are 16 channels, numbered from 0 to 15. The channel forms the last four bits (the

least significant bits) of that do refer to the channel. The idea is that each channel is associated with a particular instrument—for example, channel 1 is the piano, channel 10 is the drums. Nevertheless, you can switch instruments midstream, if desired, and associate another instrument with any channel.

- Along with *channel messages* (which include a channel number), several other types of messages are sent, such as a general message for all instruments indicating a change in tuning or timing; these are called *system messages*. It is also possible to send a special message to an instrument's channel that allows sending many notes without a channel specified. We will describe these messages in detail later.
- The way a synthetic musical instrument responds to a MIDI message is usually by simply ignoring any "play sound" message that is not for its channel. If several messages are for its channel, say several simultaneous notes being played on a piano, then the instrument responds, provided it is *multi-voice*—that is, can play more than a single note at once.

MIDI Terminology

- A *synthesizer* was, and still can be, a stand-alone sound generator that can vary pitch, loudness, and tone color. (The pitch is the musical note the instrument plays—a C, as opposed to a G, say. Whereas frequency in Hz is an absolute musical sound, pitch is relative: e.g., tuning your guitar to itself may sound fine but not have the same absolute notes as another guitar.) It can also change additional music characteristics, such as attack and delay time. A good (musician's) synthesizer often has a microprocessor, keyboard, control panels, memory, and so on. However, inexpensive synthesizers are also included on PC sound cards. Units that generate sound are referred to as *tone modules* or sound modules.
- A *sequencer* started off as a special hardware device for storing and editing a *sequence* of musical events, in the form of MIDI data. Now it is more often a software *music editor* on the computer.
- A *MIDI keyboard* produces no sound, instead generating sequences of MIDI instructions, called *MIDI messages* (but can also include a synthesizer for generating sound). MIDI messages are rather like assembler code and usually consist of just a few bytes. Stored as a sequence of MIDI messages, you might have 3 minutes of music, say, stored in only 3 kB. In comparison, a wave table file (WAV) stores 1 minute of music in about 10 MB. In MIDI parlance, the keyboard is referred to as a *keyboard controller*.
- It is easy to confuse the term *voice* with the term *timbre*. The latter is MIDI terminology for just what instrument we are trying to emulate—for example, a piano as opposed to a violin. It is the quality of the sound. An instrument (or sound card) that is *multi-timbral* is capable of playing many different sounds at the same time, (e.g., piano, brass, drums).
- On the other hand, the term "voice", while sometimes used by musicians to mean the same thing as timbre, is used in MIDI to mean every different timbre and pitch that the tone module can produce at the same time. Synthesizers can have

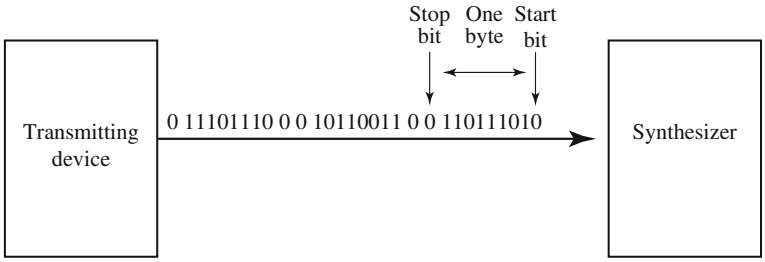

Fig. 6.9 Stream of 10-bit bytes; for typical MIDI messages, these consist of {status byte, data byte, data byte} = {Note On, Note Number, Note Velocity}

many (typically 16, 32, 64, 256, etc.) voices. Each voice works independently and simultaneously to produce sounds of different timbre and pitch.

- The term *polyphony* refers to the number of voices that can be produced at the same time. So a typical tone module may be able to produce "64 voices of polyphony" (64 different notes at once) and be "16-part multi-timbral" (can produce sounds like 16 different instruments at once).

MIDI Specifics

How different timbres are produced digitally is by using a *patch*, which is the set of control settings that define a particular timbre. Patches are often organized into databases, called *banks*. For true aficionados, software patch editors are available.

A standard mapping specifying just what instruments (patches) will be associated with what channels has been agreed on and is called *General MIDI*. In General MIDI, there are 128 patches associated with standard instruments, and channel 10 is reserved for percussion instruments.

For most instruments, a typical message might be Note On (meaning, e.g., a keypress), consisting of what channel, what pitch, and what *velocity* (i.e., volume). For percussion instruments, the pitch data means which kind of drum. A Note On message thus consists of a *status* byte—which channel, what pitch—followed by two data bytes. It is followed by a Note Off message (key release), which also has a pitch (which note to turn off) and—for consistency, one supposes—a velocity (often set to zero and ignored).

The data in a MIDI status byte is between 128 and 255; each of the data bytes is between 0 and 127. Actual MIDI bytes are 8 bit, plus a 0 start and stop bit, making them 10-bit "bytes". Figure 6.9 shows the MIDI datastream.

A MIDI device often is capable of *programmability*, which means it has filters available for changing the bass and treble response and can also change the "envelope" describing how the amplitude of a sound changes over time. Figure 6.10 shows a model of a digital instrument's response to Note On/Note Off messages.

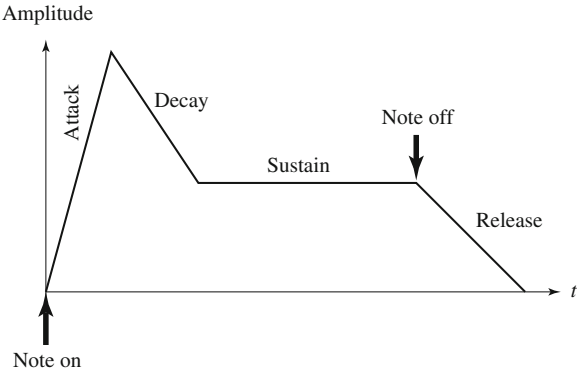

Fig. 6.10 Stages of amplitude versus time for a music note

MIDI sequencers (editors) allow you to work with standard music notation or get right into the data, if desired. MIDI files can also store wave table data. The advantage of wave-table data (WAV files) is that it much more precisely stores the exact sound of an instrument. A sampler is used to sample the audio data—for example, a "drum machine" always stores wave table data of real drums. So one could have a music editor using MIDI on one track plus digital audio such as vocals, say, on another track.

Of interest to computer science or engineering students is the fact that MIDI provides a full messaging protocol that can be used for whatever one likes, for example for controlling lights in a theater. We shall see below that there is a MIDI message for sending any kind and any number of bytes, which can then be used as the programmer desires.

Sequencers employ several techniques for producing more music from what is actually available. For example, looping over (repeating) a few bars can be more or less convincing. Volume can be easily controlled over time—this is called *time-varying amplitude modulation*. More interestingly, sequencers can also accomplish time compression or expansion with no pitch change.

While it is possible to change the pitch of a sampled instrument, if the key change is large, the resulting sound begins to sound displeasing. For this reason, samplers employ *multisampling*. A sound is recorded using several bandpass filters, and the resulting recordings are assigned to different keyboard keys. This makes frequency shifting for a change of key more reliable, since less shift is involved for each note.

MIDI Machine Control (MMC) is a subset of the MIDI specification that can be used for controlling recording equipment, e.g. multitrack recorders. The most common use of this facility is to effectively press "Play" on a remote device. An example usage would be if a MIDI device has a poor timer, then the master could activate it at just the right time. In general, MIDI can be used for controlling and synchronizing musical instrument synthesizers and recording equipment, and even control lighting.

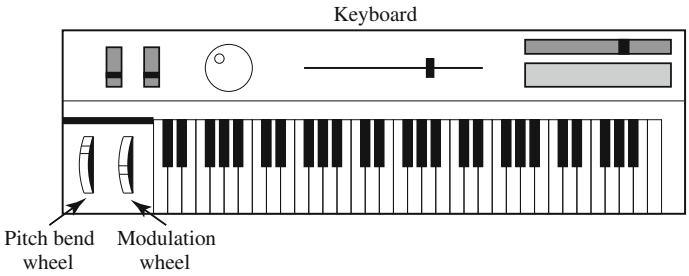

Fig. 6.11 A MIDI synthesizer

6.2.2 Hardware Aspects of MIDI

The MIDI hardware setup consists of a 31.25 kbps (kilobits per second) serial connection, with the 10-bit bytes including a 0 start and stop bit. Usually, MIDI-capable units are either input devices or output devices, not both.

Figure 6.11 shows a traditional synthesizer. The modulation wheel adds vibrato. Pitch bend alters the frequency, much like pulling a guitar string over slightly. There are often other controls, such as foots pedals, sliders, and so on.

The physical MIDI ports consist of 5-pin connectors labeled IN and OUT and there can also be a third connector, THRU. This last data channel simply copies data entering the IN channel. MIDI communication is half-duplex. MIDI IN is the connector via which the device receives all MIDI data. MIDI OUT is the connector through which the device transmits all the MIDI data it generates itself. MIDI THRU is the connector by which the device echoes the data it receives from MIDI IN (and only that—all the data generated by the device itself is sent via MIDI OUT). These ports are on the sound card or interface externally, either on a separate card or using a special interface to a serial port.

Figure 6.12 shows a typical MIDI sequencer setup. Here, the MIDI OUT of the keyboard is connected to the MIDI IN of a synthesizer and then THRU to each of the additional sound modules. During recording, a keyboard-equipped synthesizer sends MIDI messages to a sequencer, which records them. During playback, messages are sent from the sequencer to all the sound modules and the synthesizer, which play the music.

MIDI Message Transmission

The 31.25 kbps data rate is actually quite restrictive. To initiate playing a note, a 3-byte message is sent (with bytes equal to ten bits). If my hands are playing chords with all ten fingers, then a single crashing chord will take ten notes at 30 bits each, requiring transmission of 300 bits. At 31.25 kbps transmission of this chord will take about 0.01 sec, at a speed of about 0.001 seconds per note—and all this not counting the additional messages we'll have to send to turn *off* these ten notes. Moreover,

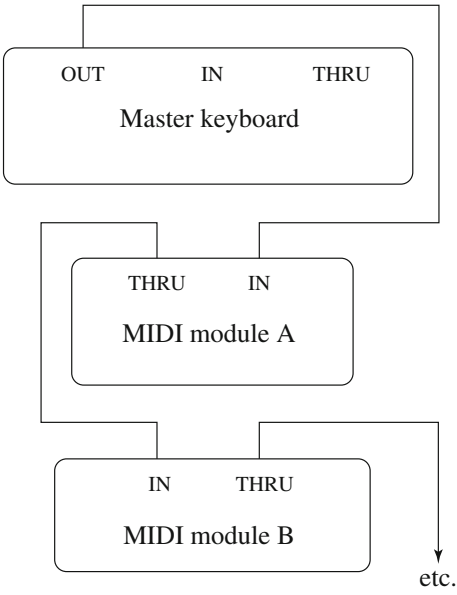

Fig. 6.12 A typical MIDI setup

using the pitch bend and modulation wheels in a synthesizer could generate many messages as well, all taking time to transmit. Hence, there could be an audible time lag generated by the slow bit transmission rate.

A trick used to tackle this problem is called *Running Status*: MIDI allows sending just the data, provided the command from the previous message has not changed. So instead of three bytes—command, data, data—MIDI would use just two bytes for the next message having the same command.

6.2.3 Structure of MIDI Messages

MIDI messages can be classified into two types, as in Fig. 6.13—channel messages and system messages—and further classified as shown. Each type of message will be examined below.

Channel Messages. A channel message can have up to 3 bytes; the first is the status byte (the opcode, as it were), and has its most significant bit set to 1. The four low-order bits identify which of the 16 possible channels this message belongs to, with the three remaining bits holding the message. For a data byte, the most significant bit is set to zero.

Voice Messages. This type of channel message controls a voice—that is, sends information specifying which note to play or to turn off—and encodes key pressure. Voice messages are also used to specify controller effects, such as sustain, vibrato, tremolo, and the pitch wheel. Table 6.3 lists these operations.

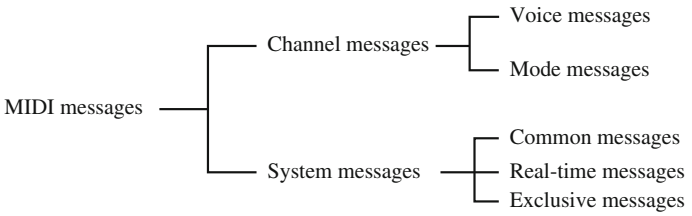

Fig. 6.13 MIDI message taxonomy

Table 6.3 MIDI voice messages

Voice message	Status byte	Data byte1	Data byte2
Note off	&H8n	Key number	Note off velocity
Note on	&H9n	Key number	Note on velocity
Polyphonic key Pressure	&HAn	Key number	Amount
Control change	&HBn	Controller number	Controller value
Program change	&HCn	Program number	None
Channel pressure	&HDn	Pressure value	None
Pitch bend	&HEn	MSB	LSB

&H indicates hexadecimal, and *n* in the Status byte hex value stands for a channel number. All values are in 0 ... 127 except Controller number, which is in 0 ... 120

For *Note On* and *Note Off* messages, the *velocity* is how quickly the key is played. Typically, a synthesizer responds to a higher velocity by making the note louder or brighter. Note On makes a note occur, and the synthesizer also attempts to make the note sound like the real instrument while the note is playing. *Pressure* messages can be used to alter the sound of notes while they are playing. The *Channel Pressure* message is a force measure for the keys on a specific channel (instrument) and has an identical effect on all notes playing on that channel. The other pressure message, *Polyphonic Key Pressure* (also called *Key Pressure*), specifies how much volume keys played together are to have and can be different for each note in a chord. Pressure is also called *aftertouch*.

The Control Change instruction sets various controllers (faders, vibrato, etc.). Each manufacturer may make use of different controller numbers for different tasks. However, controller 1 is likely the modulation wheel (for vibrato).

For example, a Note On message is followed by two bytes, one to identify the note and one to specify the velocity. Therefore, to play note number 80 with maximum velocity on channel 13, the MIDI device would send the following three hex byte values: &H9C &H50 &H7F. (A hexadecimal number has a range 0 .. 15. Since it is used to denote channels 1 to 16, "&HC" refers to channel 13). Notes are numbered such that middle C has number 60.

To play two notes simultaneously (effectively), first we would send a Program Change message for each of two channels. Recall that Program Change means to load a particular patch for that channel. So far, we have attached two timbres to two

Table 6.4 MIDI mode messages

1st data byte	Description	Meaning of 2nd data byte
& H79	Reset all controllers	None; set to 0
& H7A	Local control	0 = off; 127 = on
& H7B	All notes off	None; set to 0
& H7C	Omni mode off	None; set to 0
& H7D	Omni mode on	None; set to 0
& H7E	Mono mode on (Poly mode off)	Controller number
& H7F	Poly mode on (Mono mode off)	None; set to 0

different channels. Then sending two Note On messages (in serial) would turn on both channels. Alternatively, we could also send a Note On message for a particular channel and then another Note On message, with another pitch, before sending the Note Off message for the first note. Then we would be playing two notes effectively at the same time on the same instrument.

Recall that the Running Status method allows one to send one status byte, e.g., a Note On message, followed by a stream of data bytes that all are associated with the same status byte. For instance, a Note On message on channel 1, "&H90", could be followed by two data bytes as indicated in Table 6.3. But with Running Status we need not send another Note On but instead simply keep sending data-byte pairs for the next stream of Note On data. As well, in fact Running Status has another trick: if the Velocity data byte is 0 then that Note On message is interpreted as a Note Off. Hence one can send a single "&H90" followed by numerous Note On and Note Off data sets. So for example a Note On, Note Off pair for playing middle C on channel 1 could be sent as `&H90 &H3C &H7F; &H3C &H00` (middle C is note number 60 = "&H3C").

Polyphonic Pressure refers to how much force simultaneous notes have on several instruments. Channel Pressure refers to how much force a single note has on one instrument.

Channel Mode Messages. Channel mode messages form a special case of the Control Change message, and therefore all mode messages have opcode B (so the message is "&HBn," or 1011nnnn). However, a Channel Mode message has its first data byte in 121 through 127 (&H79–7F).

Channel mode messages determine how an instrument processes MIDI voice messages. Some examples include respond to all messages, respond just to the correct channel, don't respond at all, or go over to local control of the instrument.

Recall that the status byte is "&HB*n*," where *n* is the channel. The data bytes have meanings as shown in Table 6.4. *Local Control Off* means that the keyboard should be disconnected from the synthesizer (and another, external, device will be used to control the sound). *All Notes Off* is a handy command, especially if, as sometimes happens, a bug arises such that a note is left playing inadvertently. *Omni* means that

Table 6.5 MIDI system common messages

System common message	Status byte	Number of data bytes
MIDI timing code	&HF1	1
Song position pointer	&HF2	2
Song select	&HF3	1
Tune request	&HF6	None
EOX (terminator)	&HF7	None

Table 6.6 MIDI system real-time messages

System real-time message	Status byte
Timing clock	&HF8
Start sequence	&HFA
Continue sequence	&HFB
Stop sequence	&HFC
Active sensing	&HFE
System reset	&HFF

devices respond to messages from all channels. The usual mode is OMNI OFF—pay attention to your own messages only, and do not respond to every message regardless of what channel it is on. *Poly* means a device will play back several notes at once if requested to do so. The usual mode is POLY ON.

In POLY OFF—monophonic mode—the argument that represents the number of monophonic channels can have a value of zero, in which case it defaults to the number of voices the receiver can play; or it may set to a specific number of channels. However, the exact meaning of the combination of OMNI ON/OFF and *Mono/Poly* depends on the specific combination, with four possibilities. Suffice it to say that the usual combination is OMNI OFF, POLY ON.

System Messages. System messages have no channel number and are meant for commands that are not channel-specific, such as timing signals for synchronization, positioning information in prerecorded MIDI sequences, and detailed setup information for the destination device. Opcodes for all system messages start with "&HF." System messages are divided into three classifications, according to their use.

System Common Messages. Table 6.5 sets out these messages, which relate to timing or positioning. Song Position is measured in beats. The messages determine what is to be played upon receipt of a "start" real-time message (see below).

System Real-Time Messages. Table 6.6 sets out system real-time messages, which are related to synchronization.

System Exclusive Message. the final type of system message, *System Exclusive* messages, is included so that manufacturers can extend the MIDI standard. After the initial code, they can insert a stream of any specific messages that apply to their own

product. A System Exclusive message is supposed to be terminated by a terminator byte "&HF7," as specified in Table 6.5. However, the terminator is optional, and the datastream may simply be ended by sending the status byte of the next message.

6.2.4 General MIDI

For MIDI music to sound more or less the same on every machine, we would at least like to have the same patch numbers associated with the same instruments—for example, patch 1 should always be a piano, not a flugelhorn. To this end, General MIDI [5] is a scheme for assigning instruments to patch numbers. A standard percussion map also specifies 47 percussion sounds. Where a "note" appears on a musical score determines just what percussion element is being struck. This book's website includes both the General MIDI Instrument Path Map and the Percussion Key map.

Other requirements for General MIDI compatibility are that a MIDI device must support all 16 channels; must be multi-timbral (i.e., each channel can play a different instrument/program); must be polyphonic (i.e., each channel is able to play many voices); and must have a minimum of 24 dynamically allocated voices.

General MIDI Level2

An extended General MIDI, GM-2, was defined in 1999 and updated in 2003, with a standard SMF *Standard MIDI File* format defined. A nice extension is the inclusion of extra character information, such as karaoke lyrics, which can be displayed on a good sequencer.

6.2.5 MIDI-to-WAV Conversion

Some programs, such as early versions of Adobe Premiere, cannot include MIDI files—instead, they insist on WAV format files. Various shareware programs can approximate a reasonable conversion between these formats. The programs essentially consist of large lookup files that try to do a reasonable job of substituting predefined or shifted WAV output for some MIDI messages, with inconsistent success.

6.3 Quantization and Transmission of Audio

To be transmitted, sampled audio information must be digitized, and here we look at some of the details of this process. Once the information has been quantized, it can then be transmitted or stored. We go through a few examples in complete detail, which helps in understanding what is being discussed.

6.3.1 Coding of Audio

Quantization and transformation of data are collectively known as *coding* of the data. For audio, the μ-law technique for companding audio signals is usually combined with a simple algorithm that exploits the temporal redundancy present in audio signals. Differences in signals between the present and a previous time can effectively reduce the size of signal values and, most important, concentrate the histogram of pixel values (differences, now) into a much smaller range. The result of reducing the variance of values is that lossless compression methods that produce a bitstream with shorter bit lengths for more likely values, introduced in Chap. 7, fare much better and produce a greatly compressed bitstream.

In general, producing quantized sampled output for audio is called *Pulse Code Modulation*, or *PCM*. The differences version is called *DPCM* (and a crude but efficient variant is called *DM*). The adaptive version is called *ADPCM*, and variants that take into account speech properties follow from these. More complex models for audio are outlined in Chap. 13.

6.3.2 Pulse Code Modulation

6.3.2.1 PCM in General

Audio is analog—the waves we hear travel through the air to reach our eardrums. We know that the basic techniques for creating digital signals from analog ones consist of *sampling* and *quantization*. Sampling is invariably done uniformly—we select a sampling rate and produce one value for each sampling time.

In the magnitude direction, we digitize by quantization, selecting breakpoints in magnitude and remapping any value within an interval to one representative output level. The set of interval boundaries is sometimes called *decision boundaries*, and the representative values are called *reconstruction levels*.

We say that the boundaries for quantizer input intervals that will all be mapped into the same output level form a *coder mapping*, and the representative values that are the output values from a quantizer are a *decoder mapping*. Since we quantize, we may choose to create either an accurate or less accurate representation of sound magnitude values. Finally, we may wish to *compress* the data, by assigning a bitstream that uses fewer bits for the most prevalent signal values.

Every compression scheme has three stages:

1. **Transformation.** The input data is *transformed* to a new representation that is easier or more efficient to compress. For example, in Predictive Coding, (discussed later in the chapter) we predict the next signal from previous ones and transmit the prediction error.
2. **Loss.** We may introduce *loss* of information. Quantization is the main lossy step. Here, we use a limited number of reconstruction levels fewer than in the original signal. Therefore, quantization necessitates some loss of information.

3. **Coding.** Here, we assign a *codeword* (thus forming a binary bitstream) to each output level or symbol. This could be a fixed-length code or a variable-length code, such as Huffman coding (discussed in Chap. 7).

For audio signals, we first consider PCM, the digitization method. That enables us to consider Lossless Predictive Coding as well as the DPCM scheme; these methods use *differential coding*. We also look at the adaptive version, ADPCM, which is meant to provide better compression.

Pulse Code Modulation, is a formal term for the sampling and quantization we have already been using. *Pulse* comes from an engineer's point of view that the resulting digital signals can be thought of as infinitely narrow vertical "pulses." As an example of PCM, audio samples on a CD might be sampled at a rate of 44.1 kHz, with 16 bits per sample. For stereo sound, with two channels, this amounts to a data rate of about 1,400 kbps.

PCM in Speech Compression

Recall that in Sect. 6.1.6 we considered *companding*: the so-called compressor and expander stages for speech signal processing, for telephony. For this application, signals are first transformed using the μ-law (or A-law for Europe) rule into what is essentially a logarithmic scale. Only then is PCM, using uniform quantization, applied. The result is that finer increments in sound volume are used at the low-volume end of speech rather than at the high-volume end, where we can't discern small changes in any event.

Assuming a bandwidth for speech from about 50 Hz to about 10 kHz, the Nyquist rate would dictate a sampling rate of 20 kHz. Using uniform quantization without companding, the minimum sample size we could get away with would likely be about 12 bits. Hence, for mono speech transmission the bitrate would be 240 kbps. With companding, we can safely reduce the sample size to 8 bits with the same perceived level of quality and thus reduce the bitrate to 160 kbps. However, the standard approach to telephony assumes that the highest-frequency audio signal we want to reproduce is about 4 kHz. Therefore, the sampling rate is only 8 kHz, and the companded bitrate thus reduces to only 64 kbps.

We must also address two small wrinkles to get this comparatively simple form of speech compression right. First because only sounds up to 4 kHz are to be considered, all other frequency content must be noise. Therefore, we should remove this high-frequency content from the analog input signal. This is done using a band-limiting filter that blocks out high frequencies as well as very low ones. The "band" of not-removed ("passed") frequencies are what we wish to keep. This type of filter is therefore also called a bandpass filter.

Second, once we arrive at a pulse signal, such as the one in Fig. 6.14a, we must still perform digital-to-analog conversion and then construct an output analog signal. But the signal we arrive at is effectively the staircase shown in Fig. 6.14b. This type of discontinuous signal contains not just frequency components due to the original

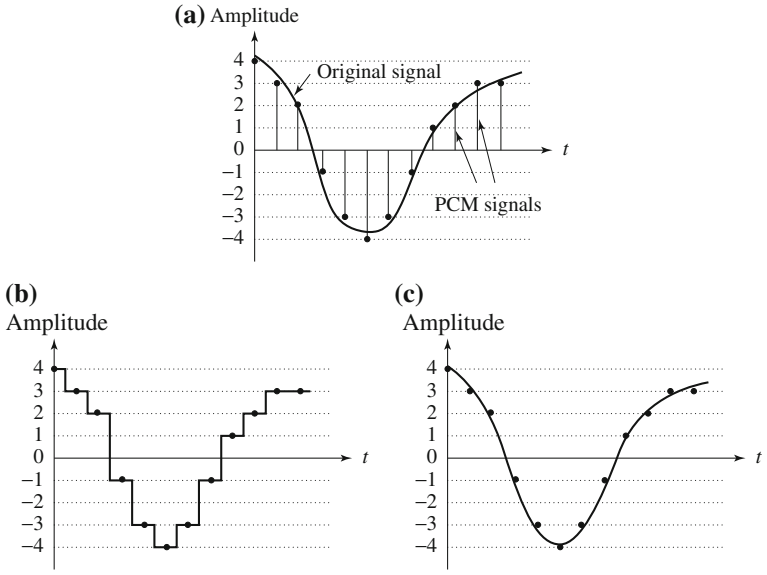

Fig. 6.14 Pulse code modulation (PCM): **a** original analog signal and its corresponding PCM signals; **b** decoded staircase signal; **c** reconstructed signal after low-pass filtering

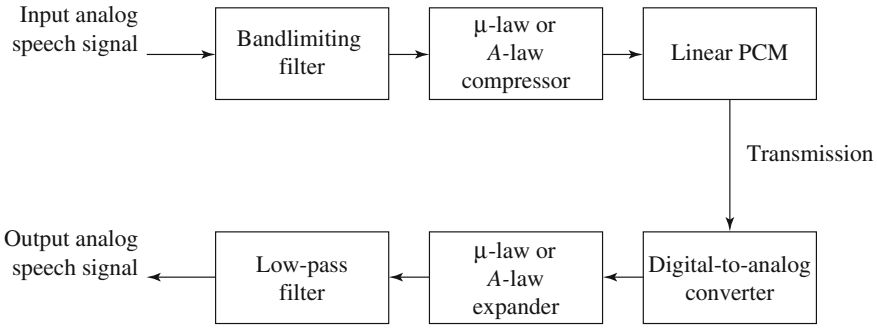

Fig. 6.15 PCM signal encoding and decoding

signal but, because of the sharp corners, also a theoretically infinite set of higher frequency components (from the theory of Fourier analysis, in signal processing). We know these higher frequencies are extraneous. Therefore, the output of the digital-to-analog converter is in turn passed to a *low-pass filter*, which allows only frequencies up to the original maximum to be retained. Figure 6.15 shows the complete scheme for encoding and decoding telephony signals as a schematic. As a result of the low-pass filtering, the output becomes smoothed, as Fig. 6.14c shows. For simplicity, Fig. 6.14 does not show the effect of companding.

A-law or μ-law PCM coding is used in the older International Telegraph and Telephone Consultative Committee (CCITT) standard G.711, for digital telephony. This CCITT standard is now subsumed into standards promulgated by a newer organization, the International Telecommunication Union (ITU).

6.3.3 Differential Coding of Audio

Audio is often stored not in simple PCM but in a form that exploits differences. For a start, differences will generally be smaller numbers and hence offer the possibility of using fewer bits to store.

An advantage of forming differences is that the histogram of a difference signal is usually considerably more peaked than the histogram for the original signal. For example, as an extreme case, the histogram for a linear ramp signal that has constant slope is uniform, whereas the histogram for the derivative of the signal (i.e., the differences, from sampling point to sampling point) consists of a spike at the slope value.

Generally, if a time-dependent signal has some consistency over time (*temporal redundancy*), the difference signal—subtracting the current sample from the previous one—will have a more peaked histogram, with a maximum around zero. Consequently, if we then go on to assign bitstring codewords to differences, we can assign short codes to prevalent values and long codewords to rarely occurring ones.

To begin with, consider a lossless version of this scheme. Loss arises when we quantize. If we apply no quantization, we can still have compression—via the decrease in the variance of values that occurs in differences, compared to the original signal. Chapter 7 introduces more sophisticated versions of lossless compression methods, but it helps to see a simple version here as well. With quantization, Predictive Coding becomes DPCM, a lossy method; we will also try out that scheme.

6.3.4 Lossless Predictive Coding

Predictive coding simply means transmitting differences—we predict the next sample as being equal to the current sample and send not the sample itself but the error involved in making this assumption. That is, if we predict that the next sample equals the previous one, then the error is just the difference between previous and next. Our prediction scheme could also be more complex.

However, we do note one problem. Suppose our integer sample values are in the range $0 \,..\, 255$. Then differences could be as much as $-255 \,..\, 255$. So we have unfortunately increased our *dynamic range* (ratio of maximum to minimum) by a factor of two: we may well need more bits than we needed before to transmit some differences. Fortunately, we can use a trick to get around this problem, as we shall see.

So, basically, predictive coding consists of finding differences and transmitting them, using a PCM system such as the one introduced in Sect. 6.3.2. First, note that

differences of integers will at least be integers. Let's formalize our statement of what we are doing by defining the integer signal as the set of values f_n. Then we *predict* values $\hat{f}_n$ as simply the previous value, and we define the error e_n as the difference between the actual and predicted signals:

$$\begin{aligned} \hat{f}_n &= f_{n-1} \\ e_n &= f_n - \hat{f}_n \end{aligned} \tag{6.13}$$

We certainly would like our error value e_n to be as small as possible. Therefore, we would wish our prediction $\hat{f}_n$ to be as close as possible to the actual signal f_n. But for a particular sequence of signal values, some *function* of a few of the previous values, f_{n-1}, f_{n-2}, f_{n-3}, etc., may provide a better prediction of f_n. Typically, a linear *predictor* function is used:

$$\hat{f}_n = \sum_{k=1}^{2 \text{ to } 4} a_{n-k} f_{n-k} \tag{6.14}$$

Such a predictor can be followed by a truncating or rounding operation to result in integer values. In fact, since now we have such coefficients a_{n-k} available, we can even change them adaptively (see Sect. 6.3.7 below).

The idea of forming differences is to make the histogram of sample values more peaked. For example, Fig. 6.16a plots 1 second of sampled speech at 8 kHz, with magnitude resolution of 8 bits per sample.

A histogram of these values is centered around zero, as in Fig. 6.16b. Figure 6.16c shows the histogram for corresponding speech signal *differences*: difference values are much more clustered around zero than are sample values themselves. As a result, a method that assigns short codewords to frequently occurring symbols will assign a short code to *zero* and do rather well. Such a coding scheme will much more efficiently code sample differences than samples themselves, and a similar statement applies if we use a more sophisticated predictor than simply the previous signal value.

However, we are still left with the problem of what to do if, for some reason, a particular set of difference values does indeed consist of some exceptional large differences. A clever solution to this difficulty involves defining two new codes to add to our list of difference values, denoted `SU` and `SD`, standing for `Shift-Up` and `Shift-Down`. Some special values will be reserved for them.

Suppose samples are in the range $0 \,..\, 255$, and differences are in $-255 \,..\, 255$. Define `SU` and `SD` as shifts by 32. Then we could in fact produce codewords for a limited set of signal differences, say only the range $-15 \,..\, 16$. Differences (that inherently are in the range $-255 \,..\, 255$) lying in the limited range can be coded as is, but if we add the extra two values for `SU, SD`, a value outside the range $-15 \,..\, 16$ can be transmitted as a series of shifts, followed by a value that is indeed inside the range $-15 \,..\, 16$. For example, 100 is transmitted as `SU, SU, SU, 4`, where (the codes for) `SU` and for `4` are what are sent.

Lossless Predictive Coding is …lossless! That is, the decoder produces the same signals as the original. It is helpful to consider an explicit scheme for such coding

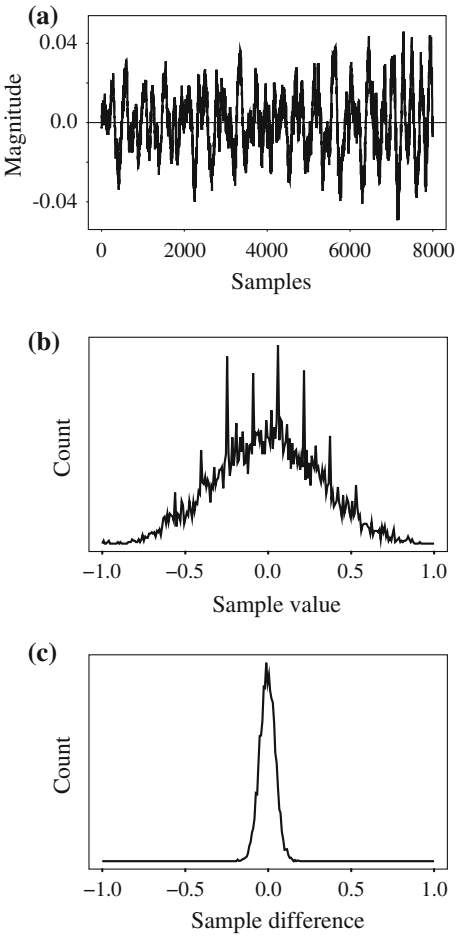

Fig. 6.16 Differencing concentrates the histogram: **a** digital speech signal; **b** histogram of digital speech signal values; **c** histogram of digital speech signal differences

considerations, so let's do that here (we won't use the most complicated scheme, but we'll try to carry out an entire calculation). As a simple example, suppose we devise a predictor for $\hat{f}_n$ as follows:

$$\hat{f}_n = \lfloor \frac{1}{2}(f_{n-1} + f_{n-2}) \rfloor$$
$$e_n = f_n - \hat{f}_n \tag{6.15}$$

Then the error e_n (or a codeword for it) is what is actually transmitted.

Let's consider an explicit example. Suppose we wish to code the sequence $f_1, f_2, f_3, f_4, f_5 = 21, 22, 27, 25, 22$. For the purposes of the predictor, we'll invent an extra signal value f_0, equal to $f_1 = 21$, and first transmit this initial value, uncoded; after all, every coding scheme has the extra expense of some header information.

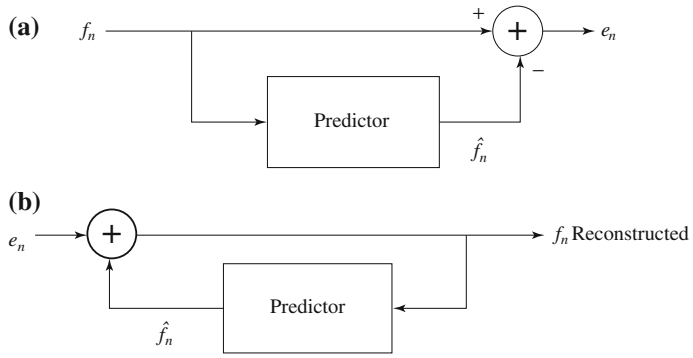

Fig. 6.17 Schematic diagram for Predictive Coding: **a** encoder; **b** decoder

Then the first error, e_1, is zero, and subsequently

$$\hat{f}_2 = 21, \quad e_2 = 22 - 21 = 1$$

$$\hat{f}_3 = \lfloor \frac{1}{2}(f_2 + f_1) \rfloor = \lfloor \frac{1}{2}(22 + 21) \rfloor = 21$$
$$e_3 = 27 - 21 = 6$$

$$\hat{f}_4 = \lfloor \frac{1}{2}(f_3 + f_2) \rfloor = \lfloor \frac{1}{2}(27 + 22) \rfloor = 24$$
$$e_4 = 25 - 24 = 1$$

$$\hat{f}_5 = \lfloor \frac{1}{2}(f_4 + f_3) \rfloor = \lfloor \frac{1}{2}(25 + 27) \rfloor = 26$$
$$e_5 = 22 - 26 = -4 \tag{6.16}$$

The error does center around zero, we see, and coding (assigning bitstring codewords) will be efficient. Figure 6.17 shows a typical schematic diagram used to encapsulate this type of system. Notice that the Predictor emits the predicted value $\hat{f}_n$. What is invariably (and annoyingly) left out of such schematics is the fact that the predictor is based on $f_{n-1}, f_{n-2}, \ldots$. Therefore, the predictor must involve a memory. At the least, the predictor includes a circuit for incorporating a delay in the signal, to store f_{n-1}.

6.3.5 DPCM

Differential Pulse Code Modulation is exactly the same as Predictive Coding, Predictive coding except that it incorporates a quantizer step. Quantization is as in PCM and can be uniform or nonuniform. One scheme for analytically determining the best set of nonuniform quantizer steps is the *Lloyd-Max* quantizer, named for Stuart

Lloyd and Joel Max, which is based on a least-squares minimization of the error term.

Here, we should adopt some nomenclature for signal values. We shall call the original signal f_n, the predicted signal $\hat{f}_n$, and the quantized, reconstructed signal $\tilde{f}_n$. How DPCM operates is to form the prediction, form an error e_n by subtracting the prediction from the actual signal, then quantize the error to a quantized version, $\tilde{e}_n$. The equations that describe DPCM are as follows:

$$\begin{aligned} \hat{f}_n &= \text{function\_of } (\tilde{f}_{n-1}, \tilde{f}_{n-2}, \tilde{f}_{n-3}, \ldots) \\ e_n &= f_n - \hat{f}_n \\ \tilde{e}_n &= Q[e_n] \\ &\quad \text{transmit codeword}(\tilde{e}_n) \\ &\quad \text{reconstruct: } \tilde{f}_n = \hat{f}_n + \tilde{e}_n \end{aligned} \tag{6.17}$$

Codewords for quantized error values $\tilde{e}_n$ are produced using entropy coding, such as Huffman coding (discussed in Chap. 7).

Notice that the predictor is always based on the reconstructed, quantized version of the signal: the reason for this is that then the encoder side is not using any information not available to the decoder side. Generally, if by mistake we made use of the *actual* signals f_n in the predictor instead of the reconstructed ones $\tilde{f}_n$, quantization error would tend to accumulate and could get worse rather than being centered on zero.

The main effect of the coder–decoder process is to produce reconstructed, quantized signal values $\tilde{f}_n = \hat{f}_n + \tilde{e}_n$. The "distortion" is the average squared error $[\sum_{n=1}^{N} (\tilde{f}_n - f_n)^2]/N$, and one often sees diagrams of distortion versus the number of bit levels used. A Lloyd-Max quantizer will do better (have less distortion) than a uniform quantizer.

For any signal, we want to choose the size of quantization steps so that they correspond to the range (the maximum and minimum) of the signal. Even using a uniform, equal-step quantization will naturally do better if we follow such a practice. For speech, we could modify quantization steps as we go, by estimating the mean and variance of a patch of signal values and shifting quantization steps accordingly, for every block of signal values. That is, starting at time i we could take a block of N values f_n and try to minimize the quantization error:

$$\min \sum_{n=i}^{i+N-1} (f_n - Q[f_n])^2 \tag{6.18}$$

Since signal *differences* are very peaked, we could model them using a Laplacian probability distribution function, which is also strongly peaked at zero [6]: it looks like $l(x) = (1/\sqrt{2\sigma^2}) exp(-\sqrt{2}|x|/\sigma)$, for variance σ^2. So typically, we assign quantization steps for a quantizer with nonuniform steps by assuming that signal differences, d_n, say, are drawn from such a distribution and then choosing steps to minimize

$$\min \sum_{n=i}^{i+N-1} (d_n - Q[d_n])^2 \, l(d_n) \tag{6.19}$$

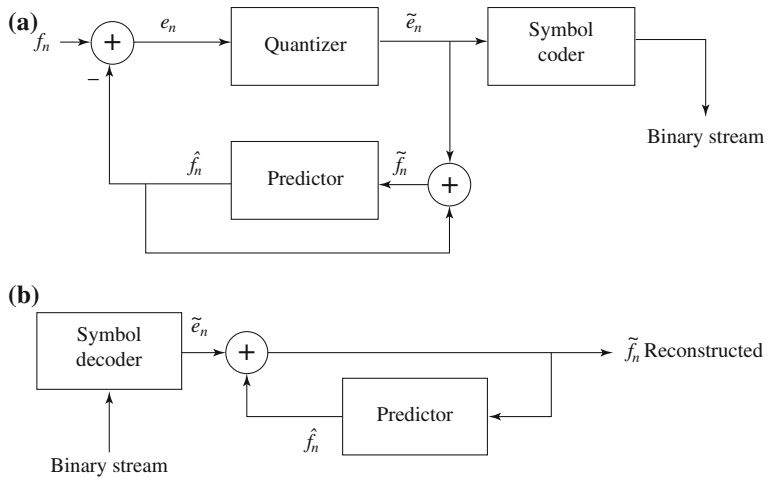

Fig. 6.18 Schematic diagram for DPCM: **a** encoder; **b** decoder

This is a least-squares problem and can be solved iteratively using the Lloyd-Max quantizer.

Figure 6.18 shows a schematic diagram for the DPCM coder and decoder. As is common in such diagrams, several interesting features are more or less not indicated. First, we notice that the predictor makes use of the reconstructed, quantized signal values $\tilde{f}_n$, not actual signal values f_n—that is, the encoder simulates the decoder in the predictor path. The quantizer can be uniform or nonuniform.

The box labeled "Symbol coder" in the block diagram simply means a Huffman coder—the details of this step are set out in Chap. 7. The prediction value $\hat{f}_n$ is based on however much history the prediction scheme requires: we need to buffer previous values of $\tilde{f}$ to form the prediction. Notice that the quantization noise, $f_n - \tilde{f}_n$, is equal to the quantization effect on the error term, $e_n - \tilde{e}_n$.

It helps us explicitly to understand the process of coding to look at actual numbers. Suppose we adopt a particular predictor as follows:

$$\hat{f}_n = \text{trunc}\left[\left(\tilde{f}_{n-1} + \tilde{f}_{n-2}\right)/2\right]$$
$$\text{so that } e_n = f_n - \hat{f}_n \text{ is an integer.} \tag{6.20}$$

Let us use the particular quantization scheme

$$\tilde{e}_n = Q[e_n] = 16 * \text{trunc}\left[(255 + e_n)/16\right] - 256 + 8$$
$$\tilde{f}_n = \hat{f}_n + \tilde{e}_n \tag{6.21}$$

First, we note that the error is in the range $-255 .. 255$—that is, 511 levels are possible for the error term. The quantizer takes the simple course of dividing the error range into 32 patches of about 16 levels each. It also makes the representative reconstructed value for each patch equal to the midway point for each group of 16 levels.

Table 6.7 DPCM quantizer reconstruction levels

e_n in range	Quantized to value
$-255 \,..\, -240$	-248
$-239 \,..\, -224$	-232
⋮	⋮
$-31 \,..\, -16$	-24
$-15 \,..\, 0$	-8
$1 \,..\, 16$	8
$17 \,..\, 32$	24
⋮	⋮
$225 \,..\, 240$	232
$241 \,..\, 255$	248

Table 6.7 gives output values for any of the input codes: 4-bit codes are mapped to 32 reconstruction levels in a staircase fashion. (Notice that the final range includes only 15 levels, not 16.)

As an example stream of signal values, consider the set of values

f_1	f_2	f_3	f_4	f_5
130	150	140	200	230

We prepend extra values $f = 130$ in the datastream that replicate the first value, f_1, and initialize with quantized error $\tilde{e}_1 \equiv 0$, so that we ensure the first reconstructed value is exact: $\tilde{f}_1 = 130$. Then subsequent values calculated are as follows (with prepended values in a box):

$$\begin{aligned}
\hat{f} &= \boxed{130},\ 130,\ 142,\ 144,\ 167 \\
e &= \quad \boxed{0},\ 20,\ -2,\ 56,\ 63 \\
\tilde{e} &= \quad \boxed{0},\ 24,\ -8,\ 56,\ 56 \\
\tilde{f} &= \boxed{130},\ 154,\ 134,\ 200,\ 223
\end{aligned}$$

On the decoder side, we again assume extra values $\tilde{f}$ equal to the correct value $\tilde{f}_1$, so that the first reconstructed value $\tilde{f}_1$ is correct. What is received is $\tilde{e}_n$, and the reconstructed $\tilde{f}_n$ is identical to the one on the encoder side, provided we use exactly the same prediction rule.

6.3.6 DM

DM stands for *Delta Modulation*, a much-simplified version of DPCM often used as a quick analog-to-digital converter. We include this scheme here for completeness.

6.3.6.1 Uniform-Delta DM

The idea in DM is to use only a *single* quantized error value, either positive or negative. Such a 1-bit coder thus produces coded output that follows the original signal in a staircase fashion. The relevant set of equations is as follows:

$$\begin{aligned}
\hat{f}_{\mathrm{n}} &= \tilde{f}_{\mathrm{n}-1} \\
e_{\mathrm{n}} &= f_{\mathrm{n}} - \hat{f}_{\mathrm{n}} = f_{\mathrm{n}} - \tilde{f}_{\mathrm{n}-1} \\
\tilde{e}_{\mathrm{n}} &= \begin{cases} +k \text{ if } e_{\mathrm{n}} > 0, \text{ where } k \text{ is a constant} \\ -k \text{ otherwise,} \end{cases} \\
\tilde{f}_{\mathrm{n}} &= \hat{f}_{n} + \tilde{e}_{\mathrm{n}}
\end{aligned} \tag{6.22}$$

Note that the prediction simply involves a delay.

Again, let's consider actual numbers. Suppose signal values are as follows:

f_1	f_2	f_3	f_4
10	11	13	15

We also define an exact reconstructed value $\tilde{f}_1 = f_1 = 10$.

Suppose we use a step value $k = 4$. Then we arrive at the following values:

$$\begin{aligned}
&\hat{f}_2 = 10,\ e_2 = 11 - 10 = 1, \quad \tilde{e}_2 = 4, \quad \tilde{f}_2 = 10 + 4 = 14 \\
&\hat{f}_3 = 14,\ e_3 = 13 - 14 = -1,\ \tilde{e}_3 = -4,\ \tilde{f}_3 = 14 - 4 = 10 \\
&\hat{f}_4 = 10,\ e_4 = 15 - 10 = 5, \quad \tilde{e}_4 = 4, \quad \tilde{f}_4 = 10 + 4 = 14
\end{aligned}$$

We see that the reconstructed set of values 10, 14, 10, 14 never strays far from the correct set 10, 11, 13, 15.

Nevertheless, it is not difficult to discover that DM copes well with more or less constant signals, but not as well with rapidly changing signals. One approach to mitigating this problem is to simply increase the sampling, perhaps to many times the Nyquist rate. This scheme can work well and makes DM a very simple yet effective analog-to-digital converter.

6.3.6.2 Adaptive DM

However, if the slope of the actual signal curve is high, the staircase approximation cannot keep up. A straightforward approach to dealing with a steep curve is to simply change the step size k *adaptively*—that is, in response to the signal's current properties.

6.3.7 ADPCM

Adaptive DPCM takes the idea of adapting the coder to suit the input much further. Basically, two pieces make up a DPCM coder: the quantizer and the predictor. Above, in Adaptive DM, we adapted the quantizer step size to suit the input. In DPCM, we

can *adaptively modify the quantizer*, by changing the step size as well as decision boundaries in a nonuniform quantizer.

We can carry this out in two ways: using the properties of the input signal (called *forward adaptive quantization*), or the properties of the quantized output. For if quantized errors become too large, we should change the nonuniform Lloyd-Max quantizer (this is called *backward adaptive quantization*).

We can also *adapt the predictor*, again using forward or backward adaptation. Generally, making the predictor coefficients adaptive is called *Adaptive Predictive Coding* (APC). It is interesting to see how this is done. Recall that the predictor is usually taken to be a linear function of previously reconstructed quantized values, $\tilde{f}_{\mathrm{n}}$. The number of previous values used is called the *order* of the predictor. For example, if we use M previous values, we need M coefficients a_i, $i = 1 \,..\, M$ in a predictor

$$\hat{f}_{\mathrm{n}} = \sum_{i=1}^{M} a_i \, \tilde{f}_{\mathrm{n}-i} \tag{6.23}$$

However we can get into a difficult situation if we try to *change* the prediction coefficients that multiply previous quantized values, because that makes a complicated set of equations to solve for these coefficients. Suppose we decide to use a least-squares approach to solving a minimization, trying to find the best values of the a_i:

$$\min \sum_{n=1}^{N} (f_{\mathrm{n}} - \hat{f}_{\mathrm{n}})^2 \tag{6.24}$$

where here we would sum over a large number of samples f_{n} for the current patch of speech, say. But because $\hat{f}_{\mathrm{n}}$ depends on the quantization, we have a difficult problem to solve. Also, we should really be changing the fineness of the quantization at the same time, to suit the signal's changing nature; this makes things problematical.

Instead, we usually resort to solving the simpler problem that results from using not $\tilde{f}_{\mathrm{n}}$ in the prediction but simply the signal f_{n} itself. This is indeed simply solved, since, explicitly writing in terms of the coefficients a_i, we wish to solve

$$\min \sum_{n=1}^{N} (f_{\mathrm{n}} - \sum_{i=1}^{M} a_i \, f_{n-i})^2 \tag{6.25}$$

Differentiation with respect to each of the a_i and setting to zero produces a linear system of M equations that is easy to solve. (The set of equations is called the Wiener-Hopf equations.)

Thus, we indeed find a simple way to adaptively change the predictor as we go. For speech signals, it is common to consider *blocks* of signal values, just as for image coding, and adaptively change the predictor, quantizer, or both. If we sample at 8 kHz, a common block size is 128 samples—16 msec of speech. Figure 6.19 shows a schematic diagram for the ADPCM coder and decoder [7].

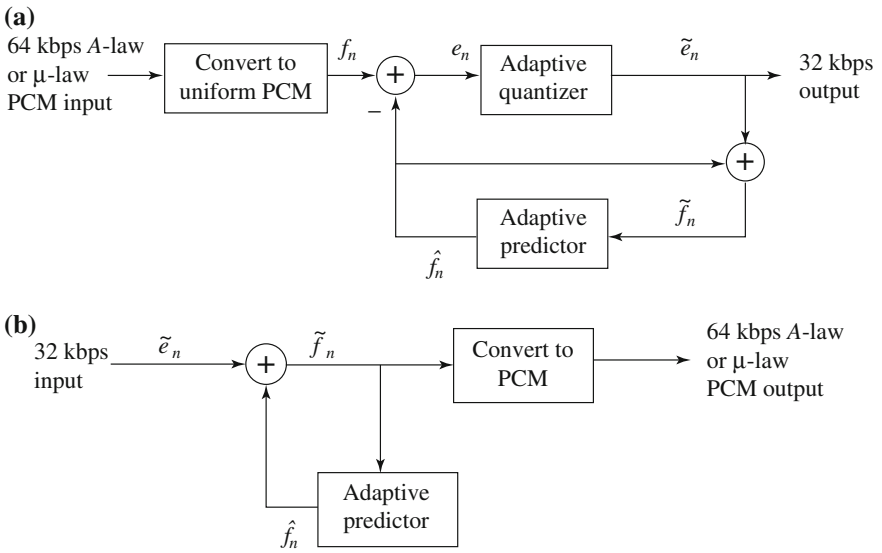

Fig. 6.19 Schematic diagram for: **a** ADPCM encoder; **b** decoder

6.4 Exercises

1. We wish to develop a new Internet service, for doctors. Medical ultrasound is in the range 2–10 MHz; what should our sampling rate be chosen as?
2. My old Soundblaster card is an 8-bit card.
   (a) What is it 8 bits of?
   (b) What is the best SQNR (Signal to Quantization Noise Ratio) it can achieve?
3. If a tuba is 20 dB louder than a singer's voice, what is the ratio of intensities of the tuba to the voice?
4. If a set of ear protectors reduces the noise level by 30 dB, how much do they reduce the intensity (the power)?
5. It is known that a loss of audio output at both ends of the audible frequency range is inevitable due to the frequency response function of audio amplifier.
   (a) If the output was 1 volt for frequencies at mid-range, after a loss of -3 dB at 18 kHz, what is the output voltage at this frequency?
   (b) To compensate the loss, a listener can adjust the gain (and hence the output) at different frequencies from an equalizer. If the loss remains -3 dB and a gain through the equalizer is 6 dB at 18 kHz, what is the output voltage now? [Hint: Assume $\log_{10} 2 = 0.3$.]
6. Suppose the Sampling Frequency is 1.5 times the True Frequency. What is the Alias Frequency?
7. In a crowded room, we can still pick out and understand a nearby speaker's voice notwithstanding the fact that general noise levels may be high. This is what is

known as the "cocktail-party effect"; how it operates is that our hearing can localize a sound source by taking advantage of the difference in phase between the two signals entering our left and right ears ("binaural auditory perception"). In mono, we could not hear our neighbor's conversation very well if the noise level were at all high.

State how you think a karaoke machine works.

Hint: the mix for commercial music recordings is such that the "pan" parameter is different going to the left and right channels for each instrument. That is, for an instrument, the left, or the right, channel is emphasized. How would the singer's track timing have to be recorded in order to make it easy to subtract out the sound of the singer? (And this is typically done.)

8. The *dynamic range* of a signal V is the ratio of the maximum to the minimum, expressed in decibels. The dynamic range expected in a signal is to some extent an expression of the signal quality. It also dictates the number of bits per sample needed in order to reduce the quantization noise down to an acceptable level; e.g., we may like to reduce the noise to at least an order of magnitude below $V_{\min}$.

   Suppose the dynamic range for a signal is 60 dB. Can we use 10 bits for this signal? Can we use 16 bits?

9. Suppose the dynamic range of speech in telephony implies a ratio $V_{\max}/V_{\min}$ of about 256. Using uniform quantization, how many bits should we use to encode speech, so as to make the quantization noise at least an order of magnitude less than the smallest detectable telephonic sound?

10. *Perceptual nonuniformity* is a general term for describing the nonlinearity of human perception, e.g., when a certain parameter of an audio signal varies, humans do not necessarily perceive the difference in proportion to the amount of change.

    (a) Briefly describe at least two types of Perceptual nonuniformities in human auditory perception.
    (b) Which one of them does A-law (or μ-law) attempt to approximate? Why could it improve the quantization?

11. Suppose we mistakenly always use the 0.75 point instead of the 0.50 point in a quantization interval as the decision point, in deciding to which quantization level an analog value should be mapped. Above, we have a rough calculation of SQNR. What effect does this mistake have on the SQNR?

12. State the Nyquist frequency for the following digital sample intervals. Express the result in Hertz in each case.

    (a) 1 ms
    (b) 0.005 s
    (c) 1 h

13. Draw a diagram showing a sinusoid at 5.5 kHz, and sampling at 8 kHz (just show 8 intervalsbetween samples in your plot). Draw the alias at 2.5 kHz and

show that in the 8 sample intervals, exactly 5.5 cycles of the true signal fit into 2.5 cycles of the alias signal.

14. In an old Western movie, we notice that a stagecoach wheel appears to be moving backwards at 5° per frame, even though the stagecoach is moving forward. To what is this effect due? What is the true situation?
15. Suppose a signal contains tones at 1 kHz, 10 kHz, and 21 kHz, and is sampled at the rate 12 kHz (and then processed with an anti-aliasing filter limiting output to 6 kHz). What tones are included in the output?
    Hint: most of the output consists of aliasing.
16. The Pitch Bend opcode in MIDI is followed by two data bytes specifying how the control is to be altered. How many bits of accuracy does this amount of data correspond to? Why?
17. (a) Can a single MIDI message produce more than one note sounding?
    (b) Is it possible that more than one note can be sounding on a particular instrument at once? How is that done in MIDI?
    (c) Is the Program Change MIDI message a Channel Message? What does this message accomplish? Based on the Program Change message, how many different instruments are there in General MIDI? Why?
    (d) In general, what are the two main kinds of MIDI messages? In terms of data, what is the main difference between the two types of messages? Within those two categories, please list the different sub-types.
18. The note "A above Middle C" (with frequency 440 Hz) is note 69 in General MIDI. What MIDI bytes (in hex) should be sent to play a note twice the frequency of (i.e., one octave above) "A above Middle C" at maximum volume on channel 1? (Don't include start/stop bits.)
    Information: An octave is 12 steps on a piano, i.e., 12 notes up.
19. Give an example (in English, not hex) of a MIDI voice message.
    Describe the parts of the "assembler" statement for the message you suggested above.
    What does a "program change" message do? Suppose "Program change" is hex &HC1 . What does the instruction &HC103 do?
20. We have suddenly invented a new kind of music: "18-tone music", that requires a keyboard with 180 keys. How would we have to change the MIDI standard to be able to play this music?
21. In PCM, what is the *delay*, assuming 8 kHz sampling? Generally, delay is the penalty associated with any algorithm due to sampling, processing, and analysis.
22. (a) Suppose we use a predictor as follows:

$$\begin{aligned} \hat{f}_{\mathrm{n}} &= \mathrm{trunc}\left(\frac{1}{2}(\tilde{f}_{\mathrm{n}-1} + \tilde{f}_{\mathrm{n}-2})\right), \\ e_{\mathrm{n}} &= f_{\mathrm{n}} - \hat{f}_{\mathrm{n}}. \end{aligned} \tag{6.26}$$

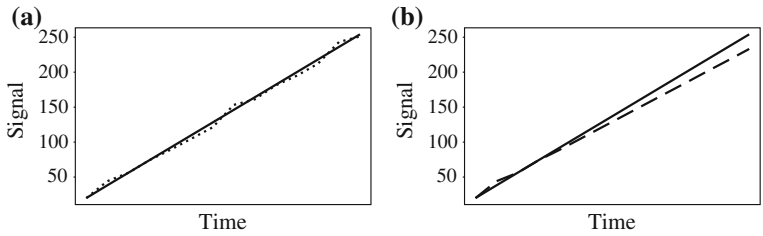

Fig. 6.20 **a** DPCM reconstructed signal (*dotted line*) tracks the input signal (*solid line*). **b** DPCM reconstructed signal (*dashed line*) steers farther and farther from the input signal (*solid line*)

Also, suppose we adopt the quantizer Eq. (6.21). If the input signal has values as follows:

$$20\ 38\ 56\ 74\ 92\ 110\ 128\ 146\ 164\ 182\ 200\ 218\ 236\ 254$$

then show that the output from a DPCM coder (without entropy coding) is as follows:

$$20\ 44\ 56\ 74\ 89\ 105\ 121\ 153\ 161\ 181\ 195\ 212\ 243\ 251.$$

Figure 6.20a shows how the quantized reconstructed signal tracks the input signal.

(b) Now, suppose by mistake on the coder side we inadvertently use the predictor for *lossless coding*, Eq. (6.15), using original values f_n instead of quantized ones, $\tilde{f}_n$. Show that on the decoder side we end up with reconstructed signal values as follows:

$$20\ 44\ 56\ 74\ 89\ 105\ 121\ 137\ 153\ 169\ 185\ 201\ 217\ 233,$$

so that the error gets progressively worse.
Figure 6.20b shows how this appears: the reconstructed signal gets progressively worse.

References

1. B. Truax, *Handbook for Acoustic Ecology*, 2nd edn. (Cambridge Street Publishing, 1999)
2. K.C. Pohlmann, *Principles of Digital Audio*, 6th edn. (McGraw-Hill, New York, 2010)
3. J.H. McClellan, R.W. Schafer, M.A. Yoder, *DSP First: A Multimedia Approach*. (Prentice-Hall PTR, Upper Saddle River, 1998)
4. J. Heckroth, Tutorial on MIDI and music synthesis. The MIDI Manufacturers Association, POB 3173, La Habra, CA 90632–3173, (1995).

5. P.K. Andleigh, K. Thakrar, *Multimedia Systems Design*. (Prentice-Hall PTR, Upper Saddle River, 1995)
6. K. Sayood, *Introduction to Data Compression*, 4th edn. (Morgan Kaufmann, San Francisco, 2012)
7. R.L. Freeman, *Reference Manual for Telecommunications Engineering*, 3rd edn. (Wiley, New York, 2001)