II Introduction to Sound Synthesis

5 *History of Digital Sound Synthesis*

Earliest Computer Sounds

Experiments at Bell Telephone Laboratories

Music I and Music II

Music III: The Modular Unit Generator Concept

Music *N* Languages

This chapter ambles through the rich history of computer-generated sound, leading to the sophisticated concept of modular unit generators.

Earliest Computer Sounds

Early digital computers did not have visual displays. In order to better understand their operation, programmers sought to sonify computational processes. For example, when a computer was properly operating, it produced audible radio interference. Technicians found it useful to put a radio near the computer to monitor its operation. When the sound stopped, it indicated that the computer had halted. Programmers soon figured out that the sound of the radio interference correlated with the logic of the program. Specifically, when a program executed a repeating loop at an audio rate, the sound was a sustained pitch. For amusement, they wrote programs containing loops of varying lengths that corresponded to the melodies of popular songs.

Some computers had a loudspeaker that could be used as an output device to signal that a particular event had occurred in a program, such as program termination. On these machines, raw 1-bit pulses on the serial output bus could be sent to a speaker. A periodic loop of pulses interspersed with delays produced a pitched tone that was called a "blurt" or "hoot."

In 1949, Frances E. "Betty" Holberton programmed the BINAC computer to play "For He's a Jolly Good Fellow" to the team who built the machine at the Eckert-Mauchly Computer Corporation in Philadelphia (Irrlichtproject 2015). In 1951, both the Australian CSIRAC and the British Manchester Mark II played popular tunes via this method (Doornbusch 2005; Link 2007). These efforts were never intended as formal research, and no scientific papers were published.

The technology of digital-to-analog converters (DACs) for sound samples did not exist at this time. This meant that there was no possibility for generalized waveform synthesis.

Experiments at Bell Telephone Laboratories

The most general way to control all aspects of a computer-generated waveform is to synthesize it according to the 1928 *theory of sampling* devised by Harold Nyquist, a communications researcher at Bell Telephone Laboratories. The first experiments in the synthesis of sound samples by computer began in 1957 by researchers at Bell Telephone Laboratories in Murray Hill,

New Jersey (David, Mathews, and McDonald 1958; Roads 1980; Wood 1991).

With a doctorate in electrical engineering from MIT, Max V. Mathews (figure 5.1) worked at Bell Labs under the direction of J. R. Pierce. Mathews set out to use a computer to produce musical tones. He wrote a program that would make a computer generate a sequence of binary numbers representing successive amplitudes of a musical sound wave (i.e., samples) (Mathews and Guttman 1959). In their experiments, Mathews and his colleagues proved that a computer could synthesize sounds according to any pitch scale or waveform, including time-varying frequency and amplitude envelopes and polyphony.

Their first programs were written for a giant IBM 704 computer (figure 5.2).

Computers were so rare at that time that the computation had to be carried out at IBM World Headquarters on Madison Avenue and 57th Street in Manhattan. Use of the 704 computer was billed at $600 per hour in 1957 dollars - $6,452 (Johnstone 1994). The logic circuits of the computer were made using vacuum tubes (figure 5.3) The 704 was a powerful machine for its day, with a 36-bit word length and a built-in floating-point processor for fast numerical

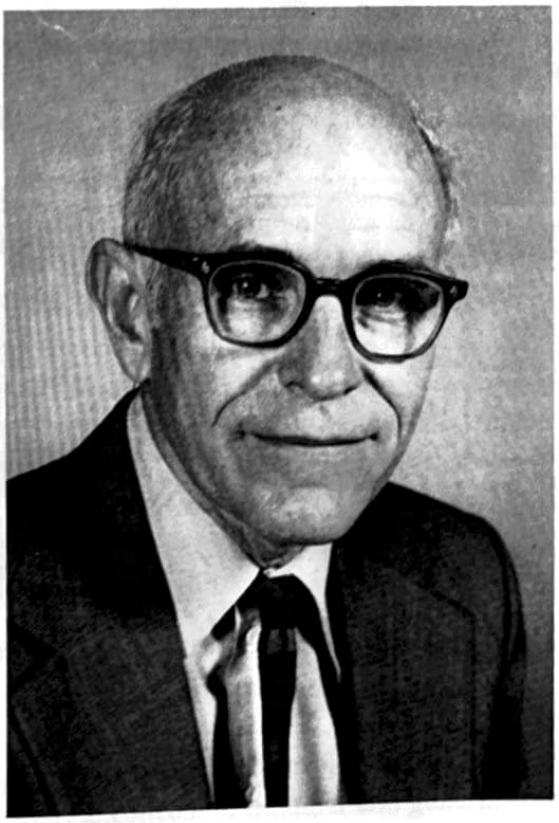


Figure 5.1 Max V. Mathews, 1981.

Figure 5.2 IBM 704 computer, 1957.

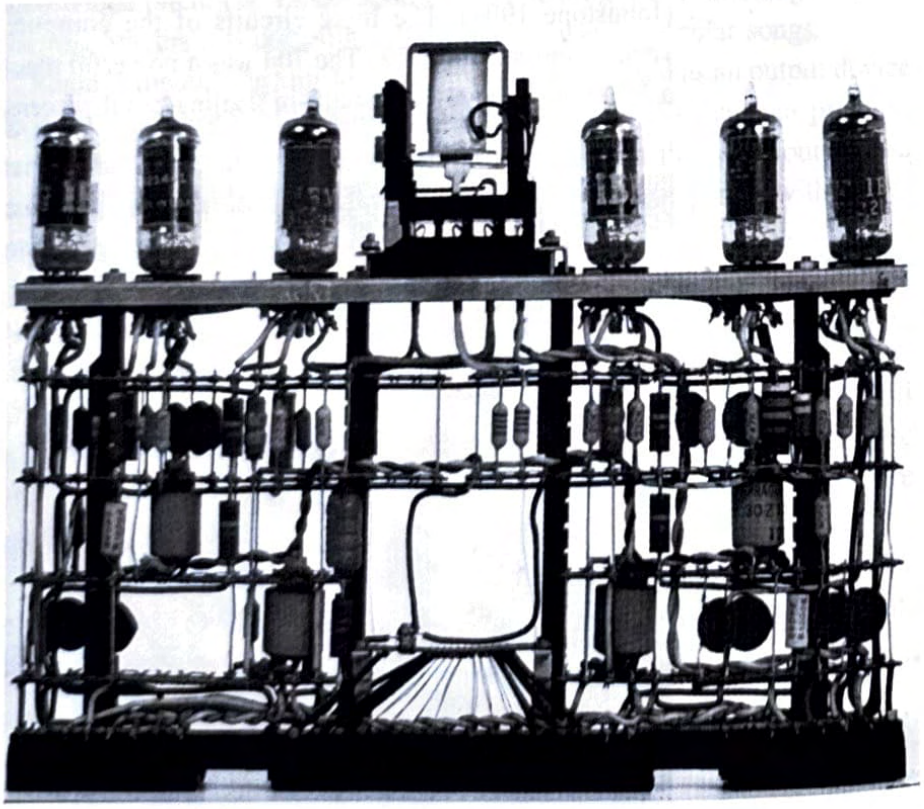


Figure 5.3 Vacuum tube logic module for the IBM 704 computer.

operations. It could be loaded with up to 32 kwords of magnetic core memory and execute up to 4,000 multiplications per second. Sound synthesis calculations took hours. The samples were written to a digital magnetic tape. A 12-bit vacuum tube DAC transformed the samples into sound. This converter, designed by Bernard Gordon, was at that time the only one in the world capable of sound production (Roads 1980).

Music I and Music II

The Music I program developed by Mathews generated a single waveform: an equilateral triangle. A user could specify notes only in terms of pitch and duration (Roads 1980). A perceptual psychology researcher named Newman Guttman made one composition with Music I, a monophonic etude called *In a Silver Scale* written on May 17, 1957 (Guttman 1980). This was the first composition synthesized by the process of digital-to-analog conversion. Recognizing the potential of the computer to generate any frequency precisely, Guttman used the piece as an experiment to contrast two microtonal scales, an equal-beating chromatic scale described by Silver (1957) and just intonation.

Mathews completed Music II in 1958. It was written in assembly language for the IBM 7090 computer, an improved computer along the lines of the IBM 704. The 7090 ran several times faster than the older machines. It was thus possible to implement more ambitious synthesis algorithms. Four independent voices of sound were available, with a choice of sixteen waveforms stored in memory. Music II was used by several researchers at Bell Telephone Laboratories, including Max Mathews, John Pierce, and Newman Guttman.

A concert of the new *computer music* was organized in 1958 in New York City, followed by a discussion panel moderated by John Cage. Later that year Guttman played his computer-synthesized composition *Pitch Variations* at Hermann Scherchen's villa in Gravesano, Switzerland, where composer Iannis Xenakis was in the audience (Guttman 1980). (We encounter Xenakis in chapters 24, 25, and 50.)

Music III: The Modular Unit Generator Concept

The most important development in the design of sound synthesis software was the concept of modular *unit generators* (UGs). UGs are signal processing modules like oscillators, filters, and amplifiers that can be interconnected to

form synthesis *instruments* or *patches* that generate and process sound signals. (In subsequent chapters we discuss UGs in more detail.)

The first synthesis language to make use of the modular unit generator concept was Music III, programmed by Mathews and his colleague Joan E. Miller in 1960. Music III let users design their own synthesis networks out of UGs. By passing the sound signal through a series of such unit generators, a large variety of synthesis algorithms could be implemented relatively easily. The flexible design of Music III also enabled multivoice composition (polyphony), albeit with a corresponding increase in computation time.

Bell Labs published the historic recording *Music From Mathematics* in 1961 (figure 5.4). It featured studies in computer-generated sound by Pierce, Mathews, Guttman, and David Lewin, along with an excerpt of the *Illiac Suite*

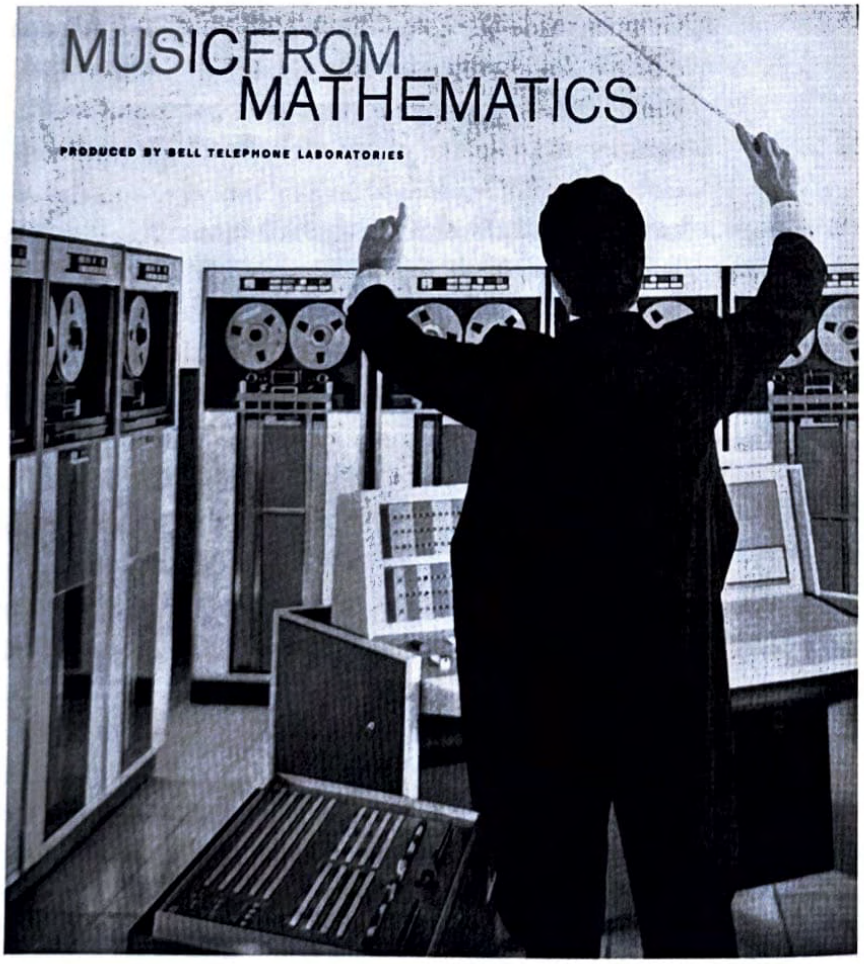
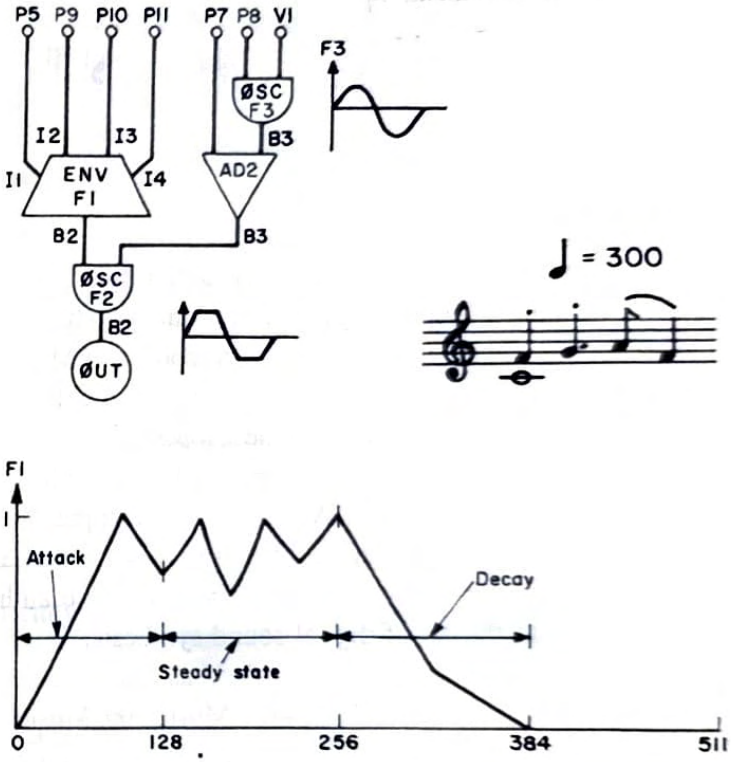


Figure 5.4 Cover of the author's copy of *Music from Mathematics* vinyl record published by Bell Telephone Laboratories in 1961.




```
1  INS 0 4
2  ENV P5 F1 B2 P9 P10 P11 P30 ;
3  ØSC P8 V1 B3 F3 P29 ;
4  AD2 P7 B3 B3 ;
5  ØSC B2 B3 B2 F2 P28 ;
6  ØUT B2 B1 ;
7  END ;
8  GEN 0 1 1 0 0 96 1 128 .7 150 1 175 .6 200 1 225 .7 256 1
     320 .3 384 0 511 0 ;
9  SV2 0 50 .050 .100 ;
10 SV3 0 1 .15 ;
11 NØT 0 4 .1 54 349 ;
12 NØT .2 4 .1 54 392 ;
13 NØT .5 4 .13 54 440 ;
14 NØT .6 4 .2 54 349 ;
15 NØT 0 4 .8 54 262 ;
```

Figure 5.5 Music V synthesis patch. Lines 1–7 define the instrument, which features an envelope generator. The envelope F1 is defined in the GEN statement in line 8. A low-frequency sine wave oscillator modulates vibrato on another oscillator (Mathews 1969).

for String Quartet, an algorithmic composition by Lejaren Hiller (Hiller and Isaacson 1959). Chapter 50 has more on Hiller and algorithmic composition.

Music *N* Languages

Since the time of Music III, a family of software synthesis systems—all based on the unit generator concept—have been developed by various researchers. Music IV was a re-coding of Music III in a new macro assembly language developed at Bell Laboratories called BEFAP (Tenney 1963, 1969).

Music V (figure 5.5), developed in 1968, was the culmination of Max Mathews's efforts in software synthesis (Mathews 1969). Written almost exclusively in FORTRAN IV—a standard computer language at the time—Music V was exported to universities and laboratories around the world in the early 1970s. For many musicians, including the author of this book, it opened a door to the art of digital sound synthesis.

Taking Music IV or Music V as a model, others have developed synthesis programs such as Music 4BF, Music 360, Music 7, Music 11, Csound, MUS10, Cmusic, Common Lisp Music, SuperCollider, ChucK, Synthesis ToolKit, Nyquist, and Max. As a general category these programs are often referred to under the rubric *Music N languages* (see chapter 48).

As the inventor of the modular unit generator synthesis paradigm for generalized waveform synthesis, the late Max V. Mathews (1926–2011) can rightly be called the father of computer-generated sound. Synthesis based on modular graphs of unit generators remains to this day the standard for flexible and experimental synthesis. Music *N* languages deliver this capability to anyone willing to learn to program.

6 *Wavetable Lookup Synthesis*

Wavetable Lookup Synthesis

Changing Waveform Frequency

Algorithm for a Digital Oscillator

Wavetable Lookup Noise and Interpolating Oscillators

Alternatives to Wavetable Lookup

Digital synthesis generates a stream of numbers representing the samples of an arbitrary audio waveform. We can hear these synthetic sounds by sending the samples through a digital-to-analog converter (DAC), which converts the numbers to a continuously varying voltage that can be amplified and sent to a loudspeaker.

Wavetable Lookup Synthesis

A flexible method of waveform synthesis is to scan a prestored wavetable in memory. This process is called *wavetable lookup synthesis.* Wavetable lookup synthesis is the core operation of a *digital oscillator*—a fundamental sound generator.

Let us now walk through the process of table lookup. Suppose that the value of the first sample is given by the first number in the wavetable (shown in figure 6.1 at index location 0). For each new sample to be produced by this simple synthesizer, take the next sample from the wavetable. At the end of the wavetable, simply go back to the beginning and start reading out the samples again. The process is also called *fixed-waveform synthesis* because the waveform does not change over the course of a sound event.

For example, let us assume that the table contains $N = 1{,}000$ entries, each of which is a 16-bit number. The entries are indexed from 0 to 999. We call

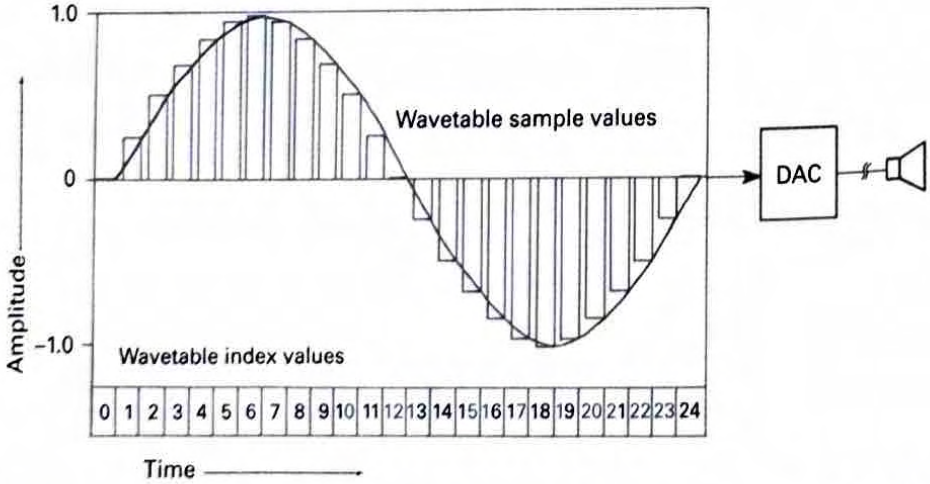


Figure 6.1 Graphical depiction of wavetable lookup synthesis. The list 0–24 in the lower portion contains numbered locations or *table index values.* An audio sample value is stored in memory for each index point. The samples are depicted as the rectangles outlining a sine wave in the top portion. For example, Wavetable[0] = 0, and Wavetable[6] = 1. To synthesize the sine wave, the computer looks up the sample values stored in successive index locations and sends them to a DAC, looping through the table repetitively.

Figure 6.2 The phase increment or *phasor* (a ramp function) goes from 0 to N two times, creating two cycles of the sine wave.

the current location in the table the *phase_index* value, with reference to the phase of the waveform. To read through the table the oscillator starts at the first entry in the table (*phase_index* = 0) and moves by an *increment* to the end of the table (*phase_index* = 999). At this point the phase index wraps around the ending point to the beginning of the wavetable and starts again (figure 6.2).

Changing Waveform Frequency

What is the frequency of the sound produced by table-lookup synthesis? It depends on the length of the wavetable and the sampling frequency. Logically, it should be obvious that if one reads through the entire wavetable (no matter its length) in one second, the result is a wave at 1 Hz; reading through it 100 times a second makes a tone at 100 Hz, and so on.

Let us now be more specific about wavetable length and sampling frequency. If the sampling frequency is 50,000 samples per second and there are 1,000 numbers in the table, the result is 50,000 / 1,000: a low tone at 50 Hz. Likewise, if the sampling frequency is 100,000 Hz and the table contains 1,000 entries, then the output frequency is 100 Hz, because 100,000 / 1,000 = 100.

How is it possible to change the frequency of the output signal? As we have just seen, one simple way is to change the sampling frequency. But this strategy has problems, particularly when one wants to process or mix signals with different sampling rates. A better solution is to scan the wavetable at different rates by skipping samples (to shift pitch up) or repeating samples (to shift pitch down). These processes, in effect, resize the wavetable in order to generate different frequencies.

For example, if we take only the even-numbered samples, then we go through the table twice as fast. This raises the pitch of the output signal by an octave. If we skip two samples, then the pitch is raised further (by an octave and a fifth, to be exact). To shift the pitch down we repeat samples. For example, to shift the pitch down an octave, we play each sample twice.

In the table-lookup algorithm, the *phase increment* determines the number of samples to be skipped or repeated. The increment is added to the current phase location in order to find the next location for reading the value of the sample. In the simplest example, where we read every sample from the table, the increment is 1. If we read only the odd- or even-numbered samples in the table, then the increment is 2.

Algorithm for a Digital Oscillator

We could say that the oscillator *resamples* the wavetable in order to generate different frequencies. That is, it skips or repeats values in the table by an increment added to the current phase location in the wavetable. Thus the most basic oscillator algorithm can be explained as the following two-step program:

1. $phase_index = \text{mod}_L\ (previous_phase + increment)$
2. $output = amplitude \times wavetable[phase_index]$

Step 1 of the algorithm contains an addition and a modulo operation (denoted mod_L). The modulo operation divides the sum by the table length L and keeps only the remainder, which is always less than or equal to L. Step 2 contains a table lookup and a multiplication. This is relatively little computation, but it assumes that the wavetables are already filled with waveform values.

If the table length and the sampling frequency are fixed, as is usually the case, then the frequency of the sound emitted by the oscillator depends on the value of the increment. The relationship between a given frequency and an increment is given by the following equation, which is the most important equation in table lookup synthesis:

$$increment = \frac{L \times frequency}{samplingFrequency} \tag{1}$$

For example, if tablelength L is 1,000, the sampling frequency is 40,000, and the specified *frequency* of the oscillator is 2,000 Hz, then the *increment* is 50.

This implies the following equation for frequency:

$$frequency = \frac{increment \times samplingFrequency}{L} \tag{2}$$

So much for the theory of digital oscillators. Now we confront the computational realities.

Wavetable Lookup Noise and Interpolating Oscillators

All the variables in the previous example were multiples of 1,000, which led to a neat integer result for the value of the phase index increment. However, for most values of the table length, frequency, and sampling frequency in equation 1, the resulting increment is not an integer but rather a real number with a fractional part after the decimal point. However, the way we look up a value in the wavetable is to locate it by its index, which is an integer. Thus we need to somehow derive an integer value from the real-valued increment.

The real value can be *truncated* to yield an integer value for the table index. This means to delete the part of the number to the right of the decimal point, so, for example, 6.99 becomes 6 when it is truncated.

Suppose that we use an increment of 1.125. Table 6.1 compares the calculated versus the truncated increments. The imprecision caused by the truncation means that we obtain a waveform value near, but not precisely the same as, the one that we actually need. As a result, small amounts of waveform distortion are introduced, called *table lookup noise* (Moore 1977; Snell 1977b). Various remedies can reduce this noise. A larger wavetable is one prescription, because a fine-grain table reduces lookup error. Another

Table 6.1 Phase index, table lookup list

Calculated	Truncated
1.000	1
2.125	2
3.250	3
4.375	4
5.500	5
6.625	6
7.750	7
8.875	8
10.000	10
11.125	11
12.250	12
13.375	13
14.500	14
15.625	15
16.750	16
17.875	17
19.000	19

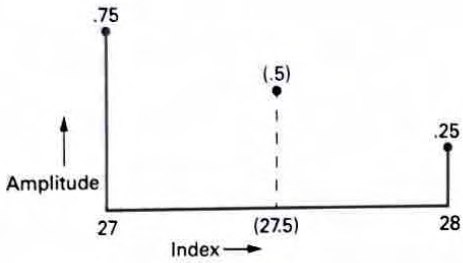


Figure 6.3 Action of an interpolating oscillator. The graph shows two x points in a wavetable, at positions 27 and 28. The oscillator phase increment indicates that the value should be 27.5. The interpolating oscillator generates a y value between 27 and 28 using a linear interpolation algorithm.

way is to *round* the value of increment up or down to the nearest integer instead of simply truncating it; in this case, an increment of 6.99 becomes 7, which is more accurate than 6. But the best performance is achieved by an *interpolating oscillator*. This is more costly from a computational standpoint, but it generates very clean signals.

An interpolating oscillator calculates what the value of the wavetable would have been, if it were possible to reference the wavetable at the exact phase specified by the increment. In other words, it interpolates between the entries in the wavetable to find the one that exactly corresponds to the specified phase index increment (figure 6.3).

With interpolating oscillators, smaller wavetables can yield the same audio quality as a larger noninterpolating oscillator. Consider that for a 1024-entry wavetable used by an interpolating oscillator, the signal-to-noise ratio for a sine wave is an excellent 109 dB (worst case), as compared with the abysmal 48 dB for a noninterpolating oscillator using the same size wavetable (Moore 1977). These figures pertain to the case of linear interpolation; even better results are possible with more elaborate interpolation schemes (Chamberlin 1985; Crochiere and Rabiner 1983; Moore 1977; Snell 1977a; Dannenberg 1998).

Alternatives to Wavetable Lookup

In certain cases, numerical operations are much faster than memory table lookups. Memories tend to be slower than microprocessor instructions. Modern microprocessors have native hardware instructions for sine and cosine generation, exponentials, square roots, and dot products. Thus, as pointed out by James McCartney (1997), waveforms like sine waves, exponentiated sine waves, formant oscillators, and chaotic oscillators can be generated much more efficiently by direct evaluation of their equation. Smith

and Cook (1992) described a highly efficient sinusoidal oscillator based on digital waveguides (see also Smith 2010). Laroche (1998) showed how using resonant filters to synthesize time-varying sinusoids could be an efficient strategy for additive synthesis.

This concludes our introduction to fixed-waveform table-lookup synthesis. The next chapter shows how aspects of synthesis can be varied over time with envelopes.

7 *Time-Varying Waveform Synthesis*

Envelopes, Unit Generators, and Patches

Graphic Notation for Synthesis Instruments

Using Envelopes in Patches

The previous chapter showed how to generate a sine wave at a fixed frequency. Because the maximum value of the sine wave does not change in time, the signal has a constant loudness. This is not terribly useful for musical purposes, because it allows control over only pitch and duration and no control over other sound parameters. Even if the oscillator reads from other wavetables, they repeat ad infinitum. The key to more interesting sounds is *time-varying* waveforms, achieved by changing one or more synthesis parameters over the duration of a sound event.

Envelopes, Unit Generators, and Patches

To create a time-varying waveform, we need a synthesis *instrument* that can be controlled by *envelopes*—functions of time. For example, if the amplitude of the sound changes over its duration, the curve that the amplitude follows is called the *amplitude envelope*. A general way of designing a synthesis instrument is to imagine it as a modular system, containing a number of specialized signal-processing units that together create a time-varying sound.

As chapter 5 pointed out, the unit generator (UG) is a fundamental concept in digital synthesis. A UG is either a signal generator or a signal modifier. A signal generator (such as an oscillator) synthesizes signals such as musical waveforms and envelopes. A signal modifier, such as a filter, takes a signal as its input and transforms that input signal in some way.

To construct an instrument for sound synthesis, one interconnects UGs into a *patch*. The term patch derives from modular synthesizers in which sound modules are connected via *patch cords*. Of course, when the connections are done in software, no physical wires or cables are connected. When a UG produces a number at its output, that number can become the input to another UG.

Graphic Notation for Synthesis Instruments

Now we introduce the graphic notation often used in publications on digital sound synthesis to illustrate patches. This notation was invented to explain the operation of the first modular languages for digital sound synthesis, such as Music 4BF (Howe 1975) and Music V (Mathews 1969) and is still useful.

The symbol for each unit generator has a unique shape. Figure 7.1 shows the graphic notation for a *table-lookup oscillator* called *osc*, a basic signal generator. It accepts three inputs: amplitude, frequency, and a waveform stored in wavetable *f*1. It produces an output signal that is *f*1 repeated at the stipulated frequency and amplitude.

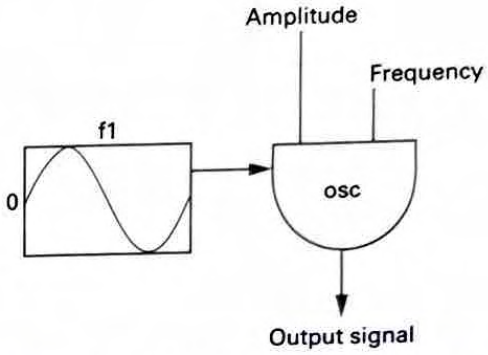


Figure 7.1 Oscillator with sound waveform input *f*1 and fixed parameters for amplitude and frequency.

Using Envelopes in Patches

If we supply a constant number (for example, 1.0) to the amplitude input of an oscillator, then the overall amplitude of the output waveform is constant over the duration of each event. By contrast, most "interesting" sounds have an amplitude envelope that varies as a function of time. Typically, a note starts with an amplitude of 0, works its way up to some maximum value (usually *normalized* to be no greater than 1.0), and dies down again more or less slowly to 0. (A normalized wave has been scaled to fall within standard boundaries such as 0 to 1 for amplitude envelopes, or −1 to +1 for other waves.) The beginning part of the envelope is called the *attack* portion, and the end of the envelope is called the *release*.

Many synthesizers define amplitude envelopes in the following four stages:

Attack (initial increase from zero to peak)

Decay (transition down to a sustained level)

Sustain (constant level)

Release (time taken for level to return to zero).

The usual acronym for such a four-stage envelope is ADSR (figure 7.2). The ADSR concept is useful for describing verbally the overall shape of an envelope; for example, "Make the attack sharper." However, for specifying a musical envelope, four stages is limiting. More flexible envelope editors allow musicians to trace arbitrary curves.

The instrument of figure 7.1 can be easily adapted to generate a time-varying amplitude by hooking up an envelope to the amplitude input of the oscillator. We are now closer to controlling the oscillator in musical terms. If we set the duration and the curve of the envelope, then the envelope controls the amplitude of each note.

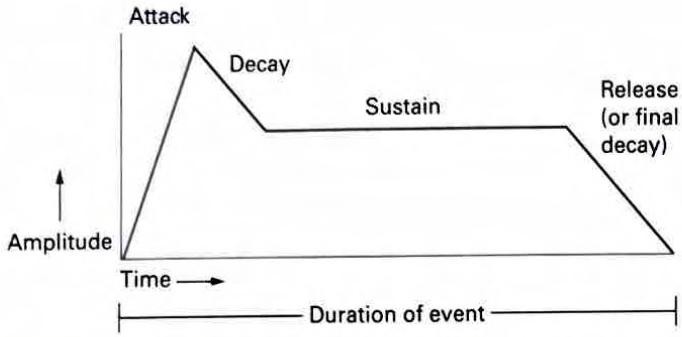


Figure 7.2 Attack, decay, sustain, release (ADSR) amplitude envelope.

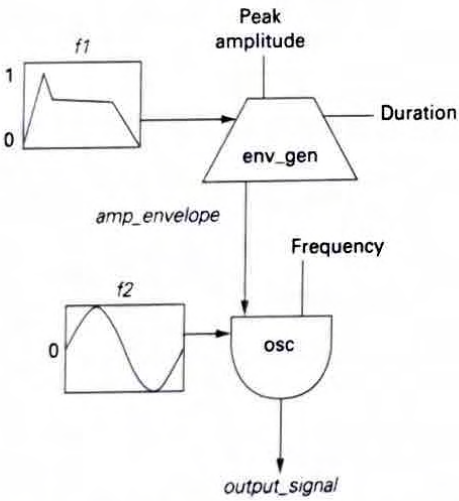


Figure 7.3 Oscillator with amplitude envelope $f1$ and sound waveform $f2$.

Designing an envelope for every event in a composition is tedious. What we seek is a simple procedure for generating an envelope that can scale itself to the duration of diverse events. Figure 7.3 shows an *envelope generator, env_gen*. This UG env_gen takes in a duration, a peak amplitude, and a wavetable. It reads through the wavetable $f1$ over the specified duration, scaling it by the peak amplitude. It is adaptable to tones of any duration. The sound waveform is $f2$ in this case.

As the reader might guess, we could also attach an envelope generator to the frequency input of osc to obtain a pitch change such as vibrato or glissando. Indeed, we can interconnect envelopes, oscillators, and other unit generators in a wide variety of ways in order to make different sounds. This is the modular approach to sound synthesis.

Human beings are naturally sensitive to undulation, so it is not surprising that time-varying envelopes are the key to interesting sounds. Envelopes are profiles of gestures. As analogies of human body movements, they infuse life into otherwise lifeless electronic signals.

8 *Software Synthesis*

What Is Software Synthesis?

Early Software Synthesis

Types of Software Synthesizers

- **Closed Apps**
- **Patchable Apps**
- **Virtual Modular Synthesizers**
- **Graphical Instrument Patch Editors**
- **Textual Synthesis Languages**
- **Custom Apps Made Using General-Purpose Programming Languages**

Real-Time and Non-Real-Time Synthesis

- **Non-Real-Time Synthesis**
- **Sound Files**
- **Real-Time Digital Synthesis**

Audio Programming

This chapter begins by comparing hardware versus software synthesis. We briefly review the history of software synthesis, review various types of software synthesizers, look at real-time and non-real-time synthesis, and introduce resources for audio programming.

What Is Software Synthesis?

In the early days of digital synthesis, computers were too slow for *real-time* operation. What do we mean by *real time*? In this context, real time means that we can complete all calculations for a sample within the duration of one sample period. If a system takes ten seconds to compute one second of sound, it must operate as a *non-real-time system.* This means that the sound output must be written to a file to be auditioned later. Early computer music systems could not be played interactively in the presence of sound—a huge liability from a musical standpoint.

Thus there was a strong motivation to design specialized digital hardware to synthesize sound in real time (Markowitz 1989; Alonso 1973; Alles and Di Giugno 1977; Snell 1977a, 1977b; Samson 1980, 1985; Asta, et al. 1980; Wallraff 1979a; Loy 2013a, 2013b). Early hardware synthesizers were rare and expensive. A breakthrough product was the Yamaha DX7 (1983), a mass-produced keyboard instrument based on specialized chips for *frequency modulation* (FM) synthesis (chapter 16). Selling for less than $2,000, over 200,000 units were sold. Much pop music of the 1980s was infused with the Yamaha FM sound.

Specialized hardware can be efficient but is limited in its flexibility. For example, the chips in the DX7 were designed for FM; they could not realize other synthesis methods. Thus, the most flexible approach to sound generation is a *software synthesis* program running on a computer. In software synthesis, all the calculations involved in computing a stream of samples are carried out by a computer without additional hardware.

Early Software Synthesis

An early example of software synthesis was the venerable Music V language (Mathews 1969), shown in figure 8.1. To create sound, one would write a program in the Music V language and then execute the program. This non-real-time process could take hours (Roads 2001a).

Figure 8.2 is a simple example of a Music V program. Figure 8.2(a) shows a conventional score of the notes to be synthesized. Figure 8.2(b) shows the

```
  264  J4=L1+J3-1
       IN1=I(J4)
  265  J5=L2+J3-1
       I(J5)=IN1+I(J5)
       IF(I21.EQ.0) GO TO 270
       ISAM=I(J5)
       IF(ISAM.GT.ICHECK.OR.ISAM.LT.-ICHECK) GO TO 90
  270  CONTINUE
       IF(TIME(2).GE.I(28)) RETURN 1
       RETURN
   90  CALL PAGE(1)
       ISAM=ISAM*SFI
       PRINT 69,ISAM,NSAM,T
   69  FORMAT(' SAMPLE OUT OF RANGE HAD VALUE',I8,' SAMPLE WAS IN BLOCK
      -F',I4,' SAMPLES PRECEDING ACTION TIME',F11.4)
       GO TO 270
C      OSCILLATOR
C      THIS OSCILLATOR ACCEPTS NEGATIVE FREQUENCY INPUTS
  102  SUM=FLOAT(I(L5))*SFI
       IF(M1)280,280,281
  280  AMP=FLOAT(I(L1))*SFI
  281  IF(M2)282,282,283
  282  FREQ=FLOAT(I(L2))*SFI
  283  CONTINUE
       LIM=L4+511
       DO293J3=1,NSAM
       J4=INT(SUM+.5)+L4
       IF(J4.LT.L4.OR.J4.GT.LIM) GO TO 295
       F=FLOAT(I(J4))
       IF(M2)285,285,286
  285  SUM=SUM+FREQ
       GOTO290
  286  J4=L2+J3-1
       SUM=SUM+FLOAT(I(J4))*SFI
   290 IF(SUM.LT.0.) SUM=SUM+XNFUN
       IF(SUM-XNFUN)288,287,287
  287  SUM=SUM-XNFUN
  288  J5=L3+J3-1
       IF(M1)291,291,292
  291  I(J5)=IFIX(AMP*F*SFXX)
       GOTO293
  292  J6=L1+J3-1
       I(J5)=IFIX(FLOAT(I(J6))*F*SFF)
  293  CONTINUE
       I(L5)=IFIX(SUM*SFID)
```

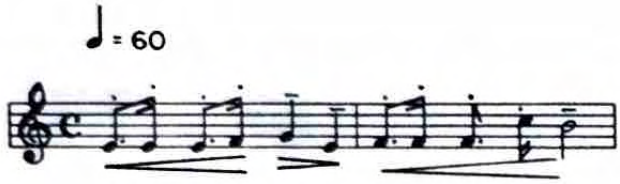
Figure 8.1 Excerpt of the author's printout of the source code of Music V from 1974. The code is classic FORTRAN IV, with program control flow by GOTO statements. The first 16 lines are the end of the OUT unit generator. The rest is the beginning of the OSCIL unit generator. The code for a more modern version of Music V for GFortran and Linux is printed in Boulanger and Lazzarini (2011). GFortran continues to be maintained.

patch for a simple instrument that will play the score. It consists solely of an oscillator and an output box. The oscillator has two inputs: the amplitude of the output is set by the fifth element in the note statement, designated P5. The waveform to be played is determined by the function F2, which is sketched in figure 8.2c.

By the mid-1970s, music synthesis languages could be run on minicomputers. Waiting times for synthesis jobs were reduced to minutes for most short-duration experiments and hours for intensive tasks like synthesizing entire pieces involving reverberation.

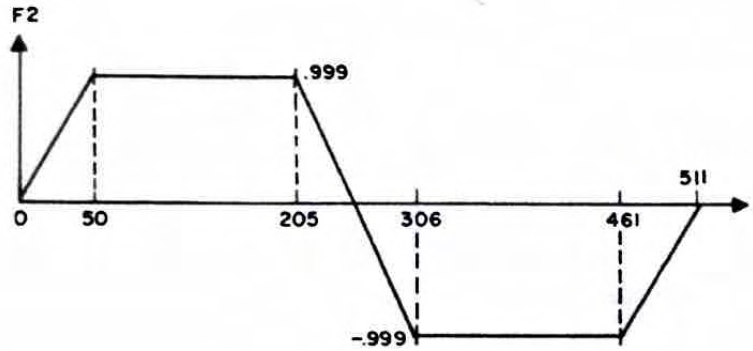
By the late 1980s, we began to see the first real-time synthesis software such as Csound (1990), SuperCollider (1996), Seer Reality (1997), and Max/MSP (1997) running on home computers.

Today a multitude of real-time software synthesizers are available. To cite one example, any user of the Native Instruments REAKTOR modular system



(a)

(b)



(c)

```
 1  INS 0 1 ;
 2  ØSC P5 P6 B2 F2 P30 ;
 3  ØUT B2 B1 ;
 4  END ;
 5  GEN 0 1 2 0 0 .999 50 .999 205 -.999 306 -.999 461 0 511 ;
 6  NØT 0 1 .50 125 8.45 ;
 7  NØT .75 1 .17 250 8.45 ;
 8  NØT 1.00 1 .50 500 8.45 ;
 9  NØT 1.75 1 .17 1000 8.93 ;
10  NØT 2.00 1 .95 2000 10.04 ;
11  NØT 3.00 1 .95 1000 8.45 ;
12  NØT 4.00 1 .50 500 8.93 ;
13  NØT 4.75 1 .17 500 8.93 ;
14  NØT 5.00 1 .50 700 8.93 ;
15  NØT 5.75 1 .17 1000 13.39 ;
16  NØT 6.00 1 1.95 2000 12.65 ;
17  TER 8.00 ;
```

Figure 8.2 A simple orchestra and score in Music V (after Mathews 1969). Line numbers 1–17 have been added to simplify the explanation. Lines 1–4 in the code define instrument number 1 (an oscillator and an output). Line 5 generates the function F2. Lines 6–16 list note statements defining the score. Each note statement gives starting time, instrument number, duration, amplitude, and pitch (encoded in this case). Line 17 terminates the program.

can design and publish a synthesizer, so that thousands of synthesizers are available for this platform alone.

Types of Software Synthesizers

Contemporary software synthesis can be divided into six general categories:

- Closed apps
- Patchable apps
- Virtual modular synthesizers
- Graphical instrument patch editors
- Textual synthesis languages
- Custom apps made using general-purpose programming languages

Chapter 19 looks at virtual modular synthesizers. Chapter 47 surveys graphical instrument patch editors. Chapter 48 addresses textual synthesis languages.

Closed Apps

Closed apps are designed to do one thing very well. They are optimized to realize a single synthesis technique or only a few variations. Their graphical interface reflects this specialization. They cannot be freely reprogrammed to realize any synthesis technique. Figure 8.3 shows the Native Instruments FM8, which emulates a Yamaha DX7 synthesizer. Even a closed app like FM8 provides some flexibility in patching. Its Expert page lets users design their own FM patches.

Many commercial software synthesizers can run either in *standalone* mode (i.e., with no other apps running), or as a *plug-in* extension to a *digital audio workstation* (DAW) or sound editor. The plug-in implementation has the advantage that multiple copies or *instances* of the plug-ins can run simultaneously, and they can be recorded by the DAW as they play.

Patchable Apps

Patchable apps offer a limited set of modules with the possibility of patching the modules to realize different synthesis techniques.

An example of a patchable app is Native Instruments Absynth, which features a *semimodular* design with twelve module slots (figure 8.4). One can choose among various types of oscillators, filters, and modulators to fill in the open slots.

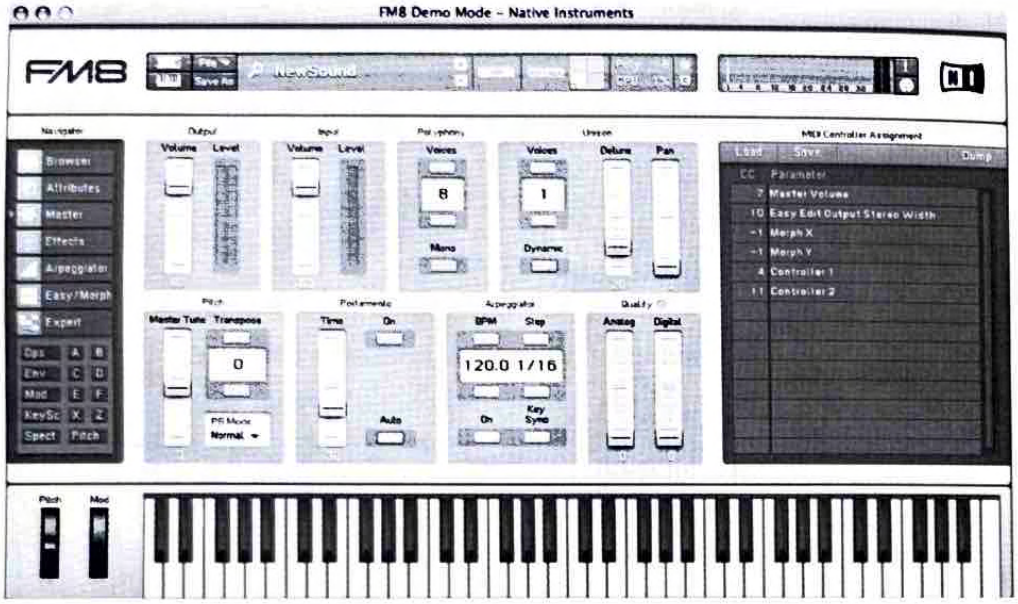


Figure 8.3 Native Instruments FM8. This software synthesizer is optimized for frequency modulation synthesis.

Figure 8.4 Absynth patch window.

Another is Madrona Labs Aalto, a self-styled West Coast synthesizer following the example of the late Don Buchla. Like a Buchla system, it couples a complex oscillator with a lowpass gate. It also features two envelope generators, a waveguide delay, a filter, and a sequencer. Although its collection of modules is fixed, it offers more than two dozen patch points for interconnecting the modules (figure 8.5). According to the developer, it is by playing with the interconnections that unexpected sounds emerge (Jones 2020).

Unfiltered Audio's versatile LION synthesizer (figure 8.6) goes further to offer an expandable number of virtual modulation sources that can be patched to control synthesis parameters.

Virtual Modular Synthesizers

The first wave of virtual modular software synthesizers were simulations of vintage analog synths. Arturia worked with Robert Moog to introduce the Modular V synthesizer in 2003. Since that time, emulations of vintage synths by Arp, EMS, and Buchla have appeared. Figure 8.7 shows the screen of Arturia's EMS Synthi V software synthesizer.

New virtual modular synthesizers include Native Instruments REAKTOR Blocks, VCV Rack, SoftTube Modular, and Cherry Audio's Voltage Modular (see chapter 19).

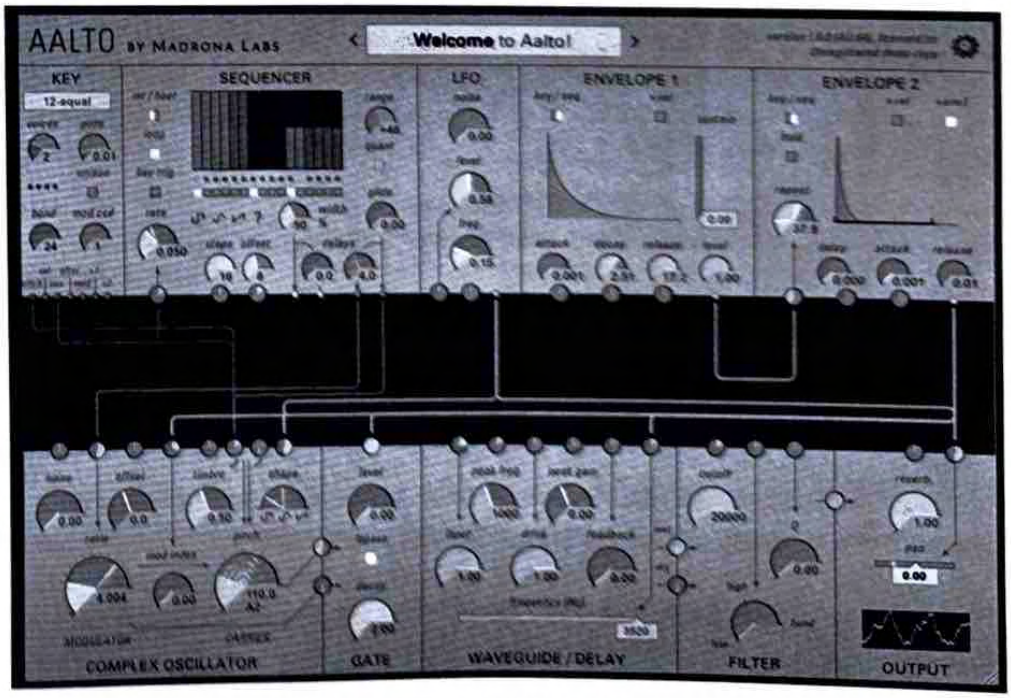


Figure 8.5 Madrona Labs Aalto, a patchable software synthesizer.

Figure 8.6 Unfiltered Audio LION, a patchable software synthesizer.

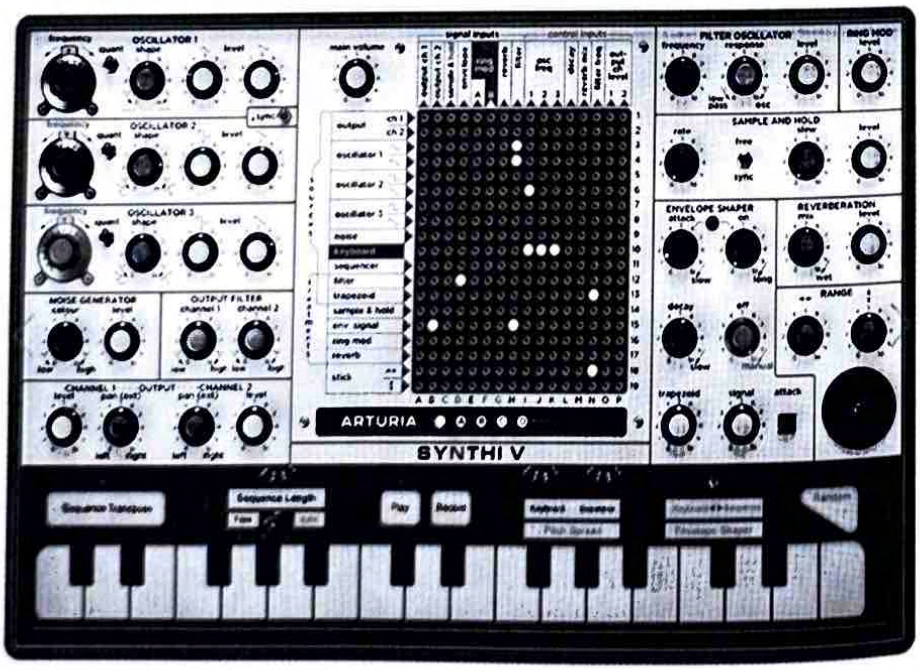


Figure 8.7 Arturia EMS Synthi V software synthesizer.

Graphical Instrument Patch Editors

A graphical instrument editor such as Max, PureData (PD), or AudioMulch lets one build synthesis patches by interconnecting icons on a screen (figure 8.8). The icons represent unit generators. The difference between what we have called a patchable app and a graphical instrument editor is that in the editor one can add any number of modules into a patch. Moreover, in a system like Max, there are hundreds of modules from which to choose. Once the editing is done, the result can be compiled to make a self-contained app.

Textual Synthesis Languages

Using a language designed for synthesis, a musician can specify sounds by writing a text that is interpreted by a synthesis engine. This category is represented by languages such as Csound, Nyquist, ChucK, Faust, and SuperCollider. Figure 8.9 shows a simple FM synthesis instrument written in SuperCollider 3.

Using a programming language like SuperCollider, one can compile new fully-functioning apps. For example, PulsarGenerator (Roads 2001a, 2001b) was compiled in SuperCollider. Our granulation app EmissionControl was originally coded in a combination of SuperCollider and C (Thall 2005).

Custom Apps Made Using General-Purpose Programming Languages

For the ultimate in customization and efficiency, one can program in a general-purpose language like C++. Most of the software synthesizers on the

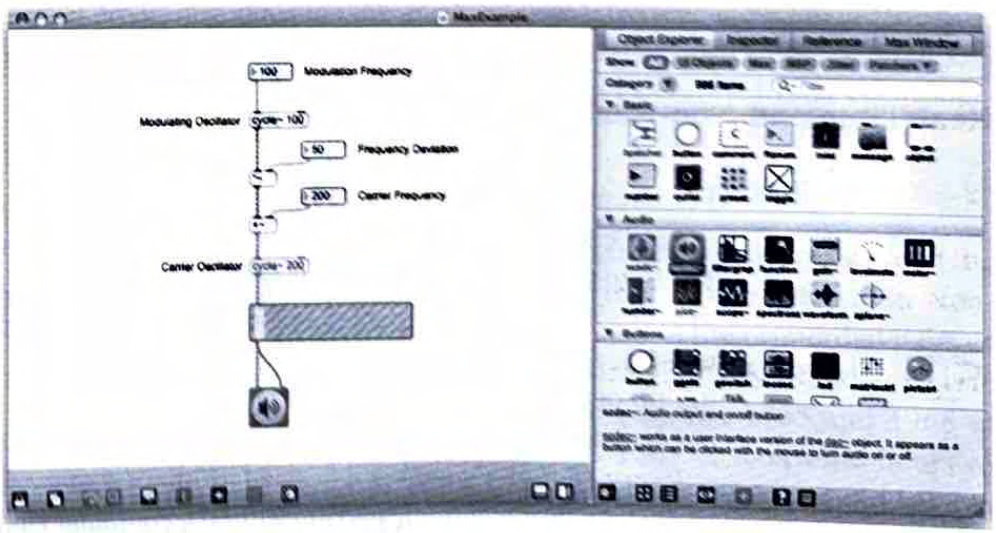


Figure 8.8 Max patch for frequency modulation.

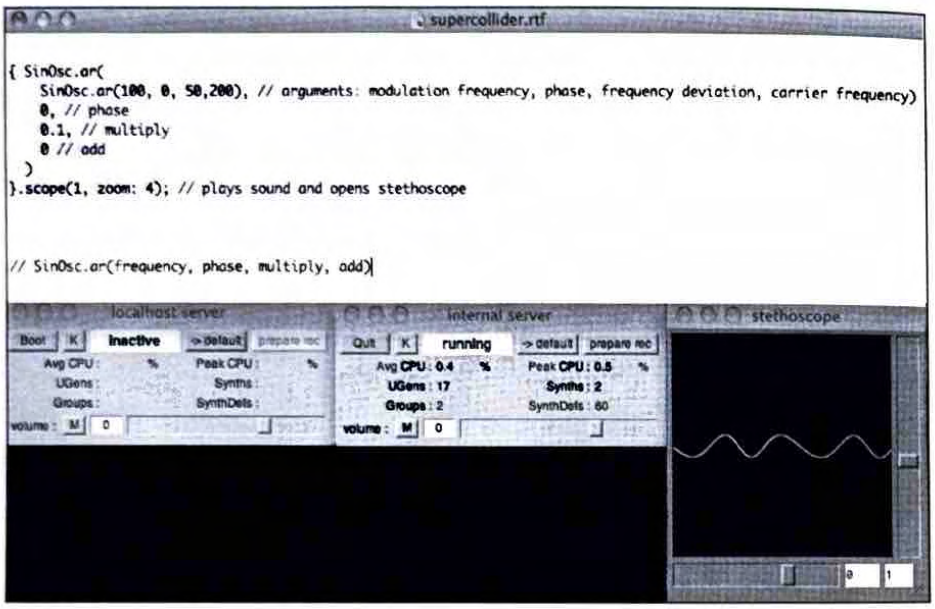


Figure 8.9 SuperCollider code for frequency modulation synthesis.

market are coded in C++. *Code libraries* are vital to programming audio applications because they provide solutions to many basic tasks that are common to these applications. Code libraries facilitate graphical user interface (GUI) design (e.g., ImGui and JUCE), audio input and output (e.g., PortAudio and RtAudio), MIDI control (e.g., RtMidi), signal processing algorithms (e.g., Pedal and STK), and music information retrieval using machine learning (e.g., FluCoMa Project). Our EmissionControl2 granular synthesis app was coded in C++ using the Allolib code library, which was designed to support immersive 3D audio and visuals.

Real-Time and Non-Real-Time Synthesis

For decades, Moore's law (1965) predicted that transistor density would double each year, with a followup effect on clock speed. By 2012, this law began to lose relevance as the clock speed of microprocessors stalled at less than 4 GHz. Chip companies responded by manufacturing *multicore* processors, so that multiple processes could run in parallel, theoretically speeding up operations. However, many real-time interactive audio algorithms do not benefit from *multithreaded* (parallel) processing.

Every step in a sound synthesis algorithm takes a certain amount of time to execute. For a complicated algorithm, a computer cannot always complete the calculations necessary for a sample in the interval of one sample period.

Moreover, the audio thread is competing with other operating system processes.

To understand this point more concretely, consider the following steps required to calculate one sample of sound by the table-lookup method of synthesis:

1. Add increment to current wavetable lookup location to obtain new location.
2. If the new location is past the end of the wavetable, subtract the wavetable length. (In other words, perform a modulo operation.)
3. Store the new location for use in calculating the next sample. (See step 1.)
4. Look up the value in the wavetable at the new location
5. Multiply that value by the amplitude input.
6. Send the product to the output.

The important point here is that each step takes time. For example, let us say it takes a computer one microsecond (one millionth of a second) to perform the six calculations in the preceding list. (This is intentionally slow in order to make the explanation simple; modern computers are faster than this.) If we use a sampling rate of 50 kHz, the time available per sample is 1 / 50,000th of a second or 20 microseconds. This means that in an ideal world, the computer could complete the calculations necessary for about twenty simple oscillators in real time.

In the real world, the process is made more complicated by using interpolation in the table lookup; increasing the sampling rate; adding filters, delays, modulating oscillators, random functions, more channels, reverberation, and spatial processing; updating the graphical user interface; and allowing the time needed to interact with a musician. Here the calculations can become impossible to realize in real time. This is the domain of non-real-time synthesis.

Non-Real-Time Synthesis

Non-real-time operation means that there is a substantial delay between the time we start computing a sound and the time that we can listen to it. Non-real-time (or *offline*) synthesis was the only option in the primordial days of computer music. For example, a two-minute portion of J. K. Randall's *Lyric Variations for Violin and Computer*, realized between 1965 and 1968 at Princeton University (Cardinal Records VCS 10057), took nine hours to compute. If a small adjustment was desired, the entire process would have to be repeated. Even though this was a laborious process, a handful of dedicated

composers were able to create computer-synthesized works of music (Tenney 1969; Von Foerster and Beauchamp 1969; Dodge 1985; Risset 1985a).

For a more modern example, the method of atomic decomposition analysis, described in chapter 39, is currently a non-real-time technique. However, it is possible that someone with a powerful multicore computer could substantially increase the speed of this technique by programming it as a multithreaded process.

Many audio computations operate faster than real time. For example, a complex multitrack mix of a piece lasting several minutes can be rendered to a sound file in seconds using a digital audio workstation like Pro Tools, Logic, or Ableton Live.

Sound Files

Whether or not it takes longer than one sample period to compute each sample, software synthesis programs can generate a *sound file* as their output. Common sound file formats include Waveform Audio File Format (WAVE) and Audio Interchange File Format (AIFF), among many others. A sound file is simply an audio data file stored on a digital storage medium. A sound file contains a *header text* and numbers representing sound samples. The header contains the name of the file and relevant information about the samples in the file (sampling rate, number of bits per sample, number of channels, etc.).

As in other computer applications, many different audio file formats coexist. The need to convert between formats is a practical fact of life in computer music studios. Sound editors and utilities like SoundHack (Erbe 1992) perform these conversions. Lazzarini (2011c) teaches how to read and write sound files.

Real-Time Digital Synthesis

The 1970s saw the introduction of the first real-time digital synthesizers based on custom hardware (Markowitz 1989; Alonso 1973; Alles and Di Giugno 1977; Snell 1977a, 1977b; Samson 1980, 1985; Asta, et al. 1980; Wallraff 1979a; Loy 1981, 2013a, 2013b; Alles 1977; Buxton et al. 1978; Strawn 1985c; Roads and Strawn 1985; Roads 1989).

By the 1980s, large circuit boards inside digital synthesizers were being replaced by tiny chips that could realize multivoice synthesis algorithms in real time. These chips could be fabricated en masse and embedded in inexpensive synthesizers made by manufacturers such as Yamaha. In the 1990s, similar chips were embedded in *sound cards* inserted into computers. As processors became faster, for a while it looked as if software synthesis would rule forever. Then the Eurorack phenomenon happened,

and hardware synthesis (both analog and digital) made a return alongside software synthesis.

Figure 8.10 shows an overview of a real-time computer music synthesis system. This system has three ways of generating digital sound:

1. Non-real-time software synthesis using a language like Csound. The sounds are calculated on the computer and written to a sound file.
2. Real-time software synthesis calculated on the computer (e.g., Max), possibly controlled by an input device that speaks MIDI or OSC.
3. Real-time synthesis using a synthesizer controlled by an input device that speaks MIDI or OSC.

The obvious advantage of real-time operation is that musical input devices or controllers can be played by the musician as sound is heard. Gestural control leads to more expressive performances. Moreover, the parameter space of a synthesis method can be explored and tested rapidly. *Sequencers* and *score editors* make it possible to record and edit performances. Chapters 40 and 41 present performance controllers and performance software, and chapters 51 and 52 cover the MIDI and OSC communication protocols.

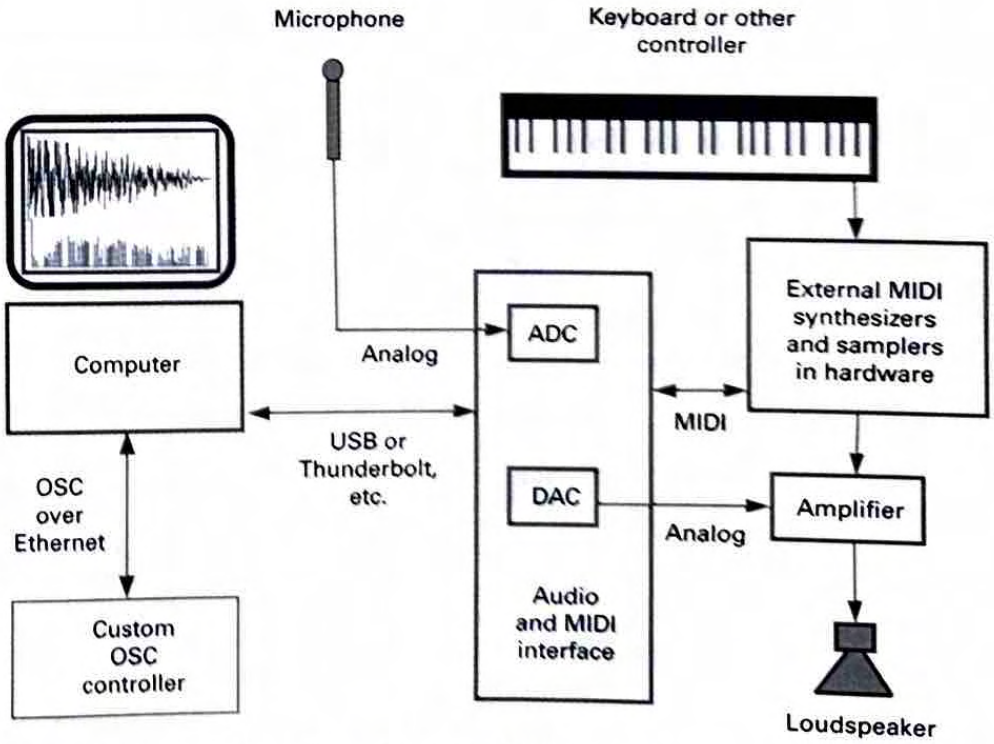


Figure 8.10 Overview of a real-time synthesis system.

Audio Programming

This section provides a brief orientation to audio programming. Many tomes address audio programming in depth and detail. For example, *The Audio Programming Book* (Boulanger and Lazzarini 2011) contains over 3,000 pages of text and thousands of lines of code. Specific books describe each audio programming language (SuperCollider, Csound, Max, Pd, ChucK, FAUST, etc.). Neukom's text (2013) uses Csound, Max, Mathematica, C/C++, and Processing. Another resource is The Audio Programmer video channel on YouTube.

A crucial element of audio programming is the management of audio I/O. Music-specific environments like SuperCollider and Max handle this automatically. By contrast, when coding in a language like C or C++, the programmer has more responsibility for ensuring the smooth stream of audio samples without stuttering. Fortunately, audio libraries such as libsndfile, Portaudio, and JUCE handle many of the low-level details of real-time input and output. In the audio software industry, ROLI's JUCE is favored by many professional developers (Jones 2020).

Real-time synthesis generates a *block* of samples at a time. Typical block sizes range from 32 samples to 2,048 samples. Large buffers provide the most protection against glitches, but they introduce *latency*: a delay from the time that the sound is recorded or synthesized to the time that it is heard.

```
# Set up variables and arrays when the program starts
ON INITIALIZE:
CREATE ARRAY "Wavetable" and FILL with [arbitrary waveform]
CREATE INTEGER "TableIndex" and SET to 0
CREATE INTEGER "BufferSize" and SET to 1024
CREATE ARRAY "BufferOut" and SET size to BufferSize
CREATE ARRAY "BufferHolding" and SET size to BufferSize
END INITIALIZE

# Audio Callback
IF (DAC is asking for a new Buffer) THEN:
      COPY BufferHolding to BufferOut
      SEND BufferOut to DAC
      # fill HoldingBuffer with samples from the wavetable
      FOR BufferIndex = 0 to BufferSize:
            INCREMENT TableIndex by 1
            # Wrap around to the beginning of the wavetable
            IF TableIndex >= Wavetable.Length THEN:
                  SUBTRACT Wavetable.Length from TableIndex
            ENDIF
           SET BufferHolding(BufferIndex) to Wavetable(TableIndex)
      ENDFOR
ENDF
```

Figure 8.11 Audio callback loop that reads a wavetable and outputs audio to the DAC. Credit to Rodney Duplessis.

The mechanism for implementing low-latency audio is the *audio callback loop,* which repeats as long as sound is being recorded or synthesized. The audio callback loop takes a block of samples from a synthesizer and puts them into a memory buffer. The block of samples plays by sending the contents of the buffer to the real-time output. However, synthesizing or applying effects to a block of samples may take longer than one sample period. Moreover, the processing speed of a software routine is not constant because it runs concurrently with operating system processes that can start at any time. Hence a scheme using a single output buffer is prone to audio dropouts. Thus it is common practice to use a *double-* or *quad-buffering* scheme in the audio callback loop. One might break a 2,048-sample buffer into two or four buffer sections. One buffer plays while the other buffer is being filled with new samples by the synthesis algorithm. In some cases buffering can be automatically managed by the sound library. Maldonado (2011) walks readers through the steps of the process. The developer of AudioMulch, Ross Bencina (2011) warns of problems in the audio callback loop and provides a list of do's and don'ts to avoid audio glitches. Figure 8.11 shows an example of an audio callback loop written in pseudocode.

Lazzarini (2011d) presents the C++ code for the basic building blocks of a synthesizer engine: interpolating oscillators, envelope generators, filters, delay lines, flangers, convolution, and pitch shifters. The same author has written a two-volume set, *Computer Music Instruments I* and *II*, with detailed examples in Python, Csound, and Faust (Lazzarini 2017, 2019)

The spread of software synthesizers has been fostered by plug-in standards. Steinberg's Virtual Studio Technology (VST) format, introduced in 1996, continues as a broadly supported protocol. Other formats include Apple's Audio Units (AU) and LV2 for Linux, among others. For tutorials on developing audio plugins, refer to Goudard and Muller (2003), Dobson (2011), and Pirkle (2019).

III Sound Synthesis

9 *Sampling*

Sampling: Background

Pierre Schaeffer and Musique Concrète
Early Sampling Instruments

Looping

Pitch Shifting

Sample-Rate Conversion without Pitch Shifting

Problems in Resampling

Sample and Wavetable Libraries

Emulating Traditional Instruments

Modeling Note-to-Note Transitions

Synthesis Methods That Model Note-to-Note Transitions

In popular parlance, *sampling* means making a digital recording of a short sound object like a one-bar drum pattern. The sample is then triggered by a MIDI note statement played on a controller or a sequencer. In the past, samples had to be short because of extremely limited memory space. Today, for all practical purposes, these limitations are gone.

The term *sampling* derives from established notions of digital *samples* and *sampling rate* explained in chapter 3. Sampling instruments are widely available in both hardware and software. All samplers are designed around the basic notion of playing back prerecorded sounds, shifted to a desired pitch.

Because they can mimic the sounds of acoustic instruments, samplers are among the most popular electronic instruments available. But emulation of a known instrument only scratches the surface of the potential of sampling, which can manipulate any sound. The prospect of live sampling onstage introduces another set of possibilities. The California artist Pamela Z samples and loops her voice in complex layers. For the musician Earl Howard, who has played with the New York Philharmonic, a sampler can be used in real time to record the sounds of other instrumentalists in an ensemble and play them back in radically transformed ways. Ultimately, the extraordinary power of a sampler to play back and pitch shift any sound in real time, whether real or synthetic, makes it an essential tool in any electronic music studio (figure 9.1).

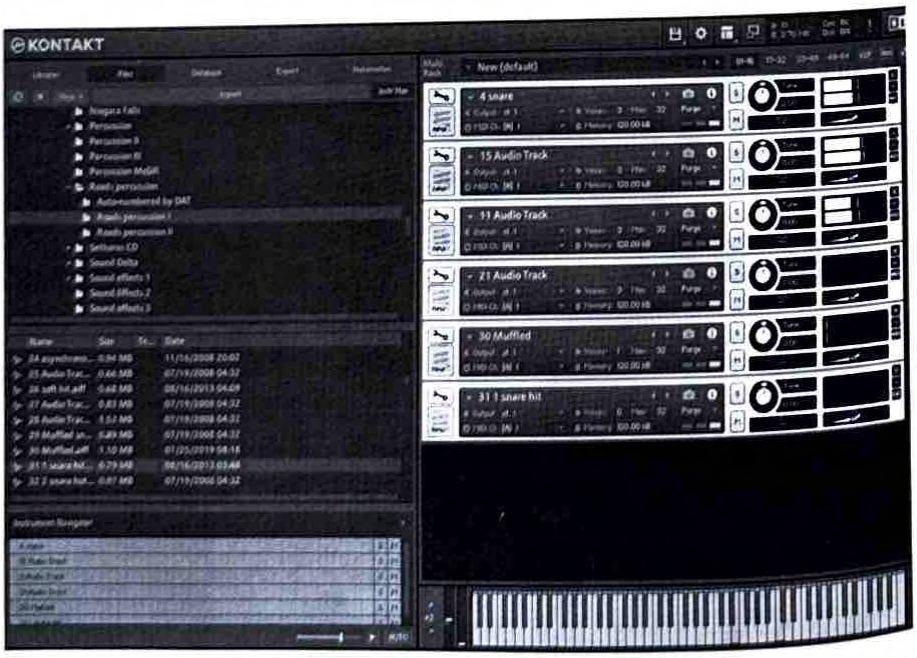


Figure 9.1 Native Instrument Kontakt, a popular software sampler. In this simple setup, six sound files are loaded in the sampler (right side). All six are triggered simultaneously by a MIDI keyboard.

Sampling is different from the classical technique of fixed-waveform synthesis explained in chapter 6. Instead of scanning a small, fixed wavetable containing one cycle of a waveform, a sampler scans a large wavetable that contains thousands of individual cycles—usually a second or more of pre-recorded sound. Because the sampled waveform changes over the attack, sustain, and decay portion of the event, the result is a rich and time-varying sound.

Sampling: Background

Composed manipulation of recorded sounds dates back to the 1920s. (See Davies [1996] for a colorful historical account.) Composers such as Darius Milhaud, Paul Hindemith, and Ernst Toch experimented with variable-speed phonographs in concert (Ernst 1977). The 1920s saw the invention of sound recording on optical film (sound photography) and the possibility of creating *sound libraries.* The musical potential of sample libraries was recognized by the Mexican composer Carlos Chavez in 1937 (89–103):

Sound photography . . . has an enormous interest for composers in that it offers new and very ample musical resource . . . Libraries of sound, in which thousands and thousands of noises, murmurings of nature and man in thousands of different places and times, have been preserved . . . Here at hand, ready to be used in our creations, are all the sound elements possible or imaginable.

The composer Halim El-Dabh created *Expressions of Zaar* (1944) from samples of an Egyptian ceremony recorded on a wire recorder, a primitive recording device. El-Dabh recognized that the low fidelity of the original recordings could be exploited artistically. He processed the sounds through echo chambers and other devices to catapult them beyond the literal and into the realm of sonic abstraction—like much electronic music today.

Magnetic tape recording, originally developed in Germany in the 1930s, is a high-fidelity recording medium when a number of technical requirements are met. Moreover, the tape medium permits cutting and splicing and therefore flexible editing and rearrangement of sequences of recorded sounds. Tape recorders became available to musicians only after World War II.

Pierre Schaeffer and Musique Concrète

After experiments with variable-speed phonographs in the late 1940s, Pierre Schaeffer founded the Studio de Musique Concrète at Paris in 1950 (figure 9.2).

Figure 9.2 Pierre Schaeffer's studio for musique concrète at rue de l'Université, Paris, 1960. The studio features three tape recorders (left) along with a disk turntable. On the right is another tape recorder and the multiple-head Phonogène device (see figure 9.3). (Photograph courtesy of the Groupe de Recherches Musicales, Paris.)

He and Pierre Henry began to use tape recorders to record and manipulate *concrète* sounds. *Musique concrète* refers to the use of microphone-recorded sounds or samples, rather than synthetically generated tones-as in pure electronic music.

The musique concrète aesthetic also refers to the manner of working directly in the presence of sound, as opposed to silently writing notes in common music notation (Schaeffer and Moles 1952; Schaeffer 1977, 2017; Chion 1982). In order to describe the vastly expanded range of sonic materials Schaeffer introduced the notion of *sound objects*. A sound object is a generalization of the note concept. It refers to any sound of a duration

between about one-tenth of a second and eight seconds (i.e., on the same time scale as musical notes) that can be used as a unit of composition (Roads 2001b).

An outstanding example of musique concrète is *De Natura Sonorum* (1975) by Bernard Parmegiani. Luc Ferrari took the concept of musique concrète in the direction of what is now called *ambient*, *soundscape*, or *environmental* music with his groundbreaking 1970 composition *Presque rien, no. 1* (Caux 2013). Horacio Vaggione combined musique concrète sources with sophisticated micromontage techniques in works such as *Mécanique des fluides* (2014).

Early Sampling Instruments

Modern sampling instruments are based on a principle of sample playback used earlier in photoelectric and tape devices (table 9.1). These devices played either optical discs (encoded with photographic images of waveforms), or magnetic tapes of sound. Depending on the disc or tape selected and the key pressed on the musical keyboard, a playback head inside these instruments would play the sound on the disc or tape that matched the pitch specified by the depressed key (figure 9.3).

The designer of the Singing Keyboard, Frederick Sammis, described the potential of such an instrument as early as 1936 (quoted in Rhea 1977):

> *Let us suppose that we are to use this machine as a special-purpose instrument for making "talkie" cartoons. At once it will be evident that we have a machine with which the composer may try out various combinations of words and music and learn at once just how they will sound in the finished work. The instrument will probably have ten or more sound tracks recorded side by side on a strip of film and featuring such words as "quack" for a duck, "meow" for a cat, "moo" for a cow . . . It could as well be the bark of a dog or the hum of a human voice at the proper pitch.*

Table 9.1 Early sampling instruments

Instrument	Inventor	Year of invention	City of invention
Light-Tone Organ	E. Welte	1936	Berlin
Singing Keyboard	F. Sammis	1936	Hollywood
Phonogène	J. Poullin	1953 first version	Paris
Special-Purpose tape Recorder	H. LeCaine	1955	Ottawa
Chamberlin	H. Chamberlin	1956	Los Angeles
Mellotron	Streetly Electronics	1963	Birmingham, UK

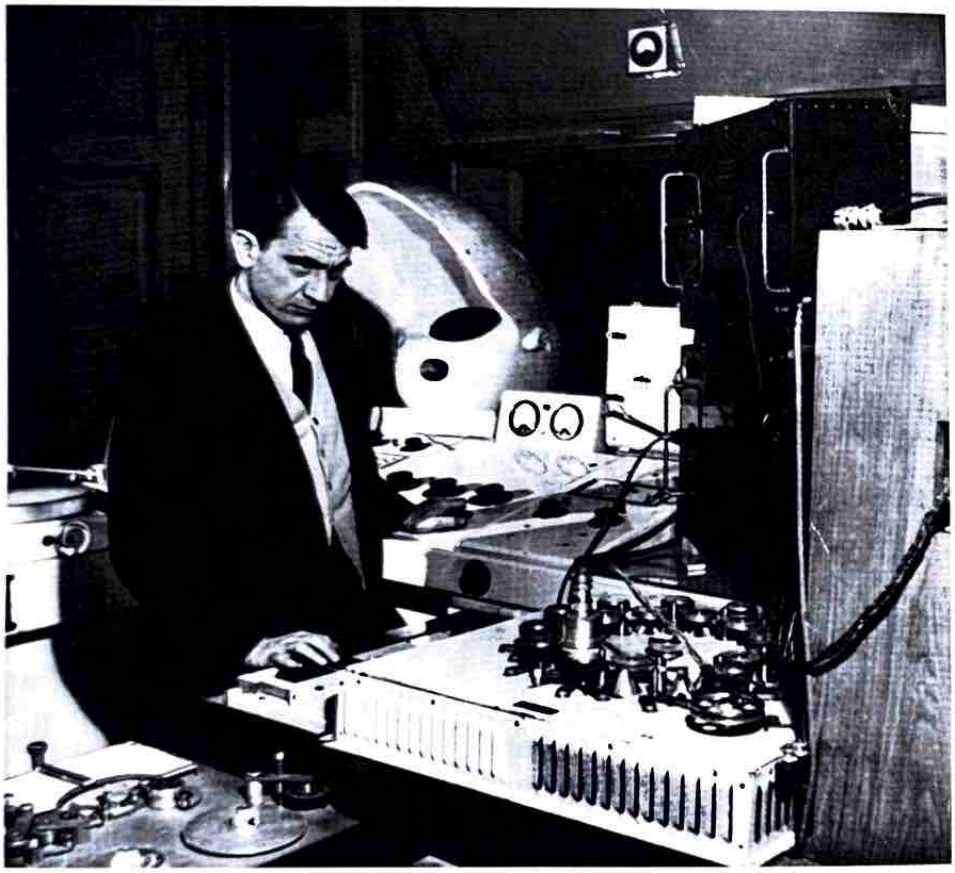


Figure 9.3 Pierre Schaeffer playing the keyboard of the Phonogène, a tape-based transposer and time stretcher, 1953, Paris. (Photograph by Lido, supplied by the courtesy of the Groupe de Recherches Musicales.)

The most famous predigital sampler was the Mellotron, an expensive and heavy instrument containing dozens of strips of tape (figure 9.4). The Mellotron enjoyed popular success with English rock groups such as the Beatles (*Strawberry Fields Forever*) and the Moody Blues (*Nights in White Satin*) in the 1960s and 1970s. They created orchestral and choral backings on popular songs. But the electromechanical design of the Mellotron made it temperamental. Tapes wore out due to head abrasion, and there were inevitable failures in the tricky mechanics for selecting and running multiple tapes. Nonetheless, the Mellotron had a signature sound that was stamped on a number of major hits. The sound of the Mellotron is alive today, as Mellotron remakes, plug-ins, and sample libraries still go strong.

By the 1980s, the rise of digital electronics made it feasible to record and store sound in digital memory chips. As digital memory chips became cheaper it was possible to store several seconds of sound for playback on a musical keyboard–based digital sampling instrument. The Fairlight

Figure 9.4 The Mellotron model 400 (1970).

Figure 9.5 The E-mu Emulator sampling keyboard instrument (1981).

Computer Music Instrument (CMI) was the first commercial keyboard sampler (1979, Australia). The CMI had a resolution of 8 bits per sample and cost over $25,000. Taking advantage of declining costs for digital hardware, the E-mu Emulator (figure 9.5), introduced in 1981, lowered the cost of 8-bit monophonic sampling (Vail 1993). For $9,000, the Emulator offered a total of 128 Kbytes of sample memory or about 10 seconds of sample time.

In order to create a commercial sampling instrument, three basic issues must be addressed: looping, pitch shifting, and data reduction, which we discuss in the next three sections.

Looping

Looping extends the duration of sampled sounds played by a musical keyboard. If the musician holds down a key, the sampler should scan seamlessly through the note until the musician releases the key. This is accomplished by specifying beginning and ending *loop points* in the sampled sound. After the attack of the note is finished, the sampler reads repeatedly through the looped part of the wavetable until the key is released; then it plays the note's final portion of the wavetable.

Commercial sample libraries are often prelooped at the factory. But for newly sampled sounds, the responsibility of specifying the begin and end loop points is usually left to the musician who sampled them. Creating a seamless loop out of a traditional instrument tone requires care. The loop should begin after the attack of the note and should end before the decay (figure 9.6).

Some samplers provide automatic methods for finding prospective loop points. One method is to perform *pitch detection* on the sampled sound (Massie 1986). (See chapter 34 for a discussion of pitch detection methods.) The pitch detection algorithm searches for repeating patterns in the wavetable that indicate a fundamental *pitch period.* The pitch period is the time interval that spans one cycle of a periodic waveform (figure 9.7). Once the pitch has been estimated, the sampler suggests a pair of loop points that match some number of pitch periods in the waveform. This kind of looping algorithm tends to generate smooth loops that are constant in pitch. If the

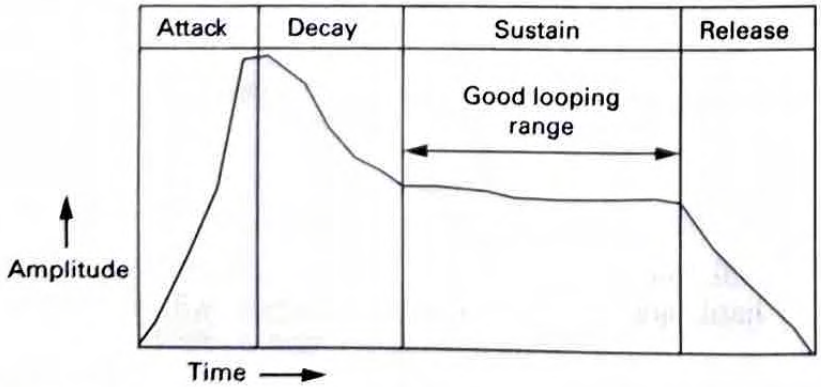


Figure 9.6 Amplitude profile of a sound with a characteristic ADSR amplitude envelope. The best area for a smooth loop is the sustained portion.

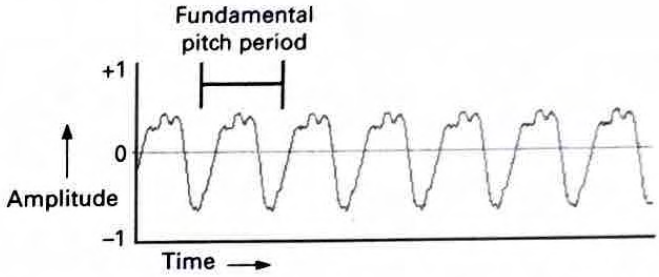


Figure 9.7 The fundamental pitch period is equal to one cycle of a periodic waveform, in this case, a waveform emitted by an alto saxophone.

body of the loop is too short, however, the result is similar to the sterile tones of fixed-waveform synthesis. For example, a loop of one or two periods of a violin note negates the time-varying qualities of a bowed string, which unfolds over hundreds of periods per second. Such a short loop yields an artificial tone that has lost its source identity.

The beginning and ending points of a loop can either be *spliced* together at a common sample point or *crossfaded*. A splice is a cut from one sound to the next. Splicing waveforms results in a click, pop, or thump at the splice point, unless the beginning and ending points are well matched. Crossfading means that the end part of each looped event gradually fades out while the beginning part slowly fades in again; the crossfade looping process repeats over and over as the note is sustained (figure 9.8). Typical crossfade times range from 10 to 100 ms but can also be longer.

When none of these techniques create a smooth loop, due to vibrato or other variations in the signal, more elaborate methods can be brought to bear, such as *bidirectional looping*. A bidirectional loop alternates between forward and backward playback (figure 9.9a). Forward and backward loops can be layered on top of one another to mask discontinuities in either direction (figure 9.9b). Even more elaborate looping techniques based on spectrum analysis are available in some looping plug-ins. For example, one can analyze the sound, randomize the phase of each spectral component in the loop, and resynthesize (Collins 1993).

Pitch Shifting

In an inexpensive sampler it may not be possible to store every note played by an acoustic instrument. These samplers store only every third or fourth semitone and obtain intermediate notes by shifting the pitch of a nearby stored note. Similarly, if one records a single sound into a sampler memory and plays it back by pressing different keys, the sampler carries out the same pitch-shifting

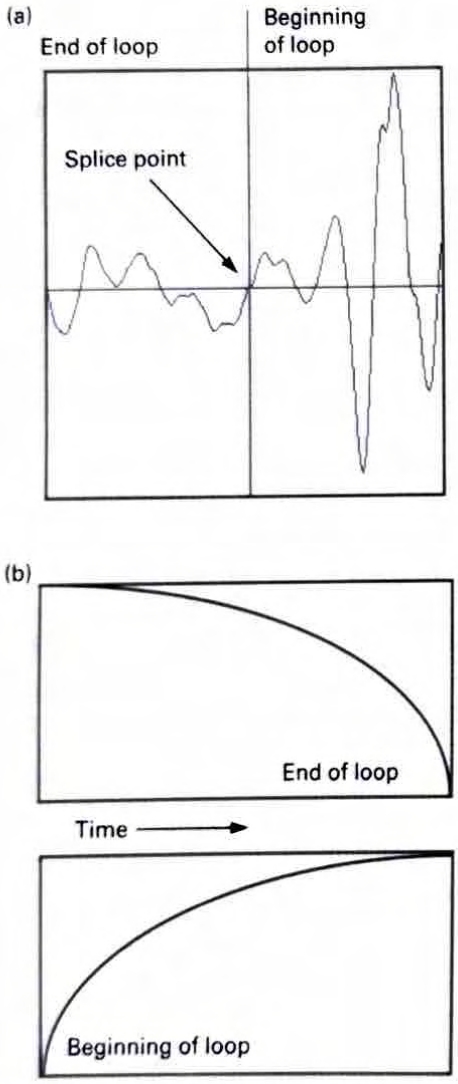


Figure 9.8 Splicing versus crossfading loops. (a) A butt splice of two parts of a waveform at a common zero point. The ending point of the loop splices to the beginning of the same wavetable loop. (b) Crossfade looping can be viewed as a fade out of the end of the loop overlapped by a fade in of the beginning of the loop.

technique. A side effect of simple pitch shifting is that the sound's duration increases or decreases, depending on the key pressed. Looping over a sustained portion of a sound compensates for the duration change.

Two methods of simple pitch shifting exist:

1. Varying the clock frequency of the output digital-to analog converter (DAC) changes the playback sampling rate; this shifts the pitch up or down.
2. *Sample-rate conversion* (*resampling* the signal in the digital domain) shifts the pitch inside the sampler and allows playback at a constant sampling rate for all pitches.

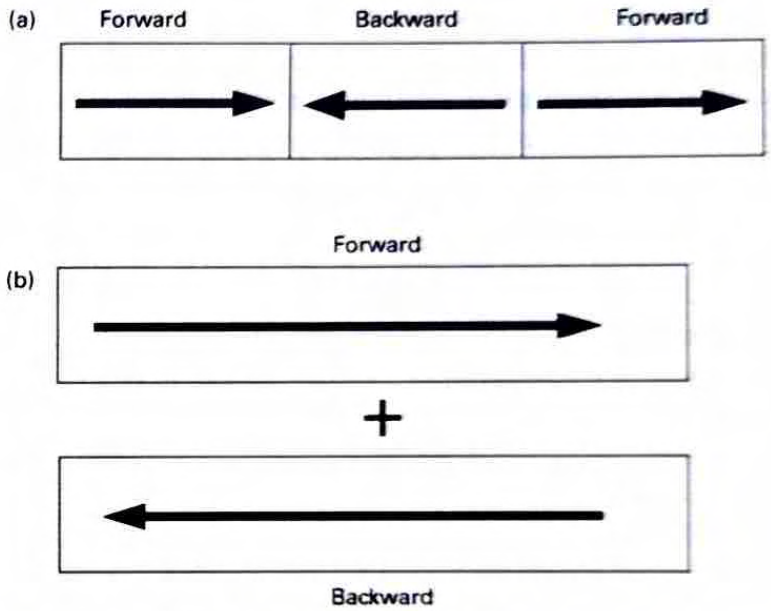


Figure 9.9 Looping methods for smoothing out variations. (a) Three cycles of a bidirectional loop. (b) In a layered forward/backward loop the two versions are added together.

Method 1 was used in early samplers like the original Emulator. Method 2 is the way that it is done today. Both these methods are called *time-domain* techniques because they operate directly on the time-domain waveform. This is different from the *frequency-domain* pitch-shifting techniques discussed in chapter 31. Next, we compare these two time-domain methods.

Because method 1 changed the playback sampling rate, it required a separate DAC for each note that can be played simultaneously on the musical keyboard (typically up to ten DACs). Each DAC had a variable clock rate and a variable-frequency smoothing filter. For full transposability, the DAC and the filter had to work over extremely wide operating ranges. To give an example, if a tone with a pitch of 250 Hz sampled at 44.1 kHz is shifted up six octaves to 16 kHz, the clock frequency of the output DAC would have to shift up six octaves to 2.82 MHz. In practice, this was not feasible. Thus the designers imposed limitations on the audio performance of the system. For example, one sampler that employed this pitch-shifting method allowed only a single semitone of transposition (less than a 6 percent change of clock frequency) for sounds recorded at its highest sampling rate of 41.67 kHz. The DAC and the filter are never forced to work at a sampling rate higher than 44.1 kHz. Other samplers did not permit any transposition above an arbitrary limit.

Pitch-shifting method 2 performs sample-rate conversion. This is essentially the same technique as used in wavetable-lookup synthesis, described in

chapter 3. Sample-rate conversion, in effect, resamples the signal in the digital domain. The output DAC's sampling frequency remains constant. Speeding up a sound and increasing its pitch is achieved by resampling the waveform at a lower sampling rate. This is analogous to time-lapse photography in which the frame rate is slowed down to achieve a speed-up on playback. In a digital audio system, samples are skipped in resampling. The number of samples that are skipped is proportional to the amount of pitch shifting that is desired (just as in wavetable-lookup synthesis). The process of skipping samples in resampling is called *decimation* (figure 9.10a). Resampling with decimation is also called *downsampling*. For example, to shift the pitch upward by three octaves, the signal is downsampled by reading every third sample in playback.

To lower the pitch of a sound and slow it down, the sound is resampled at a higher frequency to stretch it out. This is analogous to the operation of a slow-motion camera that speeds up the frame rate to achieve a slow-down effect upon playback. In a digital audio system, new intermediate samples are inserted between existing samples by means of *interpolation* (figure 9.10b). Resampling with interpolation is also called *upsampling*.

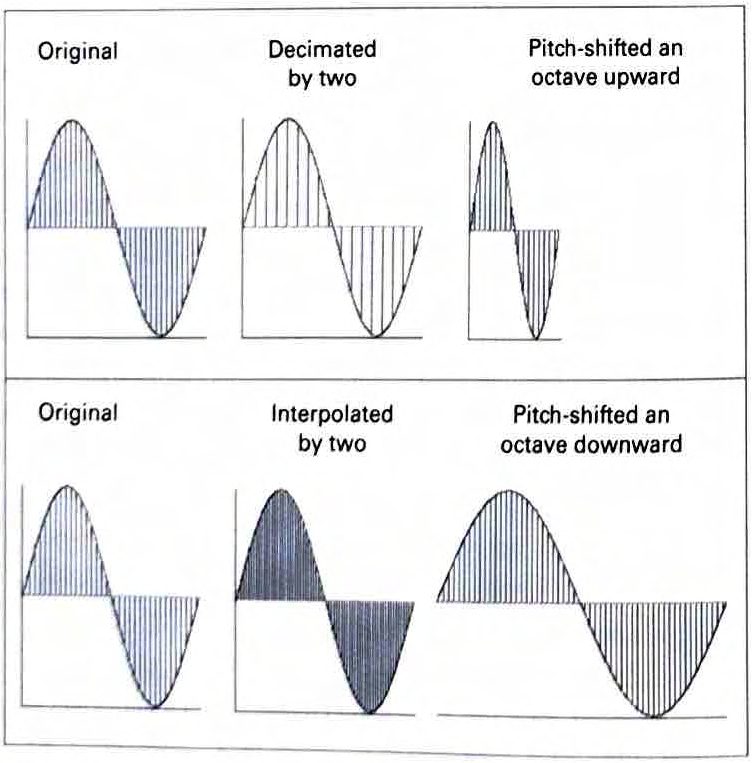


Figure 9.10 Pitch-shifting by sample-rate conversion with a constant playback sampling frequency. Top: If every other sample is skipped on playback, the signal is decimated, and the pitch is shifted up an octave. Bottom: If twice the number of samples are used by means of interpolation on playback, the signal is shifted down an octave.

The relationship between the various resampling rates and pitch shifting can be confusing at first, because pitch-shifting method 1 and method 2 seem to go in opposite directions to achieve the same aim. Method 1 raises pitch by increasing the sampling rate on playback. Method 2, however, raises pitch by decreasing the resampling rate with decimation, even though the playback sampling frequency is constant.

So far, we have seen how to shift pitch by octave intervals. To shift pitch by any integer ratio, a combination of interpolation and decimation are used (Schafer and Rabiner 1973a; Moorer 1977; Rabiner 1983; Lagadec 1983; Crochiere and Rabiner 1983; Hutchins 1986a; Duncan and Rossum 1988; Mitra 2006). In particular, to pitch shift by a ratio N/M, we first interpolate by M and then decimate by N. For example, to shift a sound down by an interval of 3/4 (a perfect fourth) we upsample and interpolate by a factor of 4 and then downsample and decimate by a factor of 3. To shift up by a factor of 4/3 we first interpolate by 3 and then decimate by 4.

Sample-Rate Conversion without Pitch Shifting

Many digital audio recorders operate at the standard sampling rates of 44.1 or 48 kHz and their multiples. How can we resample a recording at one of these frequencies so as to play it back at the other frequency with no pitch shift? In this case the resampling rate is the same as the new output DAC sampling rate.

To convert a signal between the standard sampling rates of 44.1 and 48 kHz without a pitch change, a rather elaborate conversion process is required. First the rates are factored as follows:

$$\frac{48{,}000 = 2^5 \times 5}{44{,}100 = 3 \times 7^2} = (4/3 \times 4/7 \times 10/7)$$

These ratios can be implemented as six stages of interpolations and decimations by factors of 2, 3, 5, and 7, as follows:

1. Interpolate by 4 from 44,100 to 176,400 Hz
2. Decimate by 3 from 176,400 to 58,800 Hz
3. Interpolate by 4 from 58,800 to 235,200 Hz
4. Decimate by 7 from 235,200 to 33,600 Hz
5. Interpolate by 10 from 33,600 to 336,000 Hz
6. Decimate by 7 from 336,000 to 48,000 Hz

The signal can then be played back at a sampling rate of 48 kHz with no change of pitch.

As long as the input and output sampling rates can be written as a simple fraction, then the conversion process is straightforward. If the rates do not have an integer ratio or they are changing, then more sophisticated mathematical techniques must be used, which we will not venture into here (see Crochiere and Rabiner 1983; Rabiner 1983; Lagadec 1983; Mitra 2006). A variable resampling rate applies in the case with *flanging effects* (chapter 30) and audio *scrubbing* (simulating the manual back-and-forth motion of a magnetic tape rocking across a playback head to locate a splice point).

Problems in Resampling

The audio fidelity of resampling is limited by the numerical precision used in the conversion. When there are many intermediate resampling stages, a slight loss in fidelity in the form of added noise is to be expected. Aliasing can also be a problem because, as in the original sampling process, resampling can generate unwanted spectral artifacts due to aliasing. When a sampler skips samples in decimation, it is throwing away intermediate samples. These intermediate samples may have smoothed the waveform's transition between two disjoint points. Thus a decimated signal can contain jagged discontinuities (figure 9.11). At the same time, all frequencies are shifted up, meaning that aliasing can occur on playback. This problem can be reduced to a minimal

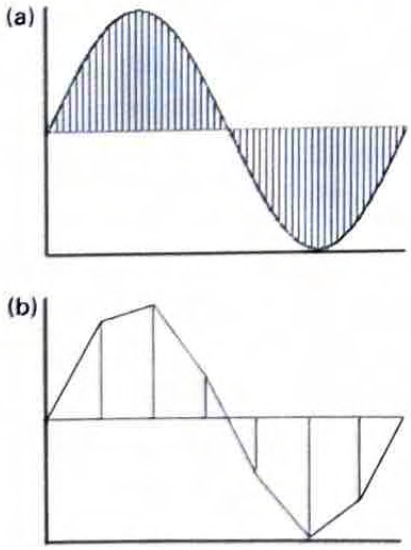


Figure 9.11 With enough decimation, even a sine wave can be turned into a jagged waveform. (a) Original sinusoidal waveform. (b) Decimation of (a) by a factor of 8.

effect by lowpass filtering the signal after decimation. Filtering smooths the jagged edges of the decimated waveform (Pohlmann 2010).

Filtering is also needed in interpolation because simple linear interpolation creates aliased components. Rather than devising a more complicated interpolation scheme, the usual approach in sample-rate conversion is to combine linear interpolation with filtering to shift the frequency content and also minimize aliasing.

Sample and Wavetable Libraries

Because a sampler is a type of recording system, the quality of the samples depends on the recording process. Making high-quality samples requires good players and instruments, excellent microphones, and favorable recording environments. Arranging all these elements for a library of sounds requires serious investment and effort. Thus most users of samplers prefer to supplement their personal collection of samples with libraries prepared by professionals—or by other amateurs, as in https://www.freesound.org (de Jong 2005).

Sample libraries are widely used in the entertainment industry, not only for music but also routinely for ambience and sound effects. Almost every radio and television show, movie, commercial, game, and pop song uses samples to create mood. Responding to commercial demand, a vast palette of sample libraries is available, ranging from orchestral instruments, choirs, and electric instruments (guitars, organs, etc.), to world instruments and sounds from natural habitats and industrial facilities. Well-known professional sound designers create and curate custom sound libraries (Devine 2008).

Some sample libraries are packaged in proprietary formats that can be read exclusively by authorized sample players. An example is the East-West product line with its PLAY sample engine. Other libraries are distributed in standard formats such as 24-bit .wav files or Apple Loops. They can be copied into folders linked with sample players like Apple Logic Pro, Reason, Native Instruments Kontakt, and Steinberg HALion. Free and open formats like SoundFonts (chapter 51) and SFZ (SFZ 2019) are royalty-free. Utilities like Translator by Chicken Systems can convert between many different sample formats.

Wavetable libraries contain thousands of single-cycle waveforms. They function as expansion packs for popular wavetable-based synthesizers like Native Instruments Absynth, Camel Audio Alchemy, U&I Software MetaSynth, and Cakewalk Rapture. An example library is Galbanum Architecture Waveforms, which offers over 25,000 waveforms.

Emulating Traditional Instruments

A sampler can play back and pitch shift any sound. However, most users are looking for emulations of traditional instruments. Thus the dream of a "symphony orchestra in a box" remains a holy grail for the creators of sample libraries. Unfortunately, sampled sound often has an artificial quality that makes it easily distinguishable from the animated sounds produced by good human performers. Most percussionists, for example, would not mistake the frozen sound of a sampled drum solo from that of a human drummer. In a live performance on acoustic drums, each drum stroke is unique, and there are major differences in the sound depending on the musical context in which the stroke is played. This is not to say that robotic performance is invalid. The commercial success of electronic drum machines like the Roland TR-808 (and myriad sampled versions of it) proves that lock-step rhythms and unvarying percussion sounds have a major audience. Original TR-808s sell for thousands of dollars.

Still, it is understandable that the naturalness or realism of a sampler or sample library should be held up as a criterion for judging between different products. Differences in sample quality are not subtle. Certain instruments, such as organs, can be simulated more or less realistically by most samplers. Other instruments such as voices, strings, woodwinds, brass, and plucked or strummed instruments are intrinsically more difficult to capture with existing sampling technology. Individual notes can be captured well, but when we string the notes together into melodies and phrases, it is apparent that major chunks of acoustic and performance information have been left out.

Factory-supplied samples model the generic singer, the generic saxophone played by the generic saxophonist, the generic orchestra played in the generic concert hall, and so on. Yet most knowledgeable listeners can tell the difference between two vocalists, saxophonists, and conductors with orchestras. It would be difficult to mistake a MIDI sequencer + sampler rendition of a saxophone solo with the signature style of the John Coltrane original. This points out a fundamental limitation in many existing samplers. Beyond a certain point, it is impossible to increase the realism of samplers without understanding the relationship between sound structure and musical performance.

In expressive instruments like voices, saxophones, strings, brass, guitars, and others, each note is created in a musical context. Within a phrase, a note is reached from another note (or from silence) and leads into the successive note (or leads to silence). In addition to these contextual cues, transitional sounds like breathing, tonguing, key clicks, and sliding fingers along strings punctuate the phrasing. Constraints of style and taste determine when context-

sensitive effects such as rubato, legato, staccato, portamento, vibrato, crescendi and diminuendi, and other nuances are applied.

How can we model the sound microstructure of note-to-note transitions? This question is the subject of the next section.

Modeling Note-to-Note Transitions

The problem of what happens in note-to-note transitions was the subject of the doctoral research of John Strawn at Stanford University (1985c). He analyzed the transitions in nine nonpercussive orchestral instruments. The time-domain and frequency-domain plots that emerged from this research graphically depicted the context-sensitive nature of tone successions.

In wind instruments, one of the ways to articulate a transition is by *tonguing*–a momentary interruption of the windstream by an action of the tongue, as if the player were pronouncing the letter *t* or *k*. Figure 9.12 shows a time-domain plot of transitions of a trumpet played tongued (a) and untongued (b). The contrast between the two types of transitions is clear.

Figure 9.13 plots the spectrum of this transition. Strawn's research demonstrated that some transitions are stamped with strong transitional cues in amplitude and spectrum changes that articulate the attack of the second note (figure 9.13a). Other transitions are very smooth, with a dip of as little as 10 dB of amplitude between notes (figure 9.13b).

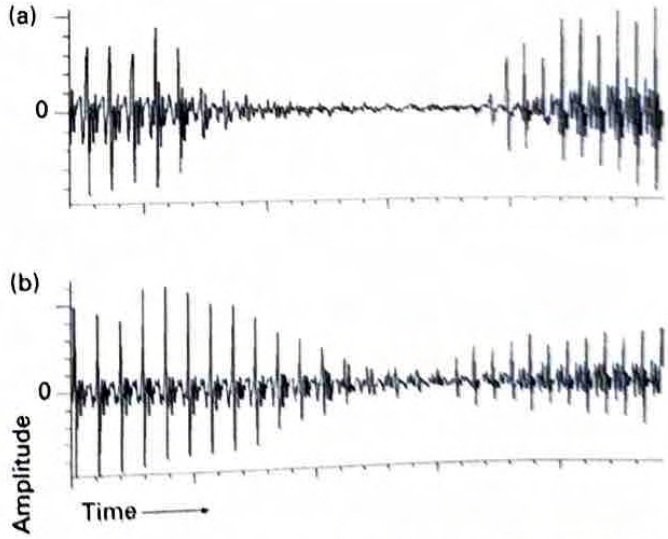


Figure 9.12 Time-domain plot of note-to-note transition of an ascending major third interval for a trumpet played tongued (a) and untongued (b). The time span for the plots is 120 ms.

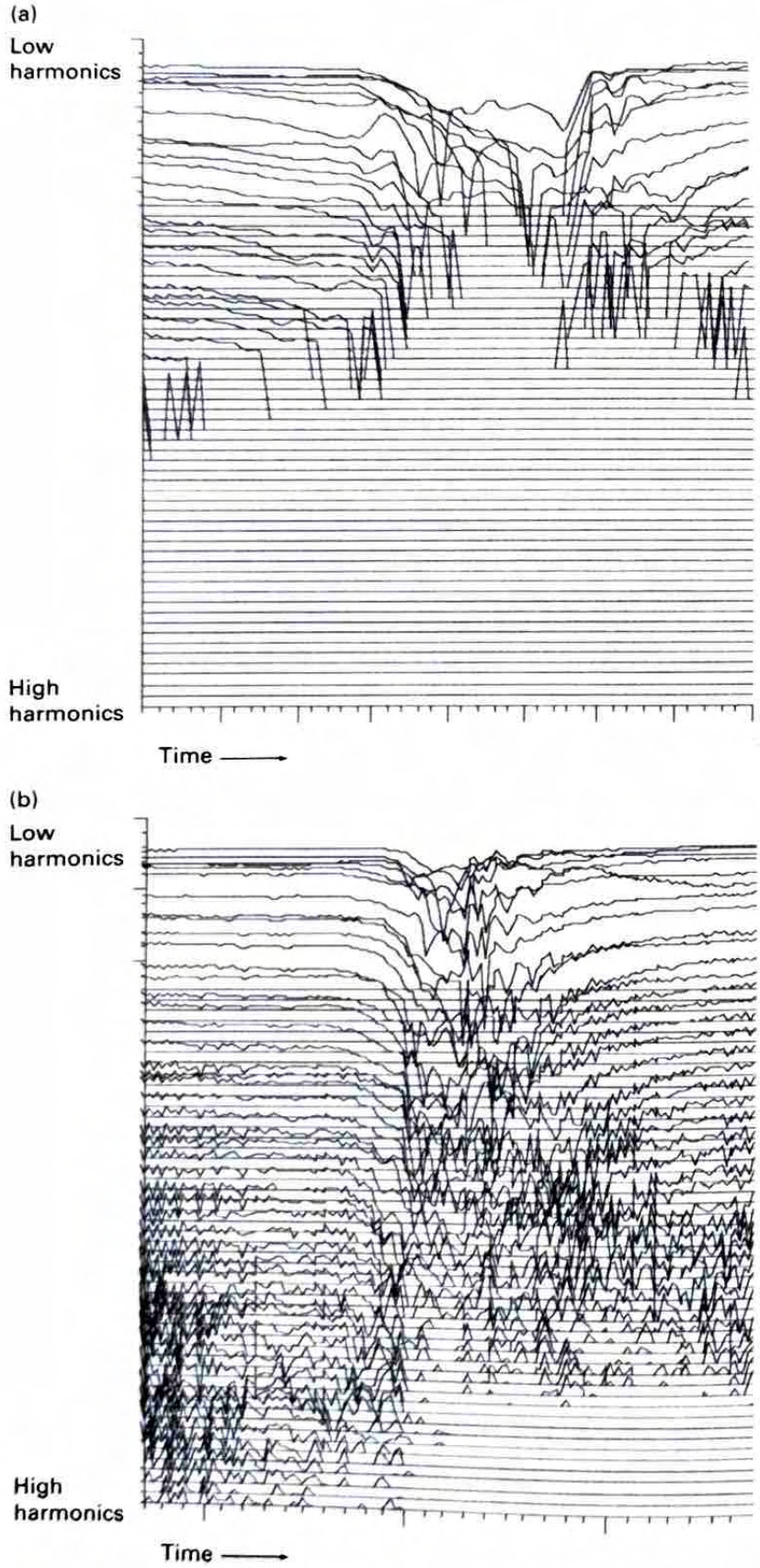


Figure 9.13 Spectrum plots of the transitions shown in figure 9.10. The plots show fifty harmonics plotted over a time span of 300 ms, with lower harmonics at the back. (a) Tongued. (b) Untongued. Notice how the "hole" in the middle of (a) is filled in when the note transition is untongued (more continuous).

Similar issues have been studied in speech, where the issue of phoneme transitions is called the *coarticulation problem*. Certain consonants, for example, are pronounced differently depending on where they are in a word. Consider the *n* in *tenth*. It is pronounced with the tongue against the teeth because the *th* that follows it is also dental, whereas the *n* in *now* is pronounced differently.

Synthesis Methods That Model Note-to-Note Transitions

Several synthesis techniques aim directly at modeling realistic note-to-note transitions. These include physical models (chapter 18 and Smith 2010), spectral models (chapter 37 and Bonada and Serra 2007), and additive synthesis (chapter 10 and Lindemann 2007, 2009).

One of the earliest methods, *diphone synthesis*, addressed the problem of phoneme transitions in speech (Peterson and Barney 1952; Peterson, Wang, and Sivertsen 1958; Olive 1977; Schwartz et al. 1979). The basic theory of diphone synthesis assumes that speech consists of a series of stable sounds separated by transition sounds. Modern speech synthesizers such as Amazon Alexa and Apple Siri use this method to create quasi-natural-sounding speech. Today we would call this *concatenative synthesis* (chapter 23 and Maestre 2009). By generalizing the concept from the limited domain of speech to the open universe of musical sounds, one can build up dictionaries of stable and transition sounds to cover a specific class of sounds, such as bowed cello tones. Researchers have developed dictionaries of transition rules for instruments that smooth the concatenation of adjacent diphones (Rodet et al. 1988; Depalle 1991).

Another way to model the microstructure of note-to-note transitions in instrumental sounds is a "brute force" calculation: capture every possible note-to-note transition as a sample. Then trigger the appropriate transition sample when going from a note to any other note.

In recent years, companies including Yamaha and Roland have introduced expressive samplers that take into account not only the instrument being emulated but also which notes follow which when they are played on a keyboard. These samplers implement *controller-switched transitions* between notes as the musician plays the keyboard. These transitions make the performance sound like a more realistic brass, woodwind, or stringed instrument. For example, the Yamaha MOTIF XF synthesizer combines sampling with physical modeling and real-time analysis of the keyboard in order to replicate the performance nuances of the selected instrument, such as flamenco strumming, plucked-string harmonics, string slides, fret noises, double-bass legato, and marimba trills.

In the same vein, NotePerformer is a sound playback engine for orchestral score editing programs like Sibelius, Finale, and Dorico, among others. Through analysis of the score, NotePerformer offers improved musical phrasing using a variety of articulations and note-to-note transition (mutes, harmonics, pizzicato, sul tasto, staccato and slur, tremolo, flutter tonguing, marcato, tenuto, etc.) on strings, brass, and woodwinds.

Finally, several companies offer sample libraries that play back entire prerecorded phrases so that note-to-note transitions are already baked into the sound. The Sonokinetic product line is a prime example. They supply libraries of hundreds of continuous phrases (beds, textures, ostinato rhythms, arpeggios, runs, rises, drops, etc.) played by orchestra in the style of composers such as Bernard Herrmann, Jerry Goldsmith, Karlheinz Stockhausen, and Krzysztof Penderecki, among others.

10 *Additive Synthesis*

Additive Synthesis: Background

Fixed-Waveform Additive Synthesis

- **The Phase Factor**
- **Addition of Partials**

Time-Varying Additive Synthesis

Demands of Additive Synthesis

Sources of Control Data for Additive Synthesis

Additive Analysis/Resynthesis

- **Musical Applications of Additive Analysis/Resynthesis**
- **Methods of Sound Analysis for Additive Synthesis**
- **Data Reduction Models in Additive Analysis/Resynthesis**
  - **Line-segment approximation**
  - **Principal components analysis**
  - **Spectral interpolation synthesis**
  - **Spectral modeling synthesis**
  - **ATS: Analysis, transformation, synthesis**
  - **Reconstructive phrase modeling**

Additive Synthesis Based on Machine Learning

Walsh Function Synthesis

Additive synthesis is based on the summation of elementary waveforms to create a complex audio signal. Additive synthesis is one of the oldest synthesis techniques, yet it continues to evolve.

Additive synthesis can simulate existing sounds or create new, previously unheard textures. However, the power of additive synthesis is that it is possible to precisely approximate any complicated waveform as a sum of elementary waveforms, a theory that can be traced back to Jean-Baptiste Joseph Fourier (1768–1830).

When coupled with a sound analysis stage, additive synthesis becomes a powerful means of simulating natural sounds and cloning variations of them. Because it is based on a sinusoidal model, sounds such as slowly varying sine waves are well modeled by additive synthesis, which means that they can be described relatively simply. Other sounds such as modulations and noise textures are not so well modeled, so that they require more complicated analysis schemes, as we explain further on.

Two other chapters cover related topics: chapter 37 deals with spectrum analysis by Fourier methods, and chapter 44 surveys spectrum editors.

Additive Synthesis: Background

The concept of additive synthesis is ancient, first applied in pipe organs by means of their multiple *register stops* (figure 10.1). By pulling on a register stop, the organist could route air to a set of pipes. The air was released into the pipe—creating sound—by pressing a key on the organ keyboard. By pulling several register stops, the organist could add together the sound of several pipes. According to one scholar, "The Middle Ages particularly favored the mixture in which every note was accompanied by several fifths and octaves based upon it" (Geiringer 1945). This idea of frequency mixtures is the essence of additive synthesis.

Additive synthesis has been used since the earliest days of electronic music (Cahill 1897; Douglas 1968; *die Reihe* 1955; Stockhausen 1964). The massive Telharmonium synthesizer unveiled in 1906 summed the signal of dozens of electrical tone generators to create additive sound complexes (Weidenaar 1991). Figure 10.2 depicts the additive scheme.

The famous Hammond organ (figure 10.3) manufactured between 1935 and 1972 was a pure additive synthesis instrument that incorporated a miniature version of the Telharmonium's electromechanical rotating tone generators.

All organs, whether mechanical, electromechanical, or purely electronic in technology, do not produce particularly realistic simulations of traditional instruments. The reason is that they produce a fixed waveform that does not

Figure 10.1 Pipe organ console with register stops on either side of the keyboards.

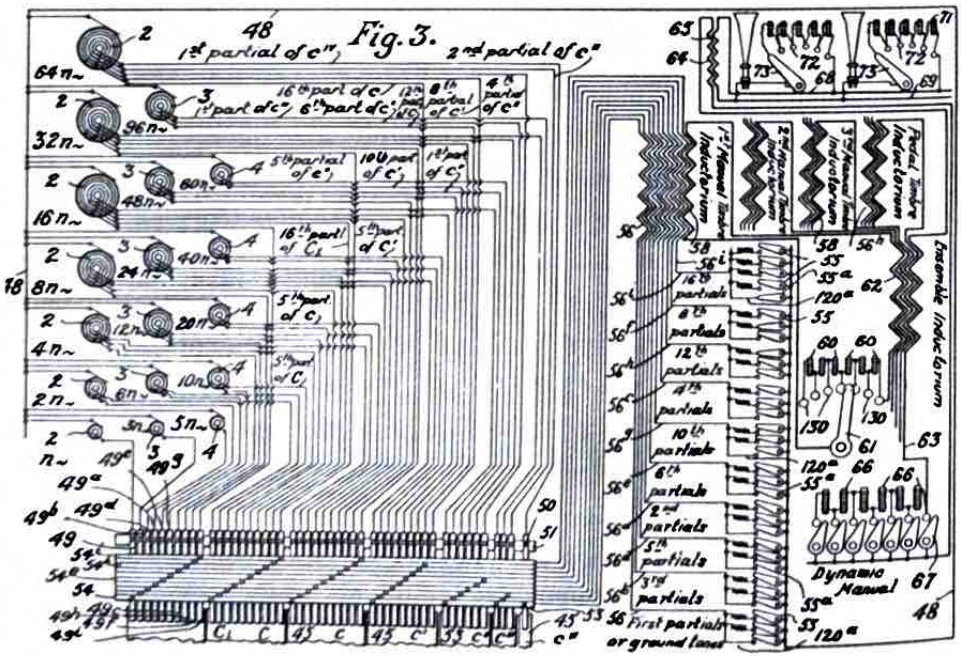


Figure 10.2 Additive synthesis of a complex tone in the Telharmonium. Sine wave harmonics from the tone-generating alternator are fed to bus bars (54). Pressing a key (C in this case) connects each harmonic to a multicoil transformer (56 Inductorium) where they mix. Each harmonic is attenuated to the desired level by the inductors in series with each winding (56a, b, etc.). The tap-switch inductors (60) regulate the amplitude of the mixing transformer output, as do the inductors near the loudspeakers (72, 73) at the listener's end of the transmission line. (Cahill patent drawing, reproduced in Johnson et al. [1970].)

Figure 10.3 The author's Hammond T422 organ, an additive synthesis instrument based on electromechanical tone-wheels. Different mixtures of the various harmonics can be adjusted by pulling drawbars above the musical keys. The instrument contains a built-in Leslie rotating speaker.

vary at the micro time scale. As we show further on, in order to produce a realistic simulation, the mixture of sine waves needs to constantly vary in detail over time.

Fixed-Waveform Additive Synthesis

Some synthesizers let musicians create waveforms by *harmonic addition*. In order to make a waveform with a given spectrum, the user adjusts the relative strengths of a set of *harmonics* of a given fundamental. (The term *harmonic* as an integer multiple of a fundamental frequency was first used by Sauveur in 1701.) For example, 400 Hz is the second harmonic of 200 Hz, because two times two hundred equals four hundred. The harmonics can be displayed as a *bar graph* or *histogram*, with the height of each bar representing the relative strength of a given harmonic. This *spectrum template* aligns to different fundamental frequencies and their harmonics when one changes the pitch of the oscillator. Figure 10.4 shows a harmonic spectrum and the corresponding waveform.

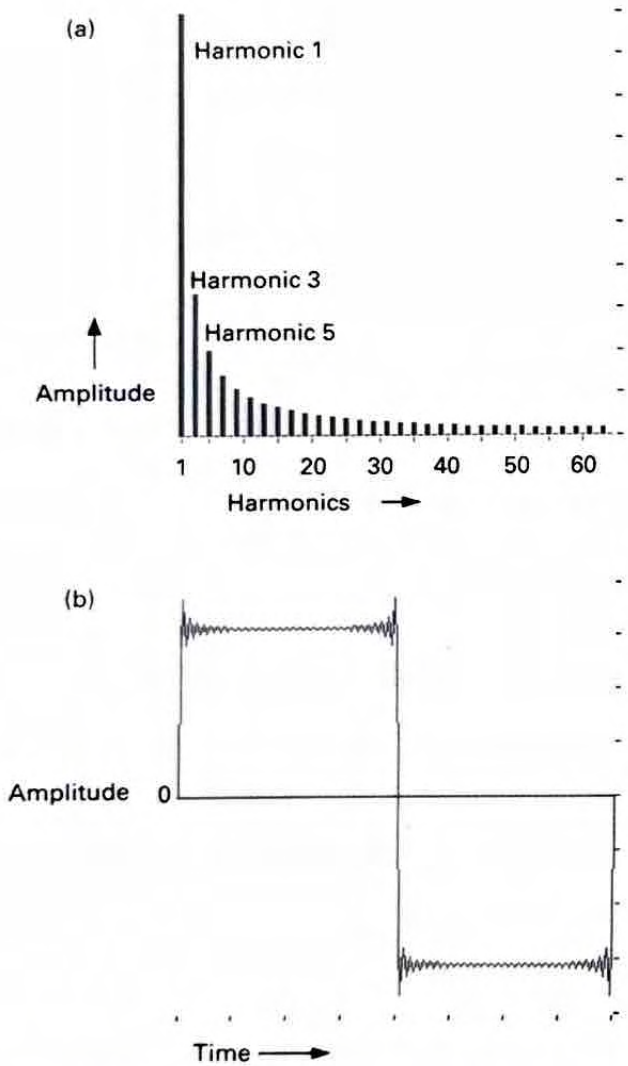


Figure 10.4 Waveform synthesis by harmonic addition. (a) Histogram plot showing the relative strength of the harmonics on a linear scale. In this case, the histogram has energy only in the odd harmonics. The amplitude of the third harmonic is one-third that of the fundamental, the amplitude of the fifth harmonic is one-fifth that of the fundamental, and so on. (b) Approximation to a square wave synthesized by harmonic addition in (a).

Once a desirable spectrum is tuned, a digital oscillator adds together the various harmonics. Figure 10.5 shows successive stages of waveform addition used to create a quasi-square wave.

The Phase Factor

Phase (presented in chapter 2) is a trickster. Depending on the context, it may or may not be a significant factor in additive synthesis. For example, if one changes the starting phases of the frequency components of a fixed

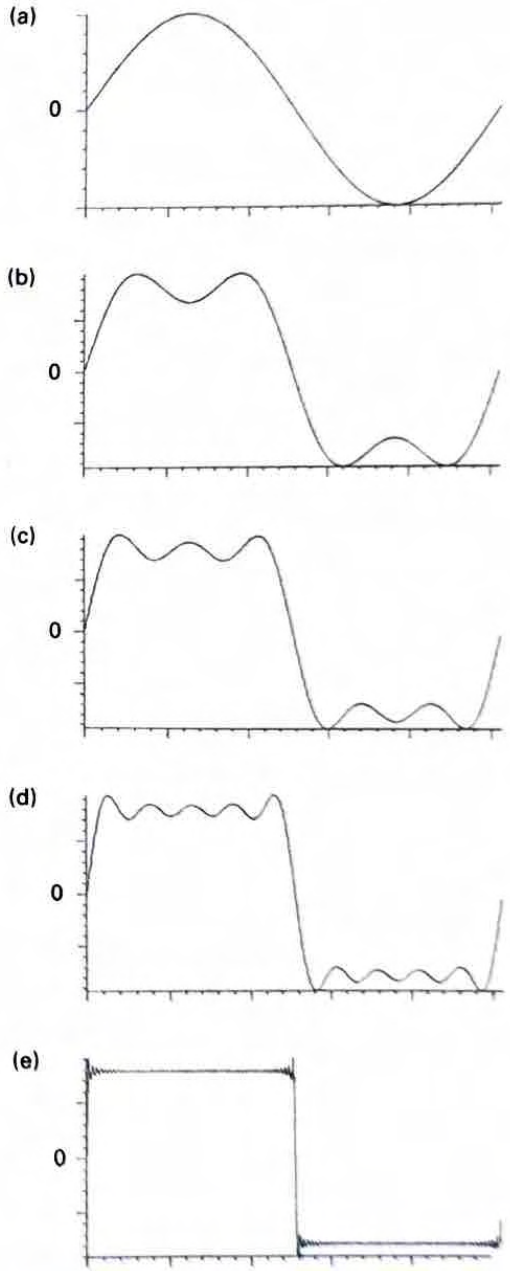


Figure 10.5 Stages of harmonic addition as seen in a series of time domain waveforms. (a) Fundamental only. (b) First and third harmonics. (c) Sum of odd harmonics through the fifth. (d) Sum of odd harmonics through the ninth. (e) Quasi-square wave created by summing odd harmonics up to the 101st.

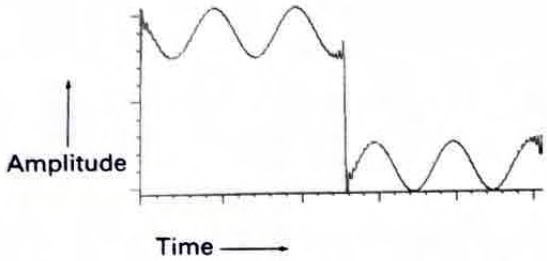


Figure 10.6 Effect of phase in additive synthesis. This waveform is the result of the same mixture of sine waves as in figure 10.5e except that the starting phase of the fifth harmonic is 90° instead of 0°.

waveform and resynthesizes the tone, this makes no difference to the listener. Yet, such a change may have a radical effect on the visual appearance of the waveform, as shown in figure 10.6.

Phase relationships become apparent in the perception of the brilliant but short life of attacks, grains, and transients. The ear is also sensitive to phase relationships in complex sounds in which the phases of certain components are shifting over time. As we show in the section on sound analysis and resynthesis, proper phase alignment is essential in reconstructing an analyzed sound.

Addition of Partials

We can generalize from addition of harmonics to addition of *partials*. In acoustics, a *partial* refers to an arbitrary frequency component in a spectrum (Benade 1990). The partial may or may not be a harmonic (integer multiple) of a fundamental frequency *f*. Figure 10.7a shows a spectrum containing four partials: two harmonic and two *inharmonic*. An *inharmonic partial* is not in an integer ratio to the fundamental frequency. Figure 10.7b is the waveform that results from the sum of the four arbitrary partials.

At the level of synthesizing a single waveform, adding partials is limited from a musical point of view. Because the spectrum in fixed-waveform synthesis is constant over the course of a note, partial addition can never reproduce accurately the sound of an acoustic instrument. It approximates only the *steady-state* portion of an instrumental tone. A time-varying timbre is more tantalizing to the ear than a constant spectrum (Grey 1975).

Time-Varying Additive Synthesis

One of the pioneers of computer music, Jean-Claude Risset (1938–2016) analyzed instrumental tones to determine the evolution of their harmonics over

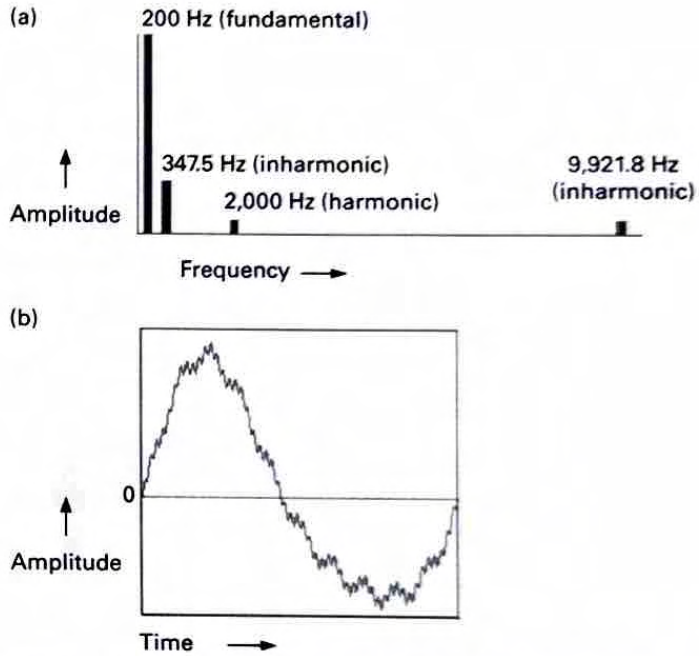


Figure 10.7 Partial addition with four components, both harmonic and inharmonic. The percentage contribution of each component is 73, 18, 5, and 4 percent, respectively. (a) Frequency-domain view. (b) Time-domain waveform.

time (figure 10.8). He discovered that variations in amplitude of each harmonic on a micro time scale was the secret to a tone's identity (Risset 1966, 1969, 1985a, 1985b, 1991; Risset and Mathews 1969; Risset and Wessel 1982). A composer as well as a scientist, Risset explored time-varying additive synthesis in many compositions, notably *Inharmonique* (1977), which charted a course through the vast space of inharmonic sine waves.

By changing the mixture of sine waves over time, one obtains more interesting synthetic timbres and more realistic instrumental tones. In the trumpet note in figure 10.9, it takes twelve sine waves to reproduce the initial attack portion of the event. After 300 ms, only three or four sine waves are needed.

We can view the process of partial addition graphically in several ways. Figure 10.10 shows additive synthesis in the analog domain, as it was originally practiced in the 1950s by composers such as Karlheinz Stockhausen (Stockhausen 1964). The figure shows several oscillator hardware modules, each with a manually controlled frequency knob. The outputs of the oscillators are routed to a mixer. To precisely realize a time-varying mixture took two people working together (Morawska-Büngler 1988). The composer and the assistant adjusted the balance of the oscillators in real time to determine the time-varying spectrum. With this equipment, manual control was the only option.

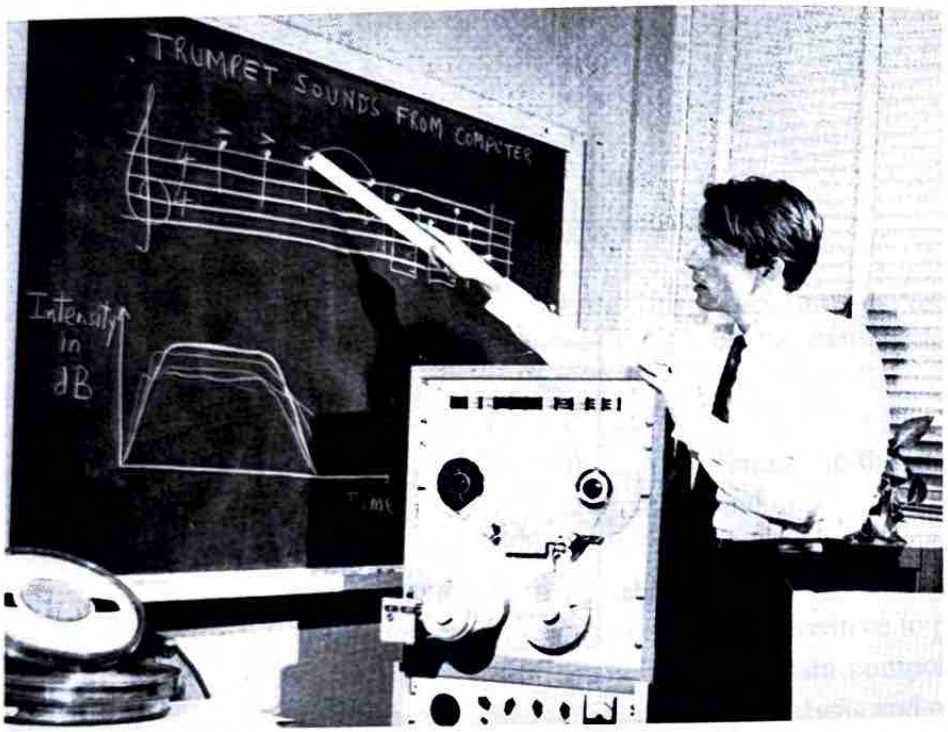


Figure 10.8 Jean-Claude Risset demonstrates a trumpet tone created by additive synthesis at Bell Telephone Laboratories, 1965. (Photograph courtesy of AT&T.)

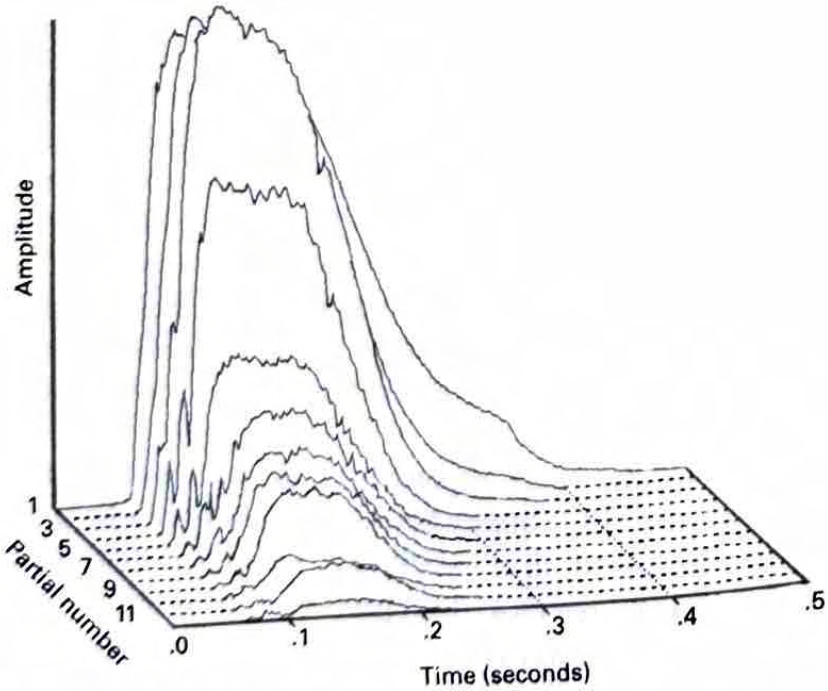


Figure 10.9 Time-varying spectrum plot of twelve partials of a trumpet tone, with the highest partials in the foreground. Time goes from left to right. Notice that the fundamental (back) is not the highest amplitude, but it lasts the longest.

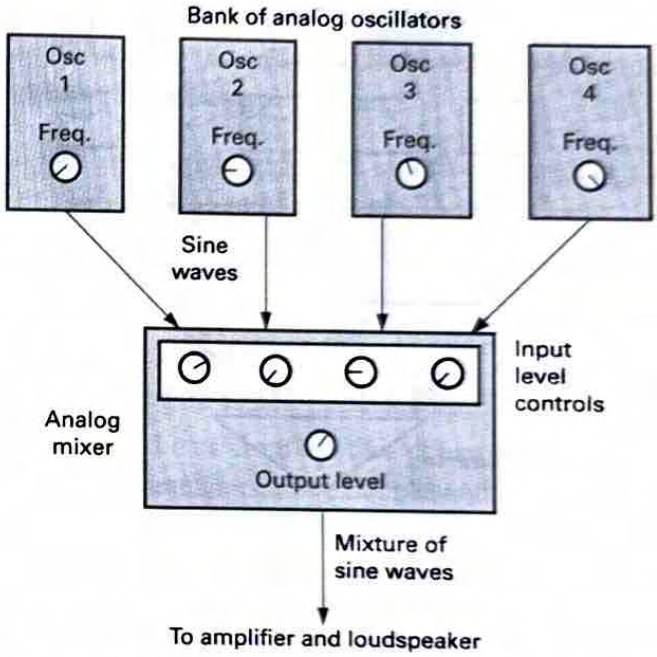


Figure 10.10 Additive synthesis in the analog domain. Oscillators feed a mixer (bottom).

Figure 10.11 shows digital additive synthesis. An audio oscillator is represented as a half-circle with a pair of inputs—one for amplitude and one for frequency. To generate a time-varying spectrum, each frequency and amplitude input to the oscillators is not a constant but a time-varying envelope function read over the duration of the event. The sine wave audio oscillators feed into a module that sums the signals. The sum module then passes the additive result to a DAC for conversion to sound.

David Wessel's *Antony* (1977) was a landmark in additive synthesis because it was the first composition to feature 256 time-varying oscillators (MacCallum, Goodheart, and Freed 2015).

Demands of Additive Synthesis

Time-varying additive synthesis imposes significant computational demands. First, it requires large numbers of oscillators. If we make the musically reasonable assumptions that each sound event in a piece can have up to sixteen partials (each generated by a separate sine wave oscillator) and that up to twenty-four events can be playing simultaneously, we need up to 384 oscillators at any given time. If the system is running at a sampling rate of 48 kHz,

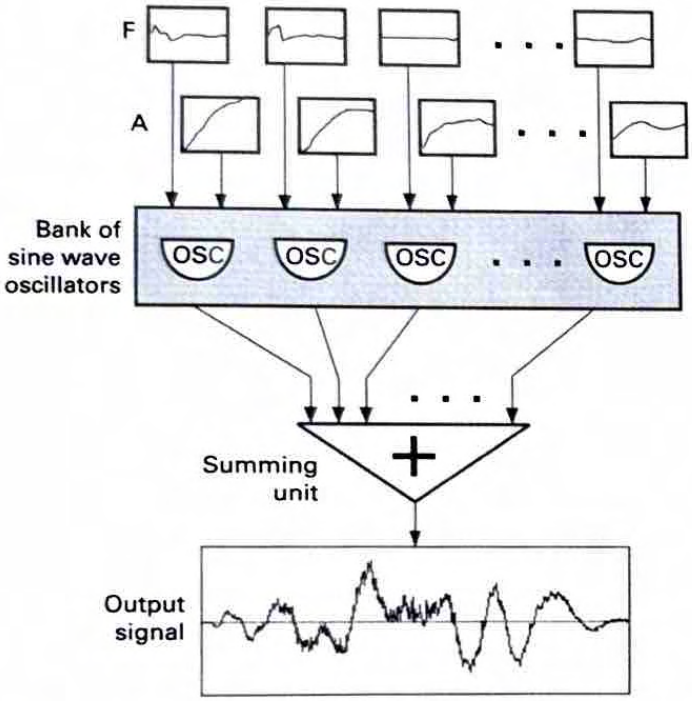


Figure 10.11 Digital time-varying additive synthesis with separate frequency (F) and amplitude (A) envelopes.

it must be capable of computing $48{,}000 \times 384 = 18{,}432{,}000$ samples/second. This is a huge amount of computation compared to twenty-four voices of frequency modulation, for example. (See chapter 16 on frequency modulation.) It is, however, within the range of today's laptop computers. Savioja et al. (2011) describes additive synthesis using *graphics processing units* (GPUs), which can run many operations in parallel. However, computation is not the only demand.

Additive synthesis has a voracious appetite for control data. If a piece contains 10,000 events (such as a typical orchestral score), each with up to sixteen partials, one needs to compute 160,000 frequency envelopes and 160,000 amplitude envelopes. Where does the control data come from? This is the subject of the next section.

Sources of Control Data for Additive Synthesis

Effective use of any digital synthesis technique, including additive synthesis, depends on having good control data for the synthesis instrument. To create animated sounds with a rich internal development, one drives the synthesizer

with time-varying control data; hence, the control data are also referred to as the *driving functions* of the synthesis instrument. Control data can be obtained from several sources:

1. An interactive app with intuitive *graphical user interface* (GUI). For example, Native Instruments Razor can generate up to 320 partials in real time. It provides both a detailed studio view that presents dozens of knobs (figure 10.12), and a simplified performance view that maps to eight knobs on any MIDI controller. These *macros* enable control of multiple partials using a single knob. Air's Loom additive synthesizer allows GUI control of up to 512 partials.
2. Imported data from another domain that is mapped to the range of synthesis parameters. For example, some composers have taken data describing the shape of mountains or urban skylines and used these curves as control functions. In essence the composer sonifies a data set or a formula. An example is the early computer music piece *Earth's Magnetic Field* (1970) by Charles Dodge.
3. Generation by an algorithmic composition program that embodies composer-specified constraints on musical microstructure.
4. Manual indication in text by a composer using intuitive, theoretical, or empirical knowledge of psychoacoustics.

Figure 10.12 Main screen of the Native Instruments Razor additive synthesizer.

5. Output of an analysis subsystem that analyzes a sampled sound and produces the control data needed to resynthesize it. The data can be edited in order to create transformations of the original sounds. Pioneers of this approach include Trevor Wishart (1988), who used sound analysis as an intermediate stage in transforming vocal sounds for his piece *Vox-5*.

The fifth method requires a subsystem for sound analysis; this is the subject of the next section.

Additive Analysis/Resynthesis

Analysis/resynthesis encompasses different techniques that have a three-step process in common (figure 10.13):

1. A recorded sound is analyzed.
2. The musician modifies the analysis data.
3. The modified data is used in resynthesizing the transformed sound.

The concept of analysis/resynthesis is not predicated solely on additive synthesis. It can also be based on subtractive synthesis, combinations of additive and subtractive synthesis, or other methods. See chapters 36 through 39.

Early experiments in additive analysis/resynthesis were carried out by Harvey Fletcher (of the famous Fletcher-Munson loudness curves) and his associates at Bell Telephone Laboratories (Fletcher, Blackham, and Stratton 1962; Fletcher, Blackham, and Christensen 1963). They used analog equipment. When digital additive methods are used for resynthesis, the system resembles that shown in figure 10.14. The analysis is carried out successively on short segments of the input signal. The process of segmenting the input signal is called *windowing* (discussed in chapter 37). We can think of each windowed segment as being sent through a bank of narrow bandpass filters, in which every filter is tuned to a specific center frequency.

In *oscillator bank resynthesis* (chapter 37), the amplitude of the signal coming out of each filter is measured, and this time-varying value becomes the *amplitude control function* for that frequency range. At the same time, the system calculates control functions corresponding to small frequency deviations by looking at the output of adjacent filters (or *analysis bins*). The frequency and amplitude envelope control functions drive a set of oscillators in the resynthesis stage. In other words, we are using the infor-

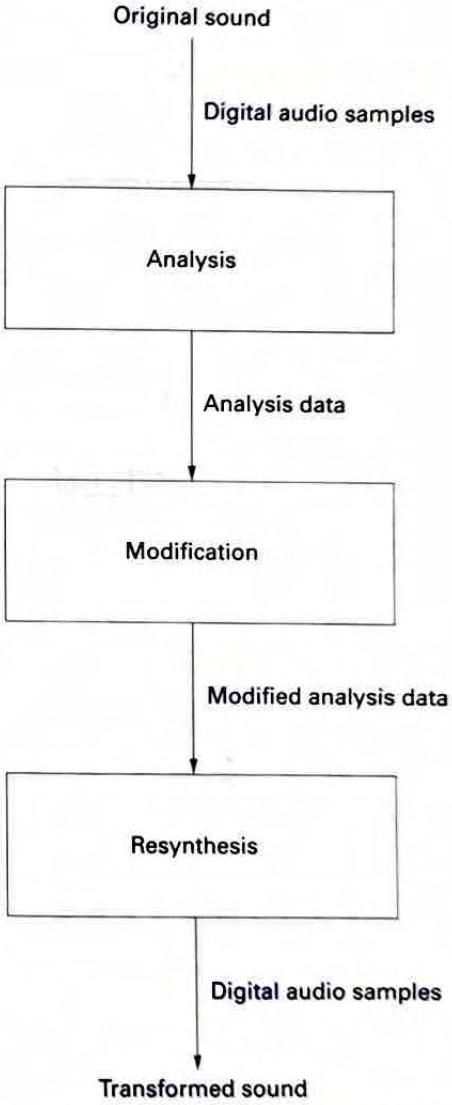


Figure 10.13 General overview of analysis/resynthesis. The modification stage may involve manual edits to the analysis data or modifications via *cross-synthesis* where the analysis data of one sound scales the analysis data from another sound.

mation gleaned from the analysis of an existing sound to create the set of control functions needed to resynthesize that sound additively with sine waves. If the input sound is well modeled as a sum of sine waves, the summed signal generated by the oscillators should sound the same as the original input signal.

Of course, analysis and resynthesis of a sound is not by itself interesting from a musical standpoint; it merely recreates the original sound. In order to create musically interesting effects, we modify the data generated by the analysis. This is the subject of the next section.

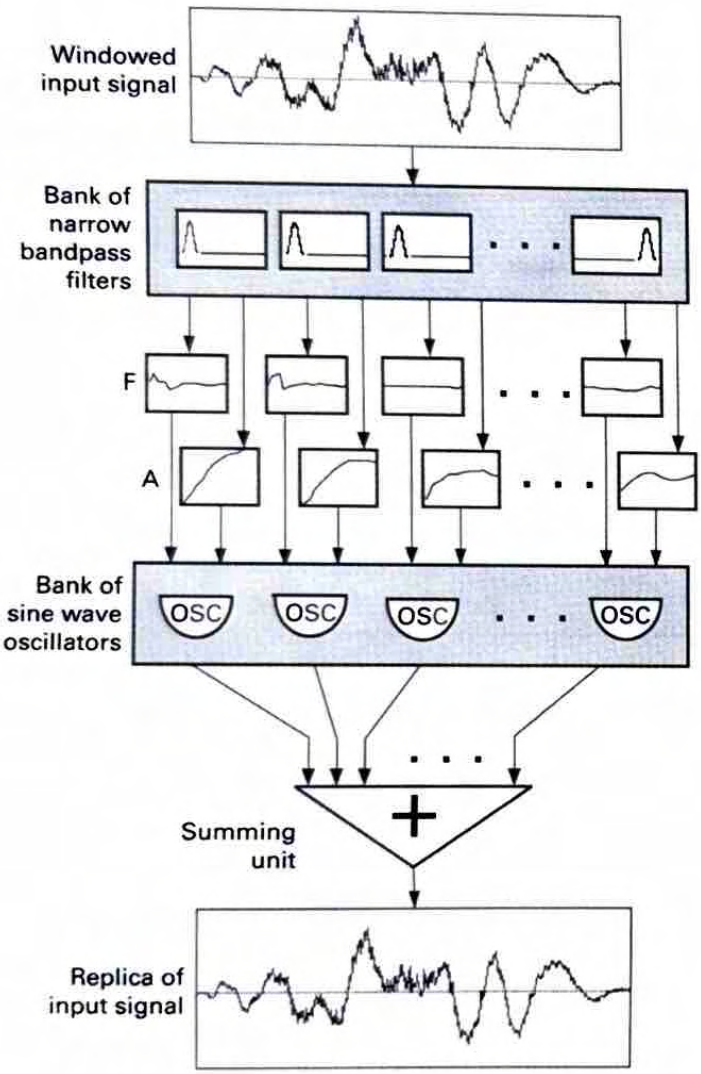


Figure 10.14 Additive analysis/synthesis. A windowed input signal is analyzed by a filter bank into a set of frequency (F) and amplitude (A) envelopes or *control functions* that drive a set of oscillators. If the analysis data is not changed, the output signal should be almost the same as the input signal.

Musical Applications of Additive Analysis/Resynthesis

After the analysis has been performed, musicians can edit the control functions to create variations of the original input signal. Many different effects are possible with this technique, as shown in table 10.1. Three compositions produced in the 1980s stand as pioneering examples of compositional manipulation of analysis data: *Mortuos Plango, Vivos Voco* (1981) by Jonathan Harvey, *Désintegrations* (1983) by Tristan Murail, and *Digital Moonscapes* (1985) by Wendy Carlos.

Table 10.1 Musical transformations using additive analysis/resynthesis

Musical effect	Technique
Variations of recorded sounds	Modify selected frequency and/or amplitude envelopes by editing or multiplication by arbitrary functions.
Spectrum scaling (without time scaling)	Multiply the frequency of all the partials (possibly excepting the fundamental) by a factor n or by arbitrary functions. Because multiplication does not preserve formant structures, vocal and instrumental sounds may lose their characteristic identity.
Spectrum shifting (without time scaling)	Add a factor n or an arbitrary function (without time scaling) to all partials (possibly excepting the fundamental). For small values this preserves formant structures.
Hybrid timbres	Replace some envelopes from one sound with selected envelopes from another sound
Time stretching and shrinking (without pitch shifting)	Adjust the duration of the frequency and amplitude envelopes, or change the *hop size* on playback (see chapter 37).
Timbral interpolation from one sound to another (morphing)	Interpolate over time between the envelopes of two tones.
Spectral filtering	Adjust the amplitudes of selected frequency partials.
Cross-synthesis	Method 1: Use the amplitude envelopes of the partials of one sound to scale the amplitude envelopes of another sound.
	Method 2: Apply the amplitude envelopes from one sound to the frequency (or phase) functions of another sound.
	Method 3: Apply the noise residual from one sound to the quasiharmonic part of another sound (see chapter 37).

In the Harvey piece, the composer analyzed the sound of a large bell. For each sinusoidal component in the resynthesis, the composer substituted the sound of a sampled boy's voice at the appropriate frequency. The voice samples followed the analyzed frequency and amplitude control functions of the chiming bells, creating an eerie effect of a boy/bell chorus. In the composition by Murail, the composer analyzed traditional instrument tones and created synthetic complements to these tones that blend seamlessly with the sounding instruments yet weave out dramatically when the instruments stop.

Désintegrations is a classic example of spectral composition techniques in which the harmonic structure of the work is based on an analysis of instrumental tones (Murail 1991). In *Digital Moonscapes,* Wendy Carlos used analysis data as the inspiration for creating a synthetic orchestra of percussion, string, woodwind, and brasslike timbres, used in an idiomatic orchestral style.

The next section briefly discusses current techniques of sound analysis with additive resynthesis with an emphasis on the data reduction problem. It serves as a prelude to the more detailed treatment in chapter 37.

Methods of Sound Analysis for Additive Synthesis

Many spectrum analysis methods are variations on the basic technique of Fourier analysis of component frequencies. These include *pitch-synchronous* analysis (Risset and Mathews 1969), the *phase vocoder* (PV) (Dolson 1983, 1986, 1989b; X. Serra 1997), *constant-Q* analysis (Petersen 1980; Schwede 1983; Stautner 1983; Loy 2007), and *frequency warping over a Bark scale* (Smith 2011) among others. The practical form of Fourier analysis is the *short-time Fourier transform* (STFT) described in chapter 37. This method can be thought of as analyzing a sampled sound by extracting successive short-duration overlapping segments (shaped by a window function) and applying a bank of filters to the selected segment. The output of each filter is measured, indicating the amplitude and the phase of the spectrum at that particular frequency. A series of these short-time analyses (akin to the frames of a film) constitute a time-varying spectrum. At the core of the STFT is the famous *fast Fourier transform* (FFT), a computationally efficient implementation of Fourier analysis (Cooley and Tukey 1965; Singleton 1967; Moore 1978a, 1978b; Rabiner and Gold 1975).

The PV (Flanagan and Golden 1966; Portnoff 1978; Holtzman 1980; Moorer 1978; Dolson 1983; Gordon and Strawn 1985; Strawn 1985b; X. Serra 1997; Loy 2007; Smith 2011; Lazzarini 2011d; Sethares 2019) deserves special mention here, as it is a popular method of sound analysis/resynthesis that has been distributed with several music software packages. The PV converts a sampled input signal into a time-varying spectral format. In particular, it generates a set of time-varying frequency and amplitude curves. Interesting sound transformations can be achieved by editing and resynthesizing PV data. For example, the PV can be used for *time stretching* or *time shrinking* without pitch change. In this effect, a sound is made longer or shorter without significantly affecting its pitch or timbre. (See chapter 31 for a discussion of various approaches to pitch-time changing.)

Contrary to the expectations of the researchers who invented them (who were searching for efficient coding techniques), sound analysis can generate

an "information explosion" (Risset and Wessel 1982). That is, the analysis data (the control functions) can take up many times more memory space than the original input signal. The amount of data depends partly on the complexity of the input sound, that is, how many sine wave functions are needed to resynthesize it, and partly on the internal data representation used in the analysis program. Using the tracking phase vocoder, for example, a short sound file containing noise that takes up two Mbytes may generate tens of Mbytes of analysis data. This situation mandates some form of data reduction of the control data, the subject of the next section.

Data Reduction Models in Additive Analysis/Resynthesis

Data reduction is important to efficient analysis/resynthesis. Data reduction takes two steps. First, the data—a set of amplitude and frequency control functions—are analyzed. Second, an algorithm transforms the original data into a more compact representation. An important goal of data reduction is to reduce data without eliminating perceptually salient features of the input signal. Another goal in computer music work can be to leave the analysis data in a form that can be edited by a composer. The aim is not simply to save bits; rather, one wants to make it easy to manipulate the data-reduced amplitude and frequency envelopes of the partials.

A large body of research work on data reduction of analysis data is recorded in the literature, including studies by Risset (1966), Freedman (1967), Beauchamp (1969, 1975), Grey (1975), Grey and Gordon (1978), Charbonneau (1981), Strawn (1980, 1985b, 1985c), Stautner (1983), Kleczkowski (1989), Serra (1989), Serra and Smith (1990), Holloway and Haken (1992), and Horner, Beauchamp, and Haken (1993). Because real-time work is so important to musicians, one goal of analysis/resynthesis research is to speed up data reduction processing and facilitate real-time synthesis from reduced data.

Many volumes explore data reduction methods. Here we glance at several techniques that have been applied in computer music: line-segment approximation, principal components analysis, spectral interpolation synthesis, spectral modeling synthesis, and reconstructive phrase modeling. (Also see Goldberg 1989 for a description of the genetic algorithm approach, which has also been applied to synthesis data reduction [Horner, Beauchamp, and Haken 1993].)

Line-segment approximation

Line-segment approximation of the amplitude and frequency control functions eliminates the need to store a distinct value for every sample analyzed.

Instead, the analysis system stores only a set of *breakpoint pairs,* which are time (x-axis) and amplitude (y-axis) points where the waveform changes significantly. Line-segment approximation represents the overall outline of a waveform by storing only the points of maximum inflection (change). In the resynthesis stage the system connects the points, usually by means of straight lines interpolated between the breakpoint pairs.

Initial work with line-segment approximation was done by hand, using an interactive graphics editor to construct functions with four to eight segments each (Grey 1975). A data reduction of a hundredfold was achieved. This manual editing work can also be partially automated, as demonstrated by Strawn (1985b, 1985c). Figure 10.15a shows a perspective plot of the sixteen harmonics of a violin tone, sampled at 25 kHz. Figure 10.15b shows a plot of an approximation to (a) using just three line segments.

Going beyond the storage of line-segment approximations, Beauchamp (1975) developed a heuristic technique for inferring the approximate amplitude curve of all harmonics of a tone from the curve of the first harmonic. For simple periodic tones, Charbonneau (1981) found that even more radical data reduction could be achieved. He used simple variations of a single envelope for all amplitude functions of a given tone. (See Kleczkowski [1989] and Eaglestone and Oates [1990] for refinements of these proposals.)

Principal components analysis

The technique of *principle components analysis* (PCA) has been applied in several analysis/resynthesis systems (Stautner 1983; Sandell and Martens 1992; Horner, Beauchamp, and Haken 1993). PCA breaks down a waveform using the mathematical technique of *covariance matrix* calculation. This results in a set of basic waveforms (the principal components) and a set of weighting coefficients for these basic waveforms. When the components are summed according to their weights, the result is a close approximation of the original waveform.

PCA analysis summarizes the underlying relationships between samples so that the fewest number of components account for the maximum possible variance in the signal. The first principal component is a fit of a single waveform to the entire data set. The second principal component is a fit to the *residual* (the difference, sometimes called the *residue*) between the original and first approximation. The third component is a fit to the residual of the second component, and so on. For further details on PCA, refer to Glaser and Ruchkin (1976).

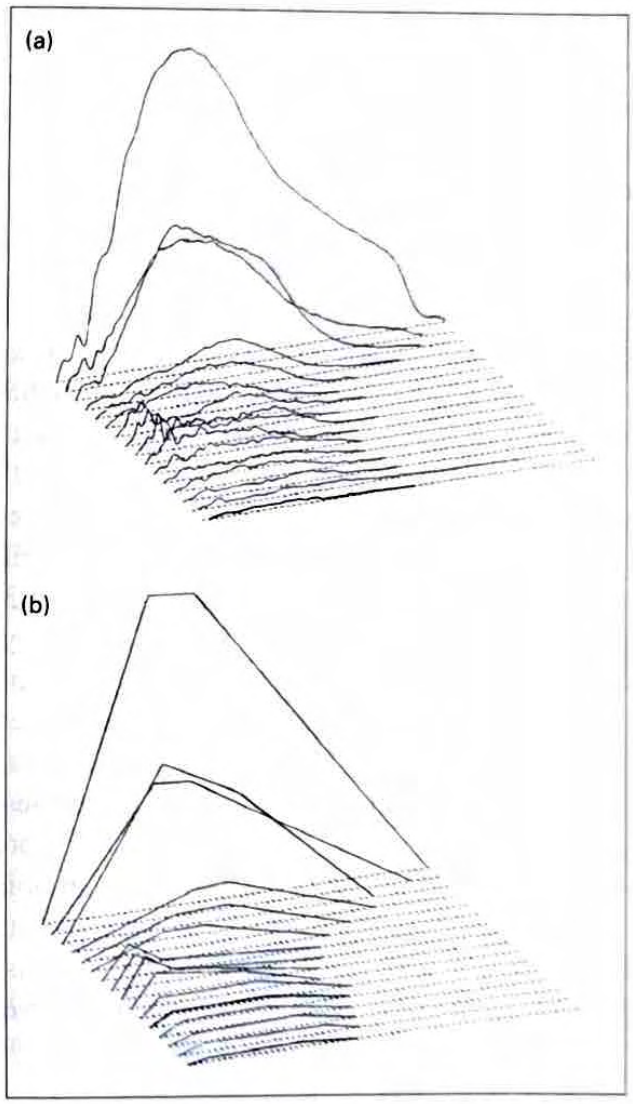


Figure 10.15 Drastic data reduction of analysis data for additive synthesis. Amplitude is plotted vertically, frequency goes from back to front, and time goes left to right. (a) Original spectrum plot of a violin tone. (b) The same violin tone as in (a), approximated with only three line segments per partial.

Spectral interpolation synthesis

Spectral interpolation synthesis (SIS) (Serra, Rubine, and Dannenberg 1990) generates time-varying sounds by interpolating between analyzed spectra. Rather than crossfading between sampled sounds in the time domain (as in multiple wavetable synthesis discussed), SIS starts from analyses of recorded sounds and uses additive synthesis to crossfade between the analyses of successive spectra in the frequency domain. An automatic data reduction algorithm

is necessary to compress the analysis data into a small set of common spectral paths between two successive sounds and a set of ramp functions that describe the transition from one spectrum to the next.

Spectral modeling synthesis

Spectral modeling synthesis (SMS) (Serra 1989; Serra and Smith 1990; Smith 2011) reduces the analysis data to a *deterministic* component (narrowband components of the original sound) and a *stochastic* component. The deterministic component is a data-reduced version of the analysis that models the most prominent frequencies in the spectrum. These frequencies are isolated by a process of *peak detection* in each frame of the analysis and *peak continuation,* which tracks each peak across successive frames. SMS resynthesizes these tracked frequencies with sine waves. This is the same method used in *tracking phase vocoders*, described in chapter 37 (see figure 10.16).

SMS goes beyond this representation, however, by also analyzing the *residual*—the difference between the deterministic component and the original signal. This is the stochastic component, which takes the form of a series of envelopes that control a bank of frequency-shaping filters through which white noise is passed. Thus one can transform the deterministic (sine) envelopes and the stochastic (filtered noise) components separately, if desired (figure 10.17). Noisy components remain noisy, even after transformations (such as filtering) have been applied to them. This stands in contrast with a pure sine wave model, in which transformations (such as time compression/expansion) on noisy components often turn them into sine wave clusters, denaturing their noisy identity.

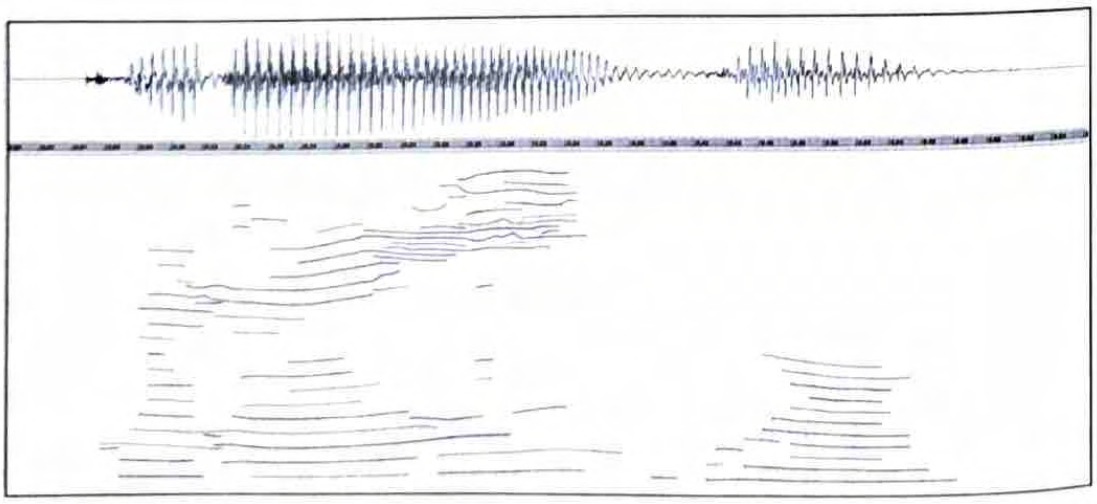


Figure 10.16 Display showing a time-domain waveform (top) of the Italian word *prego* and its tracked partials (*bottom*) in a range to 4 kHz.

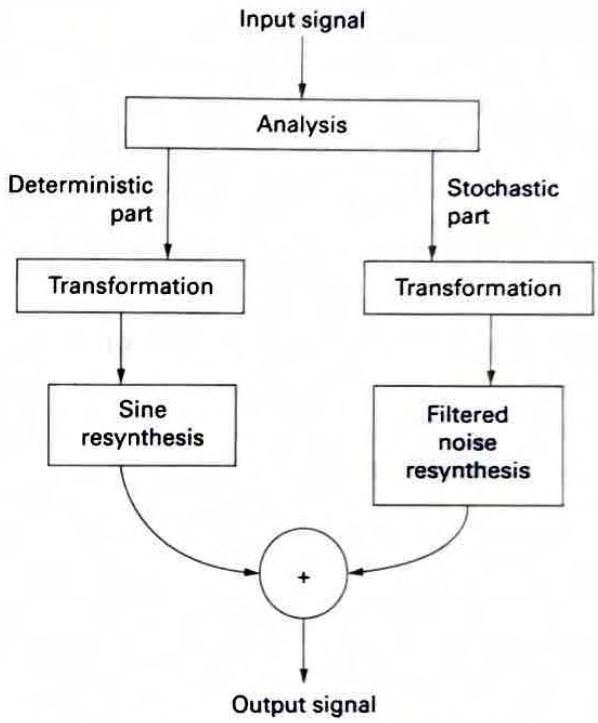


Figure 10.17 Overview of spectrum modeling synthesis. The input signal is divided into a deterministic part and a stochastic part. Each part can be modified separately before resynthesis. (See figure 37.17 for a more detailed view of the analysis stage.)

Efficient algorithms for generating *pseudorandom noise* are well known (Knuth 1973; Rabiner and Gold 1975; Wolfram 2002). Thus using filtered noise results in a tremendous data reduction. Without data reduction, noisy components must be approximated with dozens or hundreds of sine waves. The control functions for these sine waves take up significant storage space, and resynthesis of noise using sine waves is inefficient computationally.

A problem of accuracy left open by SMS is that the filtered pseudorandom noise it uses to reconstruct the stochastic component is generic and not necessarily the same quality of noise as the original source. In the real world, "noise" is the result of complicated turbulences that have an audible character and identity. For some sounds, the approximation by uniform noise leaves room for improvement. The next section addresses this issue.

ATS: Analysis, transformation, synthesis

Analysis, transformation, synthesis (ATS) is based on SMS (Pampin 2004). It goes beyond SMS through its use of psychoacoustic processing both in the sinusoid tracking and in the noise model. In particular, ATS uses perceptual

audio coding methods such as the *signal-to-mask ratio* for each spectral band. (*Masking* occurs when sound A *covers* sound B such that B is inaudible.) The model uses the Bark scale for frequency, in which successive intervals are perceptually equal. The twenty-four steps of the Bark scale correspond to the critical bands of hearing, which are used for the noise analysis phase. (A *critical band* is the range of audio frequencies within which a second tone interferes with the perception of the first tone by auditory masking.) ATS's perceptual model analyzes whether a given sinusoidal peak component masks other components. This information informs the peak continuation process. The residual is determined as in SMS. ATS divides the residual into critical bands. At each analysis frame, the algorithm detects partials in each critical band and distributes the band's noise energy among them. The result is a data-reduced model that is capable of many transformations by the manipulation of a small number of perceptually meaningful parameters. A graphical interface has been developed called ATSH (Pampin et al. 2004). ATS has been implemented in Common Lisp Music, C, Csound, SuperCollider, and Pure Data (Di Liscia 2013).

Reconstructive phrase modeling

Realistic instrument simulations depend on the transitions between notes—the connective tissue of musical expression. These transitions include slurs with varying amounts of portamento, lightly tongued or bowed note transitions, fast runs, and detached hard attacks. Eric Lindemann (2001, 2007, 2009) created an application called Synful Orchestra based on *reconstructive phrase modeling* (RPM) to produce realistic simulations of winds, horns, and strings. The technique relies on a relatively compact database of recorded fragments and phrases of orchestral instruments. These represent many types of articulation and phrasing, encoded in terms of an additive synthesis model data reduced by the method of *vector quantization* (Gersho and Grey 1992). Vector quantization encodes values from a multidimensional vector space (the time-frequency analysis data) into a finite set of values from a discrete subspace of lower dimension. This usually takes the form of a dictionary of elements that can be combined to closely approximate the original data.

The techniques used in Synful Orchestra can be seen as a combination of additive and concatenative synthesis (chapter 23). When Synful Orchestra receives MIDI input it looks at the pitch, velocity, amount of separation or overlap between notes, note duration, volume, pitch, and modulation controls to determine what kind of phrase is being played. The program uses search algorithms to scan a database of musical phrases in real time for fragments that can be concatenated to form this phrase. These fragments represent

transitions between notes, slurs with portamento, lightly tongued transitions, aggressive fast bowing, rapid runs, long sustain regions with graceful vibrato, and noise elements like flute chiffs and bow scratches.

The emphasis is on the connection between notes and the ways that performers move from one note to the next. Notes do not stand alone. The timbre and contour of a note is directly affected by notes that precede and succeed it. As previously discussed, in speech this is referred to as *coarticulation* in that the pronunciation of a syllable depends on the syllables that precede and follow it. In this sense, RPM captures the coarticulation of musical sounds.

Thus a single note can be built from multiple rapidly spliced phrase fragments concatenated in time. Splicing ordinary sampled sounds in this way would create clicks and other artifacts, so Synful Orchestra uses a form of additive synthesis in which samples are resynthesized from combinations of pure sine waves and noise elements. This gives it the ability to stretch, shift, and splice phrase fragments while preserving phrase continuity. Users can control the quality of the resynthesis by adjusting parameters like harmonic parity (emphasis on even or odd harmonics), harmonic tilt (emphasis on higher or lower harmonics), bow noise (for strings), breath noise (for winds), *sforzando* strength (strength of attack), tremolo rate spread (tremolo rate variation among different instruments in a string section), and so on.

Additive Synthesis Based on Machine Learning

Around 2015, Google's Magenta research group started publishing papers on projects based on *machine learning* (ML) methods trained using musical sound examples. (See appendix A for an introduction to machine learning.) The ostensible goal of Magenta research is to advance ML methods in the audio domain. The sonic results validate (or not) the methods.

Early experiments in sound synthesis appended fragments of sounds to form larger units of music (Kalchbrenner et al. 2018; Engel et al. 2019). These methods do not, however, fit neatly into the categories of additive, concatenative, or granular synthesis as defined in this book.

Recent Google work labeled *Differentiable Digital Signal Processing* (DDSP) uses *neural networks* (NNs) to analyze recorded sounds (Engel et al. 2020). The analyzed control functions drive a synthesizer based on the harmonics-plus-noise or SMS model presented earlier in this chapter (Serra 1989; Serra and Smith 1990; Smith 2011). Such a model combines an additive harmonic synthesizer with a subtractive filter for synthesizing noisy parts of the spectrum.

In this work, a NN analyzer extracts the synthesizer parameters (fundamental frequency, amplitude, harmonic distribution) from a training data set of violin performances. Pitch is estimated using the *convolutional representation for pitch estimation* (CREPE) model (Kim et al. 2018; chapter 34). Harmonics follow the contours of the fundamental, loudness is controlled by the amplitude envelope, and spectral structure is determined by the harmonic distribution. Violin performances are reconstructed with good fidelity.

DDSP exists as a code library that integrates neural tools with sound modules based on traditional synthesizer functions. Because the system is modular, it is possible to tap the audio output from each module to find various "decompositions" of the original audio signals. The decomposed parts can be modified and combined with other outputs to create transformations of the original sound. Transformations include loudness and pitch changes, a form of cross-synthesis that converts a singing voice into a violin timbre, and extracting reverberation from a violin performance and applying it to a singing voice.

Colonel, Curro, and Keene (2018) describes an *autoencoding neural network* that drives an additive synthesizer. The system is trained to compress and reconstruct magnitude STFT frames. The *variational autoencoder* (VAE) produces a spectrogram by activating its smallest hidden layer, and a phase response is calculated using real-time phase gradient integration. Performing an inverse STFT produces the audio signal. The algorithm is said to be lightweight in comparison with other audio-producing machine learning algorithms.

Walsh Function Synthesis

Thus far we have discussed analysis/resynthesis as a process based mainly on Fourier analysis with resynthesis based on sine wave summation. The Fourier sine wave approach has a long tradition of research and application stemming from the original theorem that states that for periodic signals, a combination of sine waves of various frequencies can be created that approximate arbitrarily closely the original signal. Mathematical research has shown that unnumerable other waveforms besides sine waves can be used to approximate signals, the so-called wavelet bases.

One such basis is the *Walsh functions,* a family of square waves that can be used to approximate a signal after it has been analyzed by means of the *Walsh-Hadamard transform.* Walsh functions, being rectangular waves, are a kind of *digital domain series* because they take on only two values, +1 and −1 (Walsh 1923).

Figure 10.18 presents the first eight Walsh functions. Just as with the Fourier series and its sine waves, an arbitrary periodic waveform can be approximated as an additive sum of a finite series of Walsh functions. Whereas the Fourier series builds up waveforms out of component frequencies, Walsh synthesis builds up waveforms using functions of different *sequencies*. *Sequency* is defined as one-half the average number of zero crossings per second (zps) (Hutchins 1973).

Figure 10.19 shows a composite waveform derived by summing several Walsh functions. It suggests how sine wave additive synthesis and Walsh

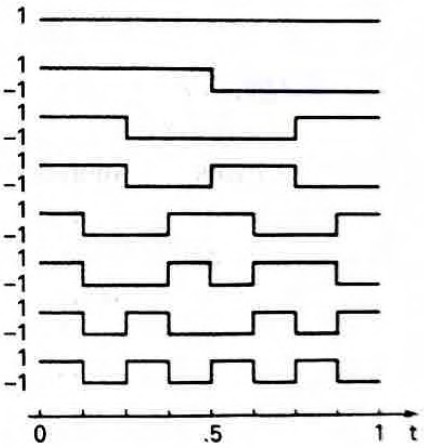


Figure 10.18 The first eight Walsh functions, 0 (*top*) to 7 (*bottom*).

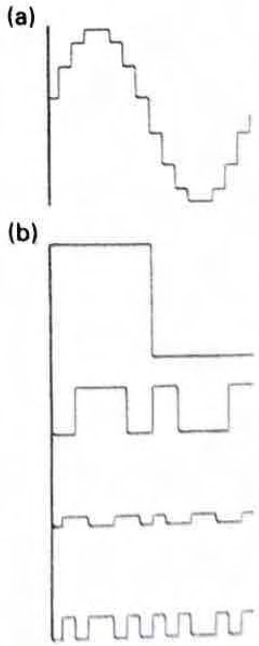


Figure 10.19 Walsh function summation. (a) A simple sine wave approximation built by adding the Walsh functions shown in (b). (After Tempelaars [1977].)

function synthesis are conceptual opposites. That is, the hardest waveform to synthesize in Walsh function synthesis is a pure sine wave. The Walsh approximation to a sine will stay jagged until a very high number of sequency terms are used. Any jaggedness gives an unsinusoidal and generally objectionable quality. By contrast, in sine wave synthesis, the hardest waveforms to synthesize are those with a rectangular corner, such as a square wave. Recall that figure 10.5e depicts a quasi–square wave constructed by summing 101 sine waves.

The main advantage of Walsh functions in digital sound synthesis is their rectangular shape, which can be computed at high speed by inexpensive digital circuits. A conceptual disadvantage of Walsh function synthesis as against sine wave synthesis is that individual Walsh functions are not associated with specific harmonics as they are in sine wave additive synthesis. One can, however, pass mathematically from the Fourier (frequency) domain to the Walsh domain (Tadokoro and Higishi 1978). Thus, one can specify a sound in terms of the addition of various frequency components (partials) and then transform this specification into a set of parameter values for a Walsh function synthesizer. Moreover, natural sounds can be sampled and transformed into the Walsh domain using the Walsh-Hadamard transform and resynthesized using the *fast Walsh transform* (FWT) (Hutchins 1973, 1975).

A number of music synthesis operations have been redesigned for Walsh signal processing circuits. For example, Hutchins (1973) designed an envelope generator using Walsh function circuits. Rozenberg (1979) and Hutchins (1975) showed how to realize amplitude modulation, subtractive synthesis, frequency modulation, frequency shifting, and reverberation—all in the Walsh domain.

Despite the potential of Walsh function synthesis, only a few experimental devices based on this technique have been built (Hutchins 1973, 1975; Insam 1974). None are commercially available. This is probably due to the fact that sine additive synthesis is now cheap (any laptop can easily handle it), so the economic advantage of Walsh function circuits has diminished. The weight of accumulated research in sine wave methods has also contributed to their continuing presence.

11 *Multiple Wavetable Synthesis*

Wavetable Cross-Fading or Vector Synthesis

Wavestacking

Multiple wavetable synthesis encompasses two simple yet sonically effective methods: *wavetable cross-fading* and *wavestacking*. These are not the only synthesis methods that can use multiple wavetables; indeed other methods (FM, for example) can be configured to do so. We distinguish the techniques discussed here because the use of multiple wavetables is the sine qua non of multiple wavetable synthesis.

Multiple wavetable synthesis is a central technique in popular synthesizers. It creates lively and animated cross-faded and stacked hybrids. Blends of sampled and synthetic waveforms can be combined to create exotic time-varying sounds.

Horner, Beauchamp, and Hakken (1993) developed the technique they called *multiple wavetable synthesis*. It is perhaps best classified as a variant of additive analysis/resynthesis (presented in chapter 37). It also can be viewed as an instance of the wavestacking method presented here, in which the wavetables are sums of sinusoids derived from an analysis and data reduction stage.

Wavetable Cross-Fading or Vector Synthesis

As chapter 6 explains, in fixed-waveform synthesis, a digital oscillator scans repeatedly through a wavetable that has been previously filled with a single waveform. This creates a static timbre because the waveform repeats without variance over time. By contrast, wavetable cross-fading is a direct way to generate time-varying timbres. Instead of scanning a single wavetable repeatedly, the oscillator *cross-fades* between two or more wavetables over the course of an event. That is, the event begins with waveform 1, and as 1 begins to fade away, waveform 2 fades in, and so on. Figure 11.1 portrays the cross-fading process. Wavetable cross-fading is the core of the technique called *vector synthesis* or *wave sequencing* (by Sequential Circuits, Korg, and Yamaha), *L/A* or *linear arithmetic* synthesis (Roland), *dynamic spectral wavetable synthesis* (Waldorf PPG Wave), or *wavetable morphing* (XferRecords Serum).

Wavetable cross-fading creates sounds that mutate or morph from one source to another over time. Figure 11.2 depicts an instrument for wavetable cross-fading that scans through four wavetables.

The first commercial synthesizer to implement wavetable cross-fading was the eight-voice Sequential Circuits Prophet VS, introduced in 1985, which could cross-fade between four waveforms. The Korg Wavestation, designed by some of the same team who made the Prophet VS, reached the market in 1990. It was reintroduced as a software synth in 2010 (figure 11.3). Rob Papen Vecto (2019) provides vector synthesis as a software plug-in with over a thousand presets. Waldorf PPG Wave 3.V is a software plug-in that realizes a variant of vector synthesis.

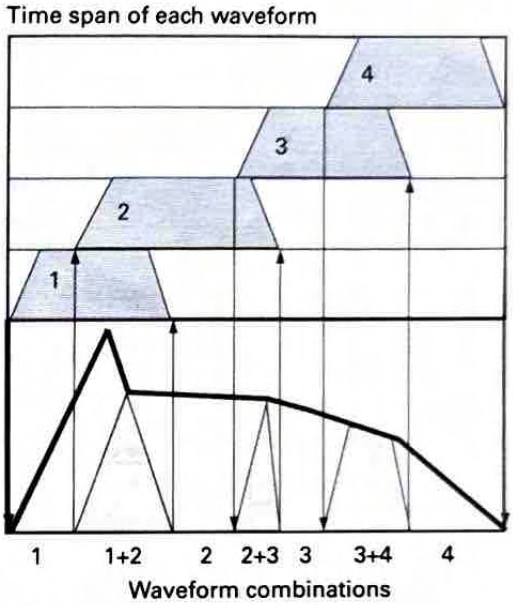


Figure 11.1 Wavetable cross-fading. The bold outline traces the amplitude of a note event. Four waveforms cross-fade over the span of the event. The numbers at the bottom indicate the sequence of waveforms alone and in combination. Each region indicated at the bottom represents a separate timbre; thus the event cross-fades through seven timbres.

In 2020, Korg introduced the 64-voice Wavestate synthesizer, its latest incarnation of vector synthesis technology (figure 11.4).

Wavetable cross-fading can be automatic (triggered by a note event and modulated by a cross-fade function), or it can be manually controlled by rotating a joystick. Some synthesizers let users specify an arbitrary number of waveforms (e.g., 128) to cross-fade during a single event, a process called *wave sequencing*. This creates lively animated sweeps of electronic sound, with radical changes in timbre but also shifts in perceived pitch because the content of the waveform is open. This type of sonic behavior is unlike any acoustic instrument.

A variant of wavetable cross-fading—linear arithmetic, also called L/A synthesis—was introduced in 1987 in the Roland D-50 and became one of the defining sounds of its era, starring in many hit records and soundtracks. L/A synthesis made a comeback in 2017 in the Roland D-05 synthesizer. L/A synthesis grafts sampled attack transients onto the front of a composite sound consisting of a subtractive synthesis waveform (sawtooth or pulse wave through a filter) mixed with a sampled waveform that can be continuously looped.

An extreme example of wavetable cross-fading is the Piston Honda morphing wavetable oscillator, a Eurorack module by Industrial Music Electronics (figure 11.5). The Piston Honda provides a bank of 512 wavetables organized as an 8-by-8-by-8 cube (X, Y, Z). The indices X, Y, and Z can be selected manually with three faders or, as is more typical, modulated using low-frequency

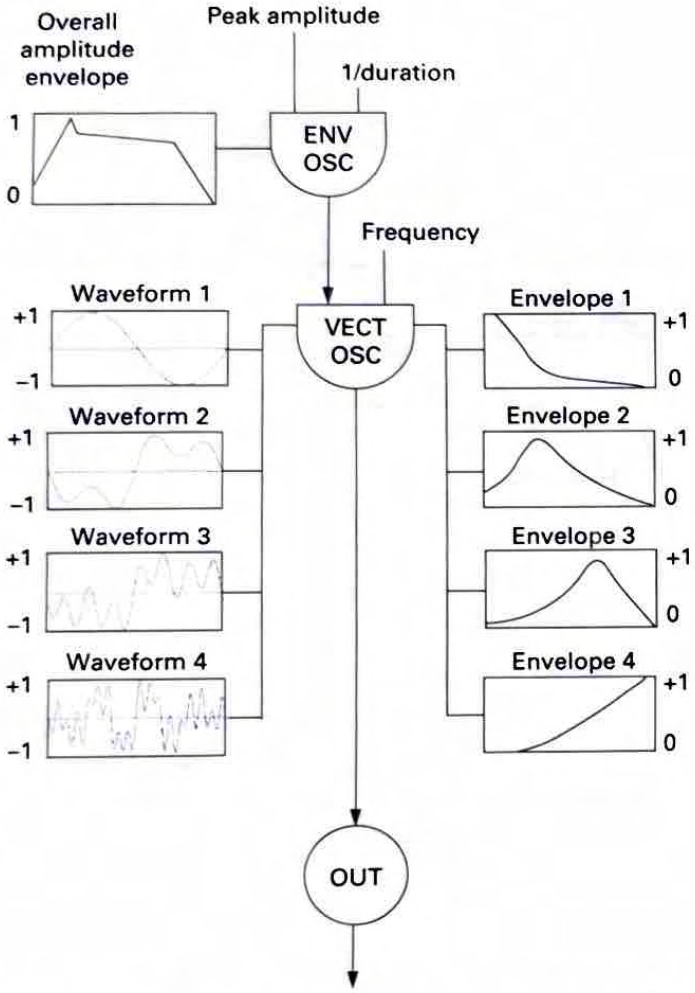


Figure 11.2 Wavetable cross-fading (vector synthesis) instrument using four wavetables. Each envelope on the right applies to a wavetable on the left.

Figure 11.3 Korg Wavestation vector synthesizer in its software incarnation.

Figure 11.4 Korg Wavestate synthesizer.

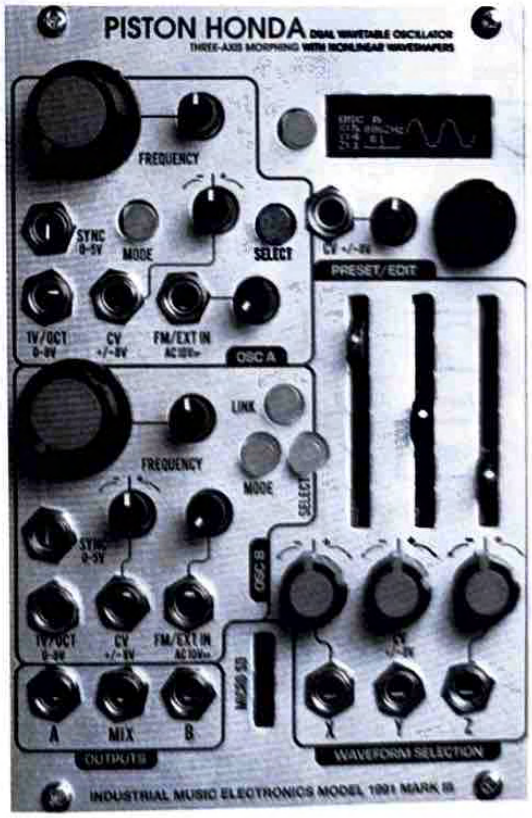


Figure 11.5 Piston Honda morphing wavetable oscillator.

oscillators (LFOs). Depending on user settings, the oscillator either morphs smoothly from one wavetable to another or steps abruptly between wavetables. Highly animated contrasts in timbre are typical of its sound.

Wavestacking

Wavetable stacking or *wavestacking* is a simple and effective variation on additive synthesis. It was implemented in the hugely successful Korg M1 synthesizer (over 250,000 sold from 1988 to 1995). The M1 was re-released as the iM1 software synthesizer for iPad in 2015 with a sound library of more than 3,000 sounds.

In wavestacking, each sound event results from the addition of several waveforms (typically four to eight). Wavestacking is done differently from

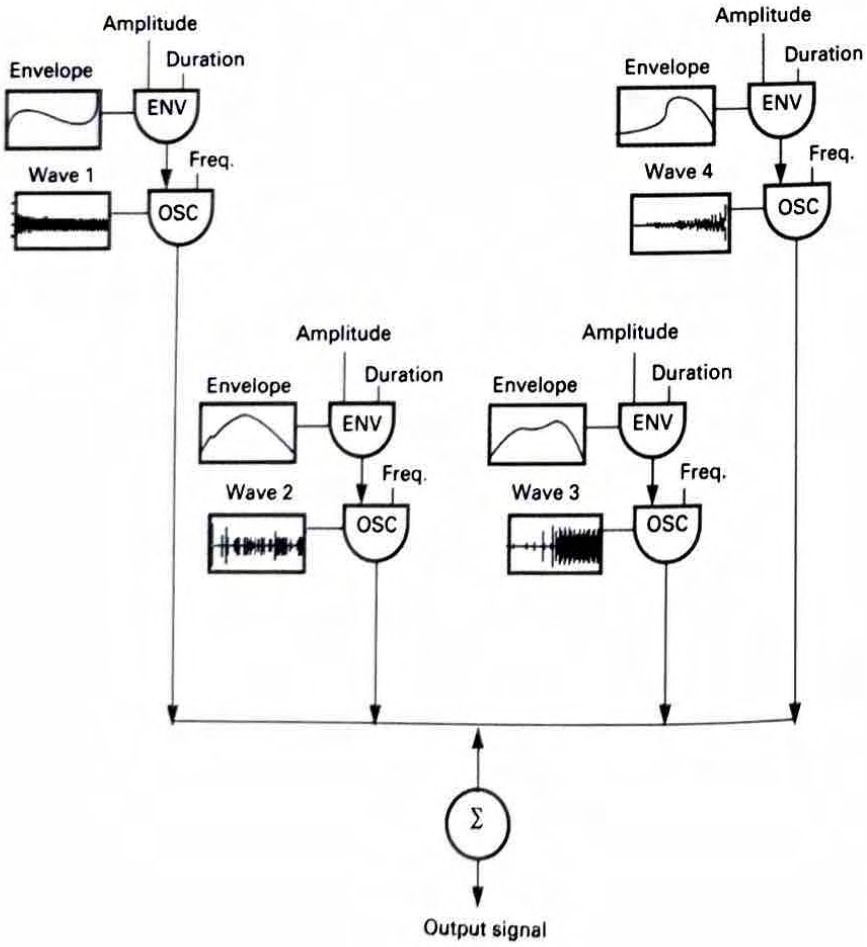


Figure 11.6 Wavetable stacking. The signals from four oscillators are added together. Notice that the wavetables contain not simple periodic functions but long sampled sounds.

classical additive synthesis, which sums sine waves. In wavestacking, each waveform can be a complicated signal, such as a sampled sound (figure 11.6). By layering several sampled sounds, one can create hybrid timbres like saxophone/flute or violin/clarinet. Each waveform in the stack has its own amplitude envelope, so sounds fade in and out of the stack in the course of a sound event. When four to eight complex waveforms are stacked, deep and rich hybrid textures can be created for each sound event.

Wavestacking is implemented by storing a library of waveforms and using table-lookup oscillators to scan them. Each waveform's envelope must be scaled by a factor of $1/n$, where n is equal to the number of stacked waveforms, to avoid numerical overflow. That is, the sum of all the waveforms should be within the quantization range of the synthesizer.

12 *Wave Terrain Synthesis*

Terrains and Trajectories

Generating Predictable Waveforms from Wave Terrains

- **Periodic Trajectories**
- **Time-Varying Trajectories**
- **Time-Varying Wave Terrain**

Extensions of Wave Terrain Synthesis

Implementations of WT Synthesis

Many synthesis techniques start from the fundamental principle of wavetable lookup, explained in chapter 6: a pointer scans a two-dimensional wavetable plotting amplitude on the x-axis versus time on the y-axis. The index to the position in the table is incremented at each sample period. What happens if we extend the principle of wavetable lookup to the scanning of a three-dimensional (x, y, z) surface, such as the undulations of mountains and valleys in an Alpine landscape? As the wave surface is scanned, the height at a given x, y position is captured as samples, in z values, for the waveform.

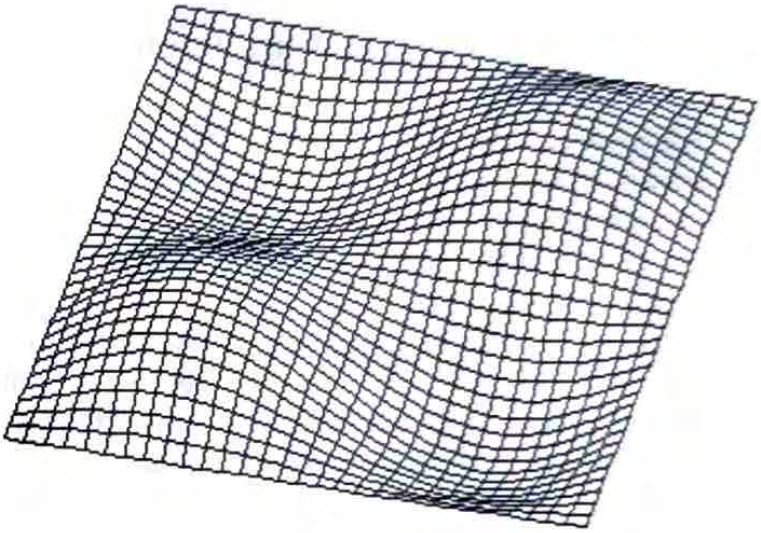
We call such a surface a *wave terrain* (WT). In the rest of this chapter we refer to wave terrain synthesis as WT. Several researchers, including Rich Gold in consultation with Leonard Cottrell (Bischoff, Gold, and Horton 1978), Mitsuhashi (1982a), and Borgonovo and Haus (1984, 1986) were the first to explore the possibilities of techniques that scan a wave terrain using two indexes. Comajuncosas (2000) and Mikelson (2000b) have described the technique in the context of Csound. The most comprehensive documents on wave terrain synthesis are the masters and doctoral theses by Stuart James (2005, 2015).

In general, the sounds WT produces are synthetic in character. Initial musical experimentation with WT focused on drones with continuously mutating timbres. James (2005) showed how WT could—under certain conditions—generate equivalent results to additive synthesis, vector synthesis, amplitude and ring modulation, phase distortion, frequency modulation, and waveshaping synthesis.

Terrains and Trajectories

A wave terrain can be plotted as a function *wave*(x,y) etched on a three-dimensional surface (figure 12.1). In this case, the z-point, which is the height of the surface at each point, represents a waveform amplitude value for a given pair (x, y). The surface stored in such a table can be a *closed-form function of two variables*, and thus the technique has also been called *two-variable function* synthesis (Borgonovo and Haus 1986).

Unlike a traditional wavetable, where the pointer scans from left to right and then repeats, a scan over a wave terrain can take an arbitrary path over the surface. It may repeat or not. A scan over the terrain is called a *trajectory* (James 2005). Early literature called this an *orbit*, but this connotes an elliptical function, when in fact the trajectory can consist of any sequence of points on the wave terrain, including a random walk.



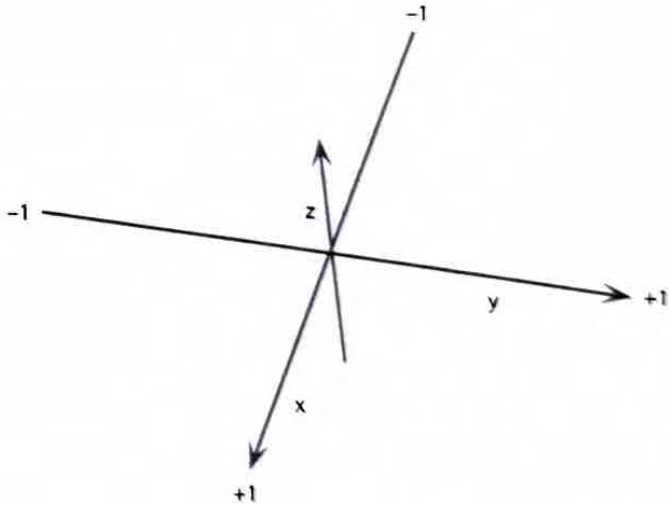


Figure 12.1 A waveform terrain is a three-dimensional surface. The height (*z*-axis) of the terrain represents the waveform value.

Pitched tones are generated by periodic repetition of trajectories. The frequency of repetition correlates to a fundamental frequency. Amplitude is correlated with the evolution of maxima and minima in the waveform.

No movement or a flat terrain will result in silence. On an undulating terrain, the frequency of the undulations determines the spectral content, and the depth of the undulations determines amplitude. The more jagged the angles of the wave terrain, regardless of amplitude, the more high frequencies will be generated. In contrast, smooth terrain features correlate with a lack of high-frequency content. However, the precise spectral content of a terrain depends on the frequencies at which the terrain is scanned.

Generating Predictable Waveforms from Wave Terrains

For musical purposes, any three-dimensional surface can serve as a wave terrain, from a tightly constrained mathematical function to an arbitrary topographical projection. Indeed one musical project used the Earth's global Digital Elevation Model database as a terrain (Thibault and Gresham-Lancaster 1992, 1997).

James (2005) distinguished between *discrete* and *continuous* terrain functions. Discrete terrains are stored in a table that contains values sampled at discrete points on the surface of the terrain. Continuous terrains can be computed on the fly by mathematical functions to an arbitrary precision. In practice, most terrains are discrete.

Scientific investigations have focused on wave terrains generated by simple mathematical functions. As in techniques like frequency modulation and waveshaping, the advantage of using simple functions is that it is possible to analytically predict exactly the output waveform and spectrum generated by a given wave terrain and trajectory. Mitsuhashi (1982c) and Borgonovo and Haus (1986) devised smooth mathematical wave terrain functions in the range $[-1 \le x \le 1, -1 \le y \le 1]$. The following conditions must be met in order to predict the output waveform:

1. Both the x and y functions and their first-order partial derivatives are mathematically continuous over the terrain.
2. Both the x and y functions are zero on the boundaries of the terrain.

The second property ensures that the functions and their derivatives are continuous when the trajectory skips from one edge of the wave terrain to another edge. Such a skip is analogous to the right-to-left wraparound that occurs in one-index wavetable scanning.

Of course, if the trajectory is kept within the edges of the wave terrain, condition 2 can be ignored.

The terrain depicted in figure 12.1 satisfies conditions 1 and 2 and is defined by the following equation:

$$\text{wave}(x, y) = \sin(\pi/2 \times x) \times \sin(\pi/2 \times y)$$

Other mathematical functions investigated include piecewise functions (Mitsuhashi 1982c), polynomials (Borgonovo and Haus 1984, 1986; Mikelson 2000a), fractal and iterative maps (Mikelson 2000a), additive and Chebyshev functions (Nelson 2000), elliptic functions (Cafagna and Vicinanza 2002) and dynamical systems of equations (Boulanger et al. 2000; Mikelson 2000a).

James (2005) devoted an appendix to a catalog of mathematically derived WTs. These include the following:

- curves that are periodic within the range $-1 \leq z \leq 1$ and continuous (figure 12.2a)
- curves that are periodic within the range $-1 \leq z \leq 1$ but are discontinuous (figure 12.2b)
- curves that include undefined values (figure 12.2c)
- curves that generally exhibit natural tendencies toward infinity (figure 12.2d)
- wavetable and tiled forms (figure 12.2e)
- other mathematical curves (figure 12.2f)

Figure 12.2 shows examples of each type in the catalog.

Periodic Trajectories

The output signal generated by WT depends on both the terrain and the trajectory. The trajectory can be any sequence of points, such as a straight or curved line across the surface, a zig-zag pattern, a random walk, a sinusoidal function, or an elliptical function generated by sinusoidal terms in both the x and y dimensions. If the trajectory repeats itself (i.e., is periodic), so will the output signal. Figure 12.3 shows a periodic elliptical trajectory defined by the functions:

$$x = 0.5 \times \sin(8\pi t + \pi/5)$$

$$y = \sin(8\pi t)$$

Figure 12.4 shows an example of another periodic trajectory that loops around the terrain.

Time-Varying Trajectories

When the trajectory is fixed, the resulting sound is a fixed waveform characterized by a static spectrum. By changing the trajectory over time one can generate a time-varying waveform (figure 12.5). Trajectories can be controlled in real time using input devices or programmed using automation (James 2005).

Time-Varying Wave Terrain

One can imagine an implementation in which the trajectory is fixed but the wave terrain is time-varying. In this case the wave-scanning process is

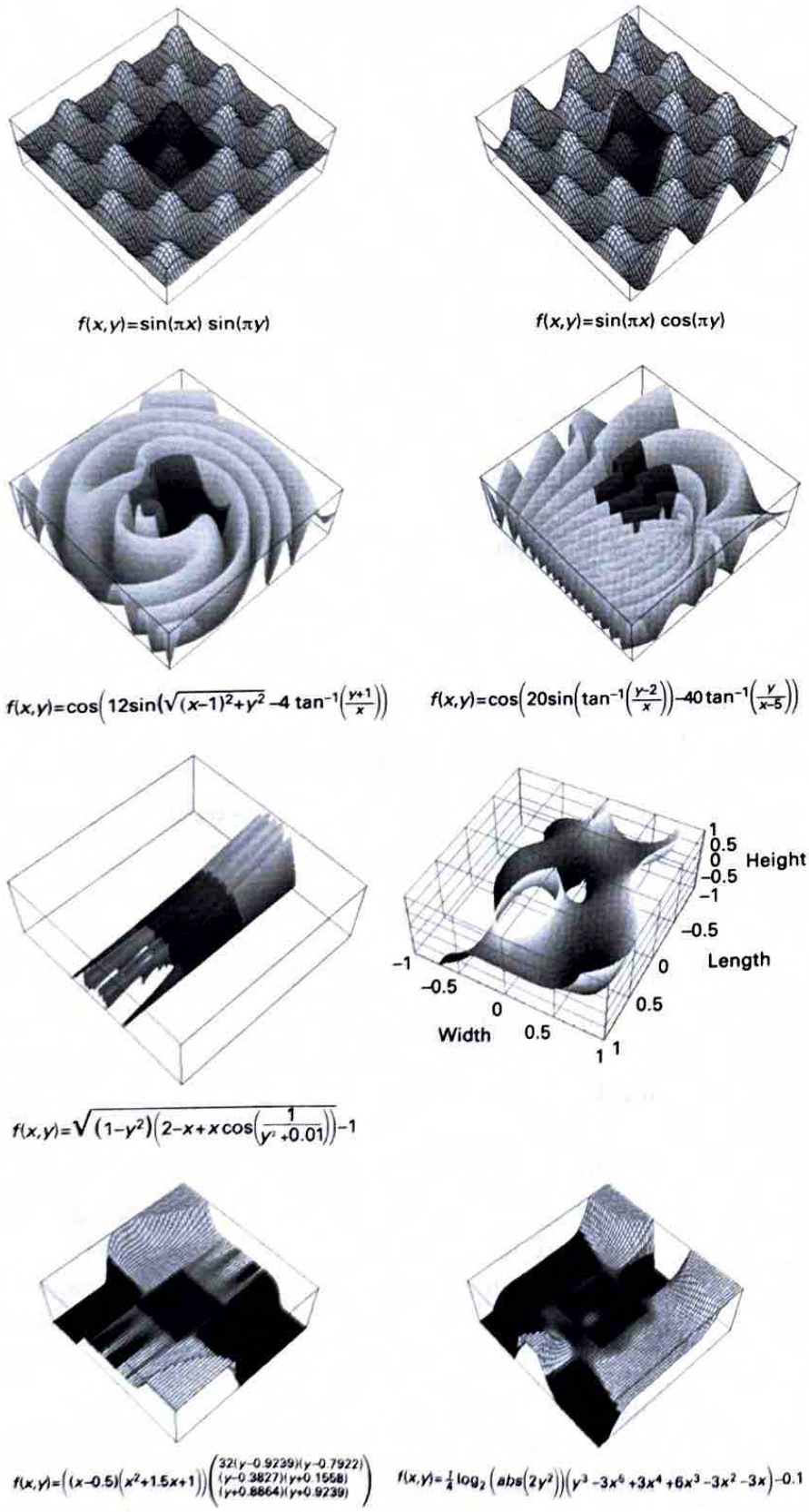


Figure 12.2 Wave terrains defined by mathematical functions. After James (2005).

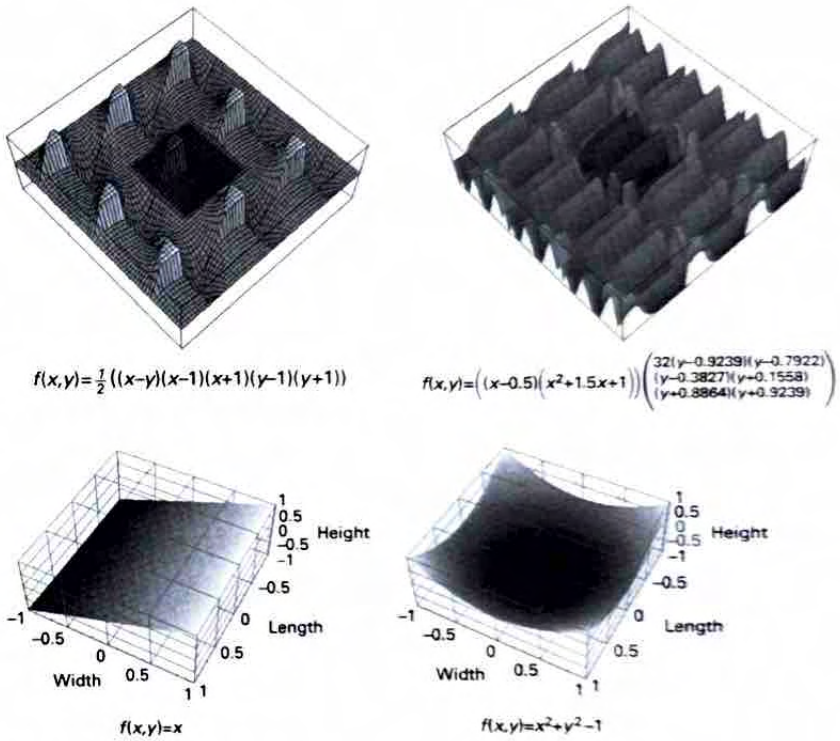


Figure 12.2 (continued)

equivalent to tracing the curves of an undulating surface, like undulations on the surface of the sea. In essence, this is the idea behind *scanned synthesis*, covered in chapter 18.

Extensions of Wave Terrain Synthesis

Basic WT synthesis is conceptually simple. However, it can be extended in many ways. The most intrepid researcher, James (2005), explored a panoply of methods to generate wave terrains: video capture, noise functions, recurrence plots, and spline surfaces. He also explored matrix operations including mappings from colored terrain surfaces, convolution processing (blurring, sharpening, and embossing), mapping sounds to different spatial locations, and video feedback. For his PhD research, the same author turned the interface of WT synthesis into a generator of control data for a spatialization scheme in which different spectral bands follow independent spatial paths (James 2015).

Zabetian (2018) created a WT synthesis instrument with interactive MIDI polyphonic expression (MPE) using a ROLI Seaboard to enhance expressivity.

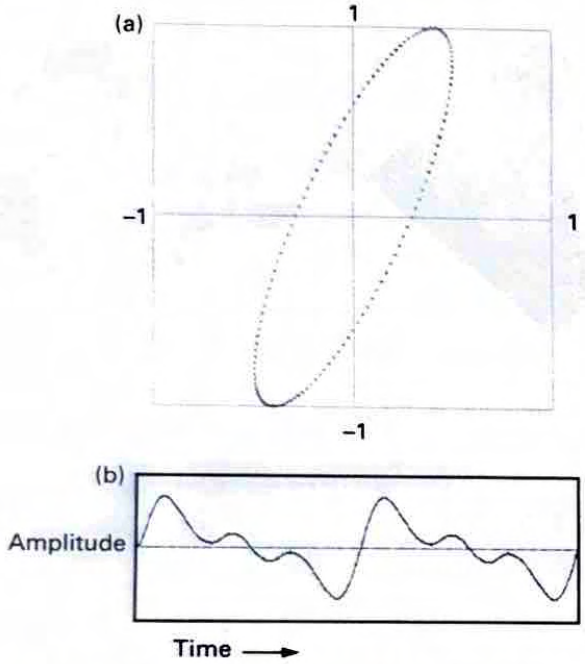


Figure 12.3 Elliptical trajectory and resulting signal. (a) Plot of the trajectory. Both the x and y dimensions vary from −1 to +1. (After Borgonovo and Haus [1986].) (b) Waveform generated by the elliptical trajectory over the wave terrain defined in equation 1. (Note: This waveform is an approximation redrawn from Borgonovo and Haus [1986].)

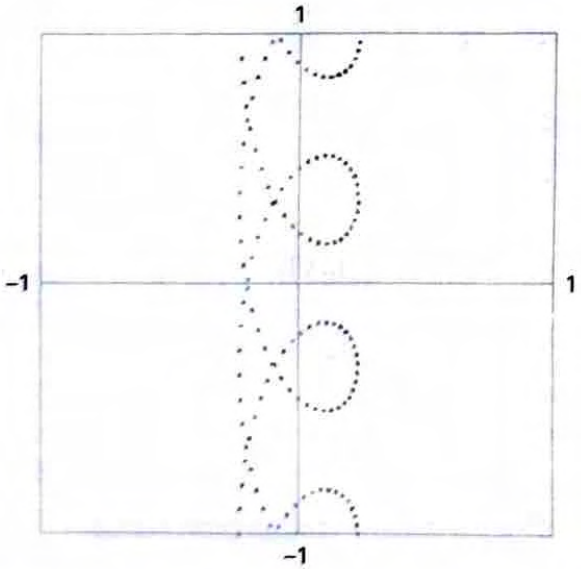


Figure 12.4 Looping trajectory. (After Borgonovo and Haus [1986].)

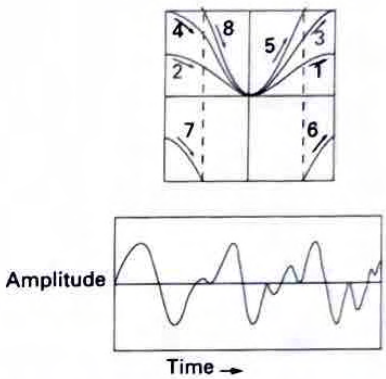


Figure 12.5 Aperiodic trajectory and resulting signal. (Top) Plot of trajectory trajectories in eight passes through the wave terrain. (Bottom) Notice the time-varying waveform. (After Mitsuhashi [1982a].)

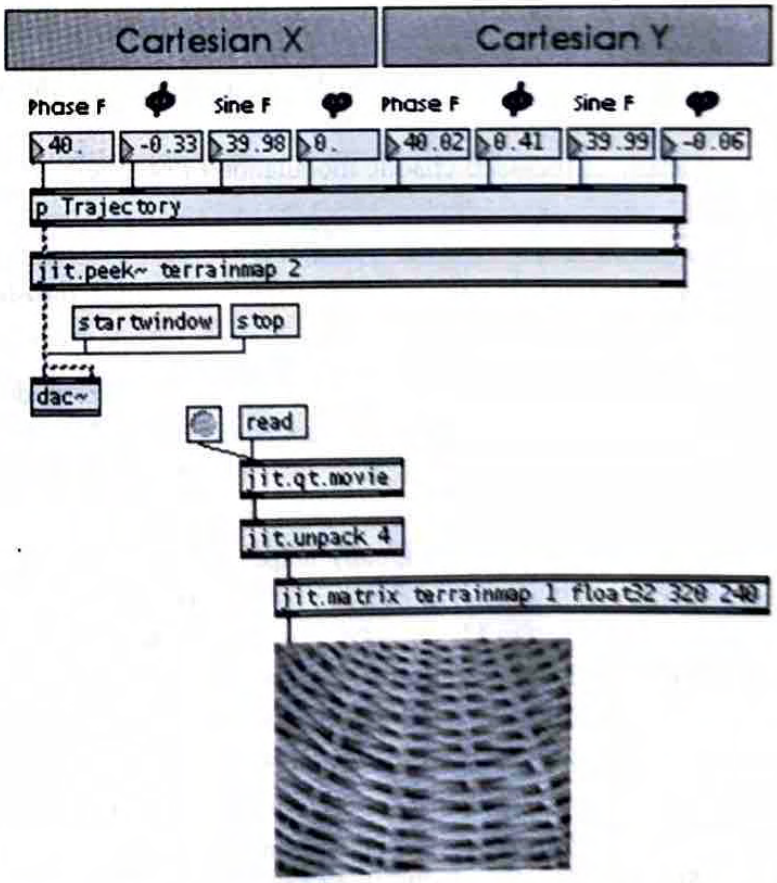


Figure 12.6 A simple WT synthesizer using the Jitter library in Max (after James [2005]). In this case, the WT used is the woven basket pictured at the bottom. All that is required is a table lookup with an interpolation routine, procedures that are performed by the jit.peek~ object.

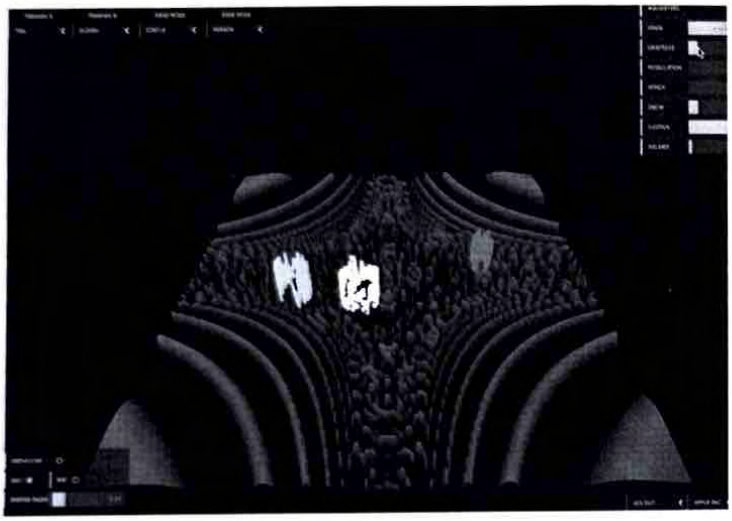


Figure 12.7 WaveTerrain Synth by Aaron Anderson. Here three small trajectories scan the surface of a complex terrain.

(See chapter 51 for more on MPE.) Combined with graphic displays, this enabled interactive control and manipulation of trajectories and terrains. Other extensions included chaotic modulation of both terrains and trajectories.

Implementations of WT Synthesis

Some implementations of WT synthesis do not provide a display. One can, for example, explore WT using Csound code. However, the advantages of visualizations became clear in searching for a "sweet spot" or a particularly smooth or jagged terrain. A user study by Zabetian (2018) showed that graphical visualization greatly helped users to understand the technique.

WT synthesis is represented by the wterrain opcode in Csound and the terrain~ object in Max and Pd. Figure 12.6 shows an implementation in Jitter using the jit.peek~ object (James 2005).

Aaron Anderson (2017) developed the Wave Terrain Synth (figure 12.7), which allowed for multiple simultaneous trajectories (Anderson 2018). Wave Terrain Synth features a large set of terrains and reading methods. It can also convert both images and sounds into terrains. The app is controllable by MIDI, with note velocity tied to trajectory size.

13 *Granular Synthesis*

Theory of Granular Synthesis

Granular Synthesis: Background

Simple Grain Generator Instrument

Parameters of Granular Synthesis

- **Per-Grain Parameters**
  - **Envelope**
  - **Duration**
  - **Effect of grain duration**
  - **Waveform**
  - **Granulation of sampled sounds**
  - **Frequency (pitch shift for samples)**
  - **Spatial position**
  - **Synchronicity (degree of periodicity)**
  - **Intermittency (degree of irregular emission)**
- **Density and Fill Factor**

High-Level Granular Organizations

- **Time-Frequency Projections**
- **Clouds**
- **Streams**
- **Pitch-Synchronous Granular Synthesis**

Sprays
Scrubbing
Micromontage
Physical and Biological Models
Abstract Generative Models
Summary of Higher-Level Organization

Granular synthesis builds up acoustic events from combinations of brief sound particles or *grains*. A single grain serves as a building block for sound objects. By combining thousands of grains over time, we can create animated sonic textures. This chapter introduces the theory, history, and parameters of the technique. A critical factor in synthesis is the choice of how to model the high-level organization of the grains. This model has a strong influence on the types of musical structures that can be produced by granular synthesis.

Theory of Granular Synthesis

A grain of sound is an acoustic event with a duration near the threshold of human auditory perception (figure 13.1) This threshold is typically between one-thousandth of a second and one-tenth of a second (from 1 to 100 ms). These are the shortest durations in which differences in duration, frequency, spectrum, amplitude, amplitude envelope, and spatial position can be perceived (Whitfield 1978; Meyer-Eppler 1959; Winckel 1967; Buser and Imbert 1992). Below 1 ms in duration, no matter how they are varied, sonic events are sensed as sub-symbolic clicks. On a granular time scale, however, we can still perceive differences in sound properties. Thus, if desired, each grain can be unique.

Granular representations are a general and useful way of viewing complex sound phenomena—as constellations of elementary units of energy, each unit bounded in time and frequency. Such *microsonic* representations are common

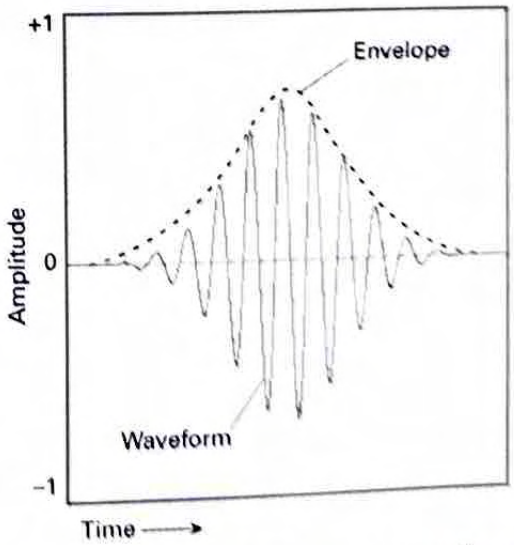


Figure 13.1 Portrait of a grain. An audio waveform is shaped by a short-duration amplitude envelope. The envelope is bell shaped with a smooth attack and decay.

inside synthesis and signal processing algorithms, although there are many different terms for similar phenomena, including *quantum* (Gabor 1946, 1947), *Gaussian elementary signal* (Helstrom 1966; Bastiaans 1980), *short-time segment* (Schroeder and Atal 1962), *short-time weighting function* (Flanagan 1972), *windowed segment* (Arfib 1991; Harris 1978; Nuttall 1981), *sliding window* (Bastiaans 1985), *window function pulse* (Bass and Goeddel 1981), *wavelet* (Kronland-Martinet and Grossmann 1991), *formant wave function* (FOF) (Rodet 1980), *VOSIM pulse* (Kaegi and Tempelaars 1978), *wave packet* (Crawford 1968), *toneburst* (Blauert 1983; Pierce 1990), *tone pulse* (Whitfield 1978), *pulsar* (Roads 2001b), *grainlet* and *trainlet* (Roads 2001a), *tone pip* (Buser and Imbert 1992), and even the *timbron* (Chen 2018). All can be described as granular representations of musical signals, operating at the micro time scale.

The grain is an apt representation for sonic microstructure because it combines time-domain information (starting time, duration, envelope shape, and waveform shape) with frequency-domain information (the frequency of the waveform inside the grain and the spectrum of the waveform inside the grain).

Granular Synthesis: Background

Atomistic views of sound as particles can be traced to antiquity (Lucretius [55] 1951). In 1616 the Dutch scholar Isaac Beekman (1588–1637) proposed a *corpuscular* theory of sound (Beeckman [1604–1634] 1953; Cohen 1984). Beeckman believed that any vibrating object, like a string, cuts the surrounding air into spherical corpuscles of air that are projected in all directions by the vibration. When these corpuscles impinge on the eardrum, Beeckman theorized, we perceive sound. While this theory is not true scientifically, it paints a colorful metaphor for the perception of granular synthesis. Refer to Roads (2001a) for a history of wave versus particle theories of sound.

Centuries later, the notion of a *quantum* or granular approach to sound was proposed by the British physicist Dennis Gabor in a pair of brilliant papers that combined theoretical insights from quantum physics with practical experiments (1946, 1947). Gabor theorized that all sound could be represented by a time-frequency grid that we now call the *Gabor matrix* (figure 13.2). This hypothesis was verified mathematically by Bastiaans (1980, 1985).

In the 1940s Gabor constructed a sound granulator based on an optical recording system adapted from a film projector. He used this to make experiments in *time stretching* and *time shrinking* with *pitch shifting*—changing the pitch of a sound without changing its duration and vice versa. (See chapter 31 for a discussion of pitch-time changing.)

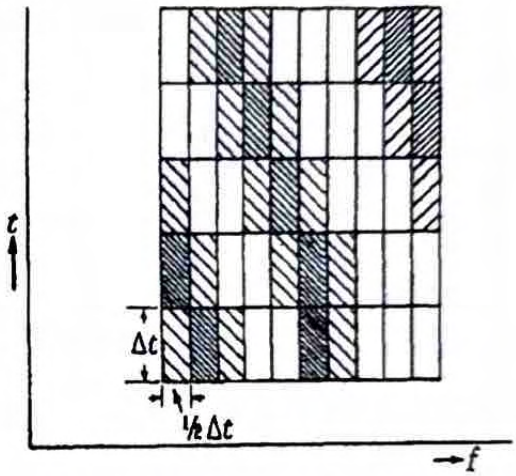


Figure 13.2 Gabor matrix. A detail of a small portion of a time-frequency grid (from Gabor [1946]). Inside each rectangle on the matrix is an elementary grain of time-frequency energy. Shading indicates intensity. Unlike in a traditional sonogram, time is plotted vertically, and frequency is plotted horizontally.

A granular representation is implicit in the *windowing* technique applied in the *short-time Fourier transform*, conceived by Gabor and extended in the 1960s (Schroeder and Atal 1962; Mallat 2009; see chapter 37). Inspired by Gabor, the MIT cybernetician Norbert Wiener (1964) and the information theorist Abraham Moles (1968) also proposed granular representations for sound. *Atomic decomposition*, described in chapter 39, can be seen as an analytical counterpart to granular synthesis. Starting from any sound as input, atomic decomposition generates a reconstruction of that sound based on a granular model (Sturm et al. 2009).

The composer Iannis Xenakis read Gabor's scientific papers and was the first to explicate a compositional theory for grains of sound (Xenakis 1960). He began by adopting the following lemma: "All sound, even continuous musical variation, is conceived as an assemblage of a large number of elementary sounds adequately disposed in time. In the attack, body, and decline of a complex sound, thousands of pure sounds appear in a more or less short interval of time Δt." Xenakis created granular sounds using analog tone generators and tape splicing. These appear in the composition *Analogique A-B* for string orchestra and tape (1959). The composition is described in Xenakis (1992) and analyzed by Di Scipio (1998). The score and tape part are available from Éditions Salabert.

Following an encounter with Xenakis, the author developed a computer-based implementation of granular synthesis using sine waves in 1974 at the University of California, San Diego (Roads 1978b). In 1981 the author made the first experiments in granulation of sampled sounds spatialized to four

loudspeakers at the Massachusetts Institute of Technology (Roads 1985c). We created the CloudGenerator app at Les Ateliers UPIC in 1995 (Roads and Alexander 1995; Roads 2001a). Later we developed the Creatovox instrument (2000), the ConstantQGranulator (2001), and EmissionControl at the University of California, Santa Barbara (Thill 2005). This was followed by the EmissionControl2 app (Roads, Kilgore, and Duplessis 2020).

Granular synthesis appears in many compositions, including the author's sets *POINT LINE CLOUD* (2004, reissued in 2019 on Presto!? Records) and *FLICKER TONE PULSE* (2019, Wergo).

Granular synthesis has been implemented in many ways. The Canadian composer Barry Truax (1987, 1988, 1990a, 1990b) developed the first real-time implementations. The book *Microsound* describes other implementations known at that time and includes sixty-eight sound examples (Roads 2001a). Alberto de Campo (2011) provides annotated code for his implementations in SuperCollider. Brandtsegg, Saue, and Johansen (2011) provides a Csound opcode for granular and other methods of particle synthesis.

In the past, granular synthesis was sometimes viewed as an exotic method, as it was not available in a commercial synthesizer. It is now mainstream. Dozens of software and hardware systems incorporate granular synthesis and processing. There are many choices in designing the synthesis engine and the user interface, thus implementations vary greatly.

Simple Grain Generator Instrument

At its most basic, granular synthesis can be implemented with a simple instrument: a sine wave oscillator controlled by an envelope generator (figure 13.3). The early theories of Gabor and Xenakis proposed this model. In the rest of this chapter, we call this model *classical granular synthesis.* By contrast, in place of the oscillator we can substitute a sample player module. In this way any sound can be granulated, from sound files or from live input. This is the prevalent model today.

Despite the simplicity of the instrument, to generate even a plain, uncomplicated sound requires a massive amount of microsonic control data—potentially thousands of parameter updates per second of sound.

Parameters of Granular Synthesis

Most parameters of granular synthesis apply to individual grains. These are the *per-grain* parameters. By contrast, the density parameter provides global control.

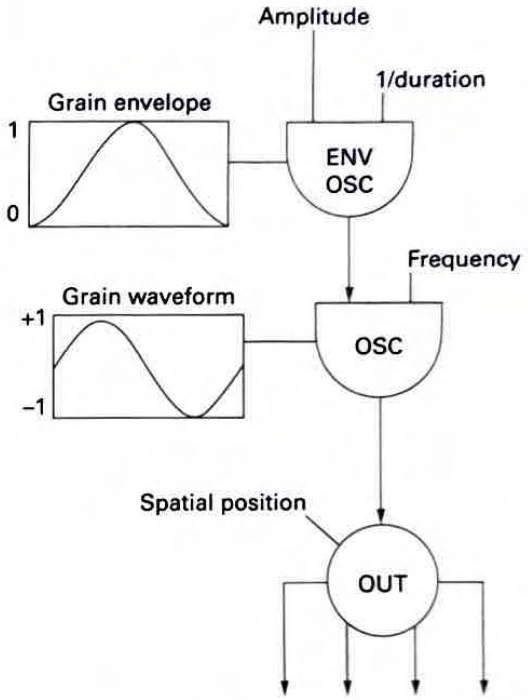


Figure 13.3 Simple granular synthesis instrument built from an envelope generator and an oscillator with four-channel spatial output.

All parameters can be modulated. Indeed, modulation is central to the animated textures produced by granular methods. The modulation sources can be traditional *low-frequency oscillators* (LFOs), random noise sources, or envelopes derived from gestural controllers. This provides another layer of control.

Per-Grain Parameters

Per-grain processing is key to the heterogeneous sonorities produced by granular synthesis. The basic per-grain parameters are the following:

- Envelope
- Duration
- Waveform (sample read pointer)
- Frequency (pitch shift for samples)
- Spatial position
- Synchronicity (degree of periodicity)
- Intermittency (degree of irregular emission)

To this can be added implementation-specific per-grain effects such as filtering, which add another layer of complexity (Roads 2001a, 2012; Roads, Kilgore, and Duplessis 2020).

Envelope

An amplitude envelope shapes each grain. Various envelopes can be used; each has its own effect on the spectrum (figure 13.4). Gabor's theory posited a Gaussian bell-shaped curve. The following equation defines a *normal* or *Gaussian* distribution $P(x)$:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

where σ determines the *standard deviation* (the spread of the bell) and μ is the *mean* or center peak.

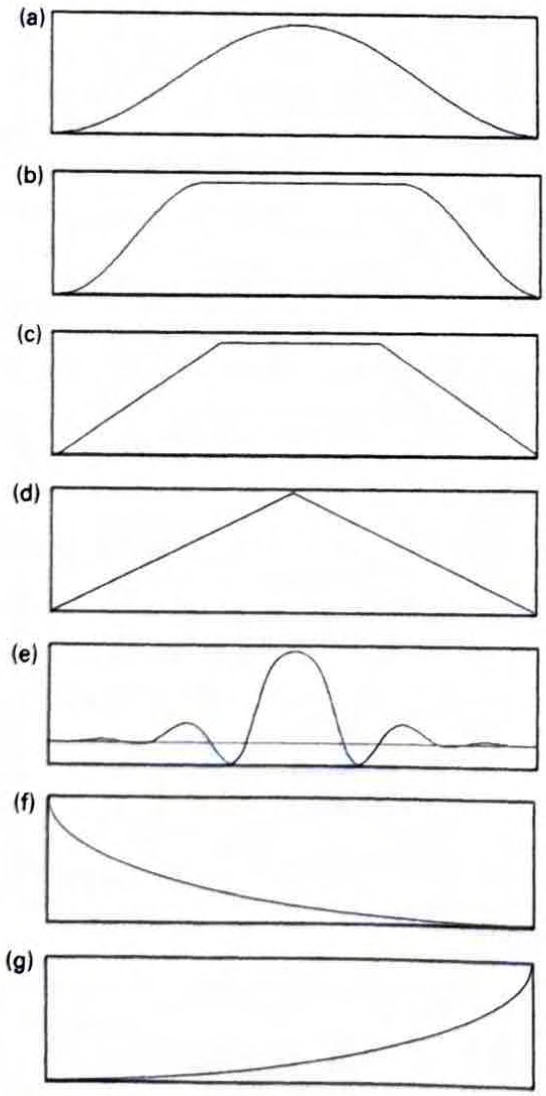


Figure 13.4 Grain envelopes. (a) Gaussian. (b) Tukey. (c) Three-stage linear. (d) Two-stage triangle. (e) Band-limited impulse. (f) Exponentially decaying (expodec). (g) Reverse exponentially decaying (rexpodec).

Figure 13.4b shows a *quasi-Gaussian* or *Tukey window* (Harris 1978), where the peak is extended over 30 to 50 percent of the duration of the grain. This shape has proved sonically effective (Roads 1985c). Simple line-segment envelopes (figure 13.4c and d) were used in the early experiments of Truax (1987, 1988).

Complicated envelopes such as a band-limited pulse (figure 13.4e) create resonant grains that sound like woodblock taps in sparse textures when the grain duration is less than 100 ms. The expodec (exponentially delaying) and rexpodec (reverse expodec) envelopes produce percussive and backward-sounding sound grains, respectively. As one would expect, sharp angles in any envelope cause strong side effects in the spectrum due to convolution. (See chapter 29.)

Duration

The duration of successive grains can be constant, variable according to some function (e.g., an envelope), or random within stipulated limits.

Another possibility is to vary duration according to a property of the grains. For example, we could link short durations to high-frequency grains. A correspondence between grain frequency and grain duration is characteristic of *wavelet* (chapter 38) and *grainlet* paradigms (Roads 2001a).

We should note that the previously mentioned upper boundary of 100 ms for duration concerning classical granular synthesis is not a hard limit. Larger grain sizes using sampled sound can be musically effective. Granular processing over a second or more spawns cascades of pitch-shifted and filtered echoes (possibly reversed) of recognizable material such as voices or fragments of music. Here grain scheduling algorithms that normally operate on the micro level scale up to the level of sound objects.

Effect of grain duration

A profound law of signal processing comes into play in setting the grain duration: the shorter the duration of an event, the greater its bandwidth. Thus short grain durations lead to crackling, explosive sonorities, whereas longer grain durations create a much smoother impression. Figure 13.5 demonstrates this important law for three elementary signals. As the duration of a signal shrinks, its spectrum broadens.

Figure 13.6 displays the dramatic spectral effects of lowering the grain duration and the resulting broadband noise.

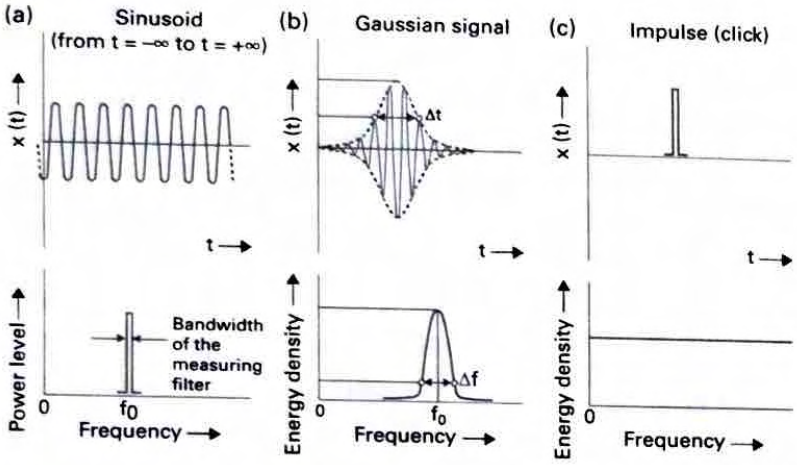


Figure 13.5 Time-domain functions (top) and spectra (bottom) of three elementary signals (after Blauert [1983]). (a) Sine wave of infinite duration corresponding to a single line in the spectrum. (b) Gaussian grain and corresponding formant spectrum. (c) Ideal impulse and corresponding infinite spectrum.

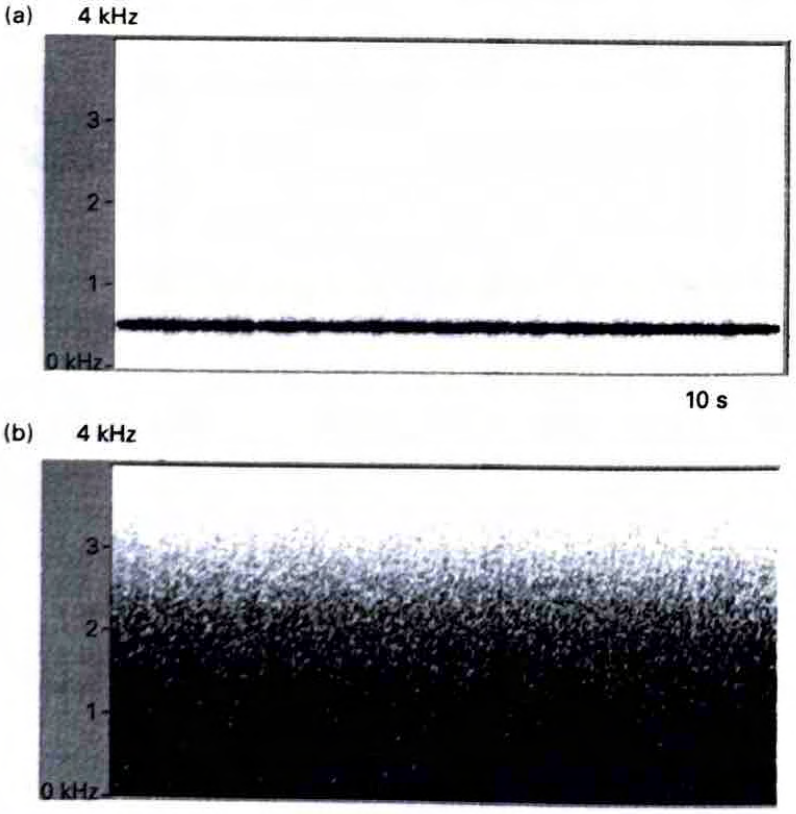


Figure 13.6 Effect of grain duration on spectrum. (a) Sonogram of a granular stream of 100 ms sinusoidal grains with a frequency of 500 Hz. (b) If we shrink the grain duration to 1 ms, the spectrum explodes dramatically into broadband noise.

Waveform

A fundamental principle of granular synthesis is that the waveform within the grain can be derived from any source: synthetic, sampled, or live. In the classical theory of granular synthesis, the waveform of each grain is a sinusoid at a specific frequency. By allowing the grain waveform to vary on a per-grain basis, we can produce colorful polychrome emissions. This enables us to choose among a number of different stored wavetables. Going further to allow sampled sound and live sources opens up a universe of possibilities.

Granulation of sampled sounds

Granulation of sampled sounds feeds acoustic material into a kind of logical threshing machine—delivering grains in a new order with a new micro-rhythm. The granulator reads in a brief segment of a sampled sound (from a sound file or directly from live input) and applies an envelope or *window* to the segment, transforming it into a grain. Listen to Roads (2004) and the CD included with Roads (2001a) for examples of sample granulation.

In sample granulation, a key parameter is the *sample read pointer* (sometimes called *file pointer, file position,* or *scan head*). This is the point in the file (or memory buffer, in the case of real-time granulation) that the granulator starts reading a sample. Once an initial pointer position is set, there are many ways to manipulate it. For example, it can scan forward smoothly to the end of the file. Or it can stutter, looping back repeatedly to its starting position. It may stagger forward slowly, time-stretching the sound. The pointer can move backward, causing the scan to play a sound in reverse. With random modulation it may jump around, causing a "scrambled sound" effect. The manipulation of this pointer is at the core of any granulation algorithm. Figure 13.7 shows three possibilities out of many.

Note that changing the position of the pointer slowly or quickly does not change the pitch. The playback sampling rate is independent of the rate of change of the sample read pointer, which merely sets the starting point. The next parameter discussed, frequency or pitch shift, changes the playback sampling rate.

Frequency (pitch shift for samples)

In classical granular synthesis, the grain waveform is a sinusoid. In this case, the frequency parameter stipulates the number of cycles per second (Hz). For a constant tone, the frequency is the same for a series of grains. In most cases,

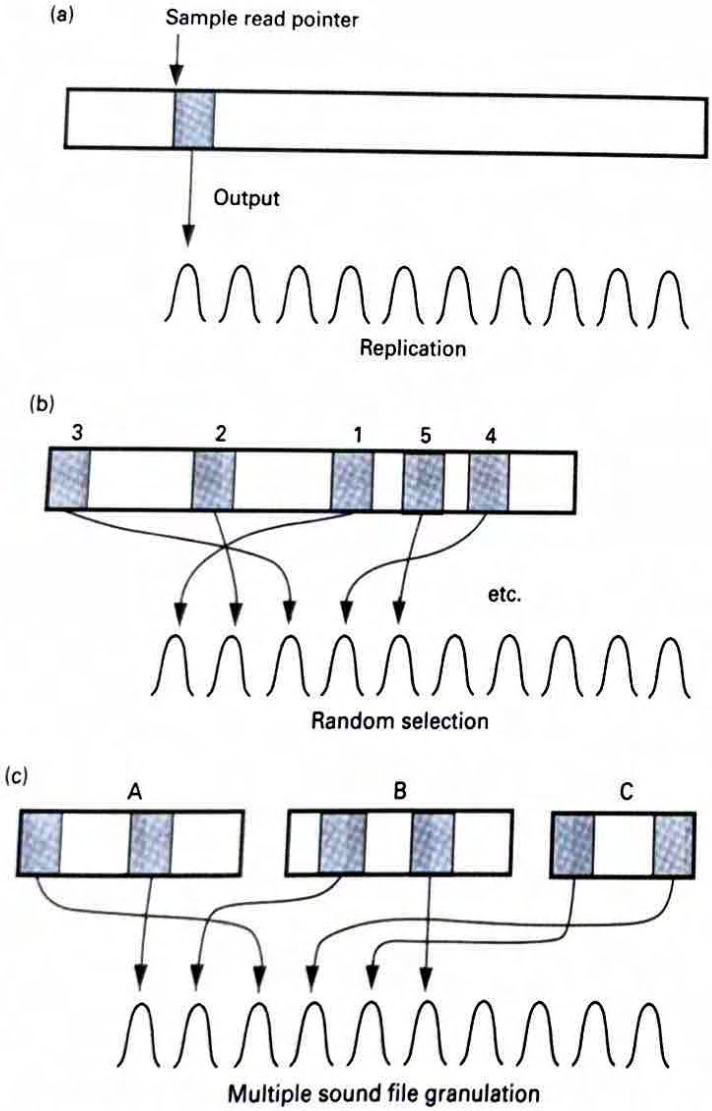


Figure 13.7 Granulation of sampled sound. The sample read pointer is the location in the file where the granulator starts reading a sample. (a) If the position does not change, a single grain is replicated. (b) If the pointer jumps around randomly, the output is scrambled. (c) In a multifile granulator, a texture can emerge from a combination of three different sound files, A, B, and C.

however, we want to be able to vary the frequency on a per-grain basis. To allow for variation in grain frequency we can stipulate a range, called a *band*. Within the upper and lower boundaries of the band, the grain generator scatters grains. This model allows for many possibilities. For example, the grains can follow narrow bands that ascend or descend in the manner of glissandi. The grains can be aligned to a pitch scale or scattered to random frequencies. When the frequency distribution is random and the band is greater than a

small interval, the result is a texture in which pitch is ambiguous or unidentifiable. The combined effects induced by the envelope and grain density also strongly influence the impression of pitch.

In the case of sampled grains the frequency parameter is a per-grain pitch shift. The pitch shift can be aligned to a scale and modulated by an LFO.

Spatial position

If the texture rendered by the granulator is monaural—every grain in the same spatial position—the impression is flat. By contrast, when each grain scatters to a unique location, the texture manifests a vivid spatial morphology. This is evident even in stereo. The grains can be scattered randomly in space, pan across two or more channels, or follow an arbitrary 3D spatial path in an immersive environment. Barrett (2016) explored the projection of grains in ambisonic systems (see chapter 32). Deleflie and Schiemer (2009) used a granular representation to render existing ambisonic encodings. Wan Rosli (2016) used spatial information extracted from the spherical harmonics of ambisonics in order to granulate space.

Selective reverberation means setting the depth of reverberation of each grain individually. Per-grain reverberation is most striking at low densities in a multiloudspeaker setup. At high densities in a stereo playback system, the individual reverberation halos fuse into a continuous background cloud, which is not much different from global wave reverberation.

Synchronicity (degree of periodicity)

Synchronicity is the degree of periodicity of the grain emissions. Thus *synchronous granular synthesis* (SGS) emits one or more streams of grains in which the grains follow each other at regular intervals (Roads 2001a). In SGS, grain density determines the frequency of grain emission. For example, a density of two grains per second means that a grain is produced every half second—a repeating tone. Synchronous densities in the range of about 0.1 to 20 grains per second generate metrical rhythms. When the densities change over time, we experience precise accelerandi/rallentandi effects. At higher densities, long grains fuse into continuous tones and textures.

Asynchronous granular synthesis (AGS) breaks the periodicity of perfectly sequential streams of grains. Instead, it randomizes the grain emission times (Roads 1991). Some implementations call this parameter *jitter*. Notice that randomizing the onset time does not change the overall density.

Intermittency (degree of irregular emission)

The amount of *intermittency* is the degree to which the regular flow of grains is interrupted. Thus intermittency is not the same as asynchronicity. As figure 13.8 shows, in an intermittent emission, grains are lost. Thus intermittency overrides the stipulated grain density, as explained next.

Density and Fill Factor

Now that we have covered the per-grain parameters, we look at a higher-level control. This is *density*—the number of grains generated per second. Density serves as a global control of texture in granular synthesis. When density is low, only a few grains are heard, resulting in a sparse rhythmic pattern. At a critical point (determined by the *fill factor*, explained further on) the individual grains fuse into a continuous texture. High densities spawn massive clouds of sound.

Density, however, is not the only determinant of texture. Density and grain duration combine to make texture. For example, a one-second cloud containing twenty 100 ms grains is continuous and opaque, whereas a one-second cloud containing twenty 1 ms grains is sparse and transparent. The difference between these two cases is their fill factor (FF). FF is the product of density and the grain duration. In the cases just cited, the fill factor of the first cloud is $20 \times 0.1 = 2$, whereas in the second cloud the fill factor is $20 \times 0.01 = 0.2$. In general, when $FF < 0.5$ the texture is sparse. When it is 1.0 or above the

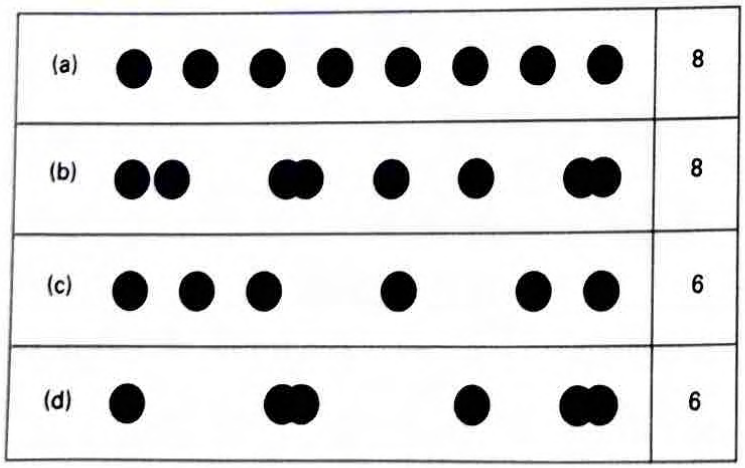


Figure 13.8 Comparison of four granular streams on a time line. In the left-hand column, a circle represents a grain. Density is the same for all four streams. (a) Synchronous. (b) Asynchronous. (c) Synchronous with intermittency. (d) Asynchronous with intermittency. The right-hand column indicates the number of grains. Intermittency results in a loss of grains.

texture is continuous. Other factors can affect continuity, such as density variations in time, asynchronous and/or intermittent grain emissions, and others.

For a grain duration of 25 ms, we can make the following observations concerning grain density as it crosses perceptual thresholds:

< 15 grains/second—Rhythmic sequences

15–25 grains/second—Fluttering, sensation of rhythm disappears. If the cloud is asynchronous, we hear intermittencies.

25–50 grains/second—We no longer perceive an acceleration of tempo in grain emissions, but we feel an increase in granular flow.

50—100 grains/second—Texture band.

> 100 grains/second—Continuous sound mass. No space between grains. Resembles reverberation in some cases.

Density and frequency band or pitch-shift effects are also synergistic, as follow:

- Narrow bands and high densities create pitched streams with formant spectra.
- Medium bands (e.g., intervals of several semitones) and high densities generate turgid colored noises.
- Wide bands (e.g., an octave or more) and high densities form massive clouds of sound.

As we have seen in the section on grain duration effects, another way to modify the bandwidth of the cloud is by changing the grain duration parameter.

High-Level Granular Organizations

The complexity of the sound generated by granular synthesis derives from the amount of control data fed to it. If n is the number of parameters for each grain and d is the average grain density per second of sound, it takes $d \times n$ parameter values to specify one second. Because d can range into the hundreds or thousands, it is clear that for the purposes of compositional control, a higher-level unit of organization for the grains is needed. These units function on the meso time scale of several seconds or more. The purpose of such a unit is to let composers stipulate massive quantities of grains using just a few global parameters.

In products that implement granular synthesis, MIDI control is an essential capability. Most synthesis instruments respond to note-on messages, for example, as well as knobs or faders for other parameters. MIDI control is not essential to all implementations of granular synthesis, however.

Table 13.1 lists various models of granular organization. Next, we look at each model in turn. A brief summary concludes the discussion.

Time-Frequency Projections

These approaches project grains onto the plane of frequency versus time. In this category we include grid-like projections such as the Gabor matrix (Gabor 1946, 1947) and screens (Xenakis 1960, 1992), as well as the short-time Fourier transform (chapter 37) and wavelet transform (Mallat 2009). Another category of time-frequency projection is atomic decompositions (chapter 39), in which there is no fixed grid in frequency or time.

We have already presented the concept of the Gabor matrix. Another grid-oriented conception is Xenakis's (1960, 1971, 1992) notion of *screens*. A screen is an amplitude-frequency grid on which grains are scattered. A

Table 13.1 High-level granular organization

Time-frequency projections	Gabor matrix (Gabor 1946, 1947); screens (Xenakis 1960, 1992); short-time Fourier transform (chapter 37); wavelet transform (Mallat 2009); atomic decompositions (chapter 39)
Clouds	Roads (1978b); CloudGenerator (Roads and Alexander 1995)
Streams	Synchronous and asynchronous (Truax 1987, 1988, 1990a, 1990b), EmissionControl (Thall 2004), EmissionControl2 (Roads, Kilgore, and Duplessis 2020); Monolake Granulator (Robert Henke); Instruo Abhar; AudioMulch; The Mangle; many others
Pitch-synchronous granular synthesis	De Poli and Piccialli (1991); Piccialli et al. (1992); Dutilleux, De Poli, et al. (2011)
Sprays	MetaSynth (Wenger and Spiegel 2010)
Scrubbing	Borderlands Granular (Carlson and Wang 2012; Carlson 2019)
Micromontage	Music of Horacio Vaggione; IRIN app (Caires 2004, 2019)
Physical/biological models	Cook (1996, 1997, 2007); Keller and Truax (1998); Norris (2013); Fischman (2003)
Abstract generative models	Di Scipio (1996, 1999); de Campo (2011); Rhys (2016)

synchronous sequence of screens (called a *book*) constitutes the evolution of a complex sound. Rather than starting from an analyzed sound, Xenakis's proposals use generative algorithms to fill the screen with grains. He proposed scattering grains randomly into screens and then deriving new screens from set-theory operations: intersections, unions, complements, and differences, among others. A basic problem with this screen paradigm is its regular 25 Hz frame rate, which introduces a periodic modulation.

Two related spectrum analysis techniques, the *short-time Fourier transform* (STFT) and the wavelet transform, take in a time-domain sound signal and measure its frequency content versus time (see chapters 37 and 38). In effect, these methods associate each point in the analysis grid with a unit of time-frequency energy—a grain or a wavelet.

The STFT is well known and can be computed using the fast Fourier transform (Rabiner and Gold 1975). The grain in this case is a set of overlapping windows within each of the N frequency bins of the Fourier analyzer. The resulting time-frequency grid can be seen by zooming in on a sonogram (figure 13.9).

The wavelet transform (Kronland-Martinet and Grossmann 1991; Mallat 2009) performs a similar operation, but the spacing of the analysis channels and the duration of the window (called the *analyzing wavelet*) is different from the STFT. In the STFT, the spacing between the channels on the frequency axis is linear, whereas in the wavelet transform it is usually logarithmic. That is, in the wavelet transform, the channel frequency interval (bandwidth) Df/f is constant. In the STFT, the window duration is fixed, whereas in the wavelet transform it can vary as a function of frequency.

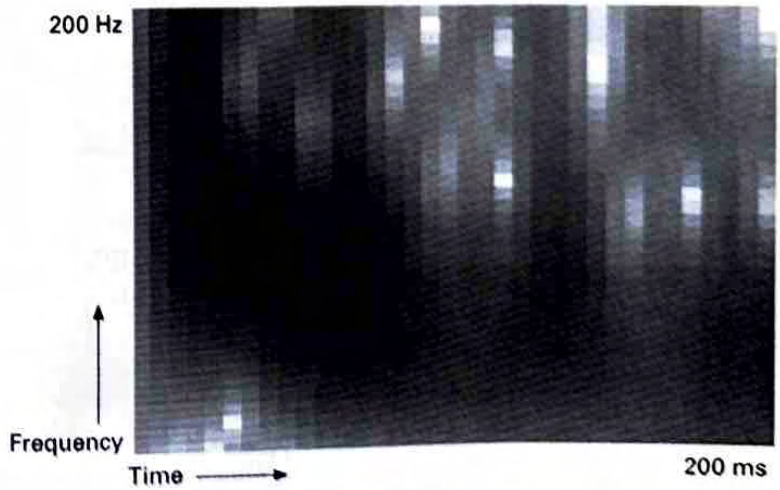


Figure 13.9 Detail of a time-frequency grid produced by the STFT. Snare drum hit, 0–200 ms, 0–200 Hz. The zoomed-in sonogram displays the rectangular cells of energy at each time frame and frequency bin.

Both the STFT and the wavelet transform permit analysis, transformation, and resynthesis, which make them potentially powerful musical tools for the manipulation of sampled sounds. However, these tools are designed to give users an illusion of sonic continuity. Their internal granular structure (quantized in time and frequency) is generally hidden to users and is therefore not controllable as such.

Clouds

The author proposed the concept of generating *clouds* of granular sound (Roads 1978b). A sound cloud can be defined as a region on the time-frequency plane, similar to what G. M. Koenig called *tendency masks* (Koenig 1970b). The region has a precise duration and adjustable frequency boundaries within which grains can be scattered at various densities (figure 13.10). This model requires a specification for each cloud and is thus a non-real-time

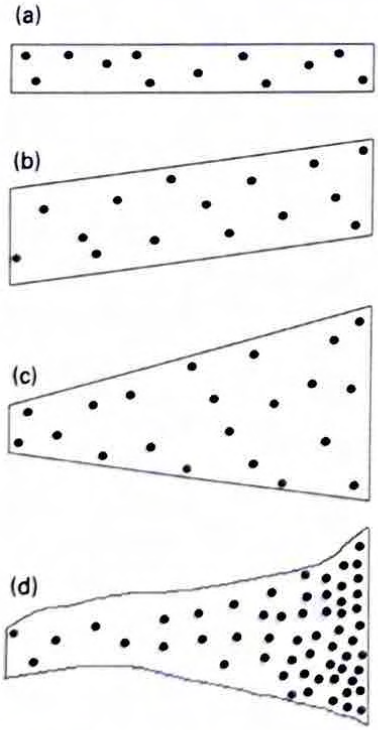


Figure 13.10 Sound clouds filled sparsely using an asynchronous algorithm that scatters the grains randomly in time. The horizontal axis is time, and the vertical axis is frequency. (a) Basic cloud with fixed upper and lower frequency bandwidth. (b) Cloud with linearly varying upper and lower bandwidth, both with positive slope. (c) Cloud with linearly varying upper (positive slope) and lower bandwidth (negative slope). (d) Cloud with continuously varying upper and lower bandwidth. Notice also that the grain density is increasing over time.

approach. In the original implementation, the author provided a script for generating multiple overlapping clouds in a sequence.

The Cloud Generator app (Roads and Alexander 1995) shown in figure 13.11 rendered a single cloud as a sound file. A composer could then arrange multiple sound files on the timeline of a DAW.

Streams

Streaming generation is a real-time process pioneered by Barry Truax (1986). In his initial system, the grain waveform was fixed. Later implementations read the grain waveform from a sound file or live input. A streaming granulator can spawn a large amount of derived sound material from a given sound

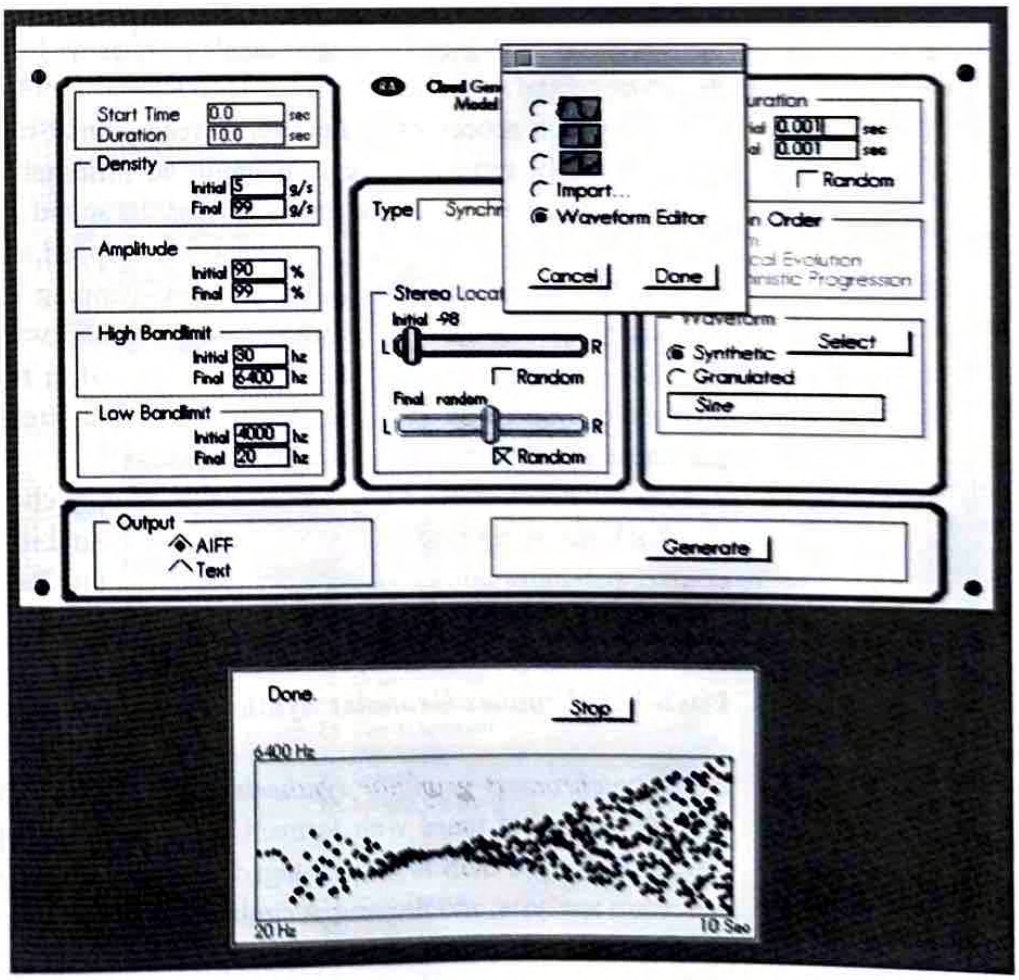


Figure 13.11 Cloud Generator app in action. The main panel sets the parameters of a cloud. The Synthetic waveform option is selected, which brings up a menu of options for the waveform. The Waveform Editor option is selected. Below these windows is another window showing the cloud generated from this specification. Individual grains are visible.

file by manipulating a sample read pointer. As explained previously, this pointer indicates where in the file to get the next grain for playback. An example of live input to streaming granulation is the music of Agostino Di Scipio (1996, 1999), who granulated a soloist on an acoustic instrument in order to produce polyphonic textures.

Figure 13.12 shows the screen of EmissionControl2, a streaming granulator (Roads, Kilgore, and Duplessis 2020). EmissionControl2 can granulate multiple sound files simultaneously, with low-frequency oscillator (LFO) control of grain parameters, including sound file selection. The third panel down shows the sound file being read. The sample read pointer is shown as a vertical line at the beginning of the shaded area. The shaded area is the current scan region.

Instruo Arbhar is an example in Eurorack hardware (figure 13.13). Twistable knobs and voltage control provide direct access to granulation processes.

In any streaming granulator, controls determine the position, speed, and direction of the sequence of grains emitted. Figure 13.14 shows some possible playback processes. If the sample read pointer remains stationary, the engine simply repeats the current grain ad infinitum. If it steps forward slowly, successive grains are repeated, and the sound is time stretched. If it steps forward quickly, successive grains are skipped, and the sound is time shrunk. The grain sequence can also loop: stepping forward and back by various ratios while also continuing forward or backward through the sound file. Stepping backward, grains can also be read in reverse. Simultaneous streams can be triggered as multiples of the first stream's clock. All these parameters can be modulated.

Manipulations of the sample read pointer simply change its position. The playback sampling rate is constant so that the sound is not pitch shifted. Of course, per-grain pitch shifting (which changes the playback sampling rate) can be another parameter of the granular engine.

Pitch-Synchronous Granular Synthesis

Pitch-synchronous granular synthesis (PSGS) is a technique designed for the generation of tones with formant regions in their spectra (De Poli and Piccialli 1991). PSGS is a multistaged operation involving pitch estimation, spectrum analysis, and filtering. A similar method is called *pitch-synchronous overlap and add* (PSOLA) (Dutilleux, De Poli, et al. 2011).

The first stage is pitch estimation of a voice or instrument (see chapter 34). Each pitch period is considered as a separate grain. Then spectrum analysis is performed on each grain. The system derives its impulse response and uses this to set the parameters for a filter. (Chapters 28 and 29 discuss impulse response measurements.)

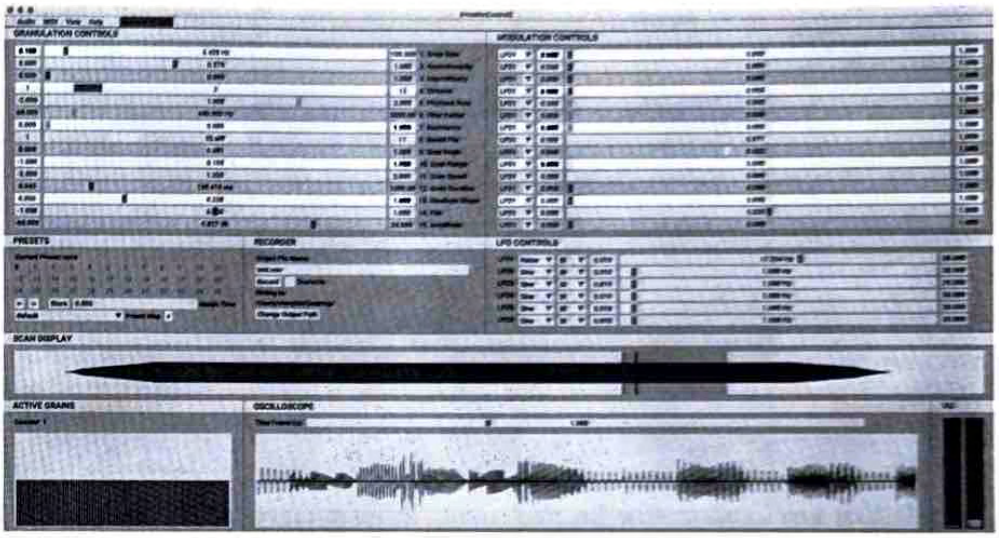
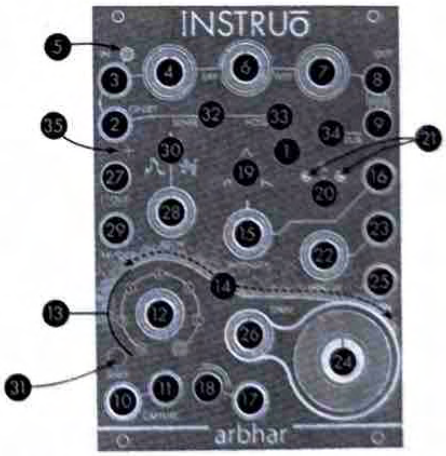


Figure 13.12 Screen of the EmissionControl2 app. Multiple sound files can be granulated simultaneously. The granulation parameter faders are on the top left. Each granulation parameter can be modulated by an LFO source. The faders at top right control the amount of modulation. The six LFOs are in the middle panel. Below it is a display of the current sound file being granulated. The bottom waveform display shows the output.



1. Condenser Microphone
2. Onset Input
3. Input
4. Input Level
5. Input Indicator
6. Dry/Wet Mix
7. Output Level
8. Output 1
9. Output 2
10. Capture Button
11. Capture Gate Input
12. Layer
13. Layer Indicators
14. Granular Stream
15. Intensity
16. Intensity CV Input
17. Strike Button
18. Strike Input
19. Grain Window
20. Grain Direction
21. Grain Direction Indicators
22. Length
23. Length CV Input
24. Scan
25. Scan CV Input
26. Spray
27. Gate Output
28. Pitch
29. 1V/Octave Input
30. Pitch Deviation
31. Shift Button
32. Sensitivity
33. Hold
34. Dub
35. Onset Indicator

Figure 13.13 Instruo Arbhar module with legend of controls and indicators.

abcdef	Orginal (*Scan Speed* = 1; *playback Rate* = 1)
aaaaaa	Frozen (*Scan Speed* = 0; *playback Rate* = 1)
------	Silence (*Scan Speed* = 1; *playback Rate* = 0)
ƒɘbɔdɑ	Time-reversed (*Scan Speed* = 1; *playback Rate* = 1)
fedcba	Time-reversed with forward grains (*Scan Speed* = 1; *playback Rate* = 1)
ɑdɔbɘƒ	Time-forward with reversed grains (*Scan Speed* = 1; *playback Rate* = 1)
aabbccddeeff	Time-stretched (*Scan Speed* = 0.5; *playback Rate* = 1)
ace	Time-shrunk (*Scan Speed* = 2; *playback Rate* = 1)
a b c d e f	Pitch-shifted down (*Scan Speed* = 1; *playback Rate* = 0.5)
abcdef	Pitch-shifted up (*Scan Speed* = 1; *playback Rate* = 2)

Figure 13.14 Some grain playback possibilities. See the text for an explanation. $\{a, \ldots f\}$ is a set of ordered grains.

In synthesis, a pulse train at the detected pitch period drives a bank of *finite impulse response* (FIR) filters. (FIR filters are discussed in chapter 28.) The output signal results from the excitation of the pulse train on the weighted sum of the impulse responses of all the filters. At each time frame, the system emits a grain that is overlapped and added with the previous grain to create a smoothly varying speech or instrument signal. The PSGS representation enables several transformations to create variations of the original sound. The includes separation of the harmonic part of the sound from the residual inharmonic part (Piccialli et al. 1992).

Sprays

Certain software-based granulators implement a flowing spray jet of sound particles. This is made visually explicit in programs like Metasynth (Wenger and Spiegel 2010) where the user literally sprays grains onto a time-frequency grid onscreen (figure 13.15). In this sense, the cloud, stream, and spray models align with the spirit of Xenakis's UPIC paradigm, in which users directly draw sounds in the time-frequency plane (Xenakis 1992; Weibel, Kanach, and Brümmer 2020).

Scrubbing

Multitouch interfaces enable a graphical *scrubbing* model, after the tape scrubbing methods of the analog era (figure 13.16). Internally, these granulators use streaming. However, they provide manual scrubbing controls for manipulation of the sample read pointer. Borderlands Granular is a prime

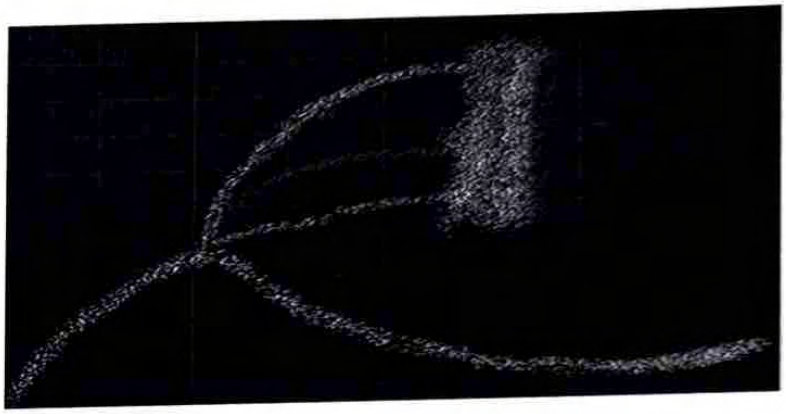


Figure 13.15 MetaSynth granular clouds painted with a spray can tool. Time is the horizontal axis and frequency is the vertical axis.

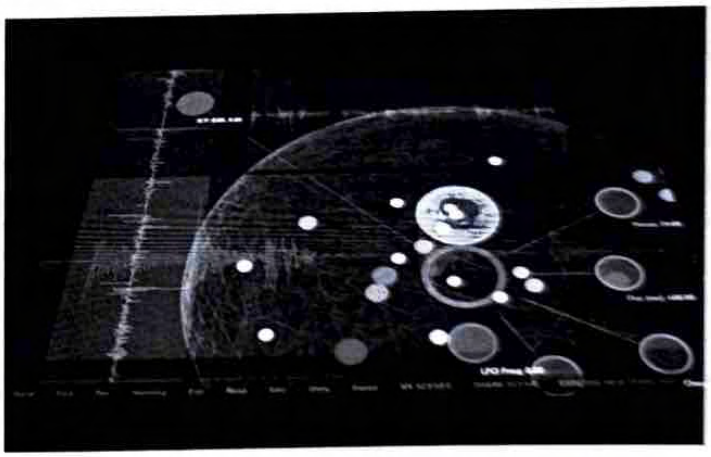


Figure 13.16 Borderlands app for iPad. The user slides or *scrubs* a circle along the waveform to select segments to granulate. The attached satellite circles function as faders for control of synthesis parameters.

example (Carlson and Wang 2012; Carlson 2019). Users begin by placing any number of sound files in a directory. When launched, the software displays waveforms (corresponding to the sound files) on the screen. Each waveform is within a rectangle and is oriented either vertically or horizontally. These rectangles can be selected, moved, and resized using one's fingers. A *grain cloud* with a set of graphic faders can be added to the screen. When the user moves a cloud over a rectangle it begins to play. The position of the cloud can be moved back and forth across the waveform. This is analogous to scrubbing audio tape across a tape head. The user can control the density of voices, their overlap, duration, pitch, envelope, and degree of random motion. The instrument is performed by creating and destroying new clouds and moving them around the landscape.

Micromontage

The technique of micromontage, as developed by the Paris-based composer Horacio Vaggione, involves arranging microsounds manually in the time line of a DAW (Vaggione 2003). The composer loaded a catalog of pre-edited sounds into the app's library. Then the composer selected clips in the library and pasted them onto a track at specific points on the time line running from left to right across the screen. By pasting a single grain multiple times, it became a sound object of a higher temporal order—a *microfigure*. Each paste operation was like a stroke of a brush in a painting, adding a touch more color over the blank space of the canvas. In this case, the collection of microsounds in the library can be thought of as a palette. Because a DAW lets users zoom in and out, the composer could paste and edit on different time scales. Grains and microfigures could be positioned on multiple simultaneous tracks, permitting a rich interplay of microevents.

This approach is extremely free in terms of compositional options, considering that material at any time scale can be processed by plug-ins and placed anywhere in the time line. Detached from real-time constraints, ideas can be tested, edited, submixed, or deleted at will.

One of the strategies that this enables is a thematic approach, that is, manipulating sound material to create repetitions and variations. For example, a copy of a given sonic entity can be pitch shifted, time scaled, ring modulated, filtered, reversed, and so on. Parts of the piece, if not the entire work, can be organized as a montage of variations of a finite number of elements. This approach forms the basis of my compositions *Never* (2010) and *Always* (2013) and all the electroacoustic works of Horacio Vaggione (Roads 2005). It was crystallized by the IRIN app developed by Vaggione's student Carlos Caires (2004, 2019).

Physical and Biological Models

A recurrent theme in computer music is setting up a mapping between a scientific equation and parameters of sound synthesis. For example, Fischman (2003) mapped the Schrödinger wave equation to parameters of granular synthesis. This kind of *sonification* project can have several aims: better scientific understanding, a purely artistic experience, or something in between.

Another approach comes from physical modeling synthesis, in which the goal is to simulate the sound of an acoustic instrument (chapter 18). An early example involving granular synthesis was a simulation of the sounds of shaken and scraped percussion: maracas, sekere, cabasa, bamboo windchime, tambourines, sleighbells, and guiro (Cook 1996, 1997, 2007). Keller and Truax

(1998) created granular models of physical processes such as liquid streams and a bouncing metallic ball. Natasha Barrett modeled processes of self-organized critical systems such as grain avalanches in her work *The Utility of Space* (2000), which was just one facet of a multilayered design.

A potentially compelling paradigm derives from the physics of granular processes (Norris 2013). A large body of scientific literature centers on modeling the micromechanics of processes such as grain vibration patterns, mixing, flow, and interactions between grains and fluids (Aronson and Tsimring 2009). These models could be repurposed for artistic ends.

Particle systems are the computer graphics counterpart to granular synthesis of sound (Reeves 1983). Fonseca (2013) adapted a particle engine from computer graphics to the control of granular synthesis. A product called Sound Particles brings this to users.

Biological models such as DNA sequences, neuron behavior, and cicadas have also been explored as models for sound synthesis (Smyth and Smith 2001). These simulations can be arbitrarily complex and extend into virtual worlds. Unfortunately, the problem with physical and biological modeling has always been the same: a model is worth little without an expert player, whether real or virtual. Virtuosity demands daily practice over a period of years. Obviously, to develop a software model of an expert player poses a daunting challenge.

Abstract Generative Models

Beyond the physical and biological lies the abstract realm of experimental algorithms that are not related to a model of a real-world phenomenon. They are sometimes derived from mathematical formulae such as recursive substitution, cellular automata, or chaotic functions. These generally low-level algorithms emit granular sound as a byproduct of a formal process from an abstract domain. A problem with such bottom-up approaches is that they do not usually incorporate a multiscale notion of time structure. Like the low-level rules in Wolfram's (2002) generative experiments, whether larger scale structures emerge out of such rules is mainly a matter of guesswork. Fractal generators are inherently multiscale (Rhys 2016). This can be easily seen in images but is more difficult to perceive in sound structures.

Abstract algorithms are not directly tethered to human action or perception. To the detached observer, often these processes tend to sound either predictable or merely random. They lack the meaningful sense of causality and effort that we associate with human gestures, the opposition between tension and resolution to which we are psychologically disposed. Human beings are highly sensitive to the virtuosic play of lively human gestures, whether live on stage or

practiced behind the scenes in a studio, and to the unfolding of a sonic narrative played out by recognizable sonic characters or themes.

Summary of Higher-Level Organization

Granular models based on time-frequency projections start from sound analysis. From a creative perspective, we want to manipulate the analysis data as a means of sound transformation. Fourier and wavelet models do not, in general provide access to the granular level. Models based on atomic decomposition do.

Cloud, stream, spray, scrubbing, and micromontage models have been effective in composition because they allow composers to agglomerate grains into multiple levels of musical structure, specifically the *object* (100 ms–8 s), *meso* (>8 s), and macro time scales (Roads 2001a).

Pitch-synchronous granular synthesis is reminiscent of convolution with pulsars (chapter 21).

Methods derived from scientific models and abstract formulas pose compositional challenges. How can one create coherent multiscale structures with these techniques? The key is finding a mechanism to transfer complex structures generated at lower levels into musically interesting structures at higher levels.

14 *Subtractive Synthesis*

Digital Filters: Background

Introduction to Filters

Filter Response Curves

Filter Types

Filter Q and Gain

Phase Response and Latency in Filtering

Filter Banks and Graphic Equalizers

Comb and Allpass Filters

Time-Varying Subtractive Synthesis

SYTER and GRM Tools

Subtractive Analysis/Resynthesis

The Vocoder

Linear Predictive Coding

What Is Linear Prediction?

LPC Analysis

Filter estimation

Pitch and amplitude analysis

Voiced/unvoiced decision

Analysis frames

LPC Synthesis

Editing LPC Frame Data

Cross-Synthesis: A Musical Extension of LPC

WaveGAN

Subtractive synthesis uses *filters* to shape the spectrum of a source signal. As the source signal passes through a filter, the filter carves selected regions of the frequency spectrum of the input signal (figure 14.1). The term *subtractive* refers to this carving effect. A filter can either cut or boost a specified region of the spectrum.

Spectrally rich source sounds include impulse trains, sawtooth waves, and white noise. Figure 14.2 shows the dramatic effect of two filters on white noise. Using white noise as the input signal, subtractive synthesis can produce many types of *colored noises*–a staple of electronic music (Roads 2015). Colored noise can be obtained by passing white noise through a filter. Works like *Vision and Prayer* (1961) and *Ensembles for Synthesizer* (1964), composed by Milton Babbitt using the RCA Synthesizer, demonstrated how a palette of colored noises could be deployed effectively in composition.

Besides their use in synthesis, filters are also central to sound editing and mixing and, for that matter, to all sound design. Hence everything we say about filters applies to those domains as well.

This chapter serves as a basic introduction to using filters, describing their main characteristics and parameters. Chapter 28 goes into greater technical

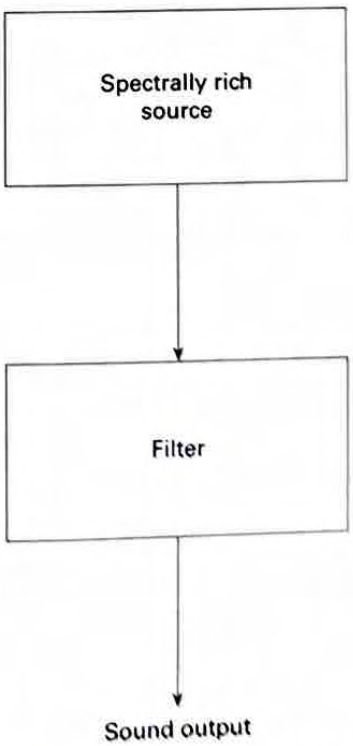


Figure 14.1 Scheme of subtractive synthesis. A spectrally rich source such as noise or pulses is fed into a filter, which shapes the output spectrum.

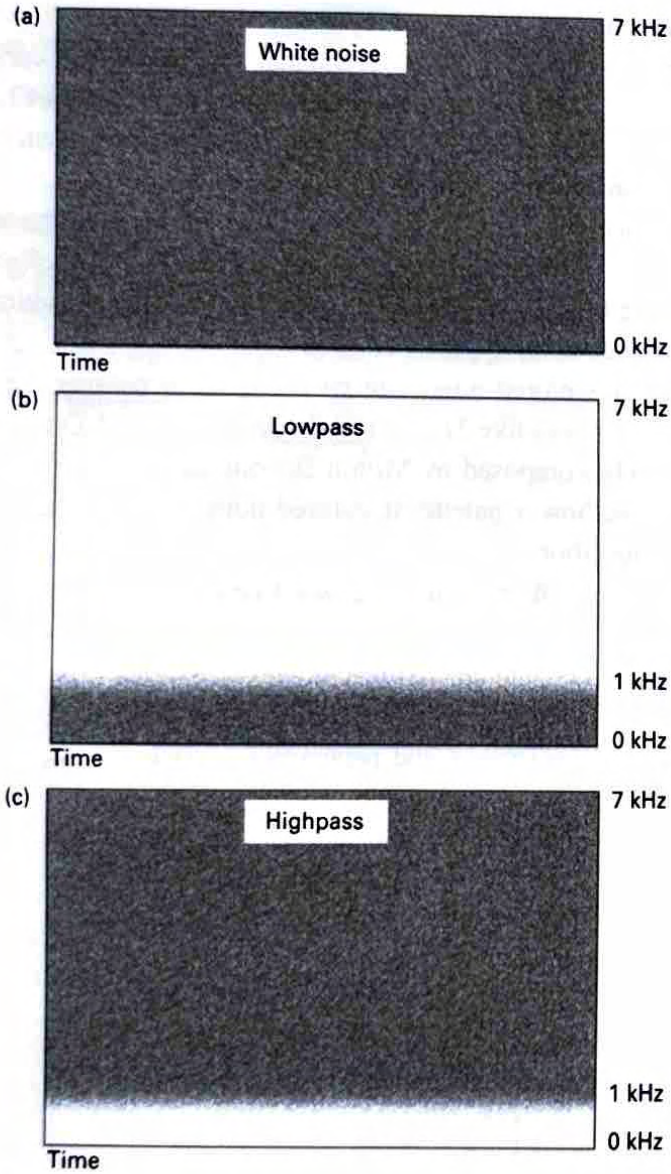


Figure 14.2 Sonogram images show the dramatic effect of two filters on a white noise signal. A sonogram plots the spectrum of a sound over time. The vertical axis is frequency, and the horizontal axis is time. The darker the trace, the more energy there is. (a) White noise lasting several seconds. The uniformly dark grey texture shows that the energy is equal in all audio frequencies. This plot is limited to 7 kHz. (b) A lowpass filter with a cutoff frequency at 1 kHz and a sharp slope eliminates the energy above 1 kHz. (c) A highpass filter with a cutoff frequency of 1 kHz and a sharp slope eliminates all the energy below 1 kHz.

detail about the internal operation of filters. Other technical introductions include Moorer (1977), Moore (1978a, b), McLellan, Schafer, and Yoder (2003), and Loy (2007). Those with an engineering background could also study more advanced texts on filters such as Moore (1990), Smith (1985a, 1985b, 2007b), Oppenheim and Willsky (1983), Rabiner and Gold (1975), and Oppenheim and Schafer (1975), among many others.

Digital Filters: Background

Primordial electronic music instruments used analog filters to shape the raw waveforms emitted by their tone generators, a process dubbed *subtractive tone forming* by Douglas (1968). Among the notable instruments that contained filters were the Mixtur-Trautonium, Solovox, Clavioline, Warbo Formant Organ, Hammond Novachord, RCA Synthesizer, and the Ondioline (Jenny 1958; Rhea 1972; Bode 1984; Roads 1996b).

Standalone analog filters like the Albis Tonfrequenz filter (figure 14.3) were staples in electronic music facilities such as the West German Radio (WDR) studio in which K. Stockhausen, G. M. Koenig, J.-C. Eloy, and other composers worked in the 1950s and 1960s. Later, the development of *voltage-controlled filters* typified the first golden era of modular analog synthesizers (Moog 1965; Chamberlin 1985; Rossum 1992; Bjørn and Meyer 2018).

Experiments with digital filter circuits began in the 1950s. The theory of digital filtering was pushed forward in the 1960s with the general adoption of the *Z-transform calculus* (Kaiser 1963; Rabiner and Gold 1975; Steiglitz 1996). The Z-transform introduced new ideas such as *stability* and *causality* that are essential to filter design (see chapter 28). Rudimentary digital filters appeared in sound synthesis languages such as Music IV, Music 4B, and Music V (Mathews and Miller 1963; Winham 1966; Mathews 1969).

By the early 1980s, engineers built digital synthesizers at several research laboratories. Examples include Stanford's Systems Concepts Digital Synthesizer (Samson 1980, 1985), IRCAM's 4X (Asta et al. 1980), and the Groupe de Recherches Musicales's SYTER (Allouis 1979; Allouis and Bernier 1982). These exotic machines could realize dozens of digital filters in real time. But it was not until a decade later that speed-ups in hardware made it possible to implement real-time digital filters in low-cost synthesizers, plug-in signal-processing cards, effects units, and digital sound mixers.

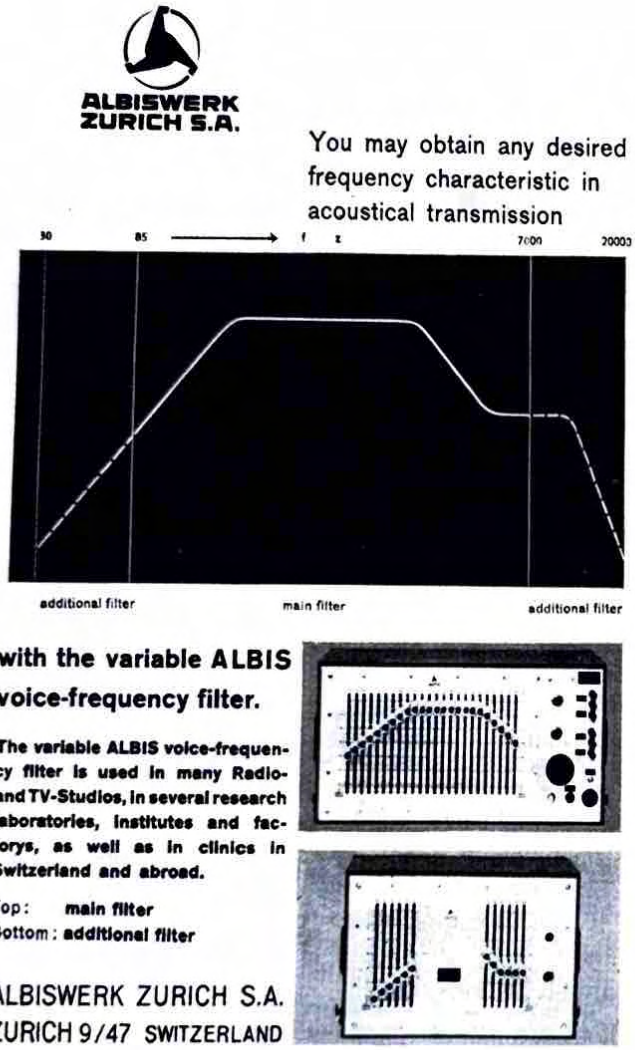


Figure 14.3 The Albis Tonfrequenz filter, a graphic equalizer used in early electronic music compositions.

Introduction to Filters

A filter can be literally any operation on a signal! That is, anything with an input and an output is a filter (Rabiner et al. 1972). But the most common use of the term in audio engineering pertains to devices that boost or attenuate regions of a sound spectrum. This is the meaning we assume here. Filters work according to one or both of the following two methods:

Delaying a copy of an *input signal* by a tiny duration of one or many sample periods and combining the delayed input signal with the new input signal (figure 14.4a).

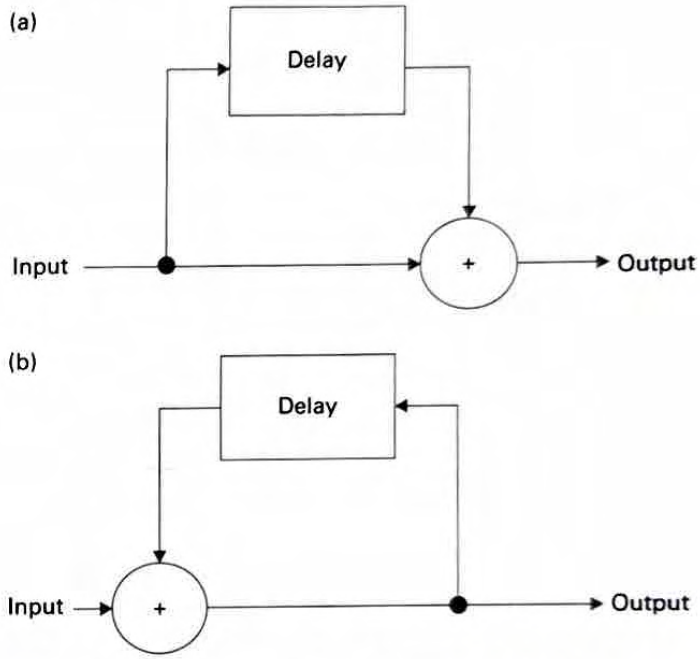


Figure 14.4 Two basic digital filters. (a) Delay the input and add it (feed-forward). (b) Delay the output and add it (feedback).

Delaying a copy of the *output signal* by a short duration and combining it with the input signal (figure 14.4b).

The first case is called a *feed-forward filter*. The second case is a *feedback filter*.

The tiny delays and combinations affect the spectrum by means of *phase cancelation* and *phase reinforcement* effects. Phase cancelation results in attenuation of selected frequencies, and phase reinforcement results in boosting of selected frequencies. Consider a signal with a frequency of 100 Hz. It has a period of 10 ms. If we delay it by10 ms and combine it with the original, we would have two in-phase signals, and the result of adding them would be a boost in amplitude. If we delay the input signal by 5 ms, then the delayed copy is 180 degrees out of phase with respect to the undelayed signal. Now if we add them together, the result will be phase cancelation. Figure 14.5 shows these effects.

Although figures 14.4 and 14.5 show combination by summation (+), the combination can also take place by subtraction (−). In either case, the combination of original and delayed signals creates a new waveform with a different spectrum. By inserting more delays or mixing sums and differences in various combinations, one can construct many different types of filters.

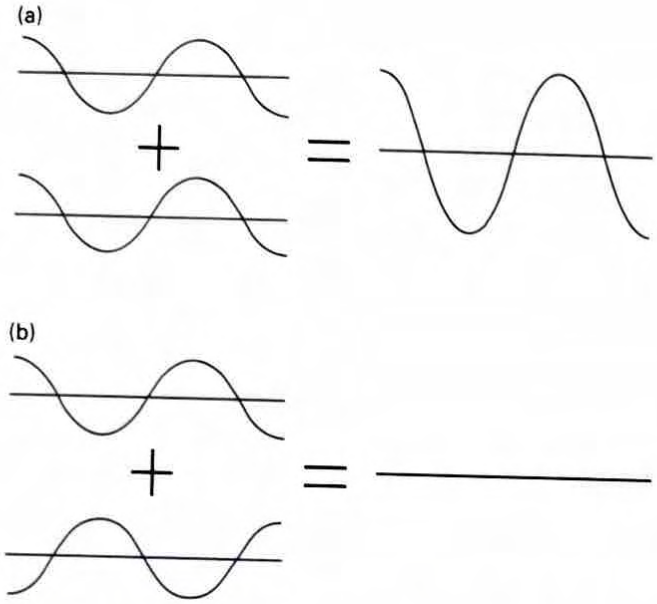


Figure 14.5 Phase effects. (a) Adding two in-phase signals results in a boost at that frequency. (b) Adding two out-of-phase signals results in phase cancelation.

Filter Response Curves

One of the main ways to characterize the various types of filters is to plot their *amplitude-versus-frequency response curve*, also referred to as *magnitude response*. Another common term for the same thing is *frequency response* (FR). The most accurate FR is a straight line, which indicates a *linear* or *flat* amplitude across the frequency spectrum. This means that any frequency within the range of the audio device is passed without any boost or attenuation. Figure 14.6a shows a nearly flat FR, typical of a high-quality audio cable or amplifier. Here we show an arbitrary upper limit of 25 kHz. For high-quality analog audio components such as preamplifiers and amplifiers, the FR may extend beyond 100 kHz. As chapter 3 explains, the frequency limits of digital audio systems depend on their sampling rate.

Many audio devices are not flat. Figure 14.6b shows the FR of a nonlinear system such as a cheap multimedia loudspeaker. We could describe the FR of this loudspeaker as +3.5, −3 dB from 100 Hz to 14 kHz. This means that the loudspeaker boosts some frequencies by as much as 3.5 dB and attenuates other frequencies by as much as 3 dB over the specified range. Below 100 Hz and above 14 kHz, the response falls off sharply. Because it alters the spectrum of a signal fed into it, the loudspeaker acts as a filter.

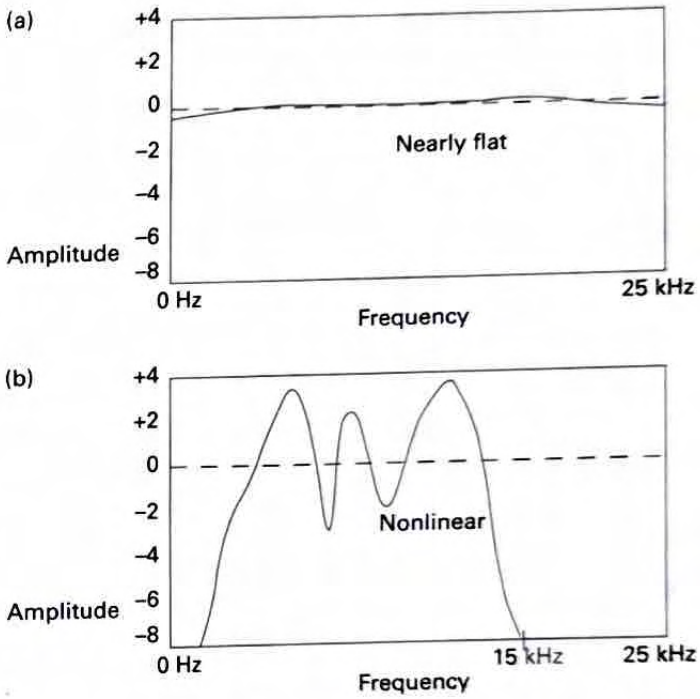


Figure 14.6 Amplitude-versus-frequency response, colloquially called *frequency response*. The vertical axis is amplitude in decibels, and the horizontal axis is frequency. (a) Nearly flat response. (b) Nonlinear response.

Filter Types

We can classify filters on the basis of their characteristic FR curve. Typical FR curves for four basic types of filters are shown in figure 14.7: *lowpass*, *highpass*, *bandpass*, and *bandreject* or *notch*. The names of the filters describe their function. Thus, a lowpass filter passes low frequencies and cuts high frequencies. A highpass filter does the opposite. A bandpass filter passes only a band or range of frequencies, usually in the middle of the spectrum. Finally, a bandreject or notch filter passes every frequency except a specific band

Shelving filters, shown in figure 14.8, boost or cut all frequencies above or below a given threshold. Their names can be confusing, because a *high shelving filter* acts like a lowpass filter when it is set to cut high frequencies, and a *low shelving filter* acts like a highpass filter when it is set to cut low frequencies.

Modern plug-ins let one combine several filters to fine tune the spectrum. Figure 14.9 shows the FabFilter Pro-Q application with its graphical interface for creating filter curves superimposed on a spectral display of the input signal.

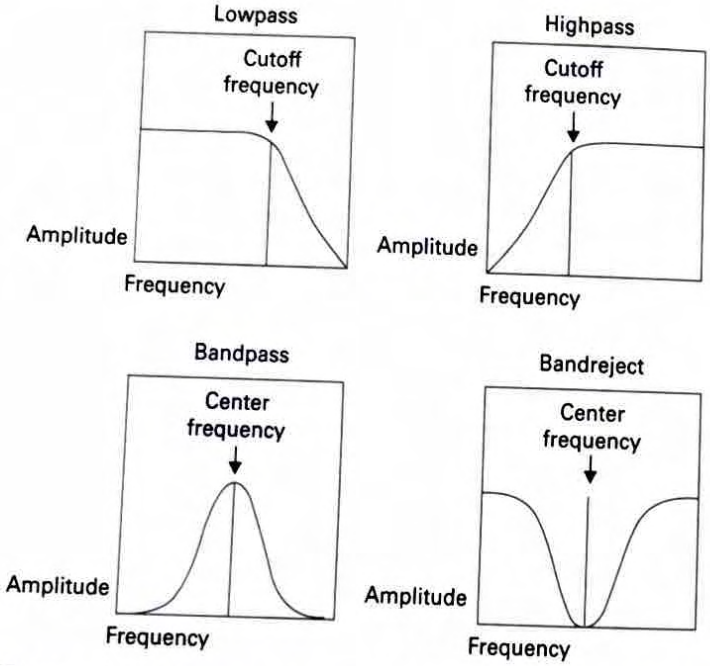


Figure 14.7 Four common types of filters. The cutoff frequency is the point at which the response diminishes. For a bandpass or bandreject filter, a common parameter is a center frequency around which a band of frequencies is affected.

An important property of a filter is its *cutoff frequency*. Figure 14.7 shows the cutoff frequency of lowpass and highpass filters. By convention, this is the point in the frequency range at which the filter reduces the signal to 0.707 of its maximum value. Why 0.707? The *power* of the signal at the cutoff frequency is proportional to the amplitude of the signal squared, because $0.707^2 = 0.5$. Thus, the cutoff frequency is also called the *half-power point.* Yet another term for the cutoff frequency is the *3 dB point* (Tempelaars 1977), so called because 0.707 relative to 1.0 is close to −3 dB.

Spectral components that are attenuated below the half-power point of a filter are said to be in the *stopband* of a filter. Those above the half-power point are said to be in the *passband* of the filter. The difference between the high and low cutoff frequencies in a bandpass filter is the *bandwidth* of the filter. The center frequency of a bandpass filter is the maximum point of amplitude; the center frequency of a bandreject filter is the minimum point of amplitude.

In an ideal sharp filter, the cutoff frequency is an absolute barrier: anything outside it is maximally attenuated, dividing the frequency response neatly into a passband and a stopband (figure 14.10a). In actual filters, the slope of the filter is not linear leading up to the cutoff frequency (there is

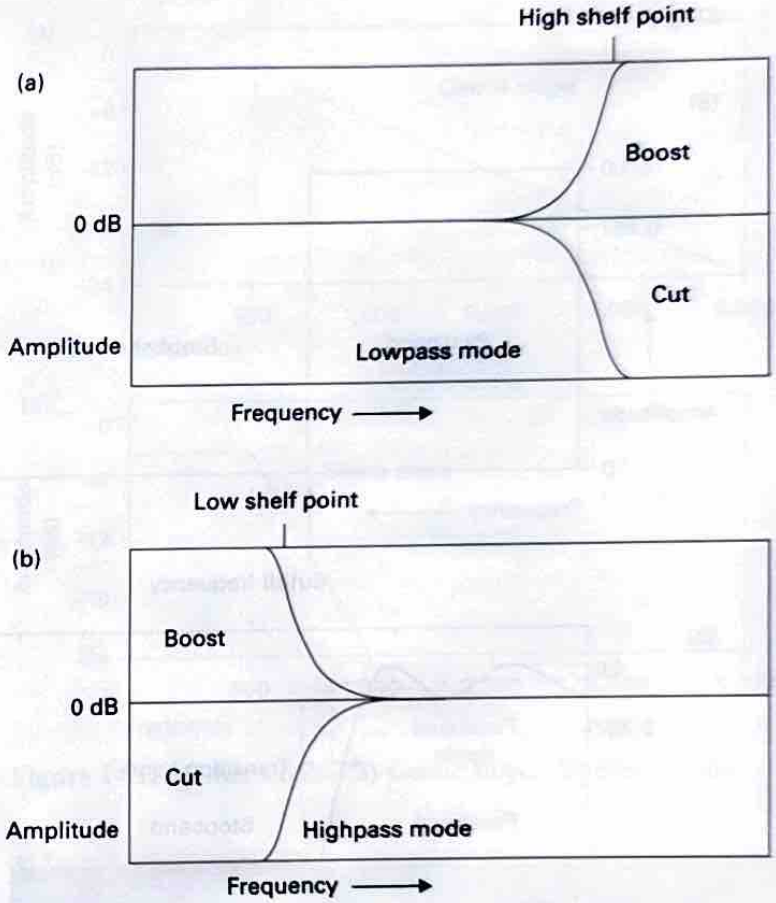


Figure 14.8 Shelving filters. (a) High shelving filter. Above the shelf point, the signal can be either boosted or cut. If the signal is cut, the effect of a high shelf filter is equivalent to a lowpass filter. (b) Low shelving filter. Below the shelf point, the signal can be either boosted or cut.

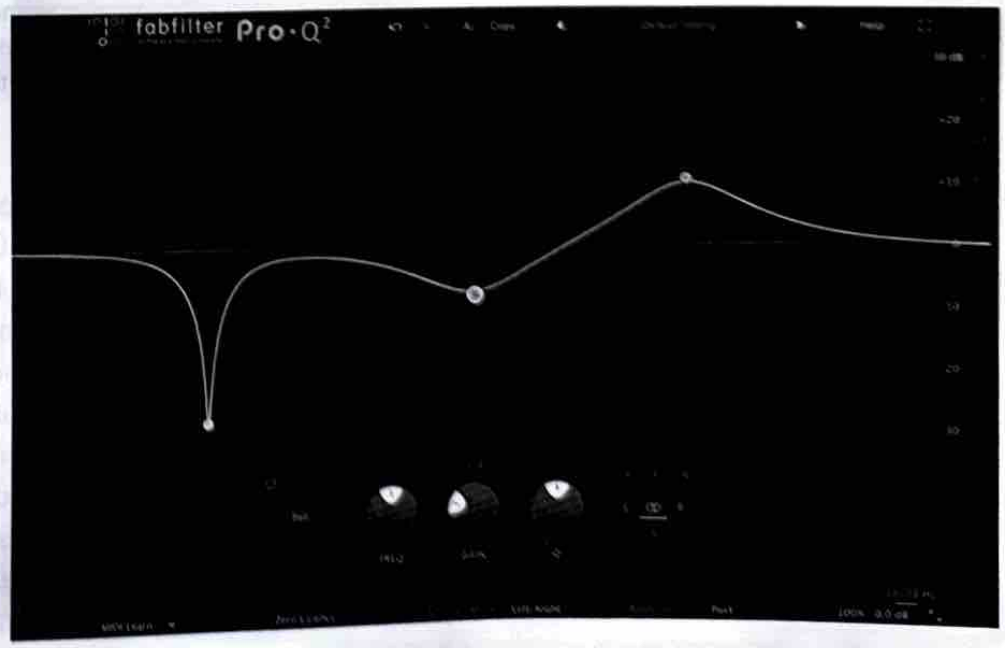


Figure 14.9 Screen image of FabFilter Pro-Q 2 with multiple filters superimposed.

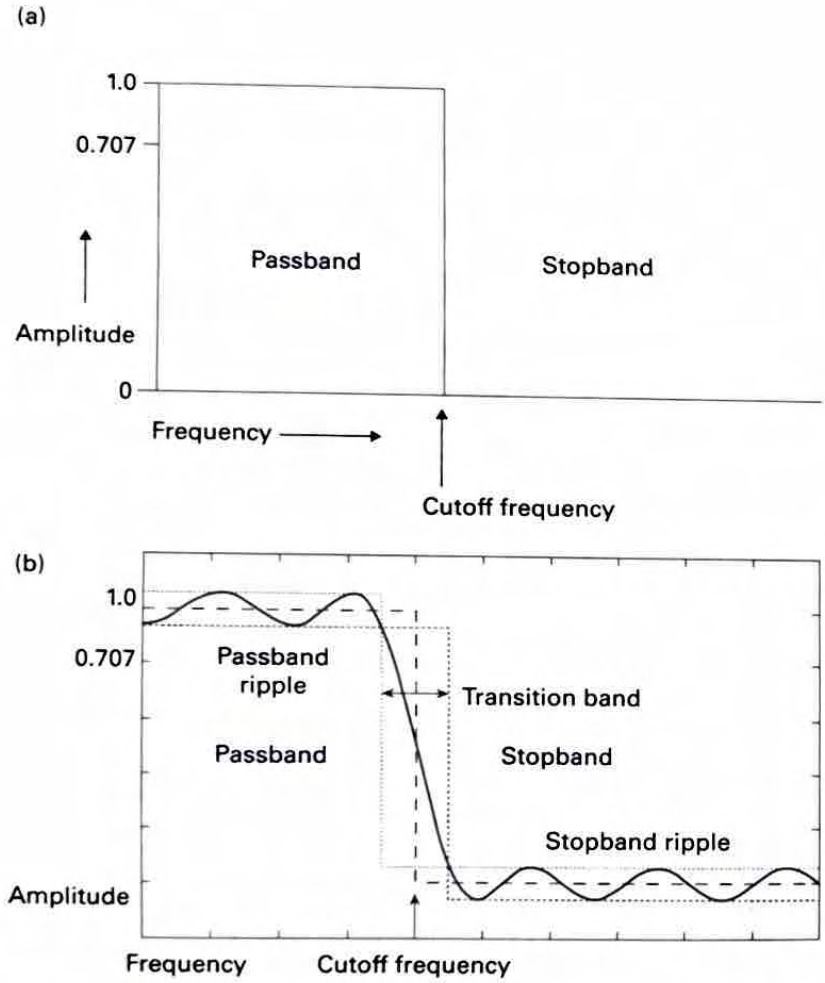


Figure 14.10 Ideal versus nonideal filters. (a) In an ideal filter, the frequencies affected by the filter can be neatly divided into a passband and a stopband, and the cutoff is rectangular. (b) In a nonideal (actual) filter, the response curve shows ripple, and there is a more or less steep transition band between the passband and the stopband.

a *ripple* in the frequency response), and the area between the passband and the stopband is called the *transition band* (figure 14.10b).

The steepness of a filter's slope is usually specified in terms of decibels of attenuation or boost per octave, compactly expressed dB/octave. For example, a 6 dB/octave slope on a lowpass filter makes a smooth attenuation (or *rolloff*), whereas a 24 dB/octave slope makes a sharp cutoff (figure 14.11).

A steep slope of 96 dB/octave is called a *brickwall* filter (figure 14.12). The use of a smooth or sharp slope depends on the musical situation. For example, a sharp notch filter might be needed to eliminate a tone centered on a particular frequency, whereas a gentle lowpass filter could be the most unobtrusive way of attenuating background noise in the high-frequency range.

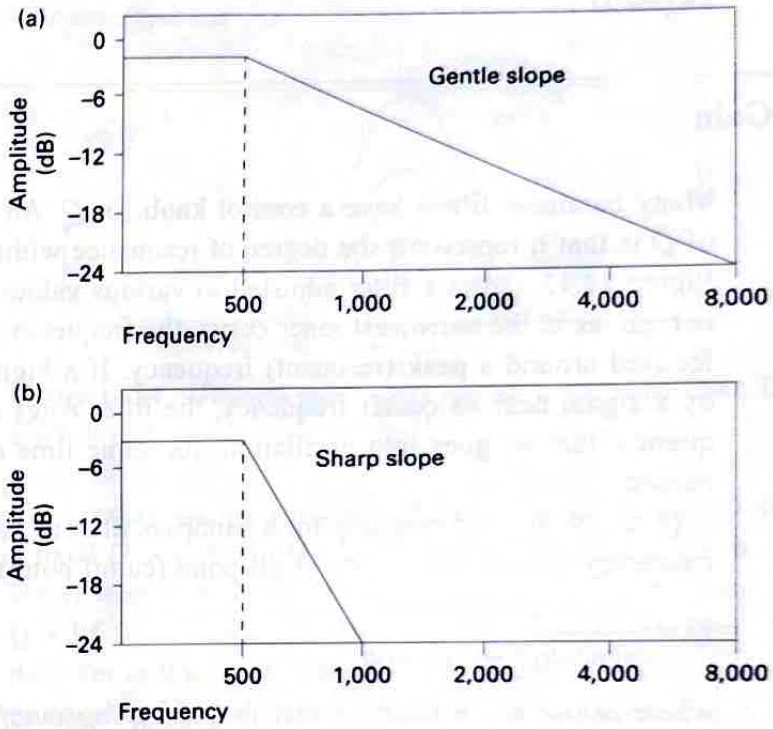


Figure 14.11 Filter slopes. (a) Gentle slope. (b) Steep slope.

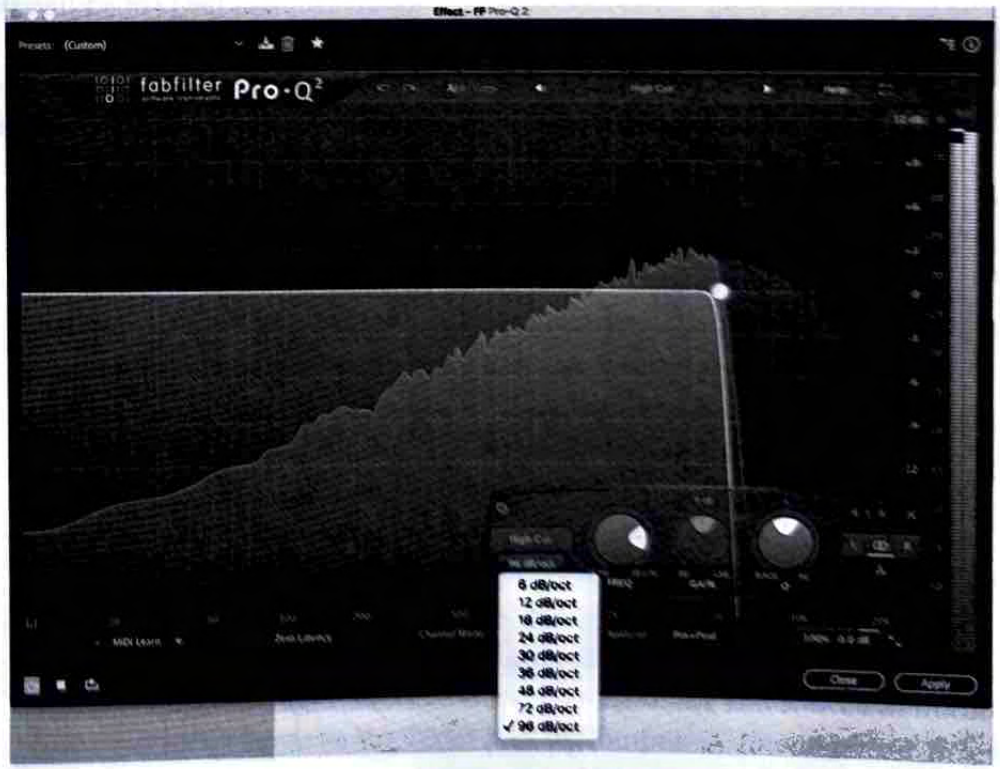


Figure 14.12 FabFilter Pro-Q 2 brickwall lowpass (or high cut) filter with a 96 dB/octave slope applied to a noise signal. The spectrum display shows before (darker) and after (lighter).

Filter Q and Gain

Many bandpass filters have a control knob for Q. An intuitive definition of Q is that it represents the degree of resonance within a bandpass filter. Figure 14.13 shows a filter adjusted to various values of Q. When the Q is high, as in the narrowest inner curve, the frequency response is sharply focused around a peak (resonant) frequency. If a high-Q filter is excited by a signal near its center frequency, the filter *rings* at the resonant frequency, that is, goes into oscillation, for some time after the signal has passed.

Q can be defined precisely for a bandpass filter as the ratio of the center frequency to the spread of its −3 dB point (cutoff point) bandwidth:

$$Q = \frac{center}{highcutoff - lowcutoff}$$

where *center* is the filter's center frequency, *highcutoff* is the upper 3 dB point, and *lowcutoff* is the lower 3 dB point. Notice that when the center frequency is constant, adjusting the Q is the same as adjusting the bandwidth. Here is an example of a calculation of the Q of a filter. We define a bandpass filter with a center frequency of 2,000 Hz and the 3 dB points of 1,800 and 2,200 Hz. This filter has a Q of $2{,}000/(2{,}200-1800)=5$. High-Q resonant filters like this are useful in generating percussive sounds. Tuned drums like tablas, wood blocks, claves, and marimba effects can be simulated by exciting a high-Q resonant filter with a pulse train.

Another property of a bandpass or bandreject filter is its *gain*. This is the amount of boost or cut of a frequency band. It shows up as the height (or depth) of the band in a response curve (figure 14.14). When passing a signal through a high-Q filter, care must be taken to ensure that the gain at the resonant frequency (the height at the peak) does not overload the system, causing distortion. Some systems have *gain-compensation* circuits in their filters that prevent this kind of overload.

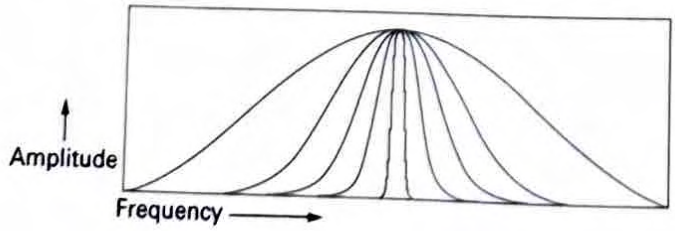


Figure 14.13 A filter set at various values for Q. A high Q corresponds to a narrow response. The gain (height of the peak) is constant.

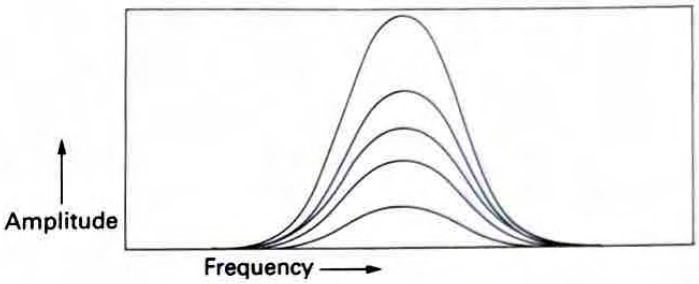


Figure 14.14 Different gain factors applied to the same filter. The bandwidth and Q remain constant.

A special type of bandpass filter is called a *constant Q filter*. To maintain a fixed Q, a constant Q filter must vary its bandwidth as a function of the center frequency. For example, when the center frequency is 30 Hz and the Q is 1.5 (or 3/2), the bandwidth is 20 Hz, because 30/20 = 1.5. But if we tune the filter to 9 kHz and keep the Q constant at 1.5, then the bandwidth must be equal to 2/3 of the center frequency, or 6,000 Hz. Figure 14.15 shows the curve of two constant Q filters plotted on linear and logarithmic frequency ranges. On a linear scale (figure 14.15a), the filter centered at 30 Hz appears as a very narrow band, whereas the filter centered at 9 kHz appears to have a much broader curve. On a logarithmic scale, the filters have the same shape (figure 14.15b).

A constant- Q filter has the musical quality that the frequency interval it spans does not change as the center frequency changes. For example, a constant Q filter centered at A440 Hz with a Q of 1.222 spans the same musical interval as a filter with a Q of 1.222 centered at A880 Hz (C260 to D620, as compared with C520 to D1240, respectively). An example of a constant Q filter is a *third-octave filter bank*, which is commonly used in sound level meters (Liski and Välimäki 2017).

Phase Response and Latency in Filtering

Audio filtering involves a trade-off between *linear phase response* and *latency*. A filter with linear phase response means that all frequencies are shifted in time by the same constant amount, preserving the waveshape without phase distortion. This time shift or *group delay* is a source of *audio latency*—a delay between the input and the output. A *linear-phase filter* processes signals without phase distortion, but it can introduce a significant latency (from a few milliseconds to more than 500 ms in the worst

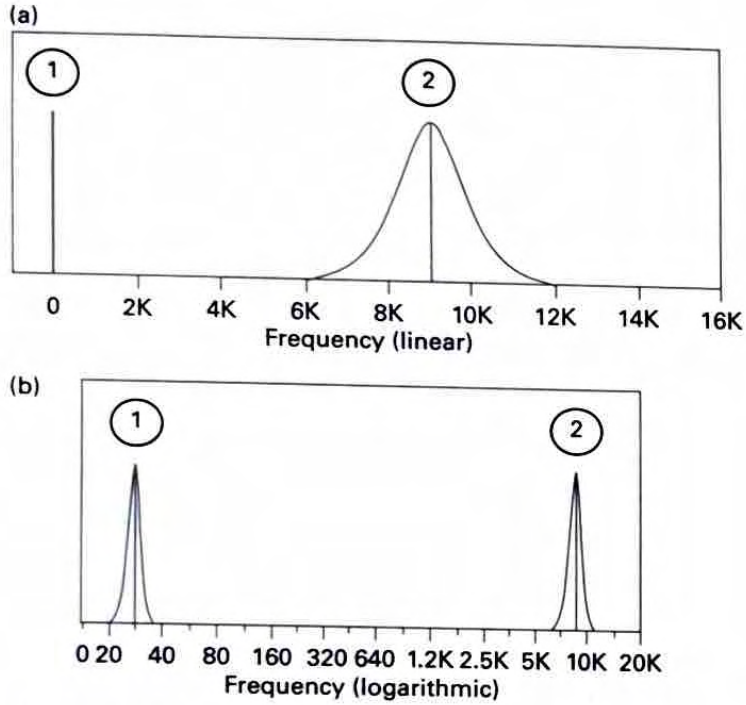


Figure 14.15 The same constant Q filters plotted on linear and logarithmic frequency ranges. Filter 1 has a center frequency of 30 Hz and extends from 20 to 40 Hz in bandwidth. Filter 2 has a center frequency of 9 KHz and extends from 6 to 12 KHz. (a) Linear. (b) Logarithmic.

case) in the sound. A linear-phase filter with a long latency can also introduce pre-echoes that can make low-frequency transients such as bass drums lose their percussive edge.

By contrast, *zero-latency filters* process signals efficiently and without delay, but they introduce *phase distortion.* This is a frequency-dependent time shift that can in some cases audibly blur the audio signal, particularly transients. Zero-latency filters are best used in live performance situations in which latency is unacceptable.

In the studio there is no fixed rule for choosing a linear phase versus a zero-latency filter; it is a matter of testing and listening.

Filter Banks and Graphic Equalizers

A *filter bank* is a group of filters that are fed the same signal in parallel (figure 14.16). The filter bank splits a sound into multiple frequency bands called *subbands* (Cook 2007). Each filter is typically a narrow bandpass filter centered around a specific frequency. The filtered signals are often

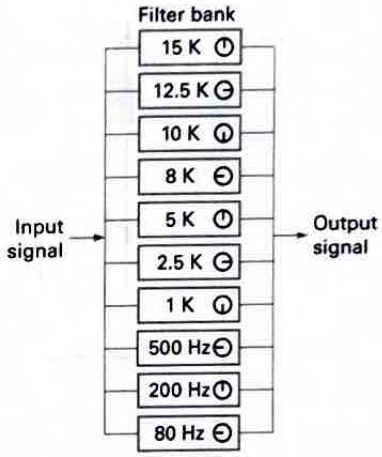


Figure 14.16 A ten-stage filter bank or spectrum shaper with a control knob (boost or attenuate) associated with each frequency band.

combined to form the output sound. When each filter has its own level control, the filter bank is called a *spectrum shaper* because the individual controls can radically modify the spectrum of the input signal. A spectrum shaper can be used to boost certain frequencies or virtually eliminate others.

A *graphic equalizer* (Liski and Välimäki 2017) is a spectrum shaper that has faders that visually mirror the shape of the filter bank's frequency response curve (figure 14.17a). Each filter has a fixed center frequency, a fixed bandwidth, and a fixed Q. (Some units can switch between several Q settings.) The response of each filter can be varied by means of a linear fader to cut or boost specific frequency bands. The potential frequency response of such a filter is shown in figure 14.17b.

A *parametric equalizer* involves a smaller number of filters, but the control of each filter is more flexible. A typical arrangement is to have three or four filters in parallel. Users can adjust independently the center frequency, the Q, and the amount of cut or boost of each filter. A *semiparametric equalizer* has a fixed Q.

Comb and Allpass Filters

Two more filter types merit mention here; they are discussed thoroughly in chapter 28. A filter that has several regularly spaced sharp curves in its frequency response is called a *comb filter*. Figure 14.18 shows the frequency response curves of two types of comb filters. The filter shown in

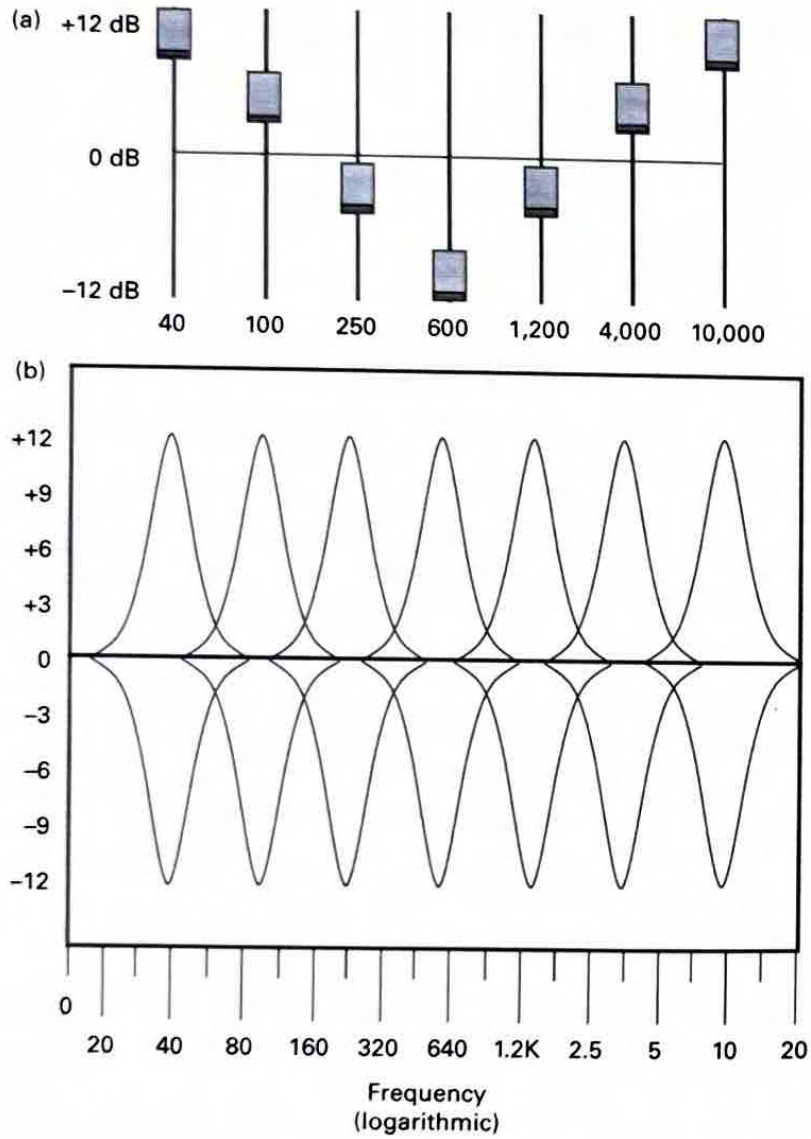


Figure 14.17 Graphic equalizer. (a) A seven-band graphic equalizer with linear potentiometers set to arbitrary levels. (b) The potential frequency response curve of a seven-band graphic equalizer.

figure 14.18a has deep notches in its response, whereas that shown in 14.18b has steep peaks. The derivation of the term *comb* should be clear from these curves.

Whereas low-, high-, band-, and notch filters are implemented with tiny time delays in the range of few microseconds, the delays for a comb filter range up to 10 ms. (See chapter 30 on delay effects.)

The sonic effect of a comb filter is distinct and recognizable. Comb filters can be used to make variations of a sound. For example, a snare drum hit passing through a comb filter will retain the identity of the drum but will

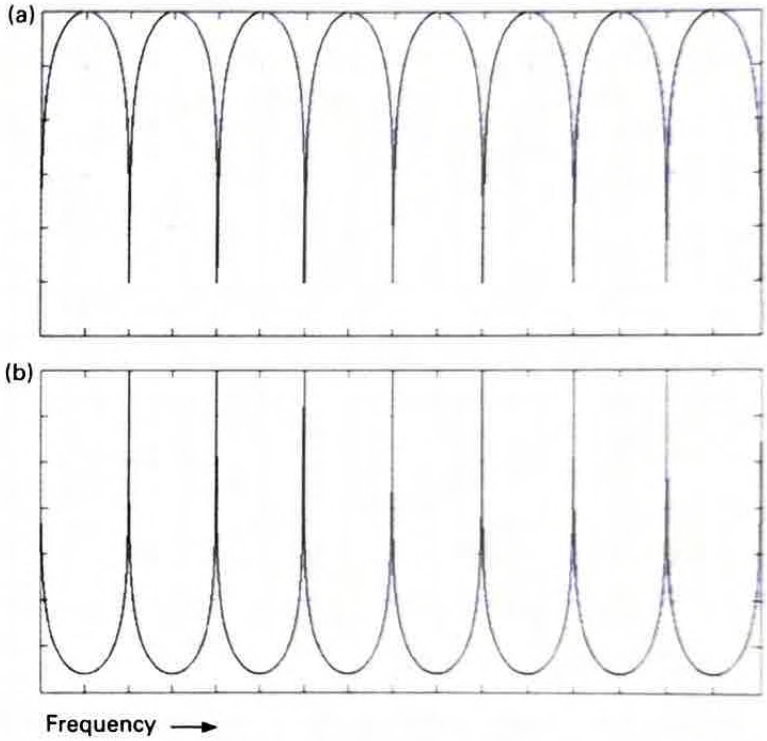


Figure 14.18 Comb filter frequency response curves. The vertical scale is amplitude, and the horizontal scale is frequency plotted linearly. This filter has peaks and troughs every 2,000 Hz. (a) Finite impulse response (FIR) comb. (b) Infinite impulse response (IIR) comb.

have a different timbre due to the comb spectral effect. By nudging the delay time slightly, one can make a variety of different snare drum hits. If we vary the delay time continuously the result is a dramatic *flanging* effect (see chapter 30).

The final filter to mention is an *allpass filter*. For a steady-state (unchanging) sound fed into it, an allpass filter passes all frequencies equally well with unity gain, hence its name. The purpose of an allpass filter is to introduce a frequency-dependent phase shift.

If the input signal varies in time (i.e., is not steady-state), the allpass filter colors the signal, due to frequency-dependent phase-shifting effects. This coloration can be subtle, but it is particularly evident on transient sounds (e.g., cymbals) for which phase relations are so important to sound quality.

A practical application of an allpass filter is to correct for the unwanted phase shift of another filter. A square wave with out-of-phase partials will not look like a square wave. An allpass filter can restore the square shape.

Allpass filters are one of the components of digital reverberators, specifically those based on Schroeder's model (see chapter 33). Allpass filters

produce frequency-dependent time shifts. When delay times within the filter are in the range of about 1 to 30 milliseconds, a series of allpass sections vastly increases the echo density or *diffusion* of the reverberated sound. For this reason, allpass filters are sometimes referred to as *impulse diffusers* (Smith 2018).

Time-Varying Subtractive Synthesis

Filters can be *fixed* or *time-varying*. In a fixed filter, the properties of the filter are set and do not change over time. For example, in mixing a pop music track, most of the time the filter settings are tuned in a setup stage and left at a fixed position once the engineer has found the preferred sound color. In the context of mixing, filtering is often referred to as *EQ*. EQ stands for *equalization* and derives from one of the earliest applications of filters, namely, to compensate for irregularities in the frequency response of telephone channels and public address systems. In effect, the filter equalizes the amplitude of all frequencies (Fagen 1975).

In music synthesis, however, we want time-varying filters. Specifically, we want to vary the center or cutoff frequency, bandwidth, gain, and Q in real time. For example, a signature sound of the famous Moog synthesizer featured a sawtooth waveform processed by a high-Q lowpass filter (the Moog 904 filter module) with a sweeping sound as the cutoff frequency moved up and down. A bandpass filter whose Q, center frequency, and bandwidth change over time can impose an enormous variety of sound colorations, particularly if the signal being filtered is also time-varying.

An example of a time-varying filter is a parametric equalizer section in a digital audio workstation or DAW. The mixing engineer can change the Q, center frequency, and amount of cut or boost at any time during the mixing process, or these parameters can all be automated.

Implementing a time-varying filter in the digital domain requires interpolating the filter's internal parameters. As chapter 28 explains, designing an arbitrary time-varying filter is a nontrivial mathematical process, but contemporary filter design tools automate much of this burden.

SYTER and GRM Tools

A notable historical example of a system for time-varying subtractive synthesis was the SYTER audio processor developed in the late 1970s at the Groupe de Recherches Musicale (GRM) studio in Paris (Allouis 1979; Allouis and Bernier 1982; Geslin 2002). SYTER realized several dozen high-Q bandpass

filters in real time with dynamic parameter changes. When full-bandwidth sounds such as water and wind were processed through the system, the resonant filters rang in musical chords and clusters. An outstanding example of the use of resonant filters is Jean-Claude Risset's 1985 composition *Sud* using filter programs by B. Maillard and Y. Geslin. Rich comb filter and phasing effects could also be created. These algorithms have since been ported to the GRM Tools software plug-in package (INA/GRM 2019).

Subtractive Analysis/Resynthesis

As with additive synthesis, the power of subtractive synthesis is enhanced by an initial sound analysis stage. This combination is called *analysis/resynthesis* or just analysis/synthesis. Most of the analysis techniques employed in subtractive analysis/resynthesis are geared toward speech synthesis, because this is where most of research has been concentrated (Flanagan et al. 1970; Flanagan 1972).

Music research using subtractive analysis/resynthesis has focused on extending speech-oriented tools (such as linear predictive coding, discussed subsequently in this chapter), to the domain of musical sound, such as singing and simulation of string ensembles.

The Vocoder

The original subtractive analysis/resynthesis system is the *vocoder* (also called *channel vocoder*) demonstrated by a talking robot at the 1936 World's Fair in New York City (Dudley 1936, 1939a, 1939b, 1955; Dudley and Watkins 1939; Schroeder 1966; Flanagan 1972; Cook 2007; Puckette 2007; Arfib et al. 2011a). We present a different system known as the *phase vocoder* in chapter 37.

A classic analog vocoder consists of two stages. The first stage is a group of fixed-frequency bandpass filters distributed over the audio bandwidth. The output of each filter is connected to an *envelope detector* that generates a voltage proportional to the amount of energy at the frequency tracked by the filter (figure 14.19). The second stage of the vocoder is a bank of bandpass filters, identical to the first stage. All filters are sent the same input signal, and the output of each filter is sent to its own *voltage-controlled amplifier* (VCA). The outputs of all the VCAs combine into an output signal. The filters and detectors in the first stage generate control signals (also called *driving functions*) that determine the amplitude of the audio signal coming from the filters in the second stage of the vocoder. Today vocoders are also implemented with digital technology (Cook 2007). Arfib et al. (2011a) provide code in MATLAB.

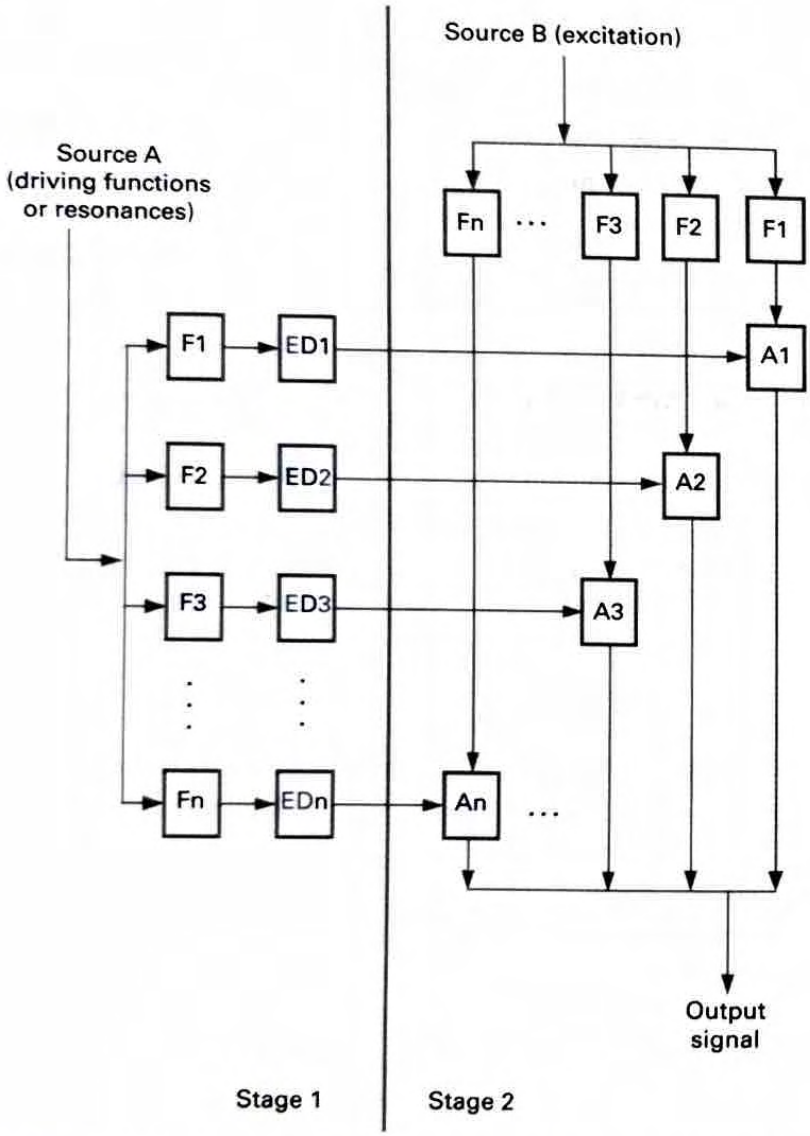


Figure 14.19 Vocoder. Stage 1 is the analysis part, and stage 2 is the synthesis. *F* stands for filter, *ED* stands for envelope detector, and *A* stands for voltage-controlled amplifier—an amplifier whose gain is determined by a control voltage fed into it from the envelope detector. The same structure can also be realized in digital form.

In figure 14.19, source A is the signal from which the formant spectrum is derived, such as a singing voice. If we were to trace an outline of this spectrum it would form the *spectrum envelope.* Source B is the *excitation function* from which the pitch is derived. The excitation function is usually a wide-bandwidth signal such as white noise or a pulse train. The output of this vocoder consists of the excitation function of source B with the time-varying spectral envelope of the singing voice of source A.

Figure 14.20 depicts graphically the process of formant filtering applied to a broadband excitation function.

In musical applications, the separation of the control signals from the excitation signal means that rhythm, pitch, and timbre are independently controllable. For example, a composer can change the pitch of a singing voice (by changing the frequency of the excitation) but retain the original spectral articulation of the voice. By stretching or shrinking the control functions over time, a piece of spoken text can be slowed down or sped up without shifting the pitch or affecting the formant structure. Another common effect is to impose the spectral shape of one sound on another: *cross-synthesis*. The classic example is a *talking orchestra* effect in which the voice provides the driving functions and the orchestra (or a keyboard synthesizer playing chords) is the excitation. Some vocoders have a highpass filter *sibilance band* that passes consonant sounds directly to the output, making for a more intelligible speech effect.

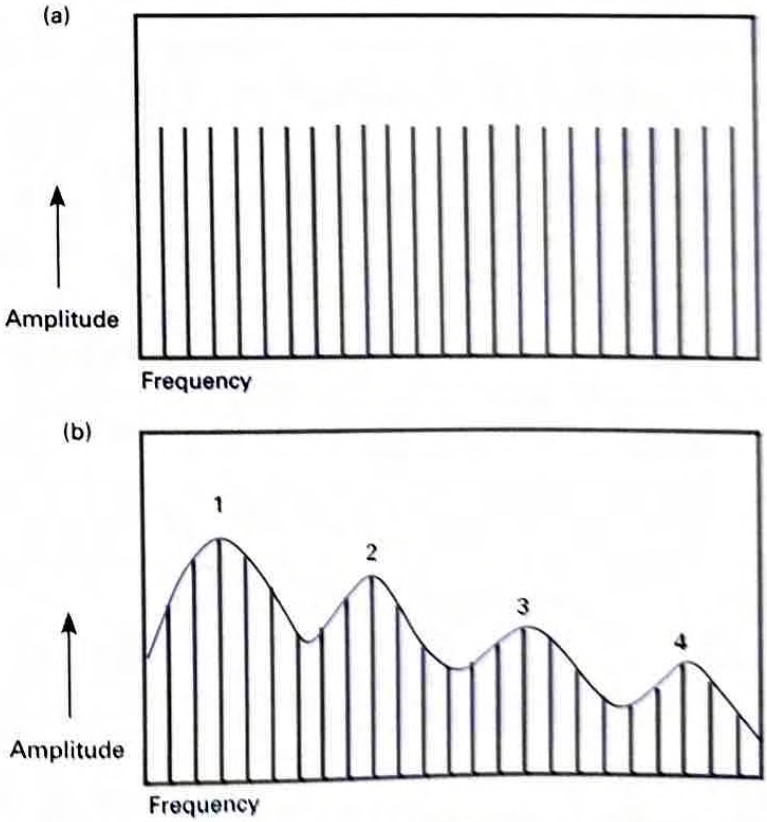


Figure 14.20 The effect of formant filters on an excitation function. (a) Simplified view of an excitation function like the spectrum produced by the open vocal cords; a buzz sound with a number of equal-strength harmonics. (b) Simplified view of the spectrum of a vowel showing four formant peaks labeled 1, 2, 3, and 4.

Linear Predictive Coding

Linear predictive coding (LPC) or *linear prediction* is a subtractive analysis/resynthesis method that has been used in speech and music applications (Atal and Hanauer 1971; Flanagan 1972; Makhoul 1975; Markel and Gray 1976; Cann 1978, 1979–80; Moorer 1977, 1979c; Dodge 1985; Lansky 1987; Lansky and Steiglitz 1981; Hutchins 1986a; Dodge 1989; Depalle 1991; Arfib et al. 2011a). Bishnu Atal, one of the inventors of LPC, wrote a history of the technique (Atal 2006). LPC can be viewed as a type of vocoder. Unlike the previously described vocoder with its fixed frequency bands, LPC's filters are more flexible.

LPC takes in a sound, such as a speaking voice, analyzes it into a greatly data-reduced form, and resynthesizes an approximation of it. LPC is computationally efficient in the sense that it requires much less data than sampled sound. For example, an integrated circuit for LPC speech was developed and built into speaking toys (Brightman and Crook 1982). From the standpoint of a musician, the power of the LPC technique derives from the fact that one can edit the analysis data and resynthesize variations on the original input signal. Like any vocoder, LPC separates the excitation signal from the resonance or spectrum envelope, making it possible to manipulate rhythm, pitch, and timbre independently and permitting a form of cross-synthesis, explained further on.

The Csound language supports LPC analysis and synthesis (Boulanger 2000), as does MATLAB. The SonicCharge Bitspeek plugin automatically analyzes an input sound such as voice and lets one adjust eight LPC playback parameters in real time using switches and knobs onscreen (figure 14.21).

In speech, the vocal cords generate a buzzy pulsatile excitation waveform. These pitched tones are called *voiced* sounds, such as *ah* and *oh*. *Unvoiced* breaths (with vocal cords silent) produce consonant sounds like *s*, *f*, *t*, and *p*. The rest of the vocal tract filters the sound to create resonances. Because LPC lets users manipulate the excitation independently, one can vary the pitch to transform a talking voice into a singing voice, for example, as the composer Charles Dodge did in his iconic *Speech Songs* (1973).

What Is Linear Prediction?

Linear prediction derives its obscure name from the fact that in the analysis part of the system, output samples are predicted by a linear combination of internal filter parameters or *coefficients* and previous samples. In predicting the next sample, the system already knows the correct result, which comes

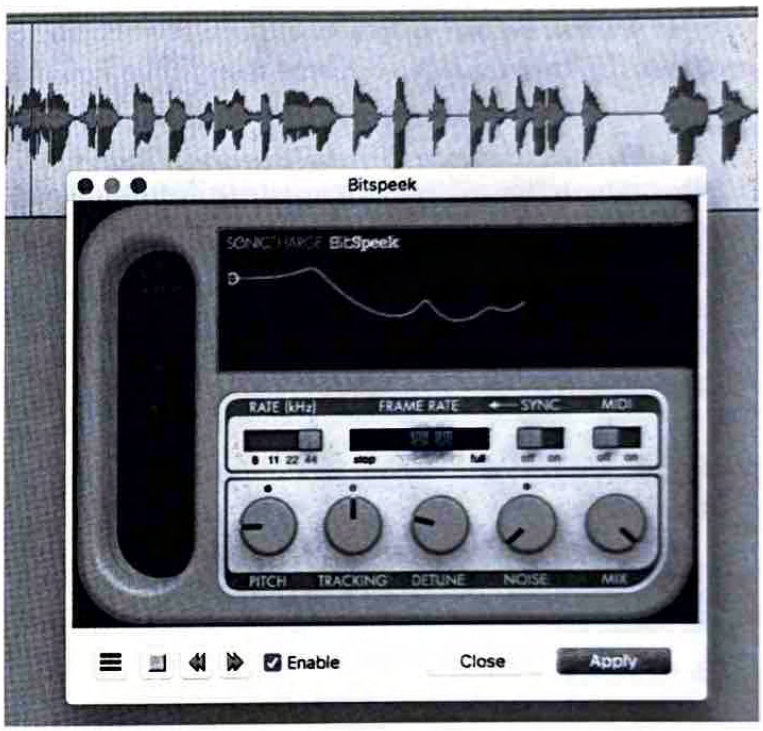


Figure 14.21 BitSpeek plug-in for LPC analysis/resynthesis. The user interface resembles the Speak & Spell toy made by Texas Instruments.

from the input signal. The goal of the analysis is to replace the input signal with a compact LPC model of it. From a musical standpoint, this compact model breaks the input signal into a handful of parameters that enable transformations of pitch, amplitude, duration, and timbre.

A *linear prediction algorithm* based on a difference equation tries to find samples at positions after a region from which one already has samples. Any extrapolation of a set of samples is prediction. Inherent in prediction is the possibility of being wrong; thus prediction algorithms always include an error estimation.

A simple predictor simply continues the slope of difference between the last sample and the sample before it. A simple predictor can be made more sophisticated by taking more samples into account. It can also factor in the error or difference between the sample it predicts and the known value of the input signal.

A key concept to understand is that because the predictor is looking at sums and differences of time-delayed samples in doing its prediction, it can be viewed as a filter that describes the waveform that it is currently processing. We will not go into the prediction algorithm here. Loy (2007) gives a detailed and musically informed description. Also refer to Arfib et al. (2011a).

If we take regular snapshots of these filter coefficients over time, invert them, and then drive the resulting filter with a rich, wide-bandwidth sound, we should have a good approximation of the time-varying spectrum of the original input signal. Thus a side effect of the prediction is to estimate the spectrum of the input signal; this is an important point. However, spectrum estimation is only one stage of LPC analysis. Pitch, amplitude, and the voiced/unvoiced decision are also analyzed. These are briefly described in the next section.

LPC Analysis

Figure 14.22 shows an overview of LPC analysis. LPC analysis branches in the following four directions:

1. Spectrum analysis in terms of formants
2. Pitch analysis
3. Amplitude analysis
4. Decision as to whether the sound was voiced (pitched, like *ah*) or unvoiced (characteristic of noisy consonants like *sh*).

Each stage of analysis is carried out on a frame-by-frame basis, where a frame is like a snapshot of the signal. Frame rates between 50 and 200 frames per second are typical in LPC analysis.

Filter estimation

The next several paragraphs describe the operation of LPC analysis in general terms. First let us state a point about filter terminology used in LPC analysis. Engineers describe bandpass and bandreject filters in terms of the position of their *poles* and *zeros*. Without going into the details of *pole-zero diagrams*, let us just say that a filter pole is a point of resonance—a peak or formant region in a spectrum plot. In contrast, a zero is a null point or notch in the spectrum. (Chapter 28 introduces pole-zero diagrams.) For more on pole-zero diagrams in filter design, refer to Puckette [2007], Loy [2007], and Smith [2007a].)

If a filter has several smooth peaks it is called an *allpole filter*. This type of filter is characteristic of LPC, which models spectra in terms of several formant peaks. Such a model is a reasonable approximation to many sounds uttered by the human voice and certain musical instruments.

Linear prediction—also called *autoregressive* analysis—takes in several input samples at a time, using the most recent sample as a reference. It tries

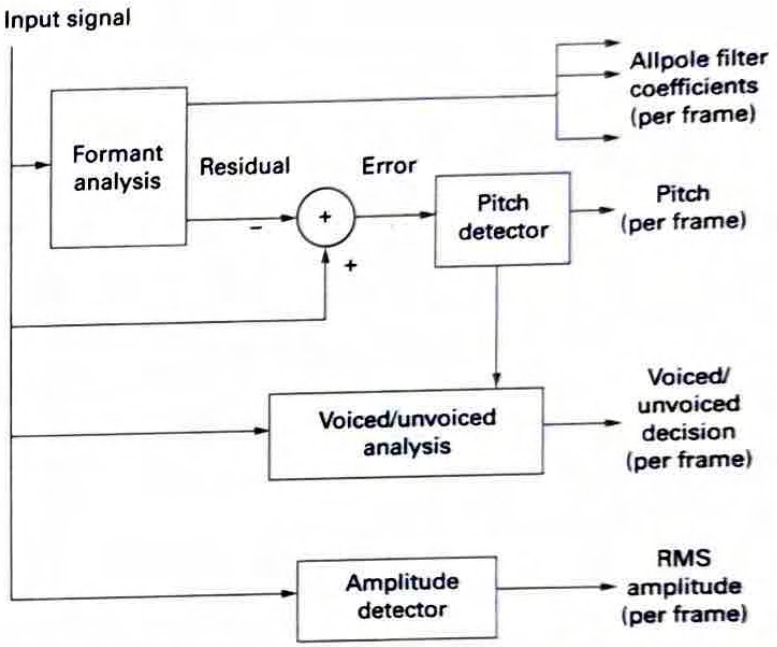


Figure 14.22 Four stages of LPC analysis. Spectrum (formant) analysis, pitch detection, amplitude detection, and voiced/unvoiced analysis.

to predict this sample from a weighted sum of filter coefficients and past samples. As a side effect of this prediction, the algorithm fits an inverse filter to the spectrum of the input signal. The inverse of an allpole filter is an *allzero filter* that creates a number of notches in the spectrum of signals sent through it.

The LPC analyzer approximates the inverse of the filter that one ultimately wants for synthesis. If the approximation is good, the result of linear prediction should be just the excitation signal (figure 14.23). In other words, the inverse filter cancels the effect of the spectrum envelope of the sound. The approximation is never perfect, so there is always a signal called the *residual* that is the excitation function (a series of impulses) plus noise. The goal of LPC spectrum analysis is to minimize the residual.

Once a good fit to the inverse filter has been found, the inverse filter is itself inverted to create a resynthesis filter. Filter inversion is mathematically straightforward. The sign of all the filter coefficients is reversed, and they are applied to past outputs instead of to past inputs (Rabiner and Gold 1975). The filter is thus converted from a FIR filter to an IIR filter (see chapter 28). For an engineering description, refer to Markel (1972), Makhoul (1975), Moore (1990), and Smith (2007a).

The reader may wonder, how does LPC know what is the excitation function of an arbitrary sound put into it? The answer is that it assumes that the

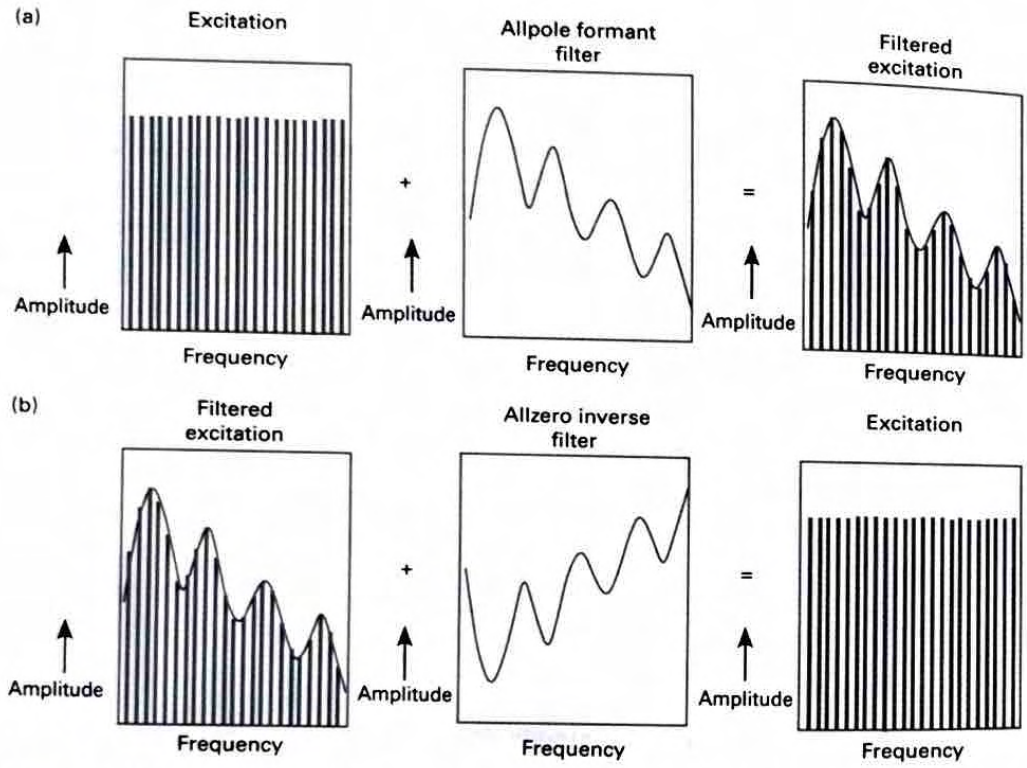


Figure 14.23 Relation of formant and inverse formant filters in the ideal case. (a) Result of formant filter. (b) Result of inverse formant filter.

excitation is either a pitched pulse train or white noise. This assumption works reasonably well in approximating speech and some instruments, but it is not a universal model for all sounds. Thus LPC usually leaves traces of artificiality in the resynthesized sounds.

Improved methods of LPC analysis create a custom pulse train that mixes pitched pulses and noise from segments stored in a codebook or dictionary. This method, called *code-excited linear prediction,* generates a unique excitation signal from analysis of the input signal. It tries to match segments of the input with segments stored in its codebook (Schroeder and Atal 1985). This reduces the artificial quality of LPC resynthesis.

Pitch and amplitude analysis

The pitch detection technique used in LPC can be any of the methods described in chapter 34. Figure 14.22 shows a scheme that tries to estimate the pitch from the residual signal.

Several techniques exist for characterizing the amplitude of each frame. A typical way is to calculate it on a frame-by-frame basis as an average value for the input waveform described by the frame.

Voiced/unvoiced decision

After pitch detection has been carried out, LPC analysis tries to make the voiced/unvoiced decision for each frame. This decision is important because it determines whether the sound will be pitched or not. As previously mentioned, a voiced sound has a pitch, like the vowels created by the buzzing of the vocal cords. An unvoiced sound is like the sibilance of *s* and *sh*, the plosive *t* and *p*, or the fricative *f* consonants. Besides voiced or unvoiced, a third category of excitation is *mixed voice*, combining a pitched tone and noise, like the *z* in *azure*.

In analyzing a wind instrument tone, the voiced/unvoiced data usually indicate the amount of breathiness, and for a violin-like sound they can indicate bow-scraping noise. In resynthesis, voiced sounds are modeled by a pitched pulse train and unvoiced sounds are modeled by white noise. Both are filtered, of course.

The voiced/unvoiced decision is typically based on statistical estimate using various features of the signal. It is hard to fully automate (Hermes 1992). In LPC systems that have been adapted for music, the analysis makes a first pass at the decision, but the composer is expected to make corrections to particular frames (Moorer 1979). The first-pass decision uses various heuristics. Figure 14.22 shows the result of pitch detection feeding into the voiced/unvoiced decision. For example, if the analysis cannot identify a pitch in the input signal, then it generates a large pitch estimation error. When this error—normalized to fall between 0 and 1.0—is greater than a certain value (around 0.2), it is likely that the sound at that moment is a noisy unvoiced sound like a consonant. The average amplitude of the residual is another clue. If the amplitude of the residual is low in comparison to the amplitude of the original input signal, then the signal is probably voiced.

Analysis frames

The result of the analysis stage is a series of frames, representing a greatly data-reduced version of the input signal. Each frame is described by a list of parameters, including the following:

Average amplitude of the residual sound (RMS1)

Average amplitude of the original sound (RMS2)

Ratio of the two amplitudes, which helps to determine whether the frame is voiced or unvoiced (ERR)

Estimated pitch (PITCH)

Frame duration (DUR)

Coefficients for the allpole filter (each pole creates a formant peak in the spectrum).

Figure 14.24 shows an example of the frame data for the word *sit* (Dodge 1985). The filter coefficients are omitted.

The ERR column is a strong clue to whether the frame is voiced or not. A large value for ERR (greater than 0.2) usually indicates an unvoiced frame. However, as mentioned previously, this indicator should be checked because a voiced/unvoiced decision is hard to automate perfectly. Notice how the ERR values change significantly at the boundary of S and I. The RMS1 and RMS2 values are a better indicator of change at the boundary of I and T.

LPC Synthesis

Figure 14.25 depicts the synthesis stage of LPC. The first parameter is the frame duration–the number of output samples generated. The next parameter determines whether the frame is voiced or unvoiced. For voiced frames, the synthesizer uses the pitch parameter to simulate the excitation function (the glottal wave) of the human voice. This is a buzzy bandlimited pulse train used for vowels and diphthongs. (A diphthong is a sequence of vowels such as the *oy* in *toy*.) For unvoiced frames, the synthesizer uses a noise generator to simulate turbulence in the vocal tract.

The output of the appropriate generator, shaped by the amplitude parameter, serves as input to the allpole filter. For speech and singing, the allpole filter simulates the resonances of the vocal tract. Up to twelve poles in the allpole filter are used in speech synthesis, and as many as 250 poles have been used in music synthesis (Moorer 2019).

Editing LPC Frame Data

The LPC technique can be adapted to musical purposes by means of an associated editing and mixing subsystem. Charles Dodge (1985) described an editing command language that performs operations on LPC parameter frames such as the following:

Set the pitch of a frame

Trill on every other frame

Stretch or shrink the duration of a frame

Stretch or shrink the duration of a group of frames

Change parameter values in a frame

Phoneme	Frame	RMS2	RMS1	ERR	PITCH	DUR
S	197	813.27	1618.21	0.252	937.50	0.010
	198	1189.36	2090.14	0.323	937.50	0.010
	199	553.71	838.38	0.436	937.50	0.010
	200	742.59	1183.17	0.393	937.50	0.010
	201	1041.95	1918.33	0.295	123.95	0.010
	202	1449.16	2677.06	0.293	123.95	0.010
	203	1454.84	2920.50	0.248	937.50	0.010
	204	1430.03	2496.88	0.348	937.50	0.010
	205	1570.88	2981.21	0.277	142.84	0.010
	206	1443.27	2665.22	0.293	142.84	0.010
	207	1172.67	2150.50	0.297	150.00	0.010
	208	1200.73	2080.20	0.333	150.00	0.010
	209	1095.51	2055.25	0.284	116.26	0.010
	210	1260.36	2408.14	0.273	116.26	0.010
	211	1105.17	2293.05	0.232	937.50	0.010
	212	809.10	1659.80	0.237	937.50	0.010
	213	428.20	784.93	0.297	250.00	0.010
I	214	419.45	3886.15	0.011	250.00	0.010
	215	925.86	6366.20	0.021	208.32	0.010
	216	746.28	8046.81	0.008	208.32	0.010
	217	829.82	8277.42	0.010	192.29	0.010
	218	754.64	8049.50	0.008	192.29	0.010
	219	771.84	8001.70	0.009	197.35	0.010
	220	726.81	7955.17	0.008	202.69	0.010
	221	807.63	7835.20	0.010	202.69	0.010
	222	874.27	7732.59	0.012	205.42	0.010
	223	776.87	7491.86	0.010	205.42	0.010
	224	684.64	7317.04	0.008	205.42	0.010
	225	560.87	6297.36	0.007	102.03	0.010
	226	175.63	1842.81	0.009	102.03	0.010
	227	46.53	1329.09	0.001	197.85	0.010
T	228	38.25	793.00	0.002	197.85	0.010
	229	39.26	316.92	0.032	202.69	0.010

Figure 14.24 A sequence of LPC frames as they might be displayed for editing purposes (after Dodge [1985]). The Phoneme column is added for clarity. The RMS2 column indicates the residual amplitude, RMS1 is the original signal amplitude. ERR is an approximation to the ratio between the two and indicates an unvoiced signal if the ratio is high. PITCH is the estimated pitch in Hz, and DUR is the frame duration in seconds.

Interpolate values between a group of frames (e.g., create a pitch glissando)

Move frames from point A to point B

Boost or diminish the amplitude of a frame

Boost or diminish the amplitude of a group of frames (crescendo/diminuendo)

One of the main applications of these operations on LPC frames is to transform a plain spoken utterance into singing. Using LPC, a word can be expanded in time, and the original spoken pitch curve can be replaced with a flowing melody. Words and phrases can be repeated or rearranged at will. Sentences can also be compressed in time without affecting their original pitch.

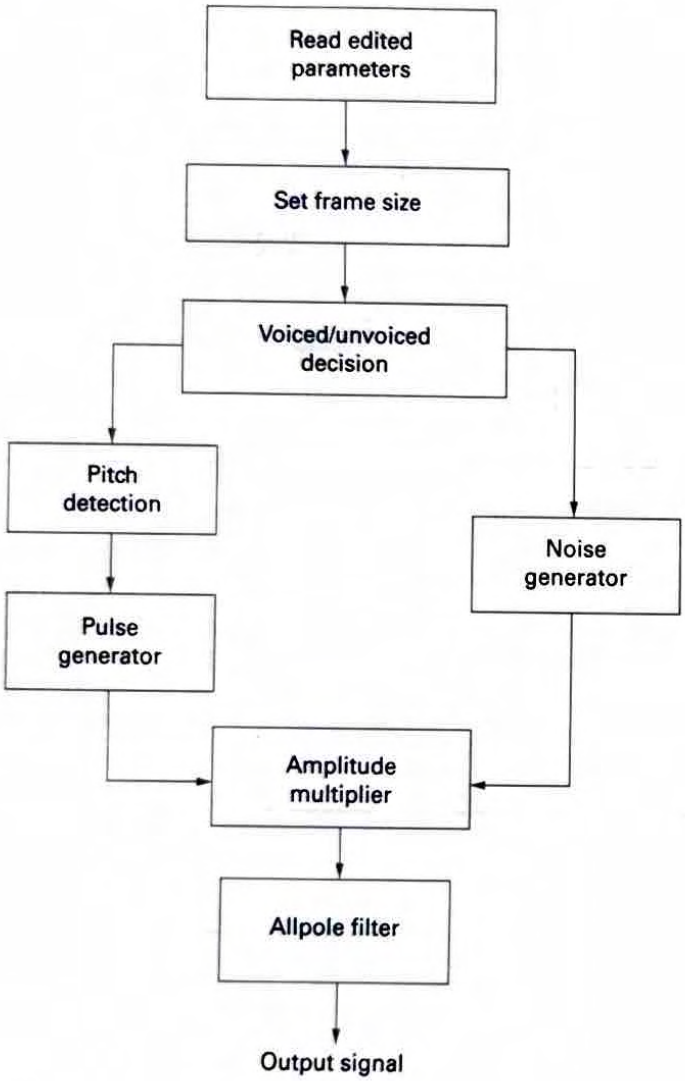


Figure 14.25 Overview of LPC synthesis.

Composers such as Charles Dodge, Paul Lansky, and James A. Moorer have used LPC to achieve all these effects. Examples include Dodge's *Speech Songs* (1973), Lansky's *Six Fantasies on a Poem by Thomas Campion* (1979), and *The Man in the Mangrove Counts to Sleep* (2019) by James A. Moorer.

Cross-Synthesis: A Musical Extension of LPC

LPC can implement a form of cross-synthesis (Mathews, Miller, and David 1961; Petersen 1975; Moorer 1979c; Arfib et al. 2011a). Cross-synthesis refers to techniques that start from an analysis of two sounds and use the characteristics of one sound to modify the characteristics of another sound,

often involving a spectrum transformation. LPC cross-synthesis takes the excitation from one source sound (pitch and event timing) to drive the time-varying spectral envelope derived from another source. For example, one can replace the simple pulse train (used for voiced speech) by a complex waveform such as the sound of an orchestra. Because the time-varying spectrum envelope of the original speech remains the same, the resulting effect is that of a "talking orchestra." Figure 14.26 is essentially the same as the vocoder in figure 14.19, except that the simple excitation function normally used in a vocoder is replaced by a wideband musical source (source B) and the internal method of analysis/resynthesis uses the LPC method.

When the desired effect is to make source B "talk," the intelligibility of the speech can be enhanced by using wide-bandwidth sources such as a full orchestra and chorus—as opposed to a narrow-bandwidth source such as a solo violin. If necessary, the excitation function can also be *whitened* to bring all spectral components up to a uniform level (Moorer 1979c; Arfib et al. 2011a).

Another extension of LPC synthesis extrapolates the filter response of a single instrument into a family of like instruments. For example, starting with an analysis of a violin, one can clone a viola, cello, and double bass to make up a string quartet (Lansky and Steiglitz 1981; Moorer 1981b, 1983a). These filter transformations can, in theory, be extended to emulate any instrument's resonances. In Lansky's music, this method called *warped linear prediction* was used to synthesize electronic versions of strings, saxophones, and harmonicas (e.g., *Guy's Harp* [1983]).

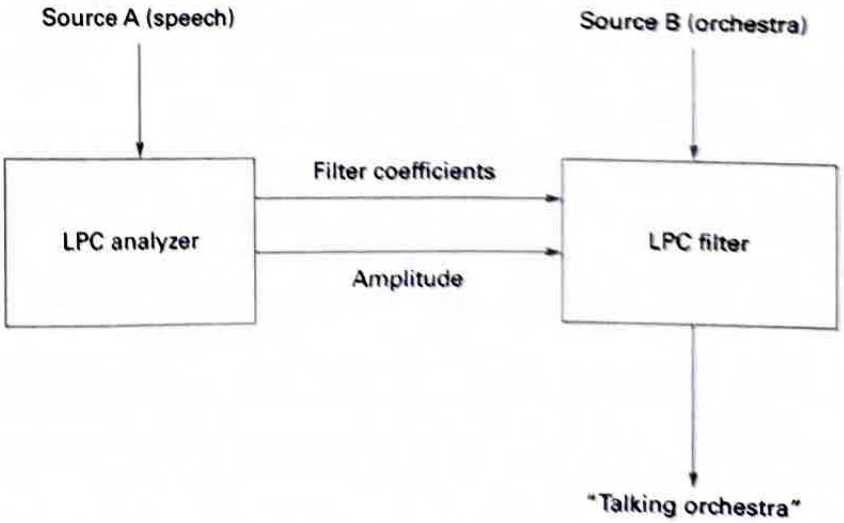


Figure 14.26 LPC cross-synthesis takes the spectral envelope from one sound and maps it onto another sound.

WaveGAN

As a final example of subtractive synthesis we mention experimental research using machine learning techniques, specifically *generative adversarial networks* (GANs). WaveGAN is a project for *adversarial audio synthesis* (Donahue, MacAuley, and Puckette 2019). This involves a neural network that filters a random input (e.g., white Gaussian noise) into a waveform that fools a *discriminator* into deciding that it is real sound, e.g., a fragment of speech. A discriminator in a GAN is simply a classifier. It tries to distinguish real data from data created by the generator using random input.

The WaveGAN algorithm learns to synthesize waveform audio by analyzing many examples of real audio. Its filters learn to reject frequencies corresponding to noise created by a generative procedure. WaveGAN is capable of learning to generate up to four seconds of audio at a 16 kHz sampling rate. It has been tested on words of speech, bird calls, drum hits, and fragments of piano music. (The code in Python and TensorFlow is available at https://github.com/chrisdonahue/wavegan.)