Theory and Techniques of Electronic Music

Miller Puckette
University of California, San Diego

DRAFT

Copyright ©2003 Miller Puckette

August 9, 2003

Contents

Introduction **vii**

1 Acoustics of digital audio signals **1**

1.1 Measures of Amplitude . . . 2
1.2 Amplitude of Combined Signals . . . 3
1.3 Units of Amplitude . . . 4
1.4 Controlling Amplitude . . . 5
1.5 Synthesizing a Sinusoid . . . 6
1.6 Superposing Sinusoids . . . 9
1.7 Frequency . . . 10
1.8 Periodic Signals . . . 11
1.9 About the Software Examples . . . 13
1.9.1 Quick Introduction to Pd . . . 13
1.9.2 How to find and run the examples . . . 15
1.10 Examples . . . 15
1.10.1 constant amplitude scaler . . . 15
1.10.2 amplitude control in decibels . . . 17
1.10.3 smoothed amplitude control with an envelope generator . 19
1.10.4 major triad . . . 20
1.10.5 conversion between frequency and pitch . . . 20

2 Wavetables and samplers **23**

2.1 The Wavetable Oscillator . . . 25
2.2 Sampling . . . 29
2.3 Enveloping samplers . . . 31
2.4 Timbre stretching . . . 35
2.5 Interpolation . . . 39
2.6 Examples . . . 43
2.6.1 wavetable oscillator . . . 43
2.6.2 wavetable lookup in general . . . 44
2.6.3 using a wavetable as a sampler . . . 46
2.6.4 looping samplers . . . 48
2.6.5 overlapping sample looper . . . 50
2.6.6 automatic read point precession . . . 52

3 Audio and control computations 55
3.1 The sampling theorem 55
3.2 Control 57
3.3 Control streams 59
3.4 Converting from audio signals to numeric control streams 63
3.5 Control streams in block diagrams 64
3.6 Event detection 65
3.7 Control computation using audio signals directly 67
3.8 Operations on control streams 70
3.9 Control operations in Pd 71
3.10 Examples 74
3.10.1 Sampling and foldover 74
3.10.2 Converting controls to signals 74
3.10.3 Non-looping sample player 76
3.10.4 Signals to controls 78
3.10.5 Analog-style sequencer 78
3.10.6 MIDI-style synthesizer 79

4 Automation and voice management 83
4.1 Envelope Generators 83
4.2 Linear and Curved Amplitude Shapes 86
4.3 Continuous and discontinuous control changes 88
4.3.1 Muting 89
4.3.2 Switch-and-ramp 90
4.4 Polyphony 92
4.5 Voice allocation 92
4.6 Voice tags 93
4.7 Encapsulation in Pd 96
4.8 Examples 97
4.8.1 ADSR envelope generator 97
4.8.2 Transfer functions for amplitude control 100
4.8.3 Additive synthesis: Risset's bell 101
4.8.4 Additive synthesis: spectral envelope control 104
4.8.5 Polyphonic synthesis: sampler 107

5 Modulation 113
5.1 Taxonomy of spectra 113
5.2 Multiplying audio signals 116
5.3 Waveshaping 120
5.4 Frequency Modulation 126
5.5 Examples 128
5.5.1 Ring modulation and spectra 128
5.5.2 Octave divider and formant adder 129
5.5.3 Waveshaping and difference tones 132
5.5.4 Waveshaping using Chebychev polynomials 133
5.5.5 Waveshaping using an exponential function 134

5.5.6 Sinusoidal waveshaping: evenness and oddness 135
5.5.7 Phase modulation (also known as FM 135

Introduction

This book is about using electronic techniques to record, synthesize, process, and analyze musical sounds, a practice which came into its modern form in the years 1948-1952, but whose technological means and artistic uses have undergone several revolutions since then. Nowadays most electronic music is made using computers, and this book will focus exclusively on what used to be called "computer music", but which should really now be called "electronic music using a computer".

Most of the available computer music tools have antecedents in earlier generations of equipment. The computer, however, is relatively cheap and the results of using one are much easier to document and re-create than those of earlier generations of equipment. In these respects at least, the computer makes the ideal electronic music instrument—until someone invents something even cheaper and more flexible than a computer.

The techniques and practices of electronic music can be studied (at least in theory) without making explicit reference to the current state of technology. Still, it's important to provide working examples of them. So each chapter starts with theory (without any reference to implementation) and ends with a series of examples realized in a currently available software package.

The ideal reader of this book is anyone who knows and likes electronic music of any genre, has plenty of facility with computers in general, and who wants to learn how to make electronic music from the ground up, starting with the humble oscillator and continuing through sampling, FM, filtering, waveshaping, delays, and so on. This will take plenty of time.

This book doesn't concern itself with the easier route of downloading precooked software to try out these techniques; instead, the emphasis is on learning how to use a general-purpose computer music environment to realize them yourself. Of the several such packages are available, we'll use Pd, but that shouldn't stop you from using these same techniques in some other environment such as Csound or Max/MSP. To facilitate this, each chapter is divided into a software-independent discussion of theory, followed by actual examples in Pd, which you can transpose into your own favorite package.

To read this book you must also understand mathematics through intermediate algebra and trigonometry, which most students should have mastered by age 17 or so. A quick glance at the first few pages of chapter one should show you if you're ready to take it on. Many adults in the U.S. and elsewhere may

have forgotten this materiel and will want to get their Algebra 2 textbooks out as a reference. A refresher by F. Richard Moore appears in [Str85, pp. 1-68].

You don't need much background in music as it is taught in the West; in particular, Western written music notation is avoided except where it is absolutely necessary. Some elementary bits of Western music theory are used, such as the tempered scale, the A-B-C system of naming pitches, and terms like "note" and "chord". Also you should be familiar with the fundamental terminology of musical acoustics such as sinusoids, amplitude, frequency, and the overtone series.

Each chapter starts with a theoretical discussion of some family of techniques or theoretical issues, followed by a a series of examples realized in Pd to illustrate them. The examples are included in the Pd distribution, so you can run them and/or edit them into your own spinoffs. In addition, all the figures were created using Pd patches, which appear in an electronic supplement. These aren't carefully documented but in principle could be used as an example of Pd's drawing capabilities for anyone interested in learning more about that aspect of things.

Chapter 1

Acoustics of digital audio signals

Digital audio processing—the analysis and/or synthesis of digital sound–is done by processing *digital audio signals*. These are sequences of numbers,

$$..., x[n-1], x[n], x[n+1], ...$$

where the index n, called the *sample number*, may range over some or all the integers. A single number in the sequence is called a *sample*. (To prevent confusion we'll avoid the widespread, conflicting use of the word "sample" to mean "recorded sound".) Here, for example, is the $r$eal sinusoid:

REAL SINUSOID

$$x[n] = a\cos(\omega n + \phi),$$

where a is the *amplitude*, ω the *angular frequency*, and ϕ the initial *phase*. At sample number n, the phase is equal to $\phi + \omega n$.

We call this sinusoid *real* to distinguish it from the *complex sinusoid* (chapter **??**), but where there's no chance of confusion we will simply say "sinusoid" to speak of the real-valued one.

Figure 1.1 shows a sinusoid graphically. The reason sinusoidal signals play such a key role in audio processing is that, if you shift one of them left or right by any number of samples, you get another one. So it is easy to calculate the effect of all sorts of operations on them. Our ears use this same magic property to help us parse incoming sounds, which is why sinusoidal signals, and combinations of them, can be used for a variety of musical effects.

Digital audio signals do not have any intrinsic relationship with time, but to listen to them we must choose a *sample rate*, usually given the variable name R, which is the number of samples that fit into a second. Time is related to sample

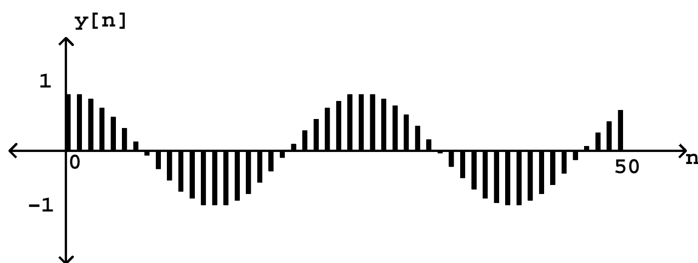


Figure 1.1: A digital audio signal, showing its discrete-time nature. This one is a REAL SINUSOID, fifty points long, with amplitude 1, angular frequency 0.24, and initial phase zero.

number by $Rt = n$, or $t = n/R$. A sinusoidal signal with angular frequency ω has a real-time frequency equal to

$$f = \frac{\omega R}{2\pi},$$

in cycles per second, because a cycle is 2π radians and a second is R samples.

A real-world audio signal's amplitude might be expressed as a time-varying voltage or air pressure, but the samples of a digital audio signal are unitless real (or in some later chapters, complex) numbers. We'll casually assume here that there is ample numerical accuracy that round-off errors are negligible, and that the numerical format is unlimited in range, so that samples may take any value we wish. However, most digital audio hardware works only over a fixed range of input and output values. We'll assume that this range is from -1 to 1. Modern digital audio processing software usually uses a floating-point representation for signals, so that the may assume whatever units are convenient for any given task, as long as the final audio output is within the hardware's range.

1.1 Measures of Amplitude

Strictly speaking, all the samples in a digital audio signal are themselves amplitudes, and we also spoke of the amplitude a of the SINUSOID above. In dealing with general digital audio signals, it is useful to have measures of amplitude for them. Amplitude and other measures are best thought of as applying to a *window*, a fixed range of samples of the signal. For instance, the window starting at sample M of length N of an audio signal $x[n]$ consists of the samples,

$$x[M], x[M+1], \ldots, x[M+N-1].$$

The two most frequently used measures of amplitude are the *peak amplitude*, which is simply the greatest sample (in absolute value) over the window:

$$A_{\text{peak}}\{x[n]\} = \max |x[n]|, \qquad n = M, \ldots, M + N - 1,$$

and the *root mean square* (RMS) amplitude:

$$A_{\text{RMS}}\{x[n]\} = \sqrt{P\{x[n]\}},$$

where Px[n] is the mean *power*, defined as:

$$P\{x[n]\} = \frac{1}{N}\left(|x[M]|^2 + \cdots + |x[M + N - 1]|^2\right).$$

In this last formula, the absolute value signs aren't necessary as long as we're working on real signals, but they are significant if the signals are complex-valued. The peak and RMS amplitude of any signal is at least zero, and is only exactly zero if the signal itself is zero.

The RMS amplitude of a signal may equal the peak amplitude but never exceeds it; and it may be as little as $1/\sqrt{N}$ times the peak amplitude, but never less than that.

Under reasonable conditions—if the window contains at least several periods and if the angular frequency is well under one radian per sample—the peak amplitude of the SINUSOID is approximately a and its RMS amplitude about $a/\sqrt{2}$.

1.2 Amplitude of Combined Signals

If a signal $x[n]$ has a peak or RMS amplitude A (in some fixed window), then the scaled signal $k \cdot a[n]$ (where $k \geq 0$) has amplitude kA. The RMS power of the scaled signal changes by a factor of k^2. The situation gets more complicated when two different signals are added together; just knowing the amplitudes of the two does not suffice to know the amplitude of the sum. The two amplitude measures do at least obey triangle inequalities; for any two signals $x[n]$ and $y[n]$,

$$A_{\text{peak}}\{x[n]\} + A_{\text{peak}}\{y[n]\} \geq A_{\text{peak}}\{x[n] + y[n]\},$$

$$A_{\text{RMS}}\{x[n]\} + A_{\text{RMS}}\{y[n]\} \geq A_{\text{RMS}}\{x[n] + y[n]\}.$$

If we fix a window from M to $N + M - 1$ as usual, we can write out the mean power of the sum of two signals:

MEAN POWER OF THE SUM OF TWO SIGNALS

$$P\{x[n] + y[n]\} = P\{x[n]\} + P\{y[n]\} + 2\text{COR}\{x[n], y[n]\},$$

where we have introduced the *correlation* of two signals:

CORRELATION

$$\mathrm{COR}\{x[n], y[n]\} = x[M]y[M] + \cdots + x[M+N-1]y[M+N-1].$$

The correlation may be positive, zero, or negative. Over a sufficiently large window, the correlation of two sinusoids with different frequencies is negligible. In general, for two uncorrelated signals, the power of the sum is the sum of the powers:

POWER RULE FOR UNCORRELATED SIGNALS

$$P\{x[n] + y[n]\} = P\{x[n]\} + P\{y[n]\}, \quad \text{whenever } \mathrm{COR}\{x[n], y[n]\} = 0.$$

Put in terms of amplitude, this becomes:

$$(A_{\mathrm{RMS}}\{x[n] + y[n]\})^2 = (A_{\mathrm{RMS}}\{x[n]\})^2 + (A_{\mathrm{RMS}}\{y[n]\})^2.$$

This is the familiar Pythagorean relation. So uncorrelated signals can be thought of as vectors at right angles to each other; positively correlated ones as having an acute angle between them, and negatively correlated as having an obtuse angle between them.

For example, if we have two uncorrelated signals both with RMS amplitude a, the sum will have RMS amplitude $\sqrt{2}a$. On the other hand if the two signals happen to be equal—the most correlated possible—the sum will have amplitude $2a$, which is the maximum allowed by the triangle inequality.

1.3 Units of Amplitude

Two amplitudes are often best compared using their ratio rather than their difference. For example, saying that one signal's amplitude is greater than another's by a factor of two is more informative than saying it is greater by 30 millivolts. This is true for any measure of amplitude (RMS or peak, for instance). To facilitate this we often express amplitudes in logarithmic units called *decibels.* If a is an amplitude in any linear scale (such as above) then we can define the decibel (dB) amplitude d as:

$$d = 20 \cdot log_{10}(a/a_0),$$

where a_0 is a reference amplitude. This definition is set up so that, if we increase the signal power by a factor of ten (so that the amplitude increases by a factor of $\sqrt{10}$), the logarithm will increase by 1/2, and so the value in decibels goes up (additively) by ten. An increase in amplitude by a factor of two corresponds to an increase of about 6.02 decibels; doubling power is an increase of 3.01 dB. In dB, therefore, adding two uncorrelated signals of equal amplitude results in one that is about 3 dB higher, whereas doubling a signal increases its amplitude by 6 dB.

Still using a_0 as a reference amplitude, a signal with linear amplitude smaller than a_0 will have a negative amplitude in decibels: $a_0/10$ gives -20 dB, $a_0/100$ gives -40, and so on. A linear amplitude of zero is smaller than that of any value in dB, so we give it a dB value of $-\infty$.

In digital audio a convenient choice of reference, assuming the hardware has a maximum amplitude of one, is

$$a_0 = 10^{-5} = 0.00001,$$

so that the maximum amplitude possible is 100 dB, and 0 dB is likely to be inaudibly quiet at any reasonable listening level. Conveniently enough, the dynamic range of human hearing—the ratio between a damagingly loud sound and an inaudibly quiet one—is about 100 dB.

Amplitude is related in an inexact way to perceived loudness of a sound. In general, two signals with the same peak or RMS amplitude won't necessarily have the same loudness at all. But amplifying a signal by 3 dB, say, will fairly reliably make it sound about one "step" louder. Much has been made of the supposedly logarithmic responses of our ears (and other senses), which may indeed partially explain why decibels are such a popular scale of amplitude.

Amplitude is also related in an inexact way to musical *dynamic*. Dynamic is better thought of as a measure of effort than of loudness or power, and the scale moves, roughly, over nine values: rest, ppp, pp, p, mp, mf, f, ff, fff. These correlate in an even looser way with the amplitude of a signal than does loudness [RMW02, pp. 110-111].

1.4 Controlling Amplitude

Conceptually at least, the simplest strategy for synthesizing sounds is by combining SINUSOIDS, which can be generated by evaluating the formula from section 1.1, sample by sample. The real sinusoid has a constant nominal amplitude a, and we would like to be able to vary that in time.

In general, to multiply the amplitude of a signal $x[n]$ by a constant $y \geq 0$, you can just multiply each sample by y, giving a new signal $y \cdot x[n]$. Any measurement of the RMS or peak amplitude of $x[n]$ will be greater or less by the factor y. More generally, you can change the amplitude by an amount $y[n]$ which varies sample by sample. If $y[n]$ is nonnegative and if it varies slowly enough, the amplitude of the product $y[n] \cdot x[n]$ (in a fixed window from M to $M + N - 1$) will be related to that of $x[n]$ by the value of $y[n]$ in the window (which we assume doesn't change much over the N samples in the window).

In the more general case where both $x[n]$ and $y[n]$ are allowed to take negative and positive values and/or to change quickly, the effect of multiplying them can't be described as simply changing the amplitude of one of them; this is considered later in chapter 5.

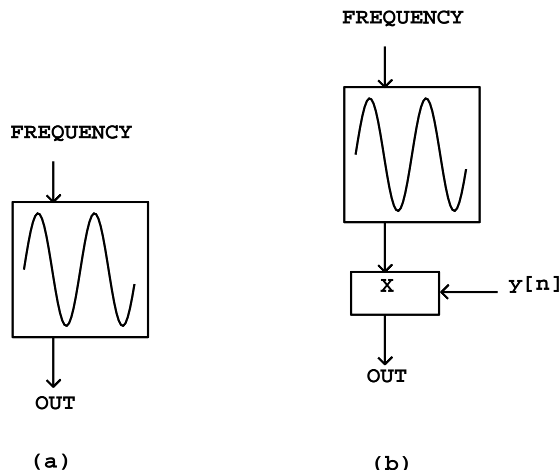


Figure 1.2: Block diagrams for (a) a sinusoidal oscillator; (b) controlling the amplitude using a multiplier and an amplitude signal $y[n]$.

1.5 Synthesizing a Sinusoid

In most widely used audio synthesis and processing packages (Csound, Max/MSP, and Pd, for instance), the audio operations are specified as networks of *unit generators* which pass audio signals among themselves. The user of the software package specifies the network, sometimes called a *patch*, which essentially corresponds to the synthesis algorithm to be used, and then worries about how to control the various unit generators in time. In this section, we'll use abstract block diagrams to describe patches, but in the "examples" section later, we'll have to choose a real implementation environment and show some of the software-dependent details.

To show how to produce a sinusoid with time-varying amplitude we'll need to introduce two unit generators. First we need a pure, SINUSOID which is produced using an *oscillator*. Figure 1.2(a) shows the icon we use to show a sinusoidal oscillator. The input is a frequency (in cycles per second), and the output is a SINUSOID of peak amplitude one.

Figure 1.2(b) shows how to multiply the output of a sinusoidal oscillator by an appropriate amplitude scaler $y[n]$ to control its amplitude. Since the oscillator's peak amplitude is 1, the peak amplitude of the product is about $y[n]$, assuming $y[n]$ changes slowly enough and doesn't become negative in value.

Figure 1.3 shows how the SINUSOID of Figure 1.1 is affected by amplitude change by two different controlling signals $y[n]$. In the first case the controlling signal shown in (a) has a discontinuity, and so therefore does the resulting amplitude-controlled sinusoid shown in (b). The second case (c, d) shows a more gently-varying possibility for $y[n]$ and the result. Intuition suggests that

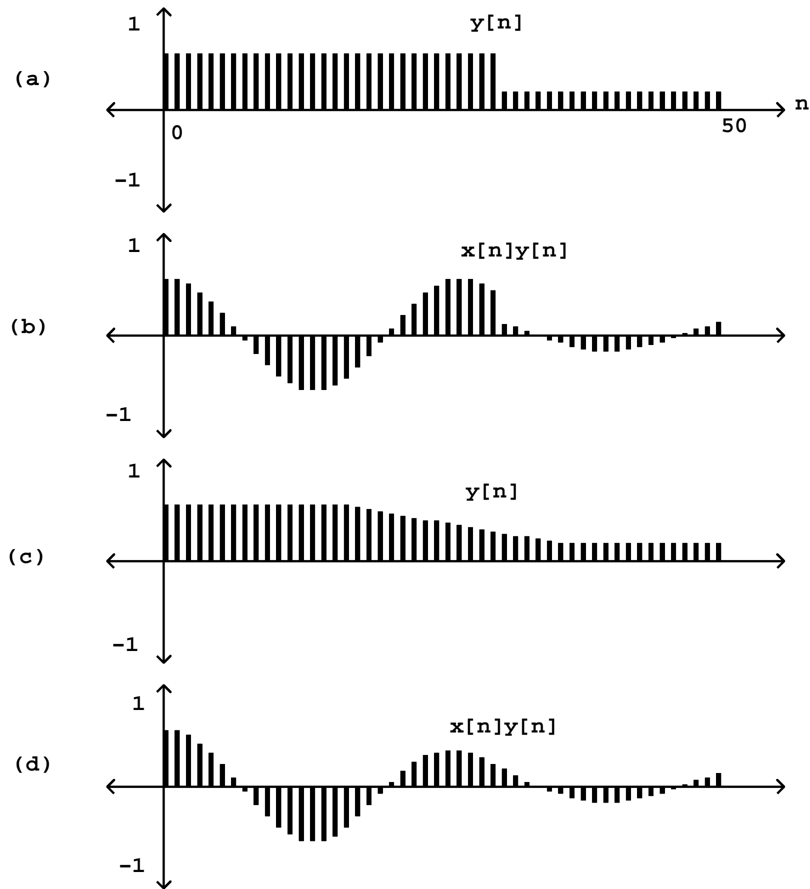


Figure 1.3: Two amplitude functions (a, c), and, in (b) and (d), the result of multiplying them by the pure sinusoid of 1.1.

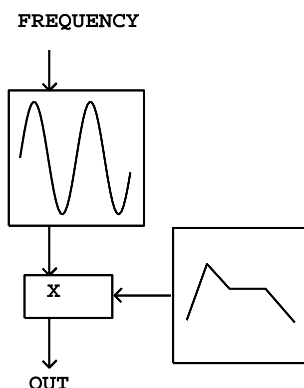


Figure 1.4: Using an envelope generator to control amplitude.

the result shown in (b) won't sound like an amplitude-varying sinusoid, but instead by a sinusoid interrupted by a fairly loud "pop" after which the sinusoid reappears more quietly. In general, for reasons that can't be explained in this chapter, amplitude control signals $y[n]$ which ramp smoothly from one value to another are less likely to give rise to parasitic results (such as the "pop" here) than are abruptly changing ones. Two general rules may be suggested here. First, pure sinusoids are the class of signals most sensitive to the parasitic effects of quick amplitude change; and second, depending on the signal whose amplitude you are changing, the amplitude control will need between 0 and 30 milliseconds of "ramp" time—zero for the most forgiving signals (such as white noise), and 30 for the least (such as a sinusoid). All this also depends (in complicated ways) on listening levels and the acoustic context.

Suitable amplitude control functions $y[n]$ may be obtained using an *envelope generator*. Figure 1.4 shows a network in which an envelope generator is used to control the amplitude of an oscillator. Envelope generators vary widely in functionality from one design to another, but our purposes will be adequately met by the simplest kind, which generates line segments, of the kind shown in fig. 1.2(b). If a line segment is specified to ramp between two output values a and b over N samples starting at sample number M, the output is:

$$y[n] = a + (b - a)\frac{n - M}{N}, \quad M \le n < M + N - 1.$$

The output may have any number of segments such as this, laid end to end, over the entire range of sample numbers n; flat, horizontal segments can be made by setting $a = b$.

In addition to changing amplitudes of sounds, amplitude control is often used, expecially in real-time applications, simply to turn sounds on and off: to

turn one off, ramp the amplitude smoothly to zero. Most software synthesis packages also provide ways to actually stop modules from computing samples at all, but here we'll use amplitude control instead.

Envelope generators are described in more detail in section 4.1.

1.6 Superposing Sinusoids

If two sinusoids have sufficiently different frequencies, they don't interact acoustically; the power of the sum is the sum of the powers, and they are likely to be heard as separate sounds. Something more complicated happens when two sinusoids of closely neighboring frequencies are combined, and something yet again when the two frequencies happen to be equal. Here we'll treat this last case.

We have seen that adding two sinusoids with the same frequency and the same phase (so that the two signals are proportional) gives a resultant sinusoid with the sum of the two amplitudes. If the two have different phases, though, we have to do some algebra.

If we fix a frequency ω, there are two useful representations of a general (real) sinusoid at frequency ω; the first is the original SINUSOID formula, which is expressed in magnitude-phase form:

$$x[n] = a \cdot \cos(\omega n + \phi)$$

and the second is the sinusoid in rectangular form:

$$x[n] = c \cdot \cos(\omega n) + s \cdot \sin(\omega n) .$$

Solving for c and s in terms of a and ϕ gives:

$$c = a \cdot \cos(\phi) ,$$

$$s = a \cdot \sin(\phi) ,$$

and *vice versa* we get:

$$a = \sqrt{c^2 + s^2},$$

$$\phi = \arctan \frac{s}{c}.$$

We can use this to find the amplitude and phase of a sum of two sinusoids at the same frequency ω but with possibly different amplitudes and phases, say, a_1, a_2, ϕ_1, and ϕ_2. We just write the sum expicitly, convert to rectangular form, add the two, and finally convert back to magnitude-phase form:

$$a_1 \cos(\omega n + \phi_1) + a_2 \cos(\omega n + \phi_1)$$

$$= a_1 \cos(\omega n) \cos(\phi_1) - a_1 \sin(\omega n) \sin(\phi_1)$$

$$+ a_2 \cos(\omega n) \cos(\phi_2) - a_2 \sin(\omega n) \sin(\phi_2)$$

$$= (a_1 \cos(\phi_1) + a_2 \cos(\phi_2)) \cos(\omega n) - (a_1 \sin(\phi_1) + a_2 \sin(\phi_2)) \sin(\omega n)$$

$$= a_3 \cos(\phi_3) \cos(\omega n) - a_3 \sin(\phi_3) \sin(\omega n)$$

$$= a_3 \cos(\omega n + \phi_3),$$

where we have chosen a_3 and phi_3 so that:

$$a_3 \cos \phi_3 = a_1 \cos \phi_1 + a_2 \cos \phi_2,$$

$$a_3 \sin \phi_3 = a_1 \sin \phi_1 + a_2 \sin \phi_2.$$

Solving for a_3 and ϕ_3 gives,

$$a_3 = \sqrt{{a_1}^2 + {a_2}^2 + 2a_1 a_1 \cos(\phi_1 - \phi_2)},$$

$$\phi_3 = \arctan\left(\frac{a_1 \sin \phi_1 + a_2 \sin \phi_2}{a_1 \cos \phi_1 + a_2 \cos \phi_2}\right).$$

In general, the amplitude of the sum can range from the difference of the two amplitudes to their sum, depending on the phase difference. As a special case, if the two sinusoids have the same amplitude $a = a_1 = a_2$, the amplitude of the sum turns out to be:

$$a_3 = a \cos\left(\frac{\phi_1 - \phi_2}{2}\right).$$

By comparing the more general formula for a_3 above with the equation for the MEAN POWER OF THE SUM OF TWO SIGNALS, we learn that the correlation of two sinusoids of the same frequency is given by:

$$\mathrm{COR}\left\{a_1 \cos(\omega n + \phi_1), a_2 \cos(\omega n + \phi_1)\right\} = 2a_1 a_1 \cos(\phi_1 - \phi_2).$$

1.7 Frequency

Frequencies, like amplitudes, are often described on a logarithmic scale, in order to emphasize proportions between frequencies, which usually provide a better description of the relationship between frequencies than do differences between frequencies. The frequency ratio between two musical tones determines the musical interval between them.

The Western musical scale divides the *octave* (the musical interval associated with a ratio of 2:1) into twelve equal sub-intervals, each of which therefore corresponds to a ratio of $2^{1/12}$. For historical reasons this sub-interval is called a *half step*. A convenient logarithmic scale for pitch is simply to count the number of half-steps from a reference pitch—allowing fractions to permit us to specify pitches which don't fall on a note of the Western scale. The most commonly used logarithmic pitch scale is MIDI, in which the pitch 69 is assigned to the frequency 440, the A above middle C. To convert between MIDI pitch and frequency in cycles per second, apply the formulas:

PITCH/FREQUENCY CONVERSION

$$m = 69 + 12\log_2(f/440),$$

$$f = 440 \cdot 2^{(m-69)/12}.$$

Middle C, corresponding to MIDI pitch 60, comes to 261.626 cycles per second.

Although MIDI itself (a hardware protocol which has unfortunately insinuated itself into a great deal of software design) allows only integer pitches between 0 and 127, the underlying scale is well defined for any number, even negative ones; for example a "pitch" of -4 is a good rate of vibrato. The pitch scale cannot, however, describe frequencies less than or equal to zero. (For a clear description of MIDI, its capabilities and limitations, see [Bal03, ch.6-8]).

A half step comes to a ratio of about 1.059 to 1, or about a six percent increase in frequency. Half steps are further divided into *cents*, each cent being one hundredth of a half step. As a rule of thumb, it takes about three cents to make a clearly audible change in pitch—at middle C this comes to a difference of about 1/2 cycle per second.

1.8 Periodic Signals

A signal $x[n]$ is said to repeat at a period τ if

$$x[n + \tau] = x[n]$$

for all n. Such a signal would also repeat at periods 2τ and so on; the smallest tau if any at which a signal repeats is called the signal's *period.* In discussing periods of digital audio signals, we quickly run into the difficulty of describing signals whose "period" isn't an integer, so that the equation above doesn't make sense. Throughout this section, we'll avoid this difficulty by supposing that the signal $x[n]$ may somehow be interpolated between the samples so that it's well defined whether n is an integer or not.

The SINUSOID has a period (in samples) of $2\pi/\omega$ where ω is the angular frequency. More generally, any sum of sinusoids with frequencies $2\pi k/\omega$, for integers k, will have this period. This is the FOURIER SERIES:

FOURIER SERIES

$$x[n] = a_0 + a_1 \cos(\omega n + \phi_1) + a_2 \cos(2\omega n + \phi_2) + \cdots + a_p \cos(p\omega n + \phi_p)$$

Moreover, if we define the notion of interpolation carefully enough, we can represent any periodic signal as such a sum. This is the discrete-time variant of Fourier analysis which will reappear in many guises later.

The angular frequencies of the sinusoids above, i.e., integer multiples of ω, are called *harmonics* of ω, which in turn is called the *fundamental.* In terms of pitch, the harmonics $\omega, 2\omega, \ldots$ are at intervals of 0, 1200, 1902, 2400, 2786, 3102, 3369, 3600, ..., cents above the fundamental; this sequence of pitches is

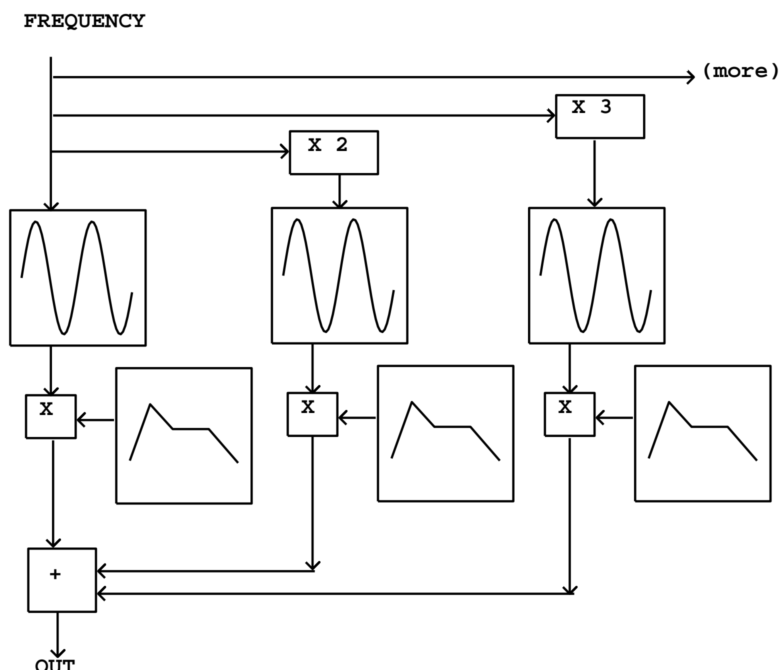


Figure 1.5: Using many oscillators to synthesize a waveform with desired harmonic amplitudes.

sometimes called the *harmonic series.* The first six of these are all oddly close to multiples of 100; in other words, the first six harmonics of a pitch in the Western scale land close to (but not always on) other pitches of the same scale; the third (and sixth) miss only by 2 cents and the fifth misses by 14.

Put another way, the frequency ratio 3:2 is almost exactly seven half-tones, 4:3 is just as near to five half tones, and the ratios 5:4 and 6:5 are fairly close to intervals of four and three half-tones, respectively. These four intervals are called the fifth, the fourth, and the major and minor thirds—again for historical reasons which don't concern us here.

Leaving questions of phase aside, we can use a bank of sinusoidal oscillators to synthesize periodic tones, or even to morph smoothly through a succession of periodic tones, by specifying the fundamental frequency and the (possibly time-varying) amplitudes of the partials. Figure 1.5 shows a block diagram for doing this. This is a special case of *additive synthesis*; more generally the term can be applied to networks in which the frequencies of the oscillators are independently controllable. The early days of computer music were full of the sound of additive synthesis.

1.9 About the Software Examples

The examples here have all been realized using Pure Data (Pd), and to use and understand them you will have to learn at least something about Pd itself. Pd is an environment for quickly assembling computer music applications, primarily intended for live music performances involving computers. Pd's utility extends to graphical and other media, although here we'll focus on Pd's audio capabilities.

Several other patchable audio DSP environments exist besides Pd. The most widely used one is certainly Barry Vercoe's Csound, which differs from Pd in being text-based–not GUI based—which is an advantage in some respects and a disadvantage in others. Csound is better adapted than Pd for batch processing and it handles polyphony much better than Pd does. On the other hand, Pd has a better developed real-time control structure than Csound. More on Csound can be found in ([Bou00]).

Another alternative in wide use is James McCartney's SuperCollider, which is also more text oriented than Pd, but like Pd is explicitly designed for real-time use. SuperCollider has powerful linguistic constructs which make it more useful than Csound as a programming language. Another major advantage is that SuperCollider's audio processing primitives are heavily optimized for the processor family it runs on (MIPS), making it perhaps twice as efficient as Pd or Csound. At this writing SuperCollider has the disadvantage that it is available only for Macintosh computers (whereas Pd and Csound both run on a variety of operating systems.)

Finally, Pd has a widely-used relative, Cycling74's commercial program Max/MSP (the others named here are all open source). Both beginners and system managers running multi-user, multi-purpose computer labs will find Max/MSP better supported and documented than Pd. It's possible to take knowledge of Pd and use it on Max/MSP and vice versa, and even to port patches from one to the other, but they aren't truly compatible.

1.9.1 Quick Introduction to Pd

Pd documents are called "patches." They correspond roughly to the boxes in the abstract block diagrams shown earlier in this chapter, but in detail they are quite different, reflecting the fact that Pd is an implementation environment and not a specification language.

A Pd patch, such as the one shown in Figure 1.6, consists of a collection of *boxes* connected in a network called a *patch*. The border of a box tells you how its text is interpreted and how the box functions. In part (a) of the figure we see three types of boxes. From top to bottom they are:

- a *message box*. Message boxes, with a flag-shaped border, interpret the text as a message to send whenever the box is activated (by an incoming message or with the mouse.) The message in this case consists simply of the number "34".

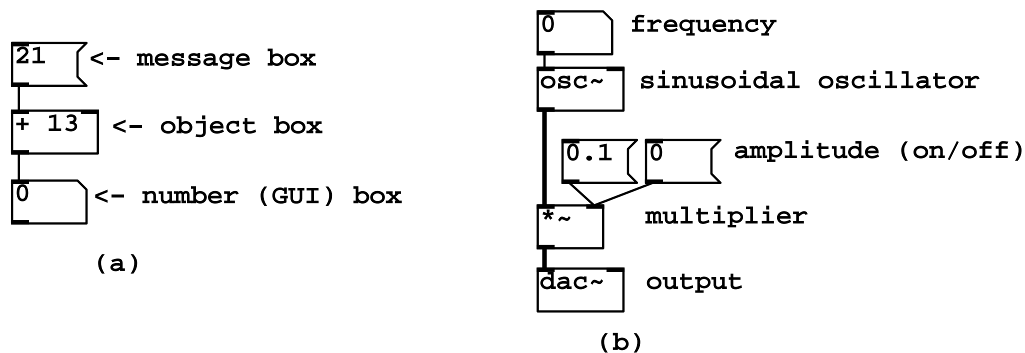


Figure 1.6: (a) three types of boxes in Pd (message, object, and GUI); (b) a simple patch to output a sinusoid.

- an *object box*. Object boxes have a rectangular border; they use the text to create objects when you load a patch. Object boxes may represent hundreds of different classes of objects—including oscillators, envelope generators, and other signal processing modules to be introduced later—depending on the text inside. In this example, the box contains an adder. In most Pd patches, the majority of boxes are of type "object". The first word typed into an object box specifies its *class*, which in this case is just "+". Any additional (blank-space-separated) words appearing in the box are called *creation arguments*, which specify the initial state of the object when it is created.

- a *number box*. number boxes are a particular case of a *GUI box*, which also include push buttons, toggle switches, sliders, and more; these will come up later in the examples. The number box has a punched-card-shaped border, with a nick out of its top right corner. Whereas the appearance of an object or message box is static when a patch is running, a number box's contents (the text) changes to reflect the current value held by the box. You can also use a number box as a control by clicking and dragging up and down, or by typing values in it.

In fig. 1.6(a) the message box, when clicked, sends the message "21" to an object box which adds 13 to it. The lines connecting the boxes carry data from one box to the next; outputs of boxes are on the bottom and inputs on top.

Figure 1.6(b) shows a Pd patch which makes a sinusoid with controllable frequency and amplitude. The connecting patch lines are of two types here; the thin ones are for carrying sporadic *messages*, and the thicker ones (connecting the oscillator, the multiplier, and the output "dac ") carry digital audio signals. Since Pd is a real-time program, the audio signals flow in a continuous stream. On the other hand, the sporadic messages appear at specific but possibly un-

predictable instants in time.

Whether a connection carries messages or signals is a function of the box the connection comes from; so, for instance, “+” outputs messages, but “*˜” outputs a signal. The inputs of objects may or may not accept signals (but they always accept messages, even if only to convert them to signals). As a naming convention, object boxes which input or output signals are all named with a trailing tilde (“˜”) as in “*˜” and “osc˜”.

1.9.2 How to find and run the examples

To run the patches, you must first download, install, and run Pd. Instructions for doing this appear in Pd’s on-line HTML documentation, which you can find at http:/crca/ucsd/edu/˜msp/software.htm.

This book should appear at: http:/crca/ucsd/edu/˜msp/techniques.htm, possibly in several revisions. Choose the revision that corresponds to the text you’re reading (go to the introduction to find the revision number) and download the archive containing the associated revision of the examples (you may also download an archive of the HTML version for easier access on your machine.) The examples should all stay in a single directory, since some of them depend on other files in that directory and might not find them if you move things around.

If you do want to copy one of the examples to another directory so that you can build on it (which you’re welcome to do), you should either include the examples directory in Pd’s search path (see the Pd documentation) or else figure out what other files are needed and copy them too. A good way to find this out is just to run Pd on the relocated file and see what Pd complains it can’t find.

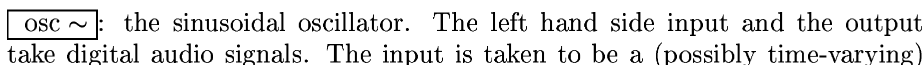
There should be dozens of files in the “examples” folder, including the examples themselves and the support files. The filenames of the examples all begin with a letter (A for chapter 1, B for 2, etc.) and a number, as in “A01.sinewave.pd”.

The example patches are also distributed with Pd, but beware that you may have a different version of the examples which might not correspond with the text you’re reading.

1.10 Examples

1.10.1 constant amplitude scaler

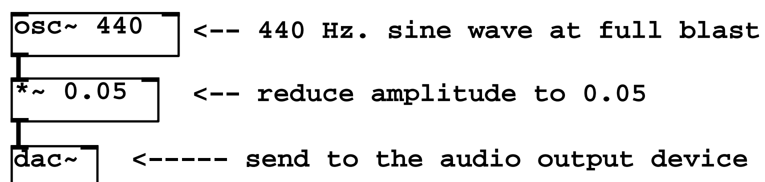
Patch A01.sinewave.pd, shown in figure 1.7, contains essentially the simplest possible noise-making patch, with only three object boxes. (There are also comments, and two message boxes to turn Pd’s “DSP” (audio) processing on and off.) The three object boxes are:



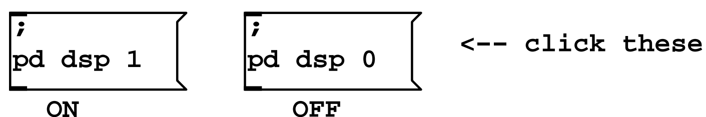
osc ~: the sinusoidal oscillator. The left hand side input and the output take digital audio signals. The input is taken to be a (possibly time-varying)

MAKING A SINE WAVE

Audio computation in Pd is done using "tilde objects" such as the three below. They use continuous audio streams to intercommunicate, as well as communicating with other ("control") Pd objects using messages.



Audio computation can be turned on and off by sending messages to the global "pd" object as follows:



You should see the Pd window change to reflect whether audio is on or off. You can also turn audio on and off using the "audio" menu, but the buttons are provided as a shortcut.

When DSP is on, you should hear a tone whose pitch is A 440 and whose amplitude is 0.05. If instead you are greeted with silence, you might want to read the HTML documentation on setting up audio.

In general when you start a work session with Pd, you will want to choose "test audio and MIDI" from the help window, which opens a more comprehensive test patch than this one.

Figure 1.7: The contents of the first Pd example patch: A01.sinewave.pd.

frequency in Hz. The output is a SINUSOID at the specified frequency. If nothing is connected to the frequency inlet, the creation argument (440 in this example) is used as the frequency. The output has peak amplitude one. You may set an initial phase by sending messages (not audio signals) to the right inlet. The left (frequency) inlet may also be sent messages to set the frequency, since any inlet that takes an audio signal may be sent messages which are automatically converted to the desired audio signal.

[$* \sim$]: the multiplier. This exists in two forms. If a creation argument is specified (as in this example; it's 0.05), this box multiplies a digital audio signal (in the left inlet) by the number; messages to the right inlet can update the number as well. If no argument is given, this box multiplies two incoming digital audio signals together.

[$dac \sim$]: the audio output device. Depending on your hardware, this might not actually be a Digital/Analog Converter—as the name suggests—but in general, it allows you to send any audio signal to your computer's audio output(s). If there are no creation arguments, the default configuration is to output to channels one and two of the audio hardware; you may specify alternative channel numbers (one or many) using the creation arguments. Pd itself may be configured to be using two or more output channels, or may not have the audio output device open at all; consult the Pd documentation for details.

The two message boxes in example 1 show a peculiarity in the way messages are parsed in message boxes. In the previous example, the message consisted only of the number 21. When clicked, that box sent the message "21" to its outlet and hence to any objects connected to it. In the current example, the text of the message boxes starts with a semicolon. This is a terminator between messages (so the first message is empty), after which the next word is taken as the name of the recipient of the following message. Thus the message here is "dsp 1" (or "dsp 0") and the message is to be sent, not to any connected objects—there aren't any anyway— but rather, to the object named "pd". This particular object is provided invisibly by the Pd program and you can send it various messages to control Pd's global state, in this case turning audio processing on ("1") and off ("0").

Many more details about the control aspects of Pd, such as the above, are explained in a different series of example patches (the "control examples") as part of the Pd release, but they will only be touched on here as necessary to demonstrate the audio signal processing aspects that are the subject of this book.

1.10.2 amplitude control in decibels

Patch A02.amplitude.pdshows how to make a crude amplitude control; the active elements are shown in figure 1.7(a). There is one new object class:

[dbtorms] : Decibels to amplitude conversion. The "RMS" is a misnomer; it should have been named "dbtoamp", since it really converts from decibels to any linear amplitude unit, be it RMS, peak, or other. An input of 100 dB

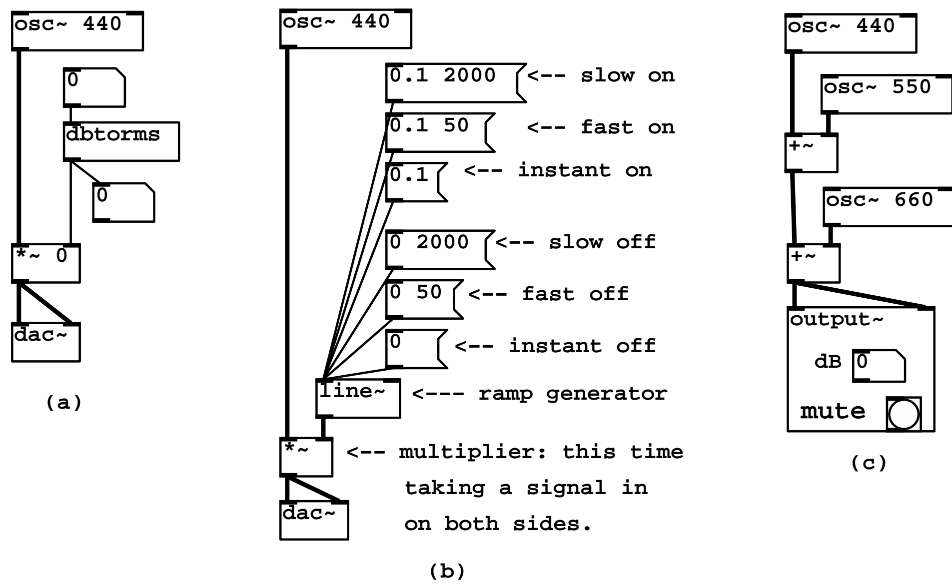


Figure 1.8: The active ingredients to three patches: (a) A02.amplitude.pd; (b) A03.line.pd; (c) A05.output.subpatch.pd.

is normalized to an output of 1. Values greater than 100 are fine (120 will give 10), but values less than or equal to zero will output zero (a zero input would otherwise have output a small positive number.) This is a control object, i.e., the numbers going in and out are messages, not signals. (A corresponding object, dbtorms ~, is the signal correlate. However, as a signal object this is expensive in CPU time and most often we'll find one way or another to avoid using it.)

The two number boxes are connected to the input and output of the `dbtorms` object. The input functions as a control; "mouse" on it (click and drag upward or downward) to change the amplitude. It has been set to range from 0 to 80; this is protection for your speakers and ears, and it's wise to build such guardrails into your own patches.

The other number box shows the output of the `dbtorms` object. It is useless to mouse on this number box, since its outlet is connected nowhere; it is here purely to display its input. Number boxes may be useful as controls, displays, or both, although if you're using it as both there is some extra work to do.

1.10.3 smoothed amplitude control with an envelope generator

As figure 1.3 shows, one way to make smooth amplitude changes in a signal without clicks is to multiply by an envelope generator; one is invoked in figure 1.4. This may be implemented in Pd using the `line~` class:

line ~: envelope generator. The output is a signal which ramps linearly from one value to another over time, as determined by the messages received. The inlets take messages to specify target values (left inlet) and time delays (right inlet). Because of a general rule of Pd messages, a pair of numbers sent to the left inlet suffices to specify a target value and a time together. The time is in milliseconds (taking into account the sample rate), and the target value is unitless, or rather, its units should conform to whatever input it may be connected to.

Patch A03.line.pd demonstrates the use of a `line~` object to control the amplitude of a sinusoid. The active part is shown in figure 1.8(b). The six message boxes are all connected to the `line~` object, and are activated by clicking on them; the top one, for instance, specifies that the `line~` ramp (starting at wherever its output was before receiving the message) to the value 0.1 over two seconds. After the two seconds elapse, unless other messages have arrived in the meantime, the output remains steady at 0.1. Messages may arrive before the two seconds elapse, in which case the `line~` object abandons its old trajectory and takes up a new one.

Two messages to `line~` might arrive at the same time or so close together in time that no DSP computation takes place between the two; in this case, the earlier message has no effect, since `line~` won't have changed its output yet to follow the first message, and its current output, unchanged, is then used as a starting point for the second segment. An exception to this rule is that, if

line~ gets a time value of zero, the output value is immediately set to the new value and further segments will start from the new value; thus, by sending two pairs, the first with a time value of zero and the second with a nonzero time value, one can independently specify the beginning and end values of a segment in line~'s output.

The treatment of line~'s right inlet is unusual among Pd objects in that it forgets old values; thus, a message with a single number such as "0.1" is always equivalent to the pair, "0.1 0". Most Pd objects will keep the previous value for the right inlet, instead of filling in zero.

Patch A04.line2.pd shows the line~ object's output graphically, so that you can see the principles of Figure 1.4 in action.

1.10.4 major triad

Patch A05.output.subpatch.pd, whose active ingredients are shown in Figure 1.8(c), presents three sinusoids with frequencies in the ratio 4:5:6, so that the lower two are separated by a major third, the upper two by a minor third, and the top and bottom by a fifth. The lowest frequency is 440, equal to A above middle C, or MIDI 69. The others are approximately four and seven half-steps higher, respectively. The three have equal amplitudes.

The amplitude control in this example is taken care of by a new object called output~. This isn't a built-in object of Pd, but is itself a Pd patch which lives in a file, output.pd. You can see the internals of output by right-clicking on the box and selecting "open". You get two controls, one for amplitude in dB (100 meaning "unit gain"), and a "mute" button. Pd's audio processing is turned on automatically when you set the output level—this might not be the right behavior in general, but it's appropriate for these example patches. The mechanism for embedding one Pd patch as an object box inside another is discussed in section 4.7.

1.10.5 conversion between frequency and pitch

Patch A06.frequency.pd (figure 1.9) shows Pd's object for converting pitch to frequency units (mtof, meaning "MIDI to frequency") and its inverse ftom. We also introduce two other object classes, send and receive:

[mtof], [ftom]: Converts MIDI pitch to frequency units according to the PITCH/FREQUENCY CONVERSION formulas. Inputs and outputs are messages (but "tilde" equivalents of the two also exist, although like dbtorms~ they're expensive in CPU time). The ftom object's output is -1500 of the input is zero or negative; and likewise, if you give mtof -1500 or lower it outputs zero.

[receive], [r]: Receive messages non-locally. The receive object, which may be abbreviated as "r" waits for non-local messages to be sent by a send object (below) or by a message box using redirection (the ";" feature discussed in the earlier example, A01.sinewave.pd). The argument (such as "frequency" and "pitch" in this example) is the name to which messages are sent. Multiple

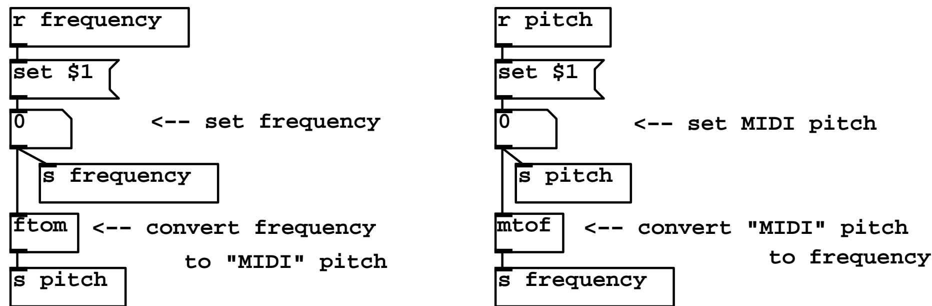


Figure 1.9: Conversion between pitch and frequency in A06.frequency.pd.

`receive` objects may share the same name, in which case any message sent to that name will go to all of them.

send, s: The `send` object, which may be abbreviated as "`s`", directs messages to `receive` objects.

Two new properties of number boxes are used here. Heretofore we've used them as controls or as displays; here, the two number boxes each function as both. If a number box gets a number in its inlet, it not only displays the number but also repeats it to its output. However, a number box may also be sent a "set" message, such as "set 55" for example. This would set the value of the number box to 55 (and display it) but not cause the output that would result from the simple "55" message. In this case, numbers coming from the two receives are formatted (using message boxes) to read "set 55" instead of just "55", and so on. (The special word "$1" is replaced by the incoming number.) This is done because otherwise we would have an infinite loop: frequency would change pitch which would change frequency and so on forever, or at least until something breaks.

Exercises

1. If 0 dB corresponds to an amplitude of 1, how many dB corresponds to amplitudes of 1.5, 2, 3, and 5? (Answer: about 3, 6, 10, and 14.)

2. Two uncorrelated signals of RMS amplitude 3 and 4 are added; what's the RMS amplitude of the sum?

3. How many uncorrelated signals, all of equal amplitude, would you have to add to get a signal that is 9 dB hotter?

4. What is the angular frequency of middle C at 44100 samples per second?

5. If $x[n]$ is an audio signal, show that:

   $$A_{\mathrm{RMS}}\{x[n]\} \le A_{\mathrm{peak}}\{x[n]\}$$

   and

   $$A_{\mathrm{RMS}}\{x[n]\} \ge A_{\mathrm{peak}}\{x[n]\}/\sqrt{N},$$

   where N is the window size. Under what conditions does equality hold for each one?

6. If $x[n]$ is the SINUSOID of Section 1.1, and making the assumptions of section 1.2, show that its RMS amplitude is approximately $a/\sqrt{2}$. *Hint:* use an integral to approximate the sum. Since the window contains many periods, you can assume that the integral covers a whole number of periods.

Chapter 2

Wavetables and samplers

In the previous chapter we treated audio signals as if they always flowed by in a continuous stream at some sample rate. The sample rate isn't really a quality of the audio signal, but rather it specifies how fast the individual samples should flow into or out of the computer. But the audio signal is at bottom just a sequence of numbers, and in practice we don't have to assume that they will be "played" linearly at all. Another, complementary view is that they can be stored in memory, and, later, they can be read back in any order—forward, backward, back and forth, or totally at random. A huge range of new possibilities opens up, one that will never be exhausted.

For many years (roughly 1950-1990), magnetic tape served as the main storage medium for sounds. Tapes were passed back and forth across magnetic pickups to render them in real time. Since 1995 or so, the predominant method of sound storage has been to keep them as digital audio signals, which are read back with much greater freedom and facility than were the magnetic tapes. Many modes of use dating from the tape era are still current, including cutting, duplication, speed change, and time reversal. Other techniques, such as *waveshaping*, have come into their own only in the digital era.

Suppose we have a stored digital audio signal, which is just a sequence of numbers $x[n]$ for $n = 0, ..., N-1$, where N is the size in samples of the stored signal. Then if we have an input signal $y[n]$ (which we assume to be flowing in real time), we can use its values as indices to look up values of the stored signal $x[n]$. This operation, called *wavetable lookup*, gives us a new signal, $z[n]$, calculated as:

$$z[n] = x[y[n]].$$

Schematically we represent this operation as shown in figure 2.1.

Two complications arise. First, the input values, $y[n]$, might lie outside the range $0, ..., N-1$, in which case the wavetable $x[n]$ has no value and the expression for the output $z[n]$ is undefined. In this situation we might choose to anything negative and $N-1$ for anything N or greater. Alternatively, we might prefer to wrap them around end to end. Here we'll adopt the convention

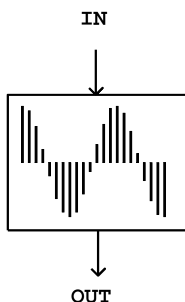


Figure 2.1: Diagram for wavetable lookup. The input is in samples, ranging approximately from 0 to the wavetable's size N, depending on the interpolation scheme.

that out-of-range samples are always clipped; when we need wraparound, we'll introduce another signal processing block to do it for us.

The second complication is that the input values need not be integers; in other words they might fall between the points of the wavetable. In general, this is addressed by choosing some scheme for interpolating between the points of the wavetable. For the moment, though, we'll just round down to the nearest integer below the input. This is called *noninterpolating* wavetable lookup, and its full definition is:

$$z[n] = \begin{cases} x[\lfloor y[n] \rfloor] & \text{if } 0 \leq y[n] < N-1 \\ x[0] & \text{if } y[n] < 0 \\ x[N-1] & \text{if } y[n] \geq N-1 \end{cases}$$

(where the symbol $\lfloor y[n] \rfloor$ means, "the greatest integer not exceeding $y[n]$").

Pictorally, we use $y[0]$ (a number) as a location on the horizontal axis of the wavetable shown in figure 2.1, and the output, $z[0]$, is whatever we get on the vertical axis; and the same for $y[1]$ and $z[1]$ and so on. The "natural" range for the input $y[n]$ is $0 \leq y[n] < N$. This is different from the usual range of an audio signal suitable for output from the computer, which ranges from -1 to 1 in our units. We'll see later that the range of input values, nominally from 0 to N, contracts slightly if interpolating lookup is used.

Figure 2.2 shows a wavetable (a) and the result of using two different input signals as lookup indices into it. The wavetable contains 40 points, which are numbered from 0 to 39. In part (b) of the figure, a *sawtooth wave* is used as the input signal $y[n]$. A sawtooth wave is nothing but a ramp function repeated end to end. In this case the sawtooth's range is from 0 to 40 (this is shown in the vertical axis). The sawtooth wave thus scans the wavetable from left to right—from the beginning point 0 to the endpoint 39—and does so every time it repeats. Over the fifty points shown in Figure 2.2(b) the sawtooth wave

makes two and a half cycles. Its period is twenty samples, or in other words the frequency in Hertz is $R/20$.

Part (c) of figure 2.2 shows the result of applying wavetable lookup, using the table $x[n]$, to the signal $y[n]$. Since the sawtooth input simply reads out the contents of the wavetable from left to right, repeatedly, at a constant rate of precession, the result will be a new periodic signal, whose waveform (shape) is derived from $x[n]$ and whose frequency is determined by the sawtooth wave $y[n]$.

Parts (d) and (e) of figure 2.2 show an example of reading the wavetable in a nonuniform way; since the inputs signal rises from 0 to N and then later recedes to 0, we see the wavetable appear first forward, then frozen at its endpoint, then backward. The table is scanned from left to right and then, more quickly, from right to left. As in the previous example the incoming signal controls the speed of precession while the output's amplitude is that of the wavetable.

2.1 The Wavetable Oscillator

Figure 2.2 suggests an easy way to synthesize any desired fixed waveform at any desired frequency, using the block diagram shown in Figure 2.3. The upper block is an oscillator—not the sinusoidal oscillator we saw earlier, but one which produces sawtooth waves instead. The output values, as indicated at the left of the block, should range from 0 to the wavetable size N. This is used as an index into the wavetable lookup block (introduced in Figure 2.1), resulting, as shown in Figure 2.2(b,c), in a periodic waveform. Figure 2.3(b) adds an envelope generator and a multiplier to control the output amplitude in the same way as for the sinusoidal oscillator shown in Chapter 1. Often, one uses a wavetable with (RMS or peak) amplitude 1, so that the amplitude of the output is just the magnitude of the envelope generator's output.

Wavetable oscillators are often used to synthesize sounds with specified, static spectra. To do this, you can precompute N samples of any waveform of period N (angular frequency $2\pi/N$) by adding up the elements of the FOURIER SERIES (section 1.8). The computation involved in setting up the wavetable at first might be significant, but this may be done in advance of the synthesis process, which can then take place in real time. Frequently, wavetables are prepared in advance and stored in files to be loaded into memory as needed for performance.

While direct additive synthesis of complex waveforms, as shown in Chapter 1, is in principle infinitely flexible as a technique for producing time-varying timbres, wavetable synthesis is much less expensive in terms of computation but requires switching wavetables to change the timbre. An intermediate technique, more flexible and expensive than simple wavetable synthesis but less flexible and less expensive than additive synthesis, is to create time-varying mixtures between a small number of fixed wavetables. If the number of wavetables is only two, this is in effect a cross-fade between the two waveforms, as diagrammed in figure 2.3. Here we suppose that some signal $0 \leq x[n] \leq 1$ is to control the

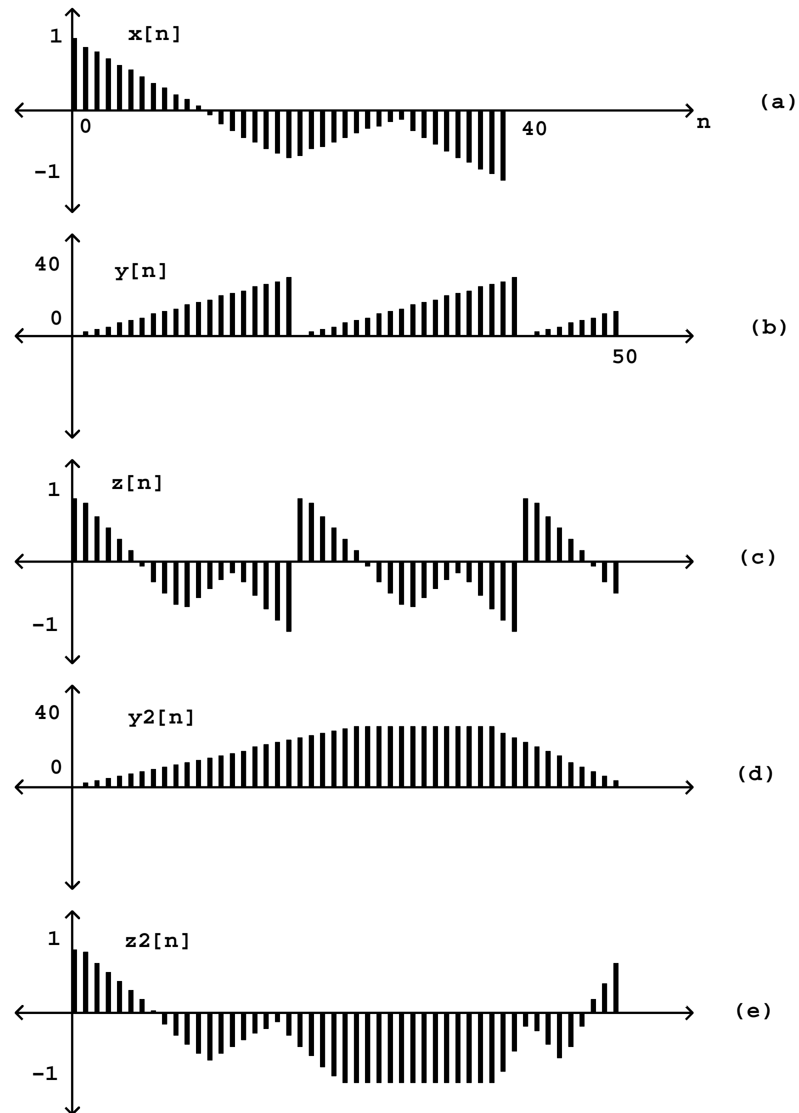


Figure 2.2: Wavetable lookup. (a): a wavetable; (b) and (d): signal inputs for lookup; (c) and (e): the corresponding outputs.

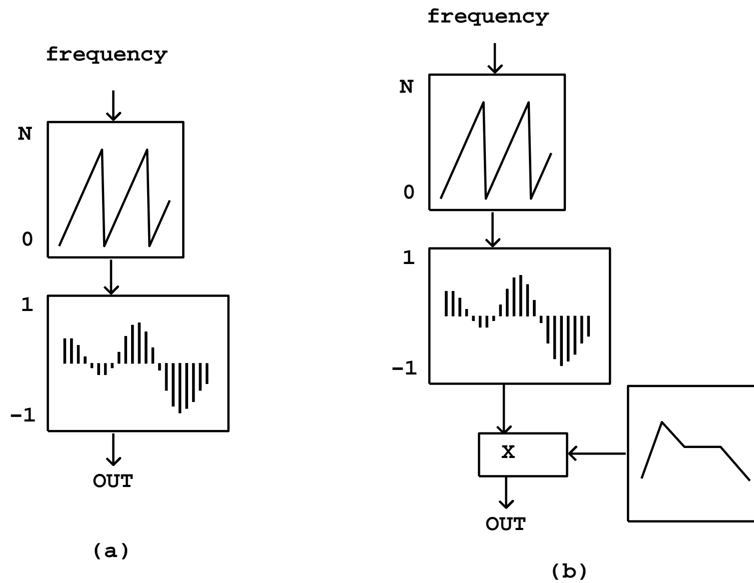


Figure 2.3: Block diagram (a) for a wavetable lookup oscillator , and (b) with amplitude control by an envelope generator.

relative strengths of the two waveforms, so that, if $x[n] = 0$, we get the first one and if $x[n] = 1$ we get the second. Supposing the two signals to be cross-faded are $y[n]$ and $z[n]$, we compute the signal

$$(1 - x[n])y[n] + x[n]z[n],$$

or, equivalently and usually more efficient to calculate,

$$y[n] + x[n](z[n] - y[n]).$$

This computation is diagrammed in figure 2.4.

In using this technique to cross-fade between wavetable oscillators, it might be desirable to keep the phases of corresponding partials the same across the wavetables, so that their amplitudes combine additively when they are mixed. On the other hand, if arbitrary wavetables are used (for instance, borrowed from a recorded sound) there will be a phasing effect as the different waveforms are mixed.

This scheme can be extended in a daisy chain to travel a continuous path between a succession of timbers. Alternatively, or in combination with daisy-chaining, cross-fading may be used to interpolate between two different timbres, for example as a function of musical dynamic. To do this you would prepare two or even several waveforms of a single synthetic voice played at different

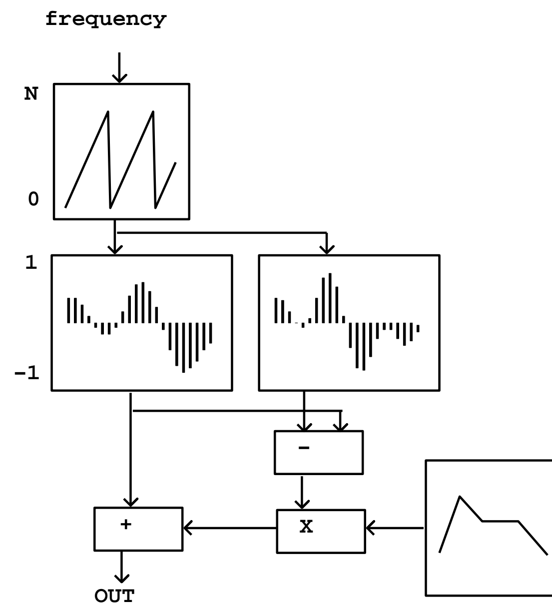


Figure 2.4: Block diagram for cross-fading between two wavetables.

dynamics. and interpolate between successive ones as a function of the output dynamic you want.

You can even use pre-recorded instrumental (or other) sounds as a waveform. In its simplest form this is called ”sampling” and is the subject of the next section.

2.2 Sampling

To make a sampler, we just record a real sound into a wavetable and then later read it back out again. In music stores the entire wavetable is usually called a “sample” but to avoid confusion we’ll only use the word “sample” here to mean a single point in an audio signal, as described in Chapter 1.

Going back to figure 2.2, suppose that instead of 40 points the wavetable $x[n]$ is a one-second recorded sample, originally recorded at a sample rate of 44100, so that it has 44100 points; and let $y[n]$ in part (b) of the figure have a period of 22050 samples. This corresponds to a frequency of 1/2 Hz. But what we hear is not a pitched sound at 2 cycles per second (that’s too slow to hear as a pitch) but rather, we’ll hear the original sample $x[n]$ played back repeatedly at double speed. We’ve just re-invented the sampler.

At its simplest, a sampler is simply a wavetable oscillator, as was shown in figure 2.3. However, in the earlier discussion we imagined playing the oscillator back at a frequency high enough to be perceived as a pitch, at least 30 Hz or so. In the case of sampling, the frequency is usually lower than 30 Hz, and so the period, at least 1/30 second and perhaps much more, is long enough that you can hear the individual cycles as separate events.

In general, if we assume the sample rate R of the recorded sample is the same as the output sample rate, if the wavetable has N samples and if we play it back using a sawtooth wave of period M, the sample is transposed by a factor of N/M, equal to Nf/R if f is the frequency in Hz of the sawtooth. As an interval, the transposition in half steps is given by the TRANSPOSITION FORMULA FOR LOOPING WAVETABLES:

$$h = 12\log_2\left(\frac{N}{M}\right) = 12\log_2\left(\frac{Nf}{R}\right).$$

Frequently the desired transposition h is known and the formula must be solved for either f or N:

$$f = \frac{2^{h/12}R}{N},$$

$$N = \frac{2^{h/12}R}{f},$$

where h is the desired transposition in half steps.

So far we have used a sawtooth as the input wave $y[t]$, but, as suggested in parts (d) and (e) of figure 2.2, we could use anything we like as an input signal.

In this case, the transposition is time dependent and is controlled by the rate of change of the input signal.

As a speed multiple the transposition multiple t and the transposition in half steps h are given by the: MOMENTARY TRANSPOSITION FORMULAS FOR WAVETABLES:

$$t[n] = |y[n] - y[n-1]|,$$

$$h[n] = 12 \log_2 |y[n] - y[n-1]|.$$

(Here the enclosing bars (|) mean absolute value.) For example, if $y[n] = n$, then $z[n] = x[n]$ so we hear the wavetable at its original pitch, and this is what the formula predicts since, in that case,

$$y[n] - y[n-1] = 1.$$

On the other hand, if $y[n] = 2n$, then the wavetable is transposed up an octave, consistent with

$$y[n] - y[n-1] = 2.$$

If values of $y[n]$ are decreasing with n, you hear the sample backward, but the transposition formula still gives a positive multiplier. This is all consistent with the earlier TRANSPOSITION FORMULA FOR LOOPING WAVETABLES; if a sawtooth ranges from 0 to N, f times per second, the difference of successive samples is just Nf/R—excepting the samples at the beginnings of new cycles.

It's well known that transposing a sample also transposes its timbre—this is the "chipmunk" effect. Not only are any periodicities (such as might give rise to pitch) in the sample transposed, but so are the frequencies of the overtones. Some timbres, notably those of vocal sounds, can be described in terms of frequency ranges in which overtones are stronger than their neighbors. These frequency ranges are also transposed, which is heard as a timbre change. In language that will be made more precise in section X.XX, we say that the *spectral envelope* is transposed along with the pitch or pitches.

In both this and the preceding section, we have considered playing wavetables periodically. In section 2.1 the playback repeated quickly enough that the repetition gives rise to a pitch, say between 25 and 4000 times per second, roughly the range of a piano. In the current section we assume a wavetable one second long, and in this case "reasonable" transposition factors (less than four octaves up) would give rise to a rate of repetition below 25, usually much lower, and going down as low as we wish.

The number 25 is significant for another reason: it is roughly the maximum number of separate events the ear can discern per second; for instance, 25 syllables of speech or melodic notes per second, or attacks of a snare drum roll, are about the most we can hope to crowd into a second before our ability to distinguish them breaks down.

A continuum exists between samplers and wavetable oscillators, in that the patch of Figure 2.3 can either be regarded as a sampler (if the frequency of repetition is less than about 20 Hz.) or as a wavetable oscillator (if the frequency is greater than about 40 Hz.) It is possible to move continuously between the

two regimes. Furthermore, it is not necessary to play an entire sample in a loop; with a bit more arithmetic we can choose sub-segments of the sample, and these can change in length and location continuously as the sample is played.

The practice of playing many small segments of a sample in rapid succession is often called *granular synthesis*. For much more discussion of the possibilities, see [Roa01].

Figure 2.5 shows how to build a very simple looping sampler. In the figure, if we call the frequency f and the segment size in samples is s, the output transposition factor is given by $t = fs/R$, where R is the sample rate at which the wavetable was recorded (which need not equal the sample rate the block diagram is working at.) In practice, this equation must usually be solved for either f or s to attain a desired transposition.

In the figure, a sawtooth oscillator controls the location of wavetable lookup, but the lower and upper values of the sawtooth aren't statically specified as they were in Figure 2.3; rather, the sawtooth oscillator simply ranges from 0 to 1 in value and the range is adjusted to select a desired segment of samples in the wavetable.

It might be desirable to specify the segment's location l either as its left-hand edge (its lower bound) else as the segment's midpoint; in either case we specify the length s as a separate parameter. In the first case, we start by multiplying the sawtooth by s, so that it then ranges from 0 to s; then we add l so that it now ranges from l to $l + s$. In order to specify the location as the segment's midpoint, we first subtract 1/2 from the sawtooth (so that it ranges from $-1/2$ to $1/2$, and then as before multiply by s (so that it now ranges from $-s/2$ to $s/2$ and add l to give a range from $l - s/2$ to $l + s/2$.

In the looping sampler, we will need to worry about the continuity between the beginning and the end of segments of a sample, which we'll consider in the next section.

A further detail is that, if the segment size and location are changing with time (they might be digital audio signals themselves, for instance), they will affect the transposition factor, and the pitch or timbre of the output signal might waver up and down as a result. The simplest way to avoid this problem is to synchronize changes in the values of s and l with the regular discontinuities of the sawtooth; since the signal jumps discontinuously there, the transposition is not really defined there anyway, and, if you are enveloping to hide the discontinuity, the effects of changes in s and l are hidden as well.

2.3 Enveloping samplers

In the previous section we considered reading a wavetable either sporadically or repeatedly to make a sampler. In most real applications we must deal with getting the samples to start and stop cleanly, so that the output signal doesn't jump discontinuously at the beginnings and ends of samples. This discontinuity can sound like a click or a thump depending on the wavetable.

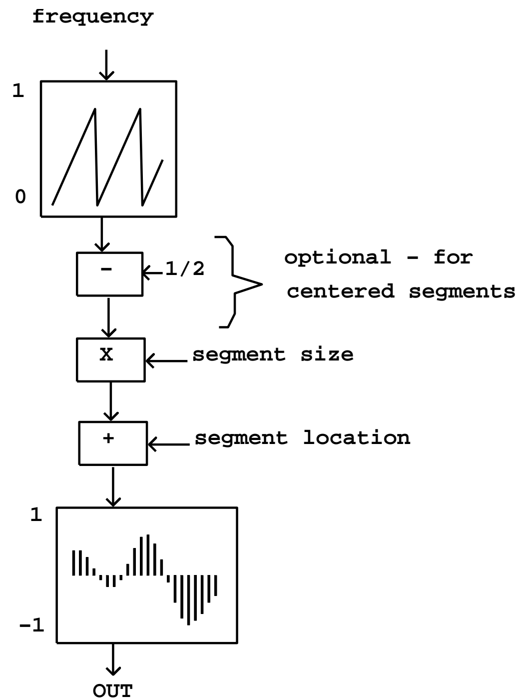


Figure 2.5: (a) A simple looping sampler, as yet with no amplitude control. There are inputs to control the frequency and the segment size and location. The "-" operation is included if we wish the segment location to be specified as the segment's midpoint; otherwise we specify the location of the left end of the segment.

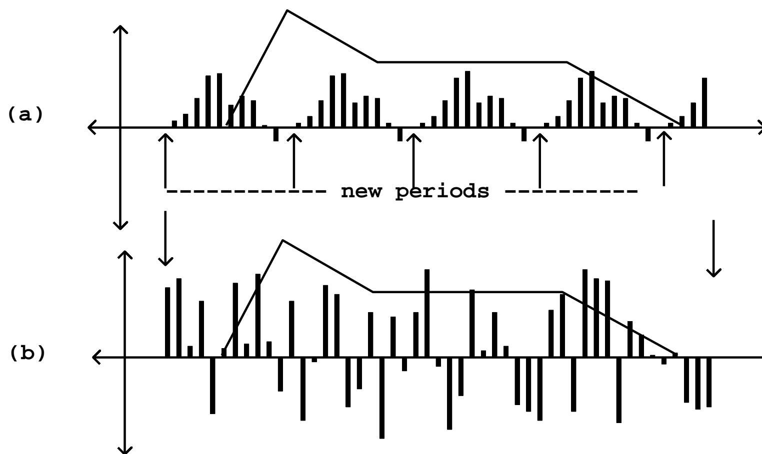


Figure 2.6: Differing envelope requirements for oscillators and samplers: (a) in an oscillator, the envelope can be chosen to conform to any desired timescale; (b) when the wavetable is a recorded sound, it's up to you to get the envelope to zero before you hit the end of the wavetable for the first time.

The easiest way to do this, assuming we will always play a wavetable completely from beginning to end, is simply to prepare the sample in advance so that it fades in cleanly at the beginning and out cleanly at the end. This may even be done in the case that the wavetable is sampled live, by multiplying the input signal by a line segment envelope timed to match the length of the recording.

In many situations, however, it is either inconvenient or impossible to pre-envelope the sample—for example, we might want to play only part of the sample back, or we may want to change the sharpness of the enveloping dynamically. In section 2.1 we had already seen how to control the amplitude of sinusoidal oscillators using multiplication by a ramp function (also known as an envelope generator), and we built this notion into the wavetable oscillators of Figures 2.3 and 2.4. This also works fine for turning samplers on and off to avoid discontinuities, but with one major difference: whereas in wavetable synthesis, we were free to assume that the waveform lines up end to end, so that we are free to choose any envelope timing we want, in the case of sampling using unprepared waveforms, we are obliged to get the envelope generator's output to zero by the time we reach the end of the wavetable for the first time. This situation is pictured in Figure 2.6.

In situations where an arbitrary wavetable must be repeated as needed, the simplest way to make the looping work continuously is to arrange for amplitude

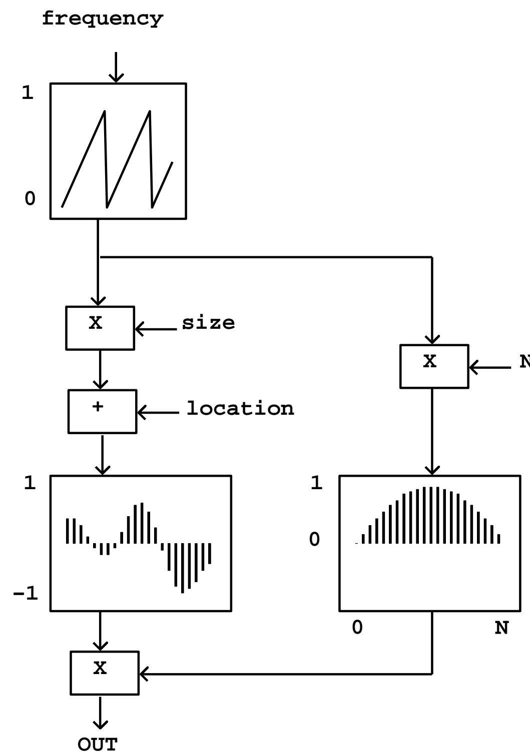


Figure 2.7: A sampler as in Figure 2.6, but with an additional wavetable lookup for enveloping.

change to be synchronized with the looping, using a separate wavetable (the envelope). This may be implemented as shown in figure 2.7. A single sawtooth oscillator is used to calculate lookup indices for two wavetables, one holding the recorded sound, and the other, an envelope shape. The main thing to worry about is getting the inputs of the two wavetables each in its own appropriate range.

In many situations it is desirable to combine two or more copies of the looping wavetable sampler at the same frequency and at a specified phase relationship. This may be done so that when any particular one is at the end of its segment, one or more others is in the middle of the same segment, so that the aggregate is continuously making sound. To accomplish this, we need a way to generate two or more sawtooth waves at the desired phase relationship that we can use in place of the oscillator at the top of figure 2.7. We can start with a single sawtooth wave and then produce others at fixed phase relationships with the first one. If we wish a sawtooth which is, say, a cycles ahead of the first one,

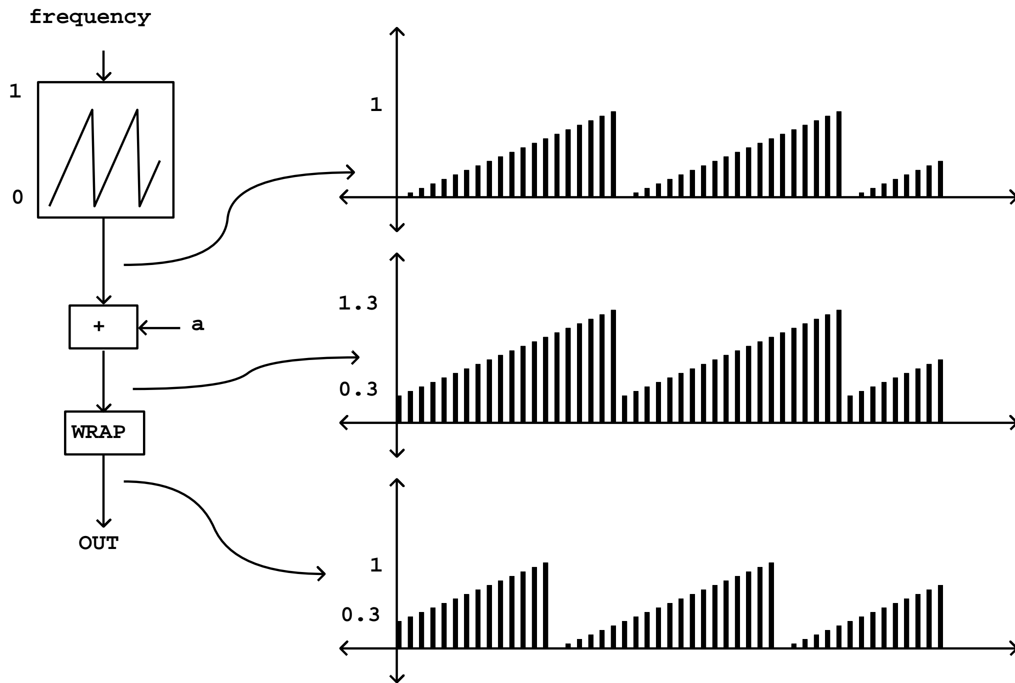


Figure 2.8: A technique for generating two or more sawtooth waves with fixed phase relationships between them. The relative phase is controlled by the parameter a.

we simply add the parameter a and then take the fractional part, which is the desired new sawtooth wave, as shown in Figure 2.8.

2.4 Timbre stretching

The waveform oscillator of section 2.1, which we extended in section 2.2 to encompass grabbing waveforms from arbitrary wavetables such as recorded sounds, may also, or simultaneously, be extended in a complementary way, that we'll refer to as *timbre stretching*, for reasons we'll develop in this section. There are also many other possible ways to extend the wavetable oscillator, using, for instance frequency modulation and waveshaping, but we'll leave those for later chapters.

The central idea of timbre stretching is to reconsider the idea of the wavetable oscillator as a mechanism of playing a stored wavetable (or part of one) end to end. There is no reason the end of one cycle has to coincide with the beginning of another. Instead, we could ask for copies of the waveform to be spaced with alternating segments of silence; or, going in the opposite direction, the

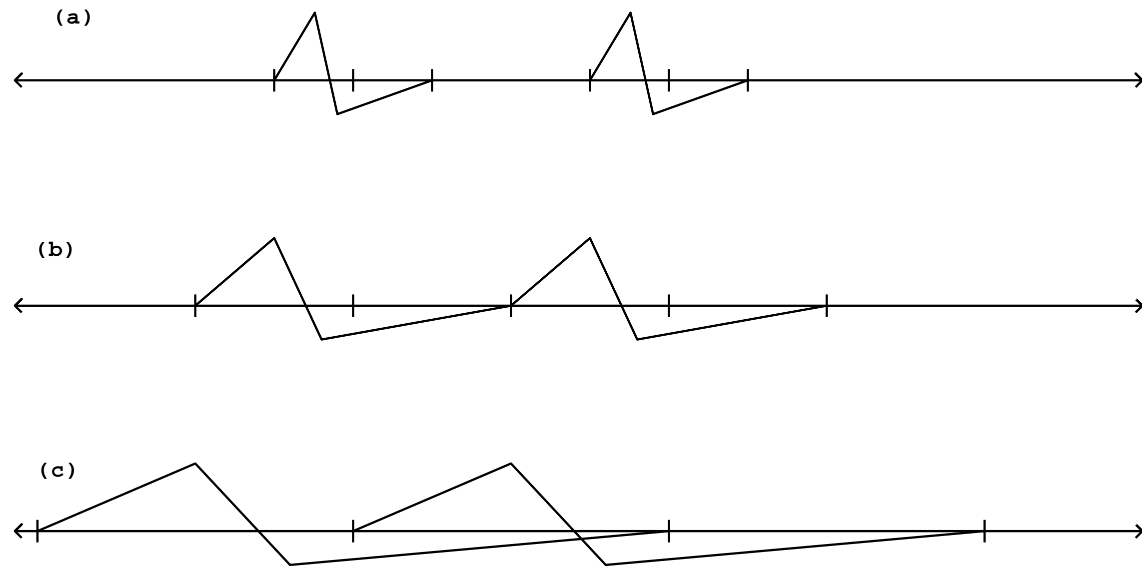


Figure 2.9: A waveform is played at a period of 20 samples: (a) at 100 percent duty cycle; (b) at 50 percent; (c) at 200 percent

waveform copies could be space more closely together so that they overlap. The single parameter available in section 2.1—the frequency—has been heretofore used to control two separate aspects of the output: the period at which we start new copies of the waveform, and also the length of each individual copy. The idea of timbre stretching is to control the two independently.

Figure 2.9 shows the result of playing a wavetable in three ways. In each case the output waveform has period 20; in other words, the output frequency is $R/20$ if R is the output sample rate. In part (a) of the figure, each copy of the waveform is played over 20 samples, so that the wave form fits exactly into the cycle with no gaps and no overlap. In part (b), although the period is still 20, the waveform is compressed into the middle half of the period (10 samples); or in other words, the *duty cycle*—the relative amount of time the waveform fills the cycle—equals 50 percent. THe remaining 50 percent of the time, the output is zero.

In part (c), the waveform is stretched to 40 samples, and since it is still repeated every 20 samples, the waveforms overlap two to one. The duty cycle is thus 200 percent.

Suppose now that the 100 percent duty cycle waveform has a Fourier series (section 1.8) equal to:

$$x_{100}[n] = a_0 + a_1 \cos(\omega n + \phi_1) + a_2 \cos(2\omega n + \phi_2) + \cdots$$

where ω is the angular frequency (equal to $\pi/10$ in our example since the period is 20.) To simplify this example we won't worry about where the series must end, and will just let it run on forever.

We would like to relate this to the Fourier series of the other two waveforms in the example, in order to show how changing the duty cycle changes the timbre of the result. For the 50 percent duty cycle case (calling the signal $x_50[n]$), we observe that the waveform, if we replicate it out of phase by a half period and add the two, gives exactly the original waveform at twice the frequency:

$$x_{100}[2n] = x_{50}[n] + x_{50}[n + \frac{\pi}{\omega}],$$

where ω is the angular frequency (and so π/ω is half the period) of both signals. So if we denote the Fourier series of $x_{50}[n]$ as:

$$x_{50}[n] = b_0 + b_1 \cos(\omega n + \theta_1) + b_2 \cos(2\omega n + \theta_2) + \cdots$$

and substitute the Fourier series for all three terms above, we get:

$$a_0 + a_1 \cos(2\omega n + \phi_1) + a_2 \cos(4\omega n + \phi_2) + \cdots = b_0 + b_1 \cos(\omega n + \theta_1) + b_2 \cos(2\omega n + \theta_2) + \cdots + b_0 + b_1 \cos(\omega n + \pi + \theta_1) +$$

$$= 2b_0 + 2b_2 \cos(2\omega n + \theta_2) + 2b_4 \cos(4\omega n + \theta_4) + \cdots,$$

so we get:

$$a_0 = 2b_0, a_1 = 2b_2, a_2 = 2b_4,$$

and so on: the even partials of x_{50}, at least, are obtained by stretching the partials of x_{100} out twice as far. (We don't yet know about the odd partials of x_{50}, and these might be in line with the even ones or not, depending on factors we can't control yet. Suffice it to say for the moment, that if the waveform connects smoothly with the horizontal axis at both ends, the odd partials will act globally like the even ones. To make this more exact we'll need to use Fourier analysis, which is developed in a later chapter.)

Similarly, x_100 and x_200 are related in exactly the same way:

$$x_{200}[2n] = x_{100}[n] + x_{100}[n + \frac{\pi}{\omega}],$$

so that, if the amplitudes of the fourier series of x_{200} are denoted by c_0, c_1, $\cdots$, we get:

$$c_0 = 2a_0, c_1 = 2a_2, c_2 = 2a_4, \cdots,$$

so that the spectrum of x_{200} is that of x_{100} shrunk by half.

We see that squeezing the waveform by a factor of 2 has the effect of stretching the Fourier series out by two, and on the other hand stretching the waveform by a factor of two squeezes the Fourier series by two. By the same sort of argument, in general it turns out that stretching the waveform by a factor of any positive number f squeezes the Fourier spectrum by the reciprocal $1/f$—at least approximately, and the approximation is at least fairly good if the waveform "behaves well" at its ends. (As we'll see later, the waveform can always be forced to behave at least reasonably well by enveloping it as in Figure 2.7.)

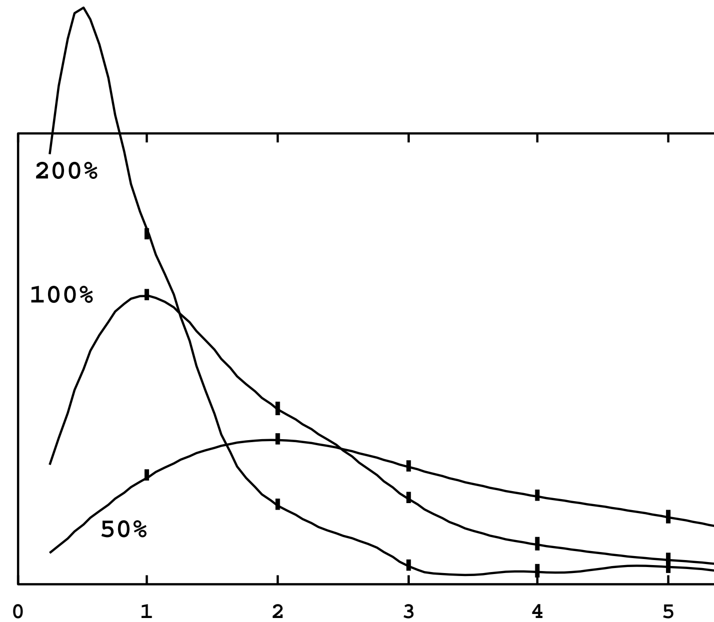


Figure 2.10: The Fourier series magnitudes for the waveforms shown in Figure 2.9. The horizontal axis is the harmonic number. We only "hear" the coefficients for integer harmonic numbers; the continuous curves are the "ideal" spectrum.

Figure 2.10 shows the spectra of the three waveforms—or in other words the one waveform at three duty cycles—of Figure 2.9. The figure emphasizes the relationship between the three spectra by drawing curves through each, which, on inspection, turn out to be the same curve, only stretched differently; as the duty cycle goes up, the curve is both compressed to the left (the frequencies all drop) and amplified (stretched upward).

The continuous curves have a very simple interpretation. Imagine squeezing the waveform into some tiny duty cycle, say 1 percent. The spectrum will be stretched by a factor of 100. Working backward, this would allow us to interpolate between each pair of consecutive points of the 100 percent duty cycle spectrum (the original one) with 99 new ones. Already in the figure the 50 percent duty cycle trace defines the curve with twice the resolution of the original one. In the limit, as the duty cycle gets arbitrarily small, the spectrum is filled in more and more densely; and the limit is the "true" spectrum of the waveform.

This "true" spectrum is only audible at suitably low duty cycles, though. The 200 percent duty cycle example actually misses the peak in the ideal (continuous) spectrum because the peak falls below the first harmonic. In general,

higher duty cycles sample the ideal curve at lower resolutions.

Timbre stretching gives us an extremely powerful technique for generating sounds with systematically variable spectra. Combined with the possibilities of mixtures of waveforms (section 2.1) and of snatching endlessly variable waveforms from recorded samples (section 2.2), it is possible to generate all sorts of sounds.

For example, the block diagram of Figure 2.7 gives us a way to to grab and stretch timbres from a recorded wavetable. When the "frequency" parameter f is high enough to be audible as a pitch, the "size" parameter s can be thought of as controlling timbre stretch, via the formula $s = tR/f$ from section 2.2, where we now reinterpret t as the factor by which the timbre is to be stretched.

2.5 Interpolation

As mentioned before, interpolation schemes are often used to increase the accuracy of table lookup. Here we will give a somewhat simplified account of the effects of table sizes and interpolation schemes on the result of table lookup.

To speak of error in table lookup, we must view the wavetable as a sampled version of an underlying function—and when we ask for a value of the underlying function which lies between the points of the wavetable, the error is the difference between the result of the wavetable lookup and the "ideal" value of the function at that point. The most revealing study of wavetable lookup error assumes that the underlying function is the SINUSOID of chapter 1. We can then understand what happens to other wavetables by considering them as superpositions (sums) of sinusoids.

The accuracy of lookup from a wavetable containing a real sinusoid depends on two factors: the quality of the interpolation scheme, and the period of the sinusoid. In general, the longer the period of the sinusoid, the more accurate the result.

In the case of a synthetic wavetable, we might know its sinusoidal components from having specified them—in which case the issue becomes one of choosing a wavetable size appropriately, when calculating the wavetable, to match the interpolation algorithm and meet the desired standard of accuracy. In the case of recorded sounds, the accuracy analysis might lead us to adjust the sample rate of the recording, either at the outset or else by resampling later.

Interpolation error for a sinusoidal wavetable can have two components: first, the continuous signal (the theoretical result of reading the wavetable continuously in time, as if the output sample rate were infinite) might not be a pure sinusoid; and second, the amplitude might be wrong. (It is possible to get phase errors as well, but only through carelessness; and we won't worry about that here.)

In this treatment we'll only consider polynomial interpolation schemes such as rounding, linear interpolation, and cubic interpolation. These schemes amount to evaluating polynomials (of degree zero, one, and three, respectively) in the interstices between points of the wavetable. The idea is that, for any index x,

we choose a nearby "good" point x_0, and let the output be calculated by some polynomial:

$$a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \cdots + a_n(x - x_0)^n.$$

Usually we choose the polynomial which passes through the $n+1$ nearest points of the wavetable. For 1-point interpolation (a zero-degree polynomial) this means letting a_0 equal the nearest point of the wavetable. For two-point interpolation, we draw a line segment between the two points of the wavetable on either side of the desired point x. We can let x_0 be the point at the left of x (which we write as $\lfloor x \rfloor$) and then the formula for linear interpolation is:

$$y[x_0] + (y[x_0 + 1] - y[x_0]) \cdot (x - x_0).$$

or in other words,

$$a_0 = y[x_0],$$

$$a_1 = (y[x_0 + 1] - y[x_0]).$$

In general, you can fit exactly one polynomial of degree $n - 1$ through any n points as long as their x values are all different.

Figure 2.11 shows the effect of using linear (two-point) interpolation to fill in a sinusoid of period 6. At the top are three traces: the original sinusoid, the linearly-interpolated result of using 6 points per period to represent the sinusoid, and finally, another sinusoid, of slightly smaller amplitude, which better matches the six-segment waveform. The error introduced by replacing the original sinusoid by the linearly interpolated version has two components: first, a (barely perceptible) change in amplitude, and second, a (very perceptible) distortion of the wave shape.

The bottom graph in the figure shows the difference between the segmented waveform and the best-fitting sinusoid. This is the distortion viewed as As the number of points increases, the error gets smaller in magnitude. Since the error is the difference between a best-fitting line segment and a curve, the magnitude of the error is roughly proportional to the square of the phase difference between each pair of points, or in other words, inversely proportional to the square of the number of points in the wavetable. Put another way, wavetable error decreases by 12 dB each time the table doubles in size. (This approximation is only good for tables with 4 or more points.)

Four-point (cubic) interpolation works similarly. The formula is:

$$-f(f-1)(f-2)/6 \cdot y[x_0 - 1] + (f+1)(f-1)(f-2)/2 \cdot y[x_0]$$

$$-(f+1)f(f-2)/2 \cdot y[x_0 + 1] + (f+1)f(f-1)/6 \cdot y[x_0 + 2],$$

where $f = x - x_0$ is the fractional part of the index. For tables with 4 or more points, doubling the number of points on the table tends to improve the RMS error by 24 dB. Table 2.5 shows the calculated RMS error for sinusoids at various periods for 1, 2, and 4 point interpolation. (A slightly different quantity

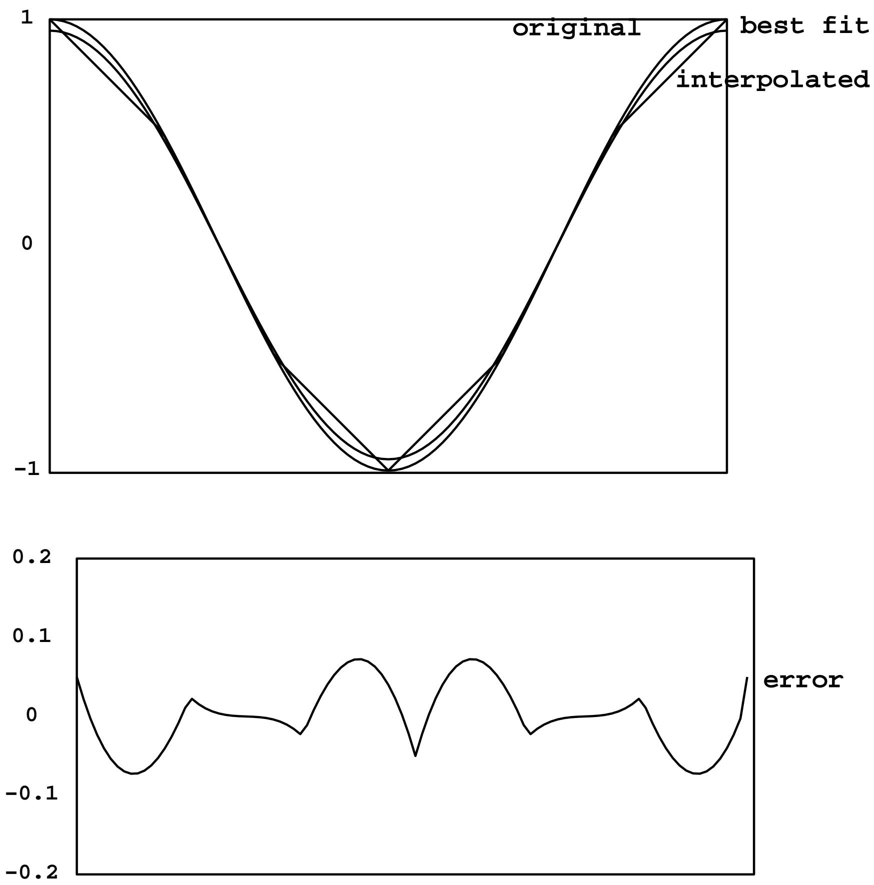


Figure 2.11: Linear interpolation of a sinusoid: (a) the original sinusoid, the interpolated sinusoid, and the best sinusoidal fit back to the interpolated version; (b) the error.

period	interpolation points		
	1	2	4
2	-1.2	-17.1	-20.2
3	-2.0	-11.9	-15.5
4	-4.2	-17.1	-24.8
8	-10.0	-29.6	-48.4
16	-15.9	-41.8	-72.5
32	-21.9	-53.8	-96.5
64	-27.9	-65.9	-120.6
128	-34.0	-77.9	-144.7

Table 2.1: RMS error for table lookup using 1, 2, and 4 point interpolation at various table sizes.

is measured in [Moo90, p.164]. There, the errors in amplitude and phase are also added in, yielding slightly more pessimistic results. See also [Har87].)

The allowable input domain for table lookup depends on the number of points of interpolation. In general, if using k-point interpolation into a table with N points, the inputs may range over an interval of $N+1-k$ points. If $k=1$ (i.e., no interpolation at all), the domain is from 0 to N (including the endpoint at 0 but excluding the one at n) if input values are truncated (and we will always use truncation in this book when doing non-interpolated table lookup). The domain is from -1/2 to $N-1/2$ if, instead, we round the input to the nearest integer instead of interpolating. In either case, the domain stretches over a length of N points.

For two-point interpolation, the inputs must lie between the first and last points, that is, between 0 and $N-1$. So the N points suffice to define the function over a domain of length $N-1$. For four-point interpolation, we cannot get values for inputs between 0 and 1 (not having the required two points to the left of the input) and neither can we for the space between the last two points ($N-2$ and $N-1$. So in this case the domain reaches from 1 to $N-2$ and has length $N-3$.

Periodic waveforms stored in wavetables require special treatment at the ends of the table. For example, suppose we wish to store a pure sinusoid of length N. For a noninterpolating table, it suffices to set, for example,

$$x[n] = cos(2\pi n/N), n = 0, \cdots, N-1.$$

For two-point interpolation, we need $N+1$ points:

$$x[n] = cos(2\pi n/N), n = 0, \cdots, N;$$

in other words, we must repeat the first ($n=0$) point at the end, so that the last segment from $N-1$ to N reaches back to the beginning value.

For four-point interpolation, the cycle must be adjusted to start at the point $n=1$, since we can't get properly interpolated values out for inputs less than one. If, then, one cycle of the wavetable is arranged from 1 to N, we must

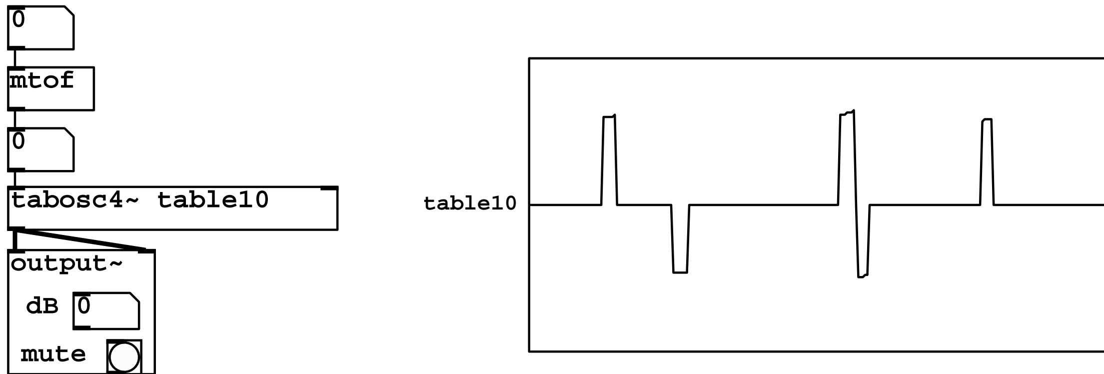


Figure 2.12: A wavetable oscillator: B01.wavetables.pd.

supply extra points for 0 (copied from N), and also $N+1$ and $N+2$, copied from 1 and 2, to make a table of length $N+3$. For the same sinusoid as above, the table should contain:

$$x[n] = cos(2\pi(n-1)/N), n = 0, \cdots, N+2.$$

2.6 Examples

2.6.1 wavetable oscillator

Patch B01.wavetables.pd, shown in figure 2.12, implements a wavetable oscillator, which plays back from a wavetable named "table10". Two new Pd primitives are shown here. First is the wavetable itself, which appears at right in the figure. You can "mouse" on the wavetable to change its shape and hear the sound change as a result. Not shown in the figure but demonstrated in the patch is Pd's facility for automatically calculating wavetables with specified partial amplitudes, which is often preferable to drawing waveforms by hand. You can also read and write tables to (text or sound) files for interchanging data with other programs. The other novelty is an object class:

tabosc4 ~ : a wavetable oscillator. The "4" indicates that this class uses 4-point (cubic) interpolation. In the example, the table's name, "table10", is specified as a creation argument to the tabosc4 ~ object. (You can also switch between wavetables dynamically by sending appropriate messages to the object.)

Wavetables used by tabosc4 ~ must always have a period equal to a power of two; but as shown above, the wavetable must have three extra points wrapped around the ends. Allowable table lengths are thus of the form $2^m + 3$, such as 131, 259, 515, etc.

Wavetable oscillators are not limited to use as audio oscillators. Patch B01.wavetables.pd shows a pair of wavetable oscillators in series. The first one's

output is used as the input of the second one, and thus controls its frequency which changes periodically in time.

2.6.2 wavetable lookup in general

The tabosc4 ~ class, while handy and efficient, is somewhat specialized and for many of the applications described in this chapter we need something more general. Patch B03.tabread4.pd (Figure 2.13) demonstrates the timbre stretching technique discussed in section 2.4. This is a simple example of a situation where tabosc4 ~ would not have sufficed. There are new classes introduced here:

tabread4 ~ : wavetable lookup. As in tabosc4 ~, the table is read using 4-point interpolation. But whereas tabosc4 ~ takes a frequency as input and automatically reads the waveform in a repeating pattern, the simpler tabread4 ~ expects the table lookup index as input. If you want to use it to do something repetitious, as in this example, the input itself has to be a repeating waveform. Like tabosc4 ~ (and all the other table reading and writing objects), you can send messages to select which table to use.

tabwrite ~ : record an audio signal into a wavetable. In this example the tabwrite ~ is used to display the output (although later on it will be used for all sorts of other things.) Whenever it receives a "bang" message from the button icon above it, tabwrite ~ begins writing successive samples of its input to the named table.

Patch B03.tabread4.pd shows how to combine a phasor ~ and a tabread4 ~ object to make a wavetable oscillator. The phasor ~'s output ranges from 0 to 1 in value. In this case the input wavetable, named "waveform12", is 131 elements long. The domain for the tabread4 ~ object is thus from 1 to 129. To adjust the range of the phasor ~ accordingly, we multiply it by the length of the domain (128) so that it reaches between 0 and 128, and then add 1, effectively sliding the interval to the right by one point. This rescaling is accomplished by the * ~ and + ~ objects between the phasor ~ and the tabread4 ~.

With only these four boxes we would have essentially reinvented the tabosc4 ~ class. In this example, however, the multiplication is not by a constant 128 but by a variable amount controlled by the "squeeze" parameter. The function of the four boxes at the right hand side of the patch is to supply the * ~ object with values to scale the phasor ~ by. This makes use of one more new object class:

pack : compose a list of two or more elements. The creation arguments establish the number of arguments, their types (usually numbers) and their initial values. The inlets (there will be as many as you specified creation arguments) update the values of the message arguments, and, if the leftmost inlet is changed (or just triggered with a `bang` message), the message is output.

In this patch the arguments are initially 0 and 50, but the number box will update the value of the first argument, so that, as pictured, the most recent message to leave the pack object was "206 50". The effect of this on the `line~` object below is to ramp to 206 in 50 milliseconds; in general the output of the

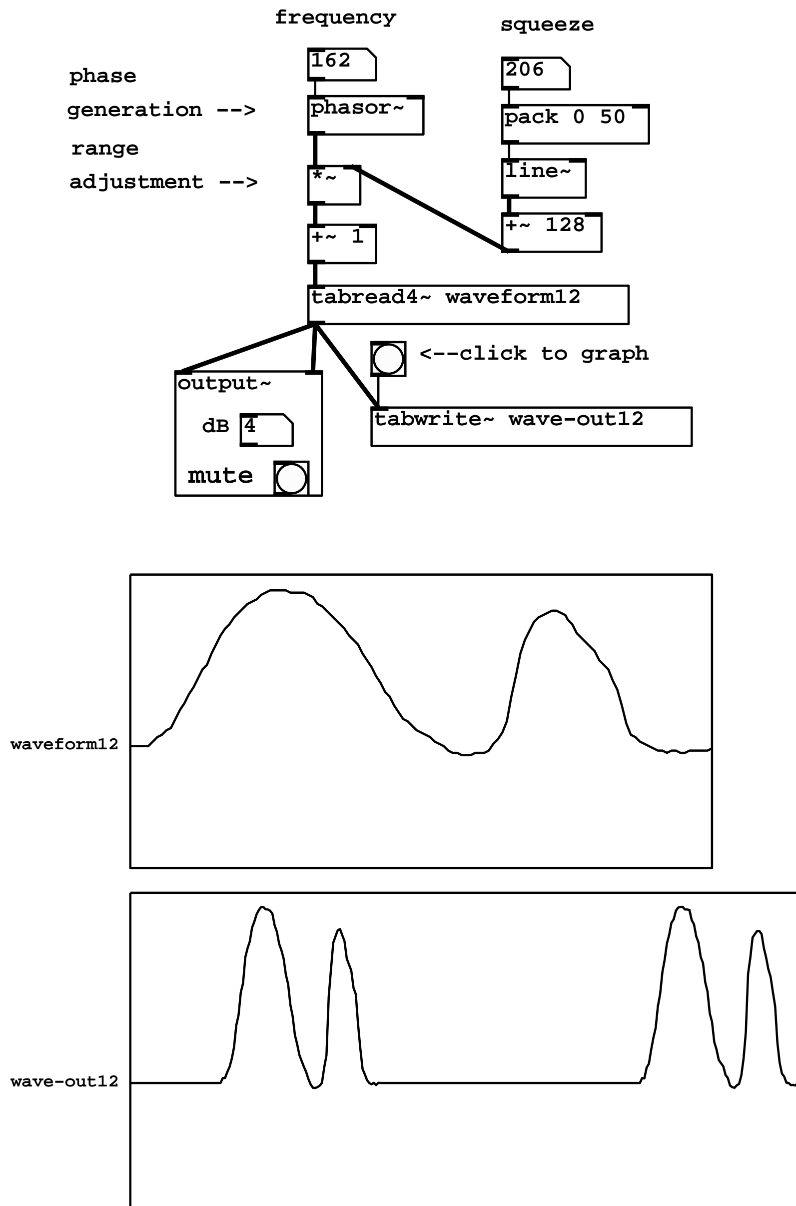


Figure 2.13: A wavetable oscillator with variable duty cycle: B03.tabread4.pd.

line~ object is an audio signal that smoothly follows the sporadically changing values of the number box labeled “squeeze”.

Finally, 128 is added to the “squeeze” value; if “squeeze” takes non-negative values (as the number box in this patch enforces), the range-setting multiplier ranges the phasor by 128 or more. If the value is greater than 128, the effect is that the rescaled phasor spends some fraction of its cycle stuck at the end of the wavetable (which clips its input to 129.) The result is that the waveform is scanned over some fraction of the cycle. As shown, the waveform is squeezed into 128/(128+206) of the cycle, so the spectrum is stretched by a factor of about 1/2.

For simplicity, this patch is subtly different from the example of section 2.4 in that the waveforms are squeezed toward the beginning of each cycle and not toward the middle. This has the effect of slightly changing the phase of the various partials of the waveform as it is stretched and squeezed; if the squeezing factor changes quickly, the corresponding phase drift will sound like a slight wavering in pitch. This can be avoided by using a slightly more complicated arrangement: subtracting 1/2 from the phasor~, multiply it by 128 or more, and then add 65 instead of one.

2.6.3 using a wavetable as a sampler

Patch B04.sampler.pd (Figure 2.14) shows how to use a wavetable as a sampler. In this case the index into the sample (the wavetable) is controlled by mousing on a number box at top. A convenient scaling for the number box is hundredths of a second; to convert to samples (as the input of tabread4~ requires) we multiply by 44100 samples/sec times 0.01 sec to get 441 samples per unit, before applying pack and line~ in much the same way as they were used in the previous example. The transposition you hear depends on how quickly you mouse up and down. This example has introduced one new object class:

hip ~ : simple high-pass (low-cut) filter. The creation argument gives the rolloff frequency in Hertz. We use it here to eliminate the constant (zero-frequency) output when the input sits in a single sample (whenever you aren't actively changing the wavetable reading location with the mouse.) Filters are discussed in chapter ??.

The pack and line~ in this example are not merely to make the sound more continuous, but are essential to making the sound intelligible at all. If the index into the wavetable lookup simply changed every time the mouse moved a pixel (say, twenty to fifty times a second) the overwhelming majority of samples would get the same index as the previous sample (the other 44000+ samples, not counting the ones where the mouse moved.) So the speed of precession would almost always be zero. Instead of changing transpositions, you would hear 20 to 50 Hertz grit. (Try it to find out what that sounds like!)

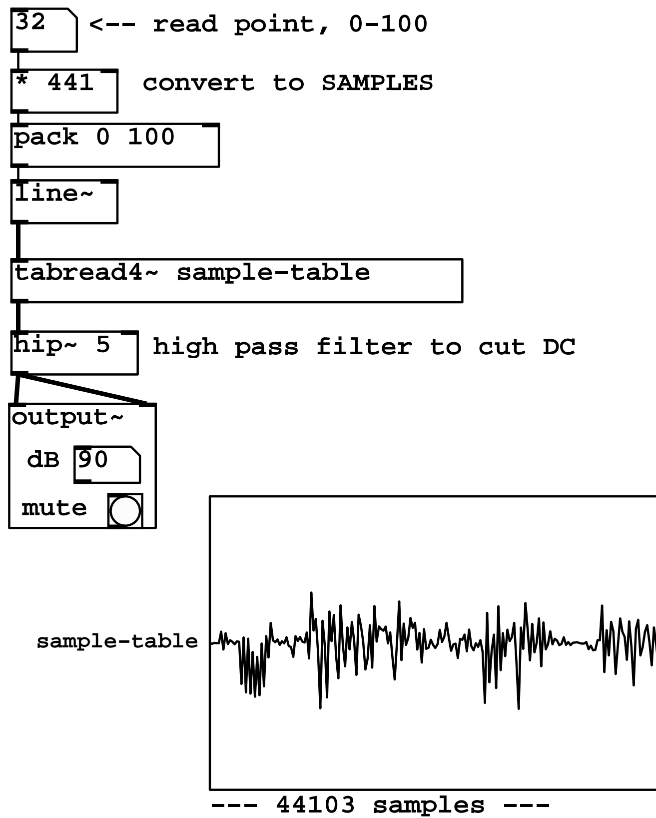


Figure 2.14: A sampler with mouse-controlled index: B04.sampler.pd.

2.6.4 looping samplers

In most situations, you'll want a more automated way than moving the mouse to specify wavetable read locations; for instance, you might want to be able to play a sample at a steady transposition; you might have several samples playing back at once (or other things requiring attention), you might want to switch quickly between samples or go to prearranged locations. In the next few examples we'll develop an automated looping sample reader, which, although only one of many possible approaches, is a powerful and often-used one.

Patches B05.sampler.loop.pd and B06.sampler.loop.smooth.pd show how to do this; B05.sampler.loop.pd in the simplest possible way and B06.sampler.loop.smooth.pd (pictured in Figure 2.14 part (a)) incorporating a second waveshape to envelope the sound as described in section 2.3. One new object class is introduced here:

cos ~ : Takes the cosine of 2π times the input signal (so that 0 to 1 makes a whole cycle.) Unlike the table reading classes in Pd, cos ~ handles wraparound so that there is no range limitation on its input.

In Figure 2.14 part (a), a phasor ~ supplies both indices into the wavetable (at right) and phases for a half-cosine-shaped envelope function at left. These two are multiplied, and the product is high-pass filtered and output. Reading the wavetable is straightforward; the phasor is multiplied by a "chunk size" parameter, added to 1, and used as an index to tabread4 ~. The chunk size parameter is multiplied by 441 to convert it from hundredths of a second to samples. This corresponds exactly to the block diagram shown in Figure 2.5, with a segment location of 1. (The segment location can't be 0 because 1 is the minimum index for which tabread4 ~ works.)

The left-hand signal path in 2.14 part (a) corresponds to the enveloping wavetable lookup shown in Figure 2.7. Here the sawtooth wave is adjusted to the range (-1/4, 1/4) (by multiplying by 0.5 and then subtracting 0.25), and then sent to cos ~. This reads the cosine function in the range $(-\pi/2, \pi/2)$, thus giving only the positive half of the waveform.

Part (b) of Figure 2.14 introduces a third parameter, the "read point", which specifies where in the sample the loop is to start. (In part (a) we always started at the beginning.) The necessary change is simple enough: simply add the "read point" control value, in samples, to the wavetable index and proceed as before. To avoid discontinuities in the index we smooth the read point value using pack and line ~ objects, just as we did in the first sampler example (Figure 2.14).

This introduces an important, though subtle, detail. The MOMENTARY TRANSPOSITION formula (section 2.2) predicts that, as long as the chunk size and read point aren't changing in time, the transposition is just the frequency times the chunk size (as always, using appropriate units; Hz. and seconds, for example, so that the product is dimensionless.) However, varying the chunk size and read point in time will affect the momentary transposition, often in very noticeable ways, as can be seen in patch B07.sampler.scratch.pd. Patch B08.sampler.nodoppler.pd (the one shown in the figure), shows one possible way of controlling this effect, while introducing a new object class:

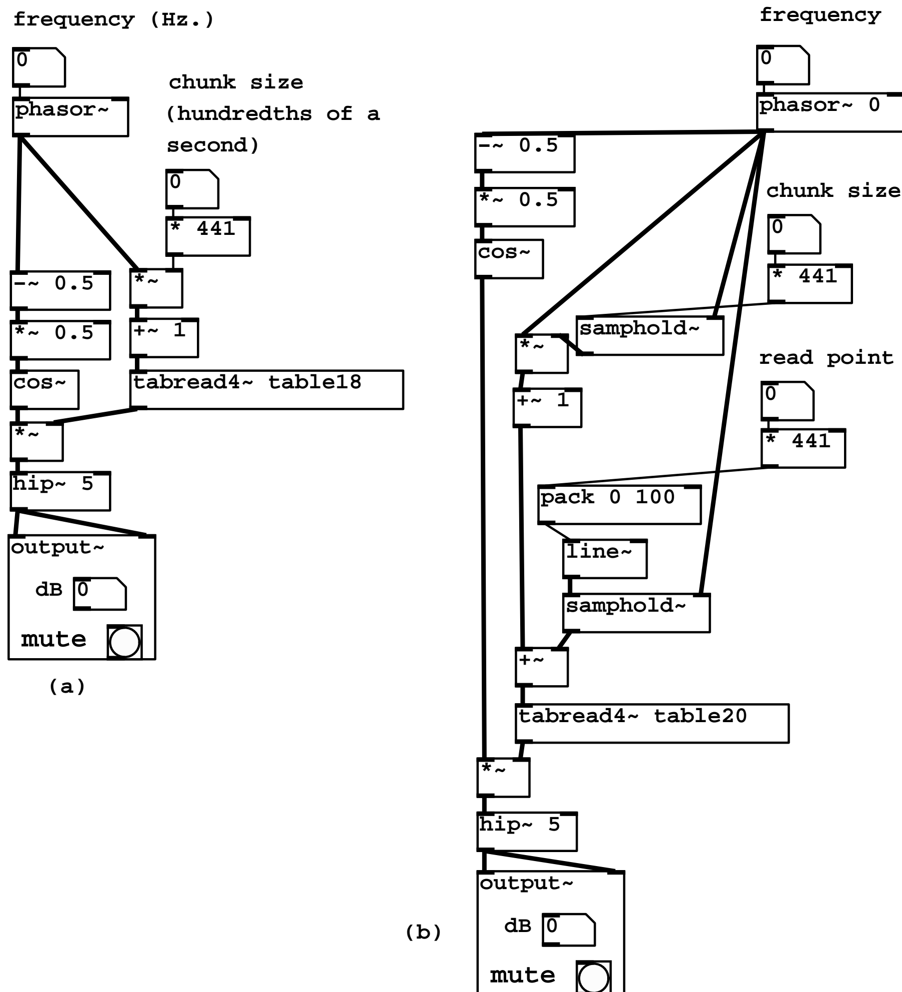


Figure 2.15: (a) a looping sampler with a synchronized envelope (B06.sampler.loop.smooth.pd); (b) the same, but with a control for read location (B08.sampler.nodoppler.pd).

samphold ~ : a sample and hold unit. (This will be familiar to analog synthesizer users, but with a digital twist.) This stores a single sample of the left-hand-side input and outputs it repeatedly, until caused by the right-hand-side input (also a digital audio signal, called the *trigger*) to overwrite the stored sample with a new one—again from the left-hand-side input. The unit acquires a new sample whenever the trigger's numerical value falls from one sample to the next. This is designed to be easy to pair with phasor ~ objects, to facilitate triggering on phase wraparounds.

Patch B08.sampler.nodoppler.pd uses two samphold ~ objects to update the values of the chunk size and read point, exactly when the phasor ~ wraps around, at which moments the cosine envelope is at zero so the effect of the instantaneous changes can't be heard. In this situation we can apply the simpler TRANSPOSITION FORMULA for looping wavetables to relate frequency, chunk size, and transposition. This is shown in patch B09.sampler.transpose.pd (not shown.)

2.6.5 overlapping sample looper

As described in section 2.3, it is sometimes desirable to use two or more overlapping looping samplers to produce a reasonably continuous sound without having to envelope too sharply at the ends of the loop. This is especially likely in situations where the chunk that is looped is short, a tenth of a second or less. Patch B10.sampler.overlap.pd, shown in Figure 2.16 (part a), realizes two looping samplers a half-cycle out of phase from each other. New object classes are:

loadbang : output a "bang" message on load. This is used in this patch to make sure the division of transposition by chunk size will have a valid transposition factor in case "chunk size" is moused on first.

expr : evaluate arithmetic expressions. Variables appear as $f1, $f2, and so on, corresponding to the object's inlets. Arithmetic expressions are allowed, with parentheses for grouping, and many library functions are supplied, such as exponentiation, which shows up in this example as "pow" (the power function.)

wrap ~ : wrap to the interval from 0 to 1. So, for instance, 1.2 becomes 0.2; 0.5 remains unchanged; and -0.6 goes to 0.4.

send ~ , s ~ : receive ~ , r ~ : signal versions of send and receive. An audio signal sent to a send ~ appears at the outlets of any and all receive ~ objects of the same name. Unlike send and receive, you may not have more than one send ~ object with the same name (in that connection, see the throw ~ and catch ~ objects).

In the example, part of the wavetable reading machinery is duplicated, using identical calculations of `chunk-size-samples` (a message stream) and `read-pt` (an audio signal smoothed as before). However, the `phase` audio signal, in the other copy, is replaced by `phase2`. The top part of the figure shows the calculation of the two phase signals: the first one as the output of a phasor ~

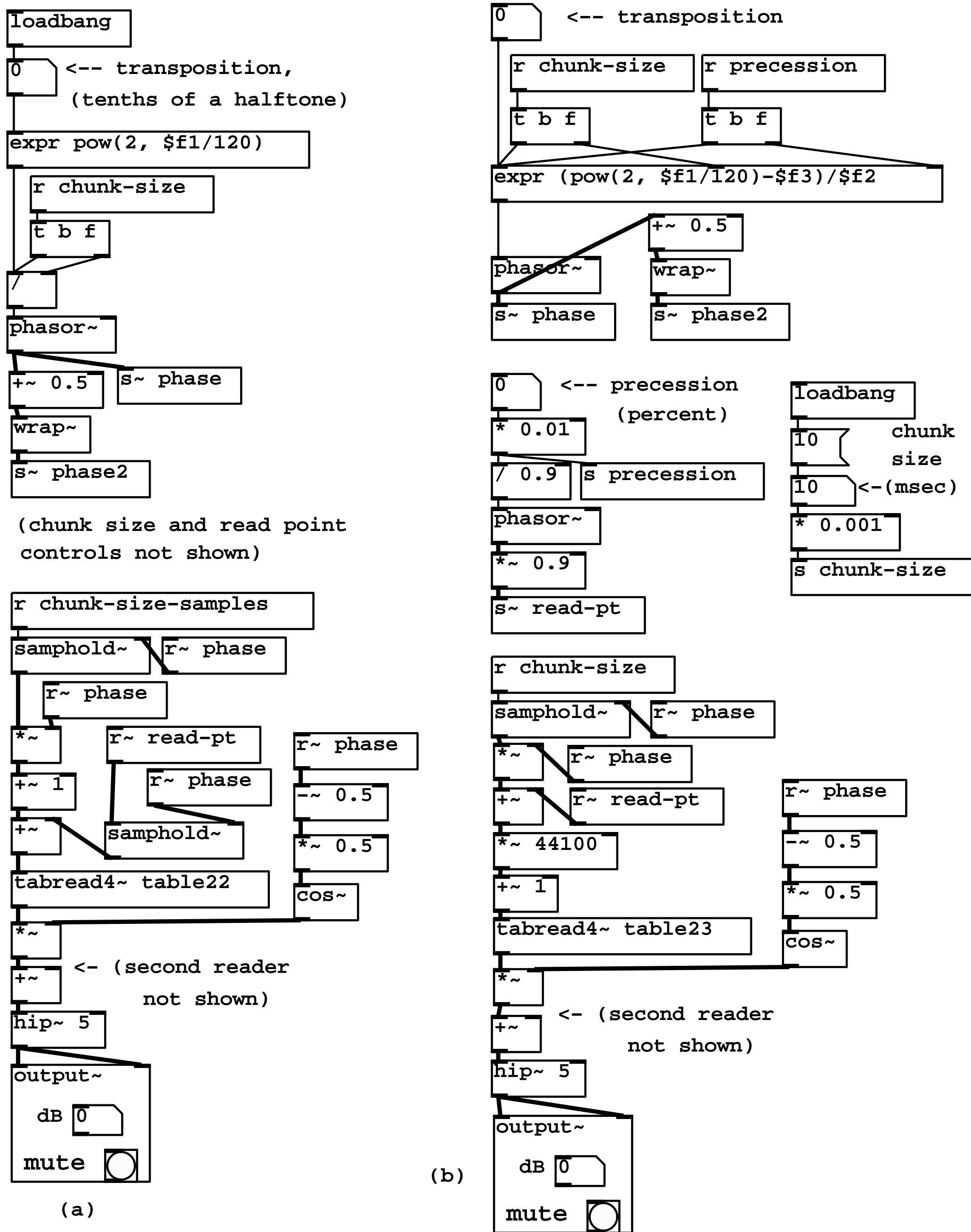


Figure 2.16: (a) two overlapped looping samplers (B10.sampler.overlap.pd); (b) the same, but with a phasor-controlled read point (B11.sampler.rockafella.pd).

object, and the second by adding 0.5 and wrapping, thereby subtracting 0.5 cycles (π radians) from the phase. The two phase signals are each used, with the same range adjustments as before, to calculate indices into the wavetable and the cos ~ object, and to control the two samphold ~ objects. Finally, the outputs of the two copies are added for output.

2.6.6 automatic read point precession

Patch B11.sampler.rockafella.pd, shown in part (b) of Figure 2.16, adapts the ideas shown above to a situation where the read point is computed automatically. Here we precess the read-point through the sample in a loop, permitting us to speed up or slow down the playback independently of the transposition.

This example addresses a weakness of the preceding one, which is that, if the relative precession speed is anywhere near one (i.e., the natural speed of listening to the recorded wavetable), and if there is not much transposition either, it becomes preferable to use larger grains and lower the frequency of repetition accordingly (keeping the product constant to achieve the desired transposition.) However, if the grain size is allowed to get large, it is no longer convenient to quantize control changes at phase wrappings, because they might be too far apart to allow for a reasonable response time to control changes.

In this patch we remove the samphold ~ object that had controlled the read point (but we leave in the one for chunk size which is much harder to change in mid-loop.) Instead, we use the (known) rate of precession of the read point to correct the sawtooth frequency, so that we maintain the desired transposition. It turns out that, when transposition factor and precession are close to each other (so that we are nearly doing the same thing as simple speed change) the frequency will drop to a value close to zero, so we will have increased the naturalness of the result at the same time.

In this patch we switch from managing read points, chunk sizes, etc., in samples and use seconds instead, converting to samples (and shifting by one) only just before the tabread4 ~ object. The wavetable holds one second of sound, and we'll assume here that the nominal chunk size will not exceed 0.1 second, so that we can safely let the read point range from 0 to 0.9; the "real" chunk size will vary, and can become quite large, because of the moving read pointer.

So the precession control sets the frequency of a phasor of amplitude 0.9, and therefore the precession must be multiplied by 0.9 to set the frequency of the phasor (so that, for a precession of one for instance, the amplitude and frequency of the read point are both 0.9, so that the slope, equal to amplitude over frequency, is one.) The output of this is named `read-pt` as before, and is used by both copies of the wavetable reader.

The precession p and the chunk size c being known, and if we denote the frequency of the upper (original) phasor ~ by f, the transposition factor is given by:

$$t = p + cf,$$

and solving for f gives:

$$f = \frac{t-p}{c} = \frac{2^{h/12} - p}{c},$$

where h is the desired transposition in half steps. This is the formula used in the `expr` object.

Exercises

1. A sinusoid is stored in a wavetable with period 4 so that the first four elements are 0, 1, 0, and -1, corresponding to indices 0, 1, 2, and 3. What value do we get for an input of 1.5: (a) using 2-point interpolation? (b) using 4-point interpolation? (c) what's the value of the original sinusoid there?

Chapter 3

Audio and control computations

3.1 The sampling theorem

We have heretofore discussed digital audio signals as if they were capable of describing any function of time, in the sense that knowing the values the function takes on the integers should somehow determine the values it takes between them. This isn't really true. For instance, suppose some function f (defined for real numbers) happens to attain the value 1 at all integers: $f(n) = 1$ for $n = \cdots, -1, 0, 1, \cdots$. We might guess that $f(t) = 1$ for all real t. But perhaps f happens to be one for integers and zero everywhere else—that's a perfectly good function too, and nothing about the function's values at the integers distinguishes it from the simpler $f(t) = 1$. But intuition tells us that the constant function is in the *spirit* of digital audio signals, whereas the one that hides a secret between the samples isn't. A function that is "possible to sample" should be one for which we can use some reasonable interpolation scheme to deduce its values for non-integers from its values for integers.

It is customary at this point in discussions of computer music to invoke the famous *Nyquist theorem.* This states (roughly speaking) that if a function is a finite or even infinite combination of REAL SINUSOIDS, none of whose angular frequencies exceeds π, then, theoretically at least, it is fully determined by the function's values on the integers. One possible way of reconstructing the function would be as a limit of higher- and higher-order polynomial interpolation.

The angular frequency π, called the *Nyquist frequency*, corresponds to $R/2$ cycles per second if R is the sample rate. The corresponding period is two samples. The Nyquist frequency is the best we can do in the sense that any real sinusoid of higher frequency is equal, at the integers, to one whose frequency is lower than the Nyquist, and it is this lower frequency that will get reconstructed by the ideal interpolation process. For instance, a REAL SINUSOID

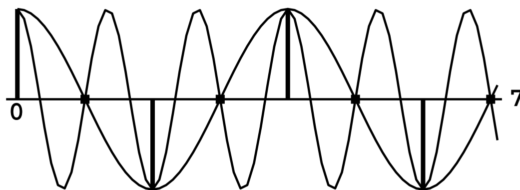


Figure 3.1: Two real sinusoids, with angular frequencies $\pi/2$ and $3\pi/2$, showing that they coincide at integers. A digital audio signal can't distinguish between the two.

with angular frequency between π and 2π, say $\pi + \omega$, can be written as

$$cos((\pi + \omega)n + \phi) = cos((\pi + \omega)n + \phi - 2\pi n)$$

$$= cos((\omega - \pi)n + \phi$$

$$= cos((\pi - \omega)n - \phi),$$

for all integers n. (If n weren't an integer the first step would fail.) So a sinusoid with frequency between π and 2π was equal, on the integers at least, to one with frequency between 0 and π; you simply can't tell the two apart. And since any conversion hardware will do the "right" thing and reconstruct the lower-frequency sinusoid, any higher-frequency one you try to synthesize will come out your speakers at the wrong frequency—specifically, you will hear the unique frequency between 0 and π that the higher frequency lands on when reduced in the above way. This phenomenon is called *foldover*, because the half-line of frequencies from 0 to inf is folded back and forth, in lengths of π, onto the interval from 0 to π. The word *aliasing* means the same thing. Figure 3.1 shows sinusoids of angular frequencies $\pi/2$ and $3\pi/2$; the higher frequency folds over to the lower one.

We conclude that when, for instance, we're computing an EXPLICIT SUM OF SINUSOIDS, either as a wavetable or as a real-time signal, we had better drop any sinusoid in the sum whose frequency exceeds π. But the picture in general is not this simple, since most techniques other than additive synthesis don't lead to neat, band-limited signals (ones whose components stop at some limited frequency.) For example, a sawtooth wave of frequency ω, of the form put out by Pd's phasor ~ object but considered as a continuous function $f(t)$, expands to:

$$f(t) = \frac{1}{2} - \frac{1}{\pi}\left(sin(\omega t) + \frac{sin(2\omega t)}{2} + \frac{sin(3\omega t)}{3} + \cdots\right),$$

which enjoys arbitrarily high frequencies; and moreover the hundredth partial is only 40 dB below the first one in level. At any but very low values of ω, the partials above π will be audibly present—and, because of foldover, they will

be heard at incorrect frequencies. (This does not mean that one shouldn't use sawtooth waves as phase generators—the wavetable lookup step magically fixes the foldover problem—but one should think twice before using a sawtooth wave itself as a digital sound source.)

Many synthesis techniques, even if not strictly band-limited, give partials which may be made to drop off more rapidly than $1/n$ as in the sawtooth example, and are thus more forgiving to work with digitally. In any case, it is always a good idea to keep the possibility of foldover in mind, and to train your ears to recognize it.

The first line of defense against foldover is simply to use high sample rates; it is a good practice to systematically use the highest sample rate that your computer can easily handle. The highest practical rate will vary according to whether you are working in real time or not, CPU time and memory constraints, and/or input and output hardware, and sometimes even software-imposed limitations.

A very non-technical treatment of sampling theory is given in [Bal03]. More detail can be found in [Mat69, pp. 1-30].

3.2 Control

So far we have dealt with audio signals, which are just sequences $x[n]$ defined for integers n, which correspond to regularly spaced points in time. This is often an adequate framework for describing synthesis techniques, but real electronic music applications usually also entail other computations which have to be made at irregular points in time. In this section we'll develop a framework for describing what we will call *control* computations. We will always require that any computation correspond to a specific *logical time.* The logical time controls which sample of audio output will be the first to reflect the result of the computation.

In a non-real-time system such as Csound, this means that logical time proceeds from zero to the length of the output soundfile. Each "score card" has an associated logical time (the time in the score), and is acted upon once the audio computation has reached that time. So audio and control calculations (grinding out the samples and handling note cards) are each handled in turn, all in increasing order of logical time.

In a real-time system, logical time, which still corresponds to the time of the next affected sample of audio output, is always slightly in advance of *real time*, which is measured by the sample that is actually leaving the computer. Control and audio computations still are carried out in alternation, sorted by logical time.

The reason for using logical time and not real time in computer music computations is to keep the calculations independent of the actual execution time of the computer, which can vary for a variety of reasons, even for two seemingly identical calculations. When we are calculating a new value of an audio signal or processing some control input, real time may pass but we require that the

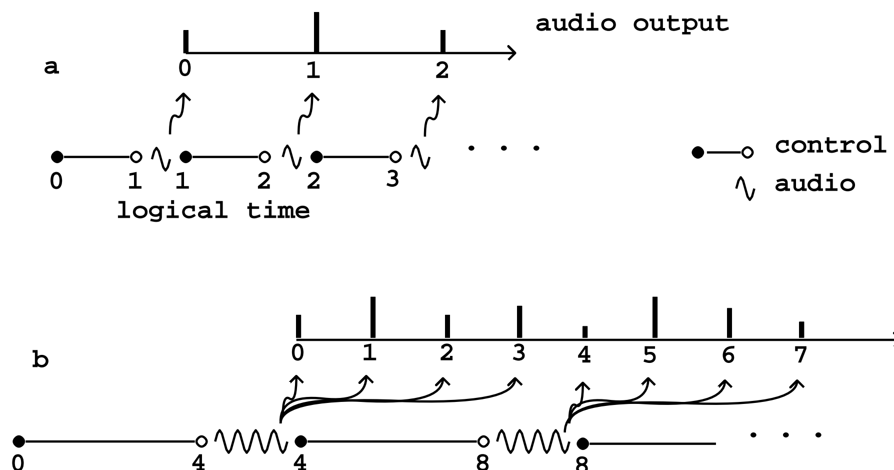


Figure 3.2: Timeline for digital audio and control computation. In (a) the block size is one sample. We compute the audio at a delay of one sample after the control computation; so sample 0 (at top of (a)) is computed at logical time 1 (logical times are shown on the number line below). In (b), the block size is four samples, and hence so is the delay for computing audio after control. Control affecting samples 0 through 3 is all computed at once, followed by the corresponding audio computation.

logical time stay the same through the whole calculation, as if it took place instantaneously. As a result of this, electronic music computations, if done correctly, are deterministic: two runs of the same real-time or non-real-time audio computation, each having the same inputs, should have identical results.

Figure 3.2 part (a) shows schematically how logical time and sample computation are lined up. Audio samples are computed periodically (as shown with wavy lines), but before the calculation of each sample we do all the control calculations (marked as straight line segments). First we do the control computations associated with logical times starting at zero, up to but not including one; then we compute the first audio sample (of index zero), at logical time one. We then do all control calculations up to but not including logical time 2, then the sample of index one, and so on. (Here we are adopting certain conventions about labeling that could be chosen differently. For instance, there is no fundamental reason control should be pictured as coming "before" audio computation but it is easier to think that way.)

Part (b) of the figure shows the situation if we wish to compute the audio output in blocks of more than one sample at a time. Using the variable B to denote the number of elements in a block (so $B = 4$ in the figure), the first audio computation will output samples $0, 1, ...B - 1$ all at once in a block. We have to do the relevant control computations for all four periods of time in advance.

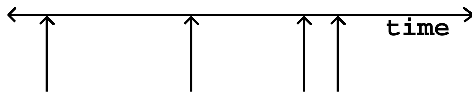


Figure 3.3: Graphical representation of a control stream as a sequence of points in time.

There is a delay of B samples between logical time and the appearance of audio output.

Audio is computed in blocks in Max/MSP, Csound and Pd, and sample by sample in SuperCollider. The reason for computing by blocks is to increase the efficiency of unit generators (Csound) or tilde objects (Pd; Max/MSP). Each tilde object incurs overhead each time it is called, equal to perhaps twenty times the cost of computing one sample on average. If the block size is one, this means an overhead of 2,000%; if it is sixty-four (as in Pd), the overhead is only some 30%.

SuperCollider overcomes this problem by compiling signal processing algorithms into machine language, thus avoiding the efficiency problem of the other environments described here. The main drawback to this scheme is the additional software development effort required, and the greater difficulty of maintaining the program. This is also perhaps the main reason SuperCollider only runs on one machine architecture; to adapt it to others will require re-writing the machine code generator.

3.3 Control streams

Control computations may come from a variety of sources, both internal and external to the overall computation. Examples of internally engendered control computations include sequencing (in which control computations must take place at pre-determined times) or feature detection of the audio output (for instance, watching for zero crossings in a signal). Externally engendered ones may come from input devices such as MIDI controllers, the mouse and keyboard, ethernet packets, and so on. In any case, they may occur at irregular intervals, unlike audio samples which correspond to a steadily ticking sample clock.

We will need a way of describing how information flows between control and audio computations, which we will base on the notion of a *control stream.* This is simply a collection of numbers—possibly empty—that appear as a result of control computations, whether regularly or irregularly spaced in logical time. The simplest possible control stream has no information other than a *time sequence*:

$$\cdots, t[0], t[1], t[2], \cdots,$$

in units of audio samples. We require to be sorted in nondecreasing order:

$$\cdots \leq t[0] \leq t[1] \leq t[2] \leq \cdots .$$

We don't insist, though, that the $t[n]$ be integers. Each item in the list is called an *event*.

Control streams may be shown graphically in Figure 3.3. A number line shows time and a sequence of arrows point to the times associated with each event. The control stream shown has no data (it is a time sequence). If we want to show data in the control stream we will write it at the base of each arrow.

A *numeric control stream.* is one that contains one number per time point, so that it appears as a sequence of ordered pairs:

$$\cdots, (t[0], x[0]), (t[1], x[1]), \cdots,$$

where the $t[n]$ are the time points and the $x[n]$ are the signal's values at those times.

A numeric control stream is roughly analogous to a "MIDI controller", whose values change irregularly, for example when a physical control is moved by a performer. Other control stream sources may have higher possible rates of change and/or more precision. On the other hand, a time sequence might be a sequence of pedal hits, which (MIDI implementation notwithstanding) shouldn't be considered as having *values*, just *times*.

Numeric control streams are like audio signals in that both are just time-varying numeric values. But whereas the audio signal comes at a steady rate (and so the time values need not be specified per sample), the control stream comes unpredictably—perhaps evenly, perhaps unevenly, perhaps never.

Let us now look at what happens when we try to convert a numeric control stream to an audio signal. As before we'll choose a block size $B = 4$. We will consider as a control stream a square wave of period 5.5:

$$(2, 1), (4.75, 0), (7.5, 1), (10.25, 0), (13, 1), \cdots$$

and demonstrate three ways it could be converted to an audio signal. Figure 3.4, part (a), shows the simplest, fast-as-possible, conversion. Each audio sample of output simply reflects the most recent value of the control signal. So samples 0 through 3 (which are computed at logical time 4 because of the block size) are 1 in value because of the point (2, 1). The next four samples are also one, because of the two points, (4.75, 0) and (7.5, 1), the most recent still has the value 1.

Fast-as-possible conversion is most appropriate for control streams which do not change frequently compared to the block size. Its main advantages are simplicity of computation and the fastest possible response to changes. As the figure shows, when the control stream's updates are too fast (on the order of the block size), the audio signal may not be a good likeness of the sporadic one. (If, as in this case, the control stream comes at regular intervals of time, we can use the sampling theorem to analyze the result. Here we have resampled a

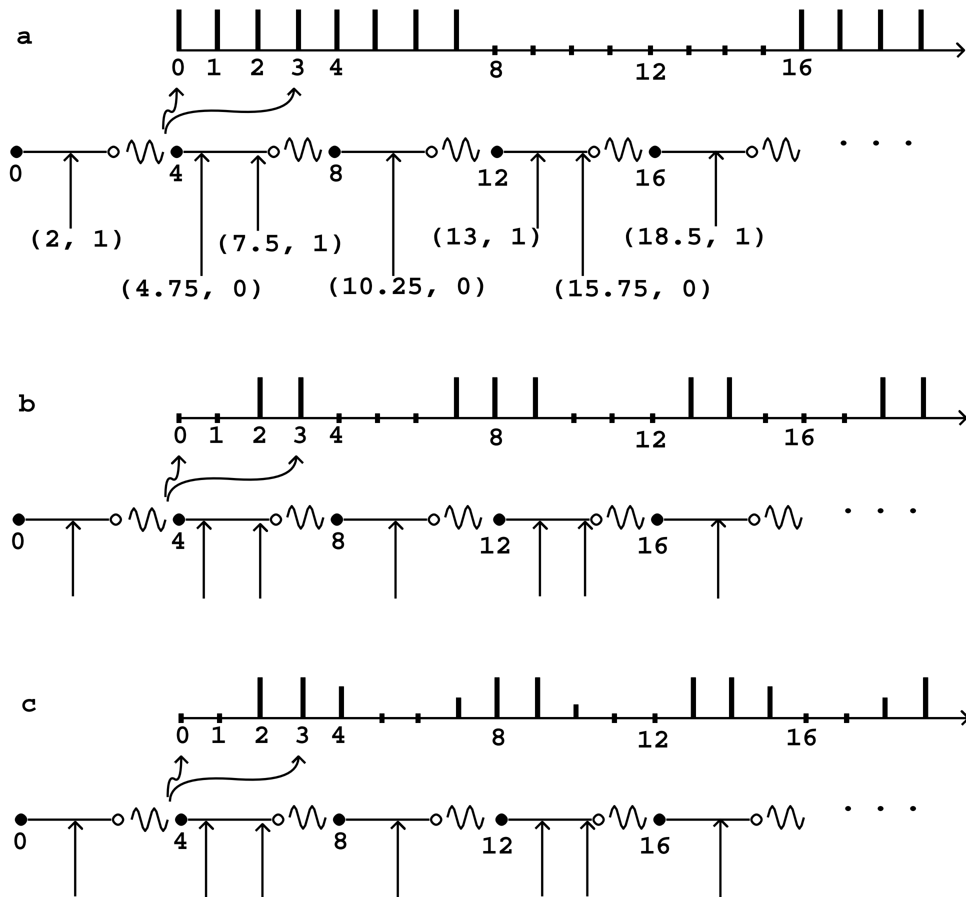


Figure 3.4: Three ways to change a control stream into an audio signal. A: as fast as possible; B: delayed to the nearest sample; C: with two-point interpolation for higher delay accuracy.

square wave to a sample rate (one sample per block) which is below its Nyquist frequency and so the output is aliased to a lower frequency than the input.)

Part (b) shows the result of nearest-sample conversion. Each new value of the control stream at a time t affects output samples starting from index $\lfloor t \rfloor$ (the greatest integer not exceeding t). This is equivalent to using fast-as-possible conversion at a block size of 1; in other words, nearest-sample conversion hides the effect of the larger block size. This is better than fast-as-possible conversion in cases where the control stream might change quickly.

Part (c) shows sporadic-to-audio conversion, again at the nearest sample, but now also using two-point interpolation to further increase the time accuracy. Conceptually we can describe this as follows. Suppose the value of the control stream was last equal to x, and that the next point is $(n + f, y)$, where n is an integer and f is the fractional part of the time value (so $0 \le f < 1$). The first point affected in the audio output will be the sample at index n. But instead of setting the output to y as before, we set it to

$$fx + (1 - f)y,$$

in other words, to a weighted average of the previous and the new value, whose weights favor the new value more if the time of the sporadic value is earlier, closer to n. In the example shown, the transition from 0 to 1 at time 2 gives

$$0 \cdot x + 1 \cdot y = 1,$$

while the transition from 1 to 0 at time 4.75 gives:

$$0.75 \cdot x + 0.25 \cdot y = 0.75.$$

This technique gives a still closer representation of the control signal (at least, the portion of it that lies below the Nyquist frequency), at the expense of more computation and slightly greater delay.

Numeric control streams may also be converted to audio signals using ramp functions to smooth discontinuities. This is often used when a control stream is used to control an amplitude, as described in section 1.5. In general there are three values to specify to set a ramp function in motion: a start time and target value (specified by the control stream) and a target time, often expressed as a delay after the start time.

In such situations it is almost always accurate enough to adjust the start and ending times to match the first audio sample computed at a later logical time, a choice which corresponds to the fast-as-possible scenario above. Figure 3.5 part (a) shows the effect of ramping from 0, starting at time 3, to a value of 1 at time 9, immediately starting back toward 0 at time 15. The times 3, 9, and 15 are truncated to 0, 8, and 12, respectively.

In real situations, where the block size might be on the order of a millisecond, this is fine for amplitude controls, which, if they reach a target a fraction of a millisecond early or late won't make an audible difference in most cases. However, other uses of ramps are more sensitive to this kind of jitter. The

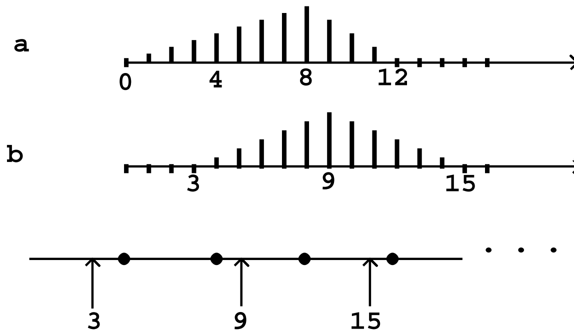


Figure 3.5: Line segment smoothing of numeric control streams: (a) aligned to block boundaries; (b) aligned to nearest sample.

most frequently encountered example of this is fast repetition driven by control computations. If we do something repetitively every few milliseconds, say. the differences in actual segment lengths will make for an audible aperiodicity.

For situations such as these, we can improve the ramp generation algorithm to start and stop at arbitrary samples, as shown in figure 3.5 part (b), for example. Here the endpoints of the line segments line up exactly with the requested samples 3, 9, and 15. We can go even further and adjust for fractional samples, making the line segments touch the values 0 and 1 at exactly specifiable points on a number line.

For example, suppose we want to repeat a recorded sound out of a wavetable 100 times per second, every 441 samples at the usual sample rate. Rounding errors due to blocking at 64-sample boundaries could detune the playback by as much as a whole tone in pitch; and even rounding to one-sample boundaries would introduce variations up to about 0.2%, or three cents. This situation would call for sub-sample accuracy in sporadic-to-audio conversion.

3.4 Converting from audio signals to numeric control streams

We sometimes need to convert in the other direction, from an audio signal to a sporadic one. To go in this direction, we somehow provide a series of logical times (a time sequence), as well as an audio signal. For output we want a control stream combining the time sequence with values taken from the audio signal. We do this when we want to incorporate the signal's value as part of a control computation.

For example, we might be controlling the amplitude of a signal using a line~ object as in Chapter 1, Example 3 (page 19). Suppose we wish to turn off the sound at a fixed rate of speed instead of in a fixed amount of time. For instance,

we might want to re-use the network for another sound and wish to mute it as quickly as possible without audible artifacts; we probably can ramp it off in less time if the current amplitude is low than if it is high. To do this we must confect a message to the line ~ object to send it to zero in an amount of time we'll calculate on the basis of its current output value. This will require, first of all, that we "sample" the line ~ object's output (an audio signal) into a control stream.

The same issues of time delay and accuracy appear as for sporadic to audio conversion. Again there will be a tradeoff between immediacy and accuracy.

Suppose as before that we are calculating audio in blocks of 4 samples, and suppose that at logical time 6 we want to look at the value of an audio signal, and use it to change the value of another one. As shown in figure 3.2 part (b), the most recently calculated value of the signal will be for index 3 and the earliest index at which our calculation can affect a signal is 4. We can therefore carry out the whole affair with a delay of only one sample. However, we can't choose exactly *which* sample—we only get the chance once out of every four of them.

As before, we can trade immediacy for increased time accuracy. If it matters exactly at which sample we carry out the audio-to-control-to-audio computation, we read the sample of index 2 and update the one at index 6. Then if we want to do the same thing again at logical time 7, we read from index 3 and update at index 7, and so on. In general, if the block size is B, and for any index n, we can always read the sample at index $n - B$ and affect the one at index n. There is thus a round-trip delay of B samples in going from audio to control to audio computation, which is the price incurred for being able to name the index n exactly.

If we wish to go further, to being able to specify a fraction of a sample, then (as before) we can use interpolation—at a slight further increase in delay. In general, as in the case of sporadic-to-audio conversion, in most cases the simplest solution is the best, but in a few cases we have to do extra work.

3.5 Control streams in block diagrams

Figure 3.6 shows how control streams are expressed in block diagrams, using control-to-signal and signal-to-control conversion as examples. Control streams are represented using dots (as opposed to audio signals which appear as solid arrows).

The *signal* operator converts from a numeric control stream to an audio signal. The exact type of conversion isn't be specified at this level of detail; when we see specific examples in Pd the choice of conversion operator will determine this.

The *snapshot* operator converts from audio signals back to numeric control streams. In addition to the audio signal, a separate, control input is needed to specify the time sequence at which the audio signal is sampled.

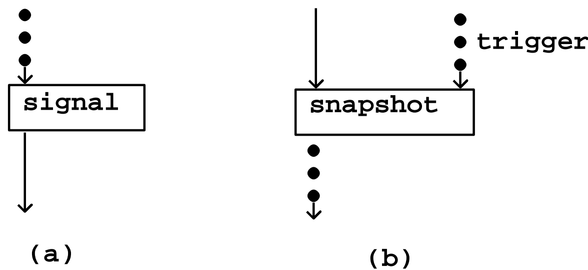


Figure 3.6: Conversion between control and audio: (a) control to signal; (b) signal to control by snapshots.

3.6 Event detection

Besides taking snapshots, a second mode of passing information from audio signals to control computations is *event detection*. Here we derive time information from the audio signal, for instance, the time at which a signal crosses a threshold. The input is an audio signal and the output will be a time sequence. Here we'll consider the example of threshold detection in some detail.

A defining situation in which we use threshold detection is to find out when some kind of activity starts and stops, such as a performer playing an instrument. We'll suppose we already have a continuous measure of activity in the form of an audio signal. (This can be done, for example, using an envelope follower). What we want is a pair of time sequences, one which marks times in which activity starts, and the other marking stops.

Figure 3.7 part (a) shows a simple realization of this idea. We assume the signal input is as shown in the continuous graph. A horizontal line shows the constant value of the threshold. The time sequence marked "onsets" contains one event for each time the signal crosses the threshold from below to above; the one marked "turnoffs" marks crossings in the other direction.

In many situations we will want to avoid getting multiple onsets and turnoffs caused by small ripples in the signal close to the threshold. This can be managed in at least two simple ways. First, as shown in part (b) of the figure, we can set two thresholds: a high one for marking onsets, and a lower one for turnoffs. In this scheme the rule is that we only report the first onset after each turnoff, and, vice versa, we only report one turnoff after each onset. Thus the third time the signal crosses the high threshold in the figure, there is no reported onset because there was no turnoff since the previous one. (At startup, we act as if the most recent output was a turnoff, so that the first onset is reported.)

A second approach to filtering out multiple onsets and turnoffs, shown in part (c) of the figure, is to associate a *dead period* to each onset. This is a constant interval of time after each reported onset, during which we refuse to report more

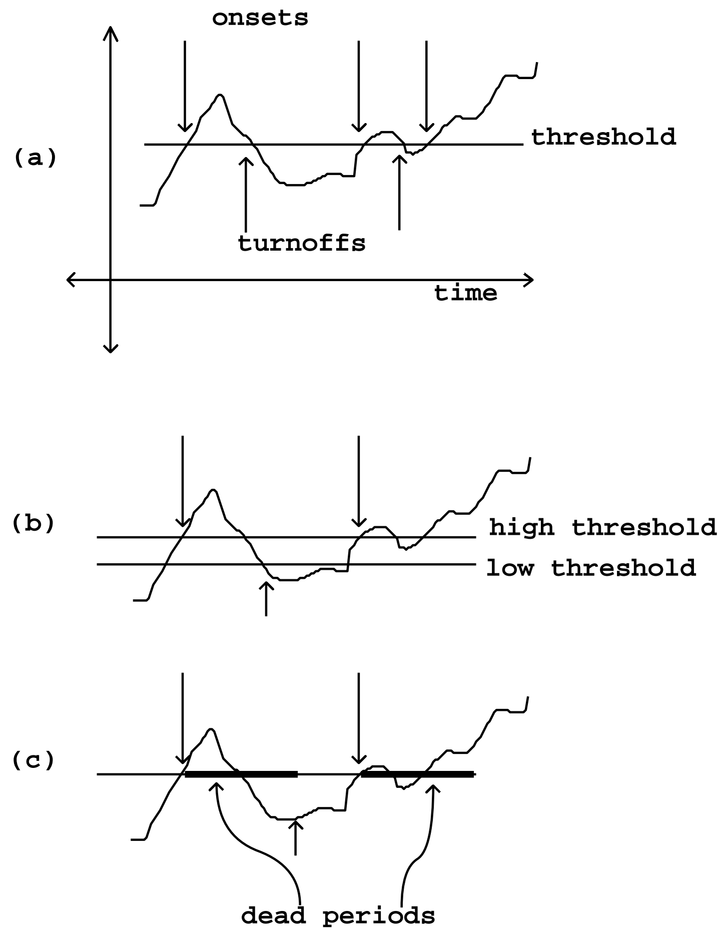


Figure 3.7: Threshold detection: (a) with no debouncing; (b) debounced using two threshold levels; (c) debounced using dead periods.

onsets or turnoffs. After the period ends, if the signal has dropped below the threshold in the meantime, we belatedly report a turnoff. Dead periods may also be associated with turnoffs, and the two time periods may have different values.

The two filtering strategies may be used separately or simultaneously. It is usually necessary to tailor the threshold values and/or dead times by hand to each specific situation in which thresholding is used.

Thresholding is often used as a first step in the design of higher-level strategies for arranging computer responses to audible cues from performers. A simple example could be to set off a sequence of pre-planned processes, each one to be set off by an onset of sound after a specified period of relative silence, such as you would see if a musician played a sequence of phrases separated by rests.

More sophisticated detectors (built on top of threshold detection) could detect continuous sound or silence within an expected range of durations, or sequences of quick alternation between playing and not playing, or periods of time in which the percentage of playing time to rests is above or below a threshold, or many other possible features. These could set off predetermined reactions or figure in an improvisation.

3.7 Control computation using audio signals directly

From the tradition of analog synthesis comes an elegant, old-fashioned approach to control problems which can be used as an alternative to the control streams we have been concerned with so far in this chapter. Instead, or in addition to using control streams, we can use audio signals themselves to control the production of other audio signals. Two specific techniques from analog synthesis lend themselves well to this treatment: analog sequencing and sample-and-hold.

The analog sequencer was often used to set off a regularly or semi-regularly repeating sequence of sounds. The sequencer itself typically put out a repeating sequence of voltages, along with a trigger signal which pulsed at each transition between voltages. One used the voltages for pitches or timbral parameters, and the trigger to control one or more envelope generators. Getting looped sequences of predetermined values in digital audio practice is as simple as sending a phasor ~ into a non-interpolating table lookup. If you want, say, four values in the sequence, scale the phasor ~ output to take values from 0 to $3.999\cdots$ so that the first fourth of the cycle reads point 0 of the table and so on.

To get repeated triggering, the first step is to synthesize another sawtooth that runs in synchrony with the phasor ~ output but four times as fast. This is done using a variant of the technique of Figure 2.8, in which we used an adder and a wraparound operator to get a desired phase shift. Figure 3.8 shows the effect of multiplying a sawtooth wave by an integer, then wrapping around to get a sawtooth at a multiple of the original frequency.

From there is is easy to get to a repeated envelope shape by wavetable

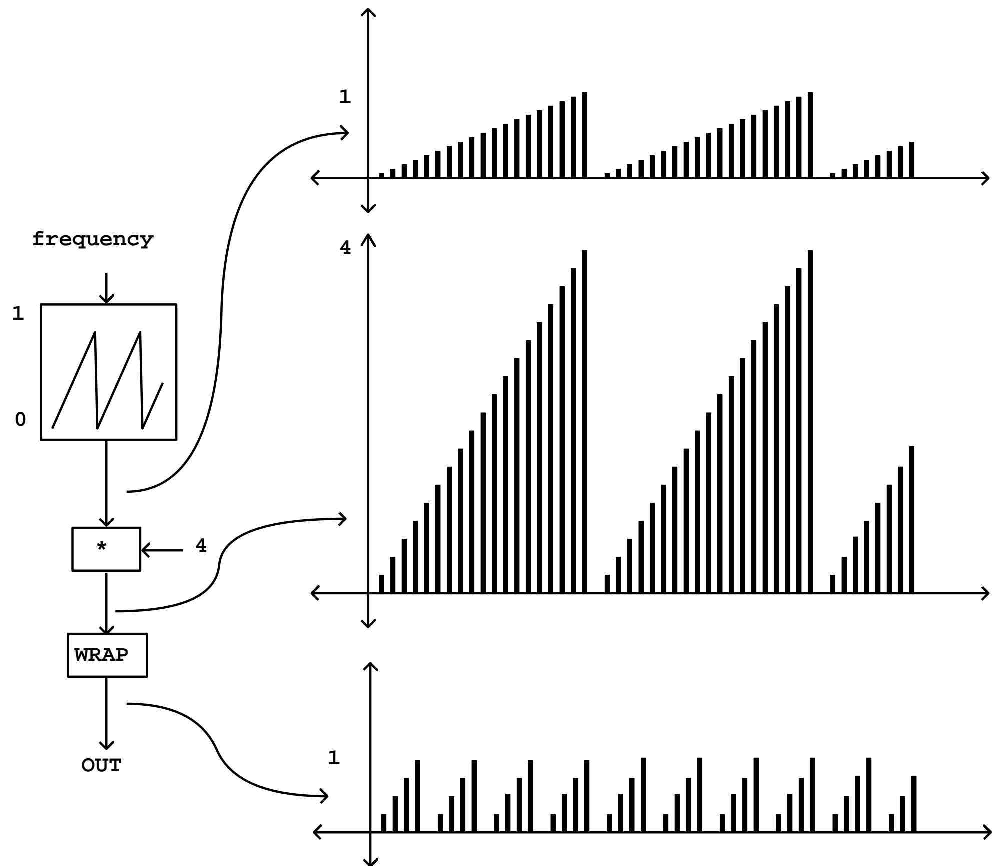


Figure 3.8: Multiplying and wrapping a sawtooth wave to generate a higher frequency.

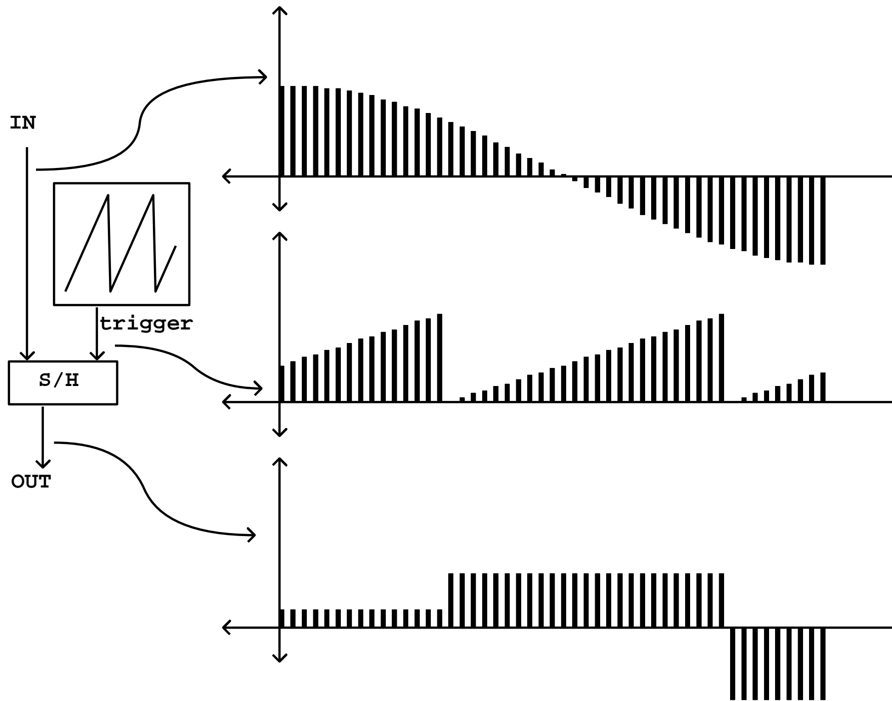


Figure 3.9: Sample and hold, using falling edges of the trigger signal.

lookup fro example (using an interpolating table lookup this time, unlike for the sequence voltages above). All the waveform generation and altering techniques used for making pitched sounds can also be brought to use here.

The other standard control technique from analog synthesizer control is the sample and hold unit. This takes an incoming signal, picks out certain instantaneous values from it, and "freezes" those values for its output. The particular values to pick out are selected by a secondary, "trigger" input. At points in time specified by the trigger input a new, single value is taken from the primary input and is output continuously until the next time point, when it is replaced by a new value of the primary input.

In digital audio it is often useful to sample a new value on falling edges of the trigger signal; for instance, we can sample whenever the current value of the trigger signal is smaller than its previous value. This is shown in Figure 3.9. Notice that this is especially well adapted for use with a sawtooth trigger, so that we can discretely sample signals in synchrony with any oscillator-driver process.

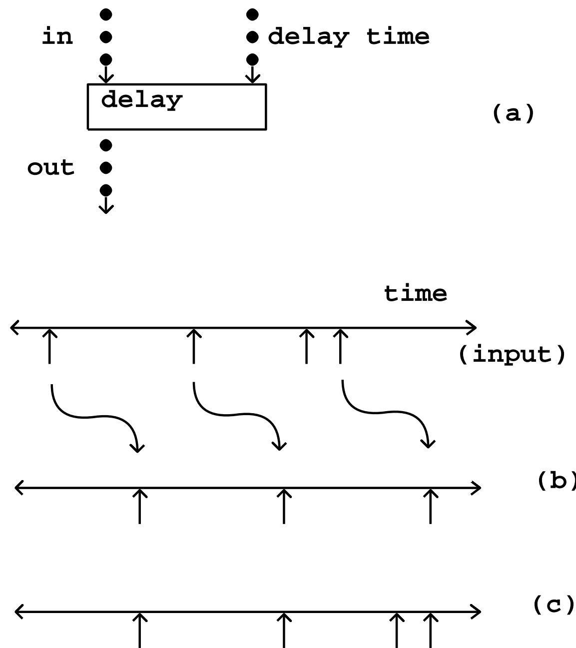


Figure 3.10: Delay as an operation on a control stream: (a) block diagram; (b) effect of a simple delay on a control stream; (c) effect of a compound delay.

3.8 Operations on control streams

So far we've discussed how to convert between control streams and audio streams. In addition to this possibility, there are four types of operations you can perform on control streams to get other control streams. These control stream operations have no corresponding operations on audio signals. Their existence explains in large part why it is useful to introduce a whole control structure in parallel with that of audio signals.

The first type consists of *delay* operations, which offset the time values associated with a control stream. In real-time systems the delays can't be negative in value. A control stream may be delayed by a constant amount, or alternatively, you can delay each event separately by different amounts.

Two different types of delay are used in practice: *simple* and *compound*. Examples of each are shown in Figure 3.10. A simple delay acting on a control stream schedules each event, as it comes in, for a time in the future. However, if another event arrives at the input before the first event is output, the first event is forgotten in favor of the second. In a compound delay, each event at the input produces an output, even if other inputs arrive before the output appears.

A second operation on control steams is *merging*. This is simply taking any

two control streams and combining all the events into a new one. Figure 3.10 (a) shows how this and the remaining operations can be shown in block diagrams, and part (b) of the figure shows the effect of merging two streams.

A subtlety arises when merging two streams: what if they contain simultaneous events? We'll say here that the output simply contains two events at the same time.

A third type of operation on control streams is *pruning*. Pruning a control stream means looking at the associated data and letting only certain elements through. Figure 3.10 (c) shows an example, in which events (which each have an associated number) are passed through only if the number is positive.

Finally, there is the concept of *resynchronizing* one control stream to another, as shown in part (d) of the figure. Here one control stream (the source) contributes values which are put onto the time sequence of a second one (the sync). The value given the output is always the most recent one from the source stream. Note that any event from the source may appear more than once (as suggested in the figure), or, on the other hand, it need not appear at all.

Again, we have to consider what happens when the two streams each contain an event at the same time. Should the sync even be considered as happening before the source (so that the output gets the value of the previous source event)? Or should the source event be considered as being first so that its value goes to the output at the same time? How this should be disambiguated is a design question, to which various software environments take various approaches.

3.9 Control operations in Pd

So far we have used Pd mostly for processing audio signals, although as early as Figure 1.6 we have had to make the distinction between Pd's notion of audio signals and of control streams: thin connections carry control streams and thick ones carry audio. Control streams in Pd appear as sequences of *messages*. The messages may contain data (most often, one or more numbers), or not. A *numeric control stream* (section 3.3) appears as a (thin) connection that carries numbers as messages.

Messages not containing data make up *time sequences*. So that you can see messages with no data, in Pd they are given the (arbitrary) symbol "bang".

The four types of control operations described in the previous section can be expressed in Pd as shown in Figure 3.12. *Delays* are accomplished using two explicit delay objects:

[del], [delay]: simple delay. You can specify the delay time in a creation argument or via the right inlet. A "bang" in the left inlet sets the delay, which then outputs "bang" after the specified delay in milliseconds. The delay is *simple* in the sense that sending a bang to an already set delay resets it to the new output time, canceling the previously scheduled one.

[pipe]: compound delay. Messages coming in the left inlet appear on the output after the specified delay, which is set by the first creation argument. If

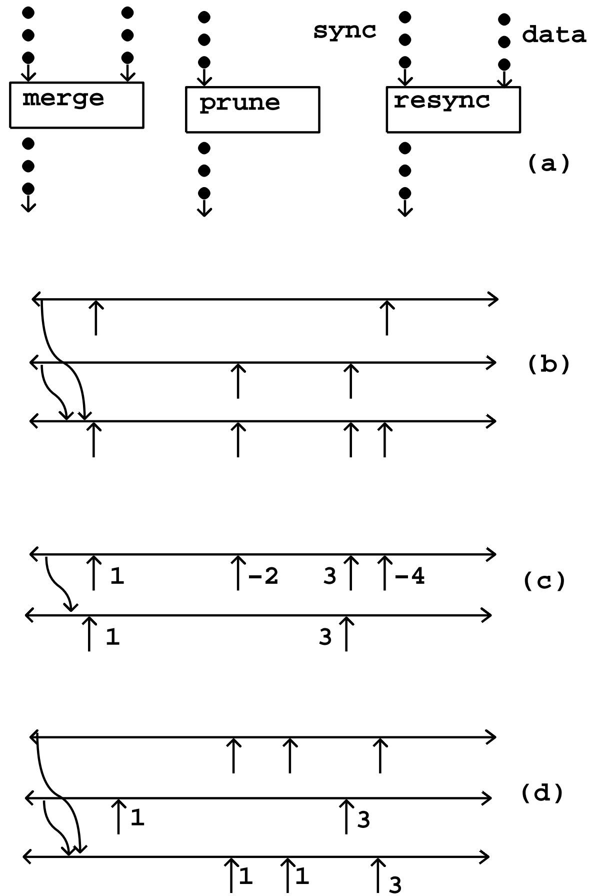


Figure 3.11: Operations on control streams (besides delay): (a) block diagrams; (b) merging; (c) pruning; (d) resynchronizing.

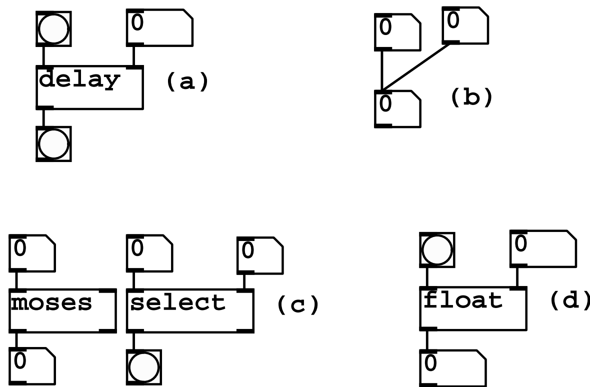


Figure 3.12: The four control operations in Pd: (a) delay; (b) merging; (c) pruning; (d) resynchronizing.

there are more creation arguments, they specify one or more inlets for numeric or symbolic data the messages will contain. Any number of messages may be stored by pipe simultaneously, and messages may be reordered as they are output depending on the various delay times given for them.

Merging of control streams in Pd is accomplished not by explicit objects but by Pd's connection mechanism itself. This is shown in part (b) of the figure.

Pd offers several objects for *pruning* control streams, of which two are shown in part (c) of the figure:

moses: prune for numeric range. Numeric messages coming in the left inlet appear on the left output if they are smaller than a threshold value (set by a creation argument or by the right inlet), and out the right inlet otherwise.

select, sel: prune for specific numbers. Numeric messages coming in the left inlet produce a "bang" on the output only if they match a test value exactly. The test value is set either by creation argument or from the right inlet.

Finally, as for the case of merging, Pd takes care of *resynchronizing* control streams implicitly in its connection mechanism, as illustrated by part (d) of the figure. Most objects with more than one inlet synchronize all other inlets to the leftmost one. So the float object shown in the figure resynchronizes its right-hand-side inlet (which takes numbers) to its left-hand-side one. Sending a "bang" to the left inlet outputs the most recent number float has received beforehand.

3.10 Examples

3.10.1 Sampling and foldover

Patch C01.nyquist.pd, shown in Figure 3.13 part (a), shows an oscillator playing a wavetable, sweeping through frequencies from 500 to 1423. The wavetable consists of only the 46th partial, which therefore varies from 23000 to 65458 Hz. At a sample rate of 44100 these two frequencies sound at 21100 and 22742 Hz, but sweeping from one to the other folds down through zero and back up.

Two other waveforms are provided to show the interesting effects of beating between partials which, although they "should" have been far apart, find themselves neighbors through foldover. For instance, at 1423 Hz, the second harmonic is 2846 Hz whereas the 33rd harmonic sounds at 1423*33-44100 = 2859 Hz—a pronounced dissonance.

Other less extreme examples can still produce audible foldover in less striking forms. Usually it is still objectionable and it is worth training ones ears to detect it. Patch C02.sawtooth-foldover.pd (not pictured here) demonstrates this for a sawtooth (the phasor ~ object). For wavetables holding audio recordings, interpolation error can create extra foldover. The effects of this can vary widely; the sound is sometimes described as "crunchy" or "splattering", depending on the recording, the transposition, and the interpolation algorithm.

3.10.2 Converting controls to signals

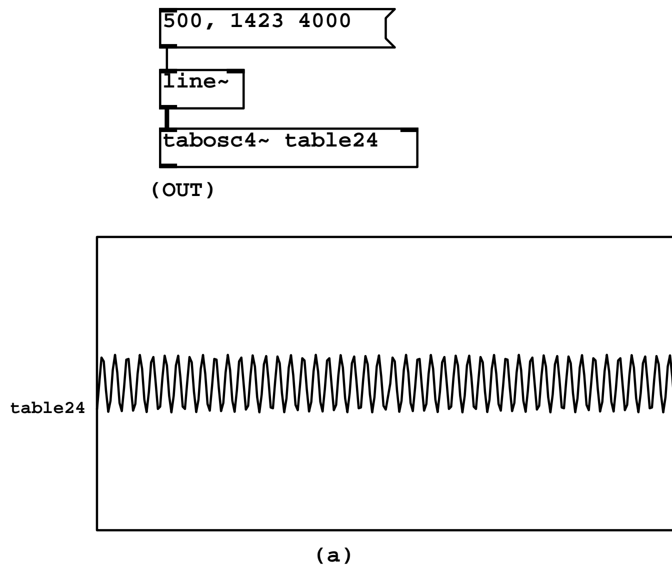
Patch C03.zipper.noise.pd (Figure 3.13 part b) demonstrates the effect of converting a slowly-updated control stream to an audio signal. This introduces a new object:

line : a ramp generator with control output. Like line ~, line takes pairs of numbers as (target, time) pairs and ramps to the target in the given amount of time; however, unlike line ~, the output is a numeric control stream, appearing, by default, at 20 msec time intervals.

In the example you can compare the sound of the rising and falling amplitude controlled by the line output with one controlled by the audio signal generated by line ~.

The output of line is converted to an audio signal at the input of the * ~ object. The conversion is implied here by connecting a numeric control stream into a signal inlet. In Pd, implicit conversions from numeric control streams to audio streams is done in the fast-as-possible mode shown in Figure 3.4 part (a). The line output becomes a staircase signal with 50 steps per second. The result is commonly called "zipper noise".

Whereas we were able to demonstrate the limitations of the line object for generating audio signals were clear even at such long time periods as 300 msec, the signal variant, line ~, does not yield audible problems until the time periods involved become much shorter. Patch C04.control.to.signal.pd (Figure 3.13 part c) demonstrates the effect of using line ~ to generate a 250 Hz. triangle wave. Here the effects shown in Figure 3.5 come into play. Since line ~ always aligns



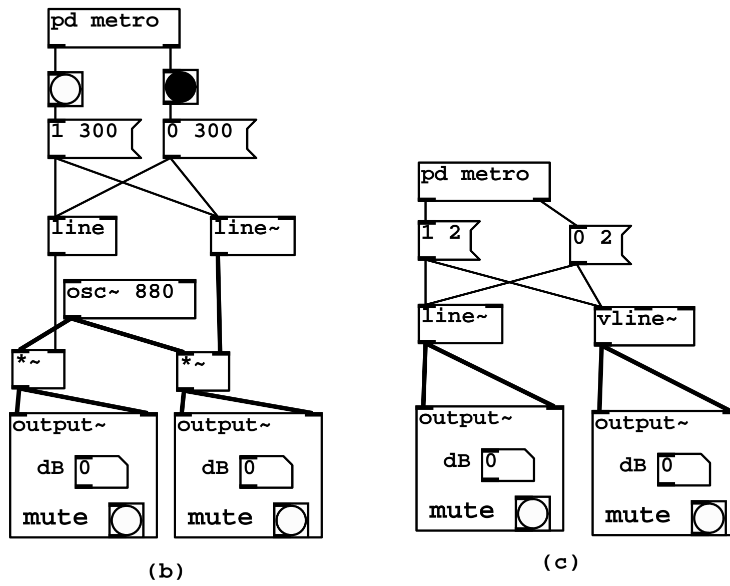


Figure 3.13: Sending an oscillator over the Nyquist frequency.

line segments to block boundaries, the exact durations of line segments vary, and in this case the variation (on the order of a millisecond) of the segments is a significant fraction of their length.

A more precise object (and a more expensive one, in terms of computation time) is provided for these situations:

vline ~ : exact line segment generator. This third member of the "line" family not only outputs an audio signal, but aligns the endpoints of the signal to the desired time points, accurate to a fraction of a sample. (The accuracy is limited only by the floating-point numerical format used by Pd.) Further, many line segments may be specified withing a single audio block; vline ~ can generate waveforms at periods down to two samples (beyond which you will just get foldover instead).

The vline ~ object can also be used for converting numeric control streams to audio streams in the nearest-sample and two-point-interpolation modes as shown in Figure 3.4 parts (b) and (c). To get nearest-sample conversion, simply give vline ~ a ramp time of zero. For linear interpolation, give it a ramp time of one sample (0.0227 msec if the sample rate is 44100 Hz.)

3.10.3 Non-looping sample player

One application area requiring careful thought about the control stream/audio signal boundary is sampling. Until now our samplers have skirted these issues by looping perpetually. This allows for a rich variety of sound that can be accessed by making continuous changes in parameters such as loop size and envelope shape. However, many uses of sampling require the internal features of a sample to emerge at predictable, synchronizable moments in time. For example, percussion samples are usually played from the beginning, are not often looped, and are usually played in some kind of determined time relationship with the rest of the music.

In this situation, control streams are better adapted than audio signals as triggers. Example patch C05.sampler.oneshot.pd (Figure 3.14) shows one possible way to accomplish this. The four tilde objects at bottom left form the signal processing network for playback. One vline ~ object generates a phase signal (actually just a table lookup index) to the tabread4 ~ object; this replaces the phasor ~ of patch B02.wavetable.FM.pd(page 45) and its derivatives.

The amplitude of the output of tabread4 ~ is controlled by a second vline ~ object. This is in order to prevent discontinuities in the output in case a new event is started while the previous event is still playing. The "cutoff" vline ~ ramps the output down to zero (whether or not it is playing) so that, once the output is zero, the index of the wavetable may be changed discontinuously.

The sequence of events for starting a new "note" is, first, that the "cutoff" vline ~ is ramped to zero; then, after a delay of 5 msec (at which point vline ~ has reached zero) the phase is reset. This is done with two messages: first, the phase is set to 1 (with no time value so that it jumps to 1 with no ramping.) This is the first readable point of the wavetable. Second, in the same message

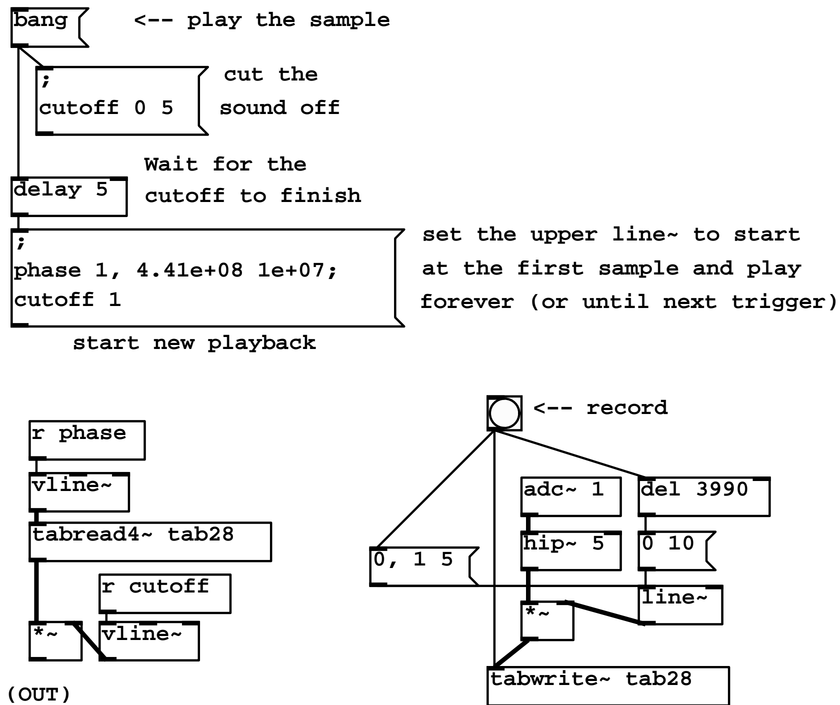


Figure 3.14: Non-looping sampler.

box, the phase is sent to 441,000,000 over a time period of 10,000,000 msec. This corresponds to 44.1 units per millisecond and thus to a transposition of one. The upper vline ~ (which generates the phase) receives these messages via the r phase object above it.

The example assumes that the wavetable is ramped smoothly to zero at either end, and the bottom right portion of the patch shows how to record a sample (in this case four seconds long) which is ramped smoothly to zero at either end. Here a regular (and computationally cheaper) line ~ object suffices. Although the wavetable should be at least 4 seconds long for this to work, you may record shorter wavetables simply by cutting the line ~ object off earlier. The only caveat is that, if you are simultaneously reading and writing from the same sample, you may have to avoid situations where read and write operations attack the same portion of the wavetable at once.

The vline ~ objects surrounding the tabread4 ~ were chosen over line ~ because the latter's rounding of breakpoints to the nearest block boundary (typically 1.45 msec) can make for audible aperiodicities in the sound if the sample is repeated more than 10 or 20 times per second, and would prevent you from getting a nice, periodic sound at higher rates of repetition.

We will return to vline ~-based sampling in the next chapter, to add transposition, envelopes, and polyphony.

3.10.4 Signals to controls

Patch C06.signal.to.control.pd (not pictured) demonstrates conversion from audio signals back to numeric control streams, via a new tilde object introduced here.

snapshot ~: convert audio signal to control messages. This always gives the most recently computed audio sample (fast-as-possible conversion), so the exact sampling time varies by up to one audio block.

It is frequently desirable to sense the audio signal's amplitude rather than peek at a single sample; patch C07.envelope.follower.pd (also not pictured) introduces another object which does this.

env ~: RMS envelope follower. Outputs control messages giving the short-term RMS amplitude (in dB) of the incoming audio signal. A creation argument allows you to select the number of samples used in the RMS computation; smaller numbers give faster (and possibly less stable) output.

3.10.5 Analog-style sequencer

Patch C08.analog.sequencer.pd (figure 3.15) realizes the analog sequencer and envelope generation described in section 3.7. The "sequence" table, with nine elements, holds a sequence of frequencies. The phasor ~ object at top cycles through the sequence table at 0.6 Hz. Non-interpolating table lookup (*mathrmtabread* ~ instead of *mathrmtabread4* ~) is used to read the frequencies in discrete steps. (Such situations, in which we prefer non-interpolating

table lookup, are rare.)

The wrap ~ object converts the amplitude-9 sawtooth to a unit-amplitude one as described earlier in Figure 3.8, which is then used to obtain an envelope function from a second wavetable. This is used to control grain size in a looping sampler (from section 2.6.4). Here the "sample" consists of six periods of a sinusoid. The grains are smoothed by multiplying by a raised cosine function (cos ~ and + ~ 1).

Patch C09.sample.hold.pd (not pictured here) shows a sample-and-hold unit, another useful device for doing control tasks in the audio signal domain.

3.10.6 MIDI-style synthesizer

Patch C10.monophonic.synth.pd (figure 3.16) also implements a monophonic, note-oriented synthesizer, but in this case oriented toward MIDI controllability. Here the tasks of envelope generation and sequencing pitches is handled using control streams instead of audio signals. Several new control objects are needed for this example.

notein: MIDI note input. Three outlets present the pitch, velocity, and channel of incoming MIDI note-on and note-off events (with note-off events appearing as velocity-zero note-on events.) The outlets appear in Pd's customary right-to-left order.

stripnote: filter out note-off messages. This passes (pitch, velocity) pairs through whenever the velocity is nonzero, dropping the others. Unlike notein, stripnote does not input or output actual MIDI messages.

trigger, t: copy a message to outlets in right to left order, with type conversion. The creation arguments ("b" and "f" in this example) specify two outlets, one giving `bang` messages, the other `float` (i.e., numbers). One outlet is created for each creation argument. Other types are "l" for `list`, "s" for `symbol`, and "a" to output anything (i.e., a copy of the input message, whatever it is).

The patch's control objects feed frequencies to the phasor ~ object whenever a note-on message is received. Controlling the amplitude (via the line ~ object) is more difficult. When a note-on message is received, the "sel 0" object outputs the velocity at right (because the input failed to match 0); this is divided by the maximum MIDI velocity of 127 and packed into a message for line ~ with a time of 100 msec.

However, when a note-off is received, it is only appropriate to stop the sound if the note-off pitch actually matches the pitch the instrument is playing. For example, suppose the messages received are "60 127", "72 127", "60 0", and "72 0". When the note-on at pitch 72 arrives the pitch should change to 72, and then the "60 0" message should be ignored, with the note playing until the "72 0" message.

To accomplish this, first we store the velocity in the upper float object. Second, when the pitch arrives, it too is stored (the lower float object) and then the velocity is tested against zero (the `bang` outlet of "t b f" recalls the velocity

```
phasor~ 0.6
*~ 9          main loop: sawtooth of amplitude 9
tabread~ sequence        read frequency sequence
phasor~       wrap~      9x original frequency sawtooth
-~ 0.5        *~ 100     adjust for reading
                                   envelope sample
              +~ 1
              tabread4~ envelope
            *~   multiply by audio-frequency sawtooth
            *~ 128   adjust amplitude
cos~                    and center for wavetable
            +~ 129
+~ 1
            tabread4~ sample
*~    multiply by raised-cosine smoothing function
(out)
```

sequence

sample

envelope

Figure 3.15: An analog-synthesizer-style sequencer.

notein
pit vel
t b f
f - store pitch below
b - bang to recall velocity
float
velocity stored here
sel 0
test for note on or off
note-on
filter
stripnote
off
on
float
recall pitch
pitch
mtof
/ 127
select
test against latest
note-on pitch
phasor~
$1 100
0 1000
-~ 0.5
line~
envelope generator now controls
amplitude as well as grain size
cos~
+~ 0.5
+~ 1
*~
*~
*~ 2
*~
cos~
This replaces the tabread4~
in the previous patch.
(OUT)

Figure 3.16: A MIDI-style monophonic synthesizer.

which is sent to "sel 0"). If this is zero, the second step is to recall the pitch and test it (the select object) against the most recently received note-on pitch. Only if these are zero (so that `bang` appears at the left-hand-side outlet of select) does the message `0 1000` go to the line ~ object.

Exercises

1. What frequency would you hear if you synthesized a sine wave at 88000 Hz. at a sample rate of 44100?

2. Draw a block diagram showing how to use thresholding to detect when one audio signal exceeds another one in value. (You might want to do this to detect and filter out feedback from speakers to microphones.)

3. Using the techniques of section 3.7, draw a block diagram for generating two phase-locked sinusoids at 500 and 700 Hz.

Chapter 4

Automation and voice management

It is often desirable to control musical objects or events in aggregates rather than individually. This might take the form of a series of events spaced in time, in which the details of the events follow from the larger arc (for instance, notes in a melody). Or the individuals might occur simultaneously, such as voices in a chord, or partials in a spectrum. In both cases the properties of the individual might be inferred from those of the whole.

A rich collection of tools and ideas has arisen in the electronic music repertory for describing individual behaviors from aggregate ones. In this chapter we cover two general classes of such tools: envelope generators and voice banks. The envelope generator automates behavior over time, and the voice bank over aggregates of simultaneous processes (such as signal generators.)

4.1 Envelope Generators

An *envelope generator* (sometimes, and more justly, called a *transient generator*) makes an audio signal that smoothly rises and falls as if to control the loudness of a musical note as it rises and falls. (Envelope generators were first introduced in section 1.5.) Amplitude control by multiplication (figure 1.4 is the most direct, ordinary way to use an envelope generator, but there are many other uses.

Envelope generators have come in many forms over the years, but the simples and the perennial favorite is the *ADSR* envelope generator. "ADSR" is an acronym for "Attack, Decay, Sustain, Release", the four segments of the ADSR generator's output. The ADSR generator is controlled by a control stream called a "trigger". Triggering the ADSR generator "on" sets off its attack, decay, and sustain segments. Triggering it "off" sets off the release segment. Figure 4.1 shows how this can appear in block diagrams.

Figure 4.2 shows some possible outputs of an ADSR generator. In part (a) we assume that the "on" and "off" triggers are well enough separated that the

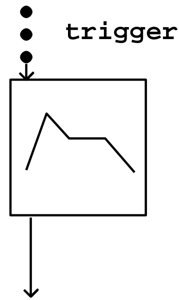


Figure 4.1: ADSR envelope as a block diagram, showing the trigger input (a control stream) and the audio output.

sustain segment is reached before the "off" trigger is received. There are five parameters controlling the ADSR generator. First, a *level* parameter sets the output value at the end of the attack segment (normally the highest value output by the ADSR generator.) Second, an *attack* parameter gives the time duration of the attack segment, and third, a *decay* gives the duration of the decay segment. Fourth, a *sustain* parameter gives the level of the sustain segment, as a fraction of the level parameter. Finally, the *release* parameter gives the duration of the release segment. These five values, together with the timing of the "on" and "off" triggers, fully determines the output of the ADSR generator. For example, the duration of the sustain portion is equal to the time between "on" and "off" triggers, minus the durations of the attack and decay segments.

Parts (b) and (c) of Figure 4.2 show the result of following an "on" trigger quickly by an "off" one: (b) during the release segment, and (c) even earlier, during the attack. The ADSR generator reacts to these situations by canceling whatever remains of the attack and decay segments and continuing straight to the release segment. Also, an ADSR generator may be retriggered "on" before the release segment is finished or even during the attack, decay, or sustain segments. Part (d) of the figure shows a reattack during the sustain segment, and part (e), during the decay segment.

The classic use of an ADSR envelope is when using a voltage-control keyboard or a sequencer to make musical notes on a synthesizer. Depressing a key (for example) would generate an "on" trigger and releasing it, an "off" trigger. The ADSR generator could then control the amplitude of synthesis so that "notes" would start and stop with the keys. In addition to amplitude, the ADSR generator can (and often is) made to control the timbre of the notes, which can then be made to evolve naturally over the course of each note.

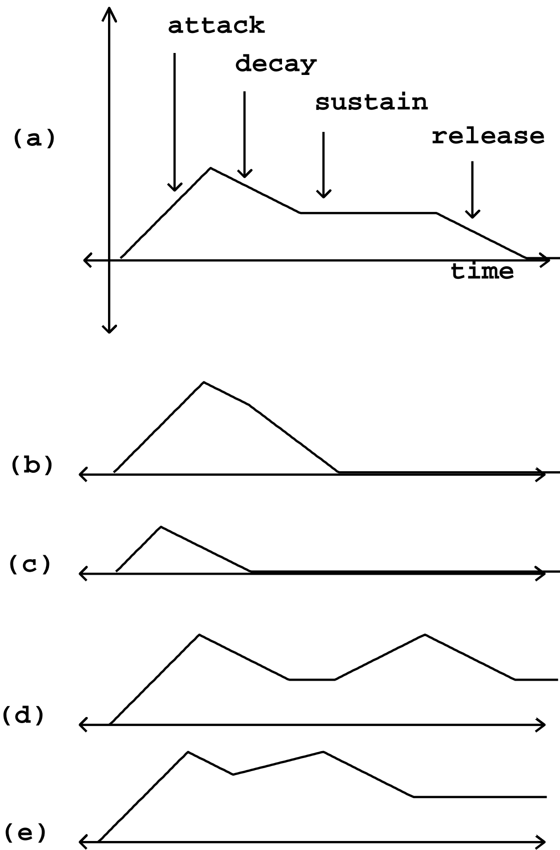


Figure 4.2: ADSR envelope output: (a) with "on" and "off" triggers separated; (b), (c) with early "off" trigger; (d), (e) re-attacked.

4.2 Linear and Curved Amplitude Shapes

Suppose you wish to fade a signal in over a period of ten seconds—that is, you wish to multiply it by an amplitude-controlling signal $y[n]$ which rises from 0 to 1 in value over $10R$ samples, where R is the sample rate. The most obvious choice would be a linear ramp: $y[n] = n/(10R)$. But this will not turn out to yield a smooth increase in perceived loudness. Over the first second $y[n]$ rises from $-\infty$ dB to -20 dB, over the next four by another 14 dB, and over the remaining five, only by the remaining 6 dB. Over most of the ten second period the rise in amplitude will be barely perceptible.

Another possibility would be to ramp $y[n]$ exponentially, so that it rises at a constant rate in dB. You would have to fix the initial amplitude to be inaudible, say 0 dB (if we fix unity at 100 dB). Now we have the opposite problem: for the first five seconds the amplitude control will rise from 0 dB (inaudible) to 50 dB (pianissimo); this amount of rise should have only taken up the first second or so.

The natural progression should perhaps have been: 0-ppp-pp-p-mp-mf-f-ff-fff, so that each increase of one dynamic marking would take roughly one second, and would correspond to one "step" in loudness.

We appear to need some scale in between logarithmic and linear. A somewhat arbitrary choice, but useful in practice, is the quartic curve:

$$y[n] = \left(\frac{n}{N}\right)^4,$$

where N is the number of samples to fade in over (in the example above, it's $10R$). So, over the second (of the ten) we would rise to -80 dB, after five seconds to -24 dB, and after nine, about -4 dB.

Figure 4.3 shows three amplitude transfer functions:

$$f_1(x) = x \qquad \text{(linear)},$$

$$f_2(x) = 10^{2x} \qquad \text{(dB to linear)},$$

$$f_3(x) = x^4 \qquad \text{(quartic)}.$$

The second function converts from dB to linear, arranged so that the input range, from 0 to 1, corresponds to 40 dB. (This input range of 40 dB corresponds to a reasonable dynamic range, allowing 5 dB for each of 8 steps in dynamic.) The quartic curve imitates the exponential (dB) curve fairly well for higher amplitudes, but drops off more rapidly for small amplitudes, reaching true zero at right (whereas the exponential curve only goes down to 1/100.)

We can think of the three curves as showing transfer functions, from an abstract control (ranging from 0 to 1) to a linear amplitude. After we choose a suitable transfer function f, we can compute a corresponding amplitude control signal; if we wish to ramp over N samples from silence to unity gain, the control signal would be:

$$y[n] = f(n/N).$$

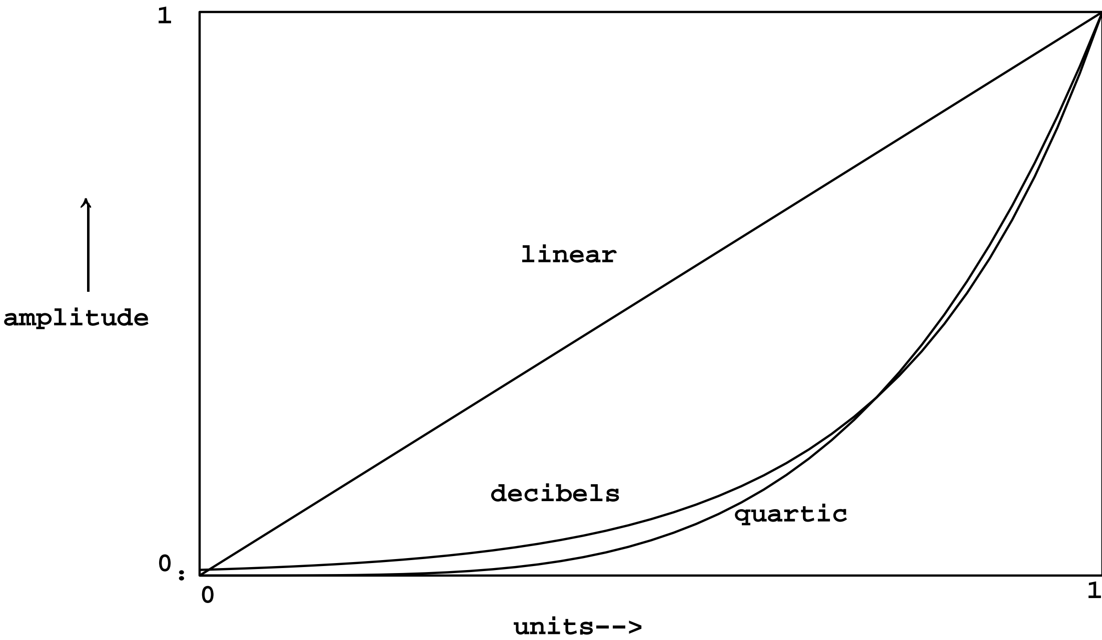


Figure 4.3: Three amplitude transfer functions. The horizontal axis is in linear, logarithmic, or fourth-root units depending on the curve.

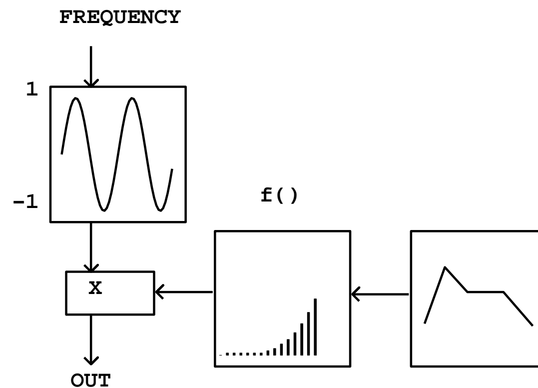


Figure 4.4: Using a transfer function to alter the shape of amplitude curves.

A block diagram for this is shown in Figure 4.4. Here we are introducing a new type of block to represent the application of a *transfer function*. For now we won't worry about its implementation; depending on the function desired, this might be best done arithmetically or using table lookup.

4.3 Continuous and discontinuous control changes

Synthesis algorithms vary widely in their ability to deal with discontinuously changing controls. Until now in this chapter we have assumed that controls must change continuously, and the ADSR envelope generator turns out to be ideally suited to controlling such a parameter. It may even happen that the maximum amplitude of a note is less than the current value of the amplitude of its predecessor (using the same generator) and the ADSR envelope will simply ramp down (instead of up) to the new value for an attack.

This isn't necessarily desirable, however, in situations where an envelope generator is in charge of some aspect of timbre: it may be that we don't want (for example) the sharpness of a note to decrease during the attack to a milder one, but rather to jump to a much lower value so as always to be able to rise during the attack.

This situation also can arise with pitch envelopes: it may be desirable to slide pitch from one note to the next, or it may be desirable that the pitch trajectory of each note start anew at a point independent of the previous sound.

Two situations arise when we wish to make discontinuous changes to synthesis parameters: either we can simply make them without disruption (for instance, making a discontinuous change in pitch); or else we can't, such as a change in a wavetable index (which makes a discontinuous change in the output). There are even parameters that can't *possibly* be changed continuously;

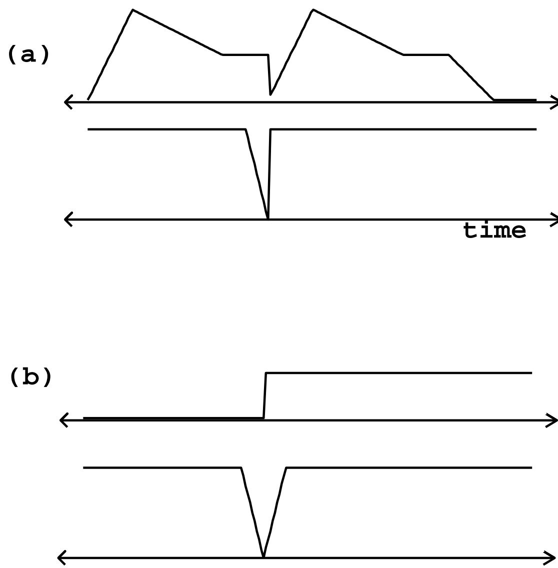


Figure 4.5: The muting technique for hiding discontinuous changes, for example of an amplitude envelope. In part (a) the envelope (upper graph) is set discontinuously to zero. The muting signal (lower graph) ramps down in advance to prepare for the change, and then is restored (discontinuously) to its previous value. In (b), the envelope changes discontinuously between two nonzero values; the muting signal must both ramp down beforehand and ramp back up afterward.

for example, a selection among a collection of wavetables. In general, we can't set the phase of an oscillator or the amplitude of a signal discontinuously, but we may change phase increments (such as pitches) almost arbitrarily without bad results.

In those cases where a parameter change can't be made continuously for one reason or another, there are at least two strategies for making the change: *muting* and *switch-and-ramp*.

4.3.1 Muting

The *muting* technique is to apply an envelope to the output amplitude, which is quickly ramped to zero before the parameter change and then restored afterward. It may or may not be true that the discontinuous changes will result in a signal that rises smoothly from zero afterward. In Figure 4.5 (part a), we take the example of an amplitude envelope (the output signal of an ADSR generator), and assume that the discontinuous change is to start a new note at amplitude zero.

To change the ADSR generator's output discontinuously we *reset* it. This is a different operation from triggering it; the result is to make it jump to a new value, after which we may either simply leave it there or trigger it anew. In the top of 4.5 (part a) we show the effect of resetting and retriggering an ADSR generator.

Below the ADSR generator output we see the muting signal, which ramps to zero to prepare for the discontinuity. The amount of time we allow for muting should be small (so as to disrupt the previous sound as little as possible) but not so small as to cause audible artifacts in the output. A working example of this type of muting was already shown in section 3.10.3; there we allowed 5 msec for ramping down.

Figure 4.5 (part b) shows the situation in which we suppose the discontinuous change is between two nonzero values. Here the muting signal must not only ramp down as before (in advance of the discontinuity) but must also ramp back up afterward. The ramp-down time need not equal the ramp-up time; these must be chosen, as always, by listening to the output sound.

In general, muting presents the difficulty that you must start the muting operation in advance of making the desired control change. In real-time settings, this often requires that we intentionally delay the control change. This is another reason for keeping the muting time as low as possible. (Moreover, it's a bad idea to try to minimize delay by conditionally omitting the ramp-down period when it isn't needed; a constant delay is much better than one that varies, even if it is smaller on average.)

4.3.2 Switch-and-ramp

The *switch-and-ramp* technique also seeks to remove discontinuities resulting from discontinuous control changes, but does so in a different way: by synthesizing an opposing discontinuity which cancels the original one out. Figure 4.6 shows an example in which a synthetic percussive sound, an enveloped sinusoid, starts a note in the middle of a previous one. The attack of the sound derives not from the amplitude envelope but on the initial phase of the sinusoid, as is often appropriate for percussive sounds. The lower graph in the figure shows a compensating audio signal with an opposing discontinuity, which can be added to the upper one to remove the discontinuity. The advantages of this technique over muting are, first, that there need be no delay between the decision to make an attack and the sound of the attack; and second, that any artifacts arising from this technique are more likely to be hidden by the new sound's onset.

Figure 4.7 shows how the switch-and-ramp technique can be realized in a block diagram. The box marked with ellipsis ("...") may hold any synthesis algorithm, which we wish to interrupt discontinuously so that it restarts from zero (as the example of the previous figure did). At the same time we trigger whatever control changes are necessary (exemplified by the top ADSR generator), we also reset and trigger another ADSR generator (middle right) to cancel out the discontinuity.

To do this we measure the level the ADSR generator must now jump to.

Figure 4.6: The switch-and-ramp technique for canceling out discontinuous changes. A discontinuity (upper graph) is measured and canceled out with a signal having the opposite discontinuity (lower graph), which then decays smoothly.

Figure 4.7: Block diagram for the switch-and-ramp technique.

This is its own current level (which may not be zero) minus the current output of the synthesis algorithm. We subtract the two and send the difference to a snapshot object, triggered at the same time as the discontinuity is expected. Finally, we use the snapshot output as a parameter to the ADSR generator, which must be reset (to the level given by the snapshot) and ramped toward zero.

4.4 Polyphony

In music, the term *polyphony* is usually used to mean "more than one separate voices singing or playing at different pitches one from another". When speaking of musical instruments (electronic or otherwise), we use the term to mean "maintaining several copies of some process in parallel." We usually call each copy a "voice" in keeping with this analogy, although the voices needn't be playing separately distinguishable sounds.

In this language, a piano is a polyphonic instrument, with 88 "voices". Each voice of the piano is normally capable of playing exactly one pitch. There is never a question of which voice to use to play a note of a given pitch, and no question, either, of playing several notes simultaneously of the same pitch.

Most polyphonic electronic musical instruments take a more flexible approach to voice management. Most software synthesis programs (like csound) use a dynamic voice allocation scheme, so that, in effect, a new voice is created for every note in the score. In systems such as Max or Pd which are oriented toward real-time interactive use, a *voice bank* is allocated in advance, and the work to be done (playing notes, or whatever) is distributed among the voices in the bank. This is diagrammed in Figure 4.8.

In this example the several voices each produce one output signal, which are all added to make the total output of the voice bank. Frequently it will occur that individual voices need several separate outputs; for instance, they might output stereo so that voices may be panned individually; or they might have individual effects sends so that each may have its own send level.

4.5 Voice allocation

It is frequently desirable to automate the selection of voices to associate with individual *tasks* (such as notes to play). For example, a musician playing at a keyboard can't practically choose which voice should go with each note played. To automate voice selection we need a voice allocation algorithm, to be used as shown in Figure 4.9.

Armed with a suitable voice allocation algorithm, the control source need not concern itself with the detail of which voice is taking care of which task; algorithmic note generators and sequencers frequently rely on this. On the other hand, musical writing for ensembles frequently specifies explicitly which

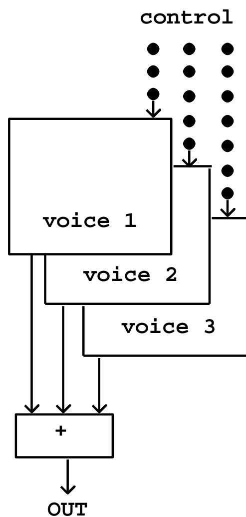


Figure 4.8: A voice bank for polyphonic synthesis.

instrument plays which note, so that the notes will connect to each other end-to-end in a desirable way.

One simple voice allocation algorithm works as shown in Figure 4.10. Here we suppose that the voice bank has only two voices, and we try to allocate voices for the tasks a, b, c, and d. Things go smoothly until task d comes along, but then we see no free voices (they are taken up by b and c). We could now elect either to drop task d, or else to steal the voice of either task b or c. In practice the best choice is usually to steal one. In this particular example, we chose to steal the voice of the oldest task, b.

We can see that, if we know the length of the tasks b and c at the outset of task d, we may be able to make a better choice of which voice to steal. In this example it might have been better to steal from c, so that d and b would be playing together at the end and not d alone. In some situations this information will be available when the choice must be made, and in some (live keyboard input, for example) it will not.

4.6 Voice tags

Suppose now that we're using a voice bank to play notes, as in the example above, but suppose the notes a, b, c, and d all had the same pitch, and furthermore that all their other parameters were identical. How would we design a control stream so that, when any one note was turned off, we would know which one it was?

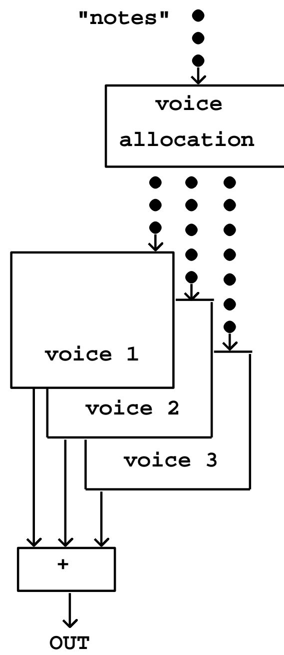


Figure 4.9: Polyphonic voice allocation

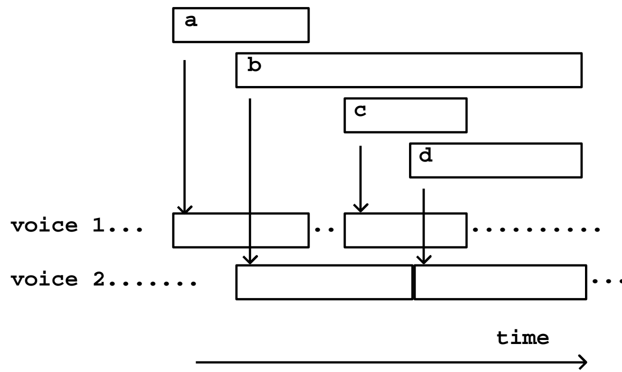


Figure 4.10: A polyphonic voice allocation algorithm, showing voice stealing.

This question doesn't come up if the control source is a clavier keyboard because it's impossible to play more than one simultaneous note on a single key. But it could easily arise algorithmically, or simply as a result of merging two keyboard streams together. Moreover, turning notes off is only the simplest example of a more general problem, which is how, once having set an task off in a voice bank, we can get back to the correct voice to guide its evolution as a function of real-time inputs or any other unpredictable factor.

To deal with situations like this we can add one or more *tags* to the message starting a process (such as a note). A tag is any collection of data with which we can later identify the process, which we can later use to search for the voice that is allocated for it.

Taking again the example of Figure 4.10, here is one way we might write those four tasks as a control stream:

```
start-time end-time   pitch   ...

    1          2        60    ...
    2          6        62
    4          2        64
    5          3        65
```

In this representation we have no need of tags because each message (each line of text) contains all the information we need in order to specify the entire task. (Here we have assumed that the tasks $a, \ldots, d$ are in fact musical notes with pitches 60, 62, 64, and 65.) In effect we're representing each task as a single event (section 3.3) in a control stream.

On the other hand, if we suppose now that we do not know in advance the length of each note, a better representation would be this one:

```
time    tag  action   parameters

  1       a   start     60 ...
  2       b   start     62 ...
  3       a   end
  4       c   start     64 ...
  5       d   start     65 ...
  6       c   end
  8       b   end
  8       d   end
```

Here each note has been split into two separate events to start and end it. The labels a, ..., d are used as tags; we know which start goes with which end since their tags are the same. Note that the tag is not necessarily related at all to the voice that will be used to play each note.

The MIDI standard does not supply tags; in normal use, the pitch of a note serves also as its tag (so tags are constantly being re-used.) If two notes having the same pitch must be addressed separately (to slide their pitches in different

ways for example), the MIDI channel may be used (in addition to the note) as a tag.

In real-time music software it is often necessary to pass back and forth between the event-per-task representation and the tagged representation above, since the first representation is better suited to storage and graphical representation, while the second is better suited to real-time operations.

4.7 Encapsulation in Pd

The examples for this chapter will use Pd's *abstraction* mechanism, which permits the building of patches that may be reused any number of times. One or more object boxes in a Pd patch may be *subpatches*, which are separate patches encapsulated inside the boxes. These come in two types: *one-off subpatches* and *abstractions*. In either case the subpatch appears as an object box in the *parent* patch.

If you type "pd" or "pd my-name" into an object box, this creates a one-off subpatch. The contents of the subpatch are saved as part of the parent patch, in one file. If you make several copies of a subpatch you may change them individually. On the other hand, you can invoke an abstraction by typing into the box the name of a Pd patch saved to a file (without the ".pd" extension). In this situation Pd will read that file into the subpatch. In this way, changes to the file propagate everywhere the abstraction is invoked.

Object boxes in the subpatch (either one-off or abstractions) may create inlets and outlets on the box in the parent patch. This is done with the following classes of objects:

[inlet], [inlet ~]: create inlets for the object box containing the subpatch. The inlet ~ version creates an inlet for audio signals, whereas inlet creates one for control streams. In either case, whatever shows up on the inlet of the box in the parent patch comes out of the inlet object in the subpatch.

[outlet], [outlet ~]: Corresponding objects for output from subpatches.

Pd provides an argument-passing mechanism so that you can parametrize different invocations of an abstraction. If in an object box you type "$1", it is expanded to mean "the first creation argument in my box on the parent patch", and similarly for "$2" and so on. The text in an object box is interpreted at the time the box is created, unlike the text in a message box. In message boxes, the same "$1" means "the first argument of the message I just received" and is interpreted whenever a new message comes in.

An example of an abstraction, using inlets, outlets, and parametrization, is shown in figure 4.11. In part (a), a patch invokes "plusminus" in an object box, with a creation argument equal to 5. The number 8 is fed to the plusminus object, and out comes the sum and difference of 8 and 5.

The plusminus object is not defined by Pd, but is rather defined by the patch residing in the file named "plusminus.pd". This patch is shown in part (b) of the figure. The one inlet and two outlet objects correspond to the inlets

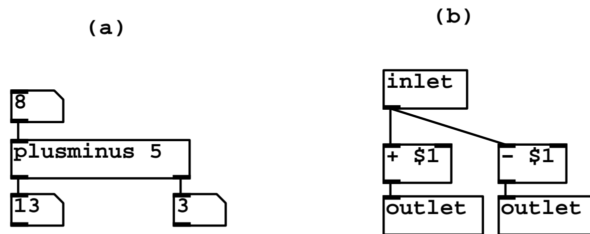


Figure 4.11: Pd's abstraction mechanism: (a) invoking the abstraction, "plusminus" with 5 as a creation argument; (b) the contents of the file, "plusminus.pd".

and outlets of the plusminus object. The two "`$1` arguments (to the + and − objects) are replaced by 5 (the creation argument of the plusminus object).

We have already seen one abstraction in the examples: the output ∼ object introduced in chapter 1, Figure 1.8(c). This shows an additional feature of abstractions, that they may display controls as part of their boxes on the parent patch; see the Pd documentation for a description of this feature.

4.8 Examples

4.8.1 ADSR envelope generator

Patch D01.envelope.gen.pd (figure 4.12) shows how the `line~` object may be used to generate an ADSR envelope to control a synthesis patch (only the ADSR envelope is shown). The "attack" button, when pressed, sets off two chains of events. The first (leftmost in the figure) is to set the `line~` object on its attack segment, with a target of 10 (the peak amplitude) over 200 msec (the attack time). Second, the attack button sets a `delay 200` object, so that after the attack segment is done, the decay segment can start. The decay segment goes to a target of 1 (the sustain level) after another 2500 msec (the decay time).

The "release" button sends the same `line~` object beck to zero over 500 more milliseconds (the release time). Also, in case the `delay 200` object happens to be set at the moment the "release" button is pressed, a `stop` message is sent to it. This prevents the ADSR generator from launching its decay segment after launching its release segment.

In patch D02.adsr.pd (figure 4.13) we encapsulate the ADSR generator in a Pd abstraction (named `adsr`) so that it can easily be replicated. The design of the `adsr` abstraction makes it possible to control the five ADSR parameters

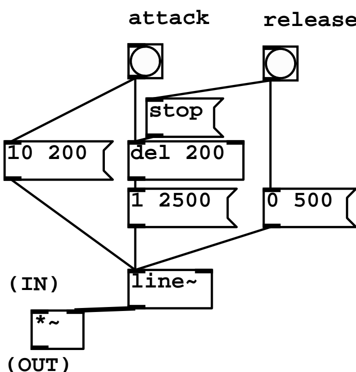


Figure 4.12: Using a `line~` object to generate an ADSR envelope.

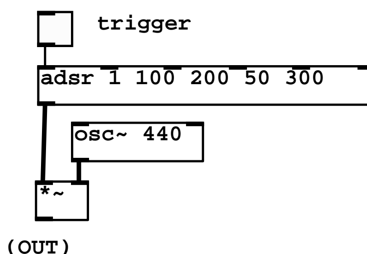


Figure 4.13: Invoking the `adsr` abstraction.

either by supplying creation arguments or by connecting control streams to its inlets.

In this example the five creation arguments (1, 100, 200, 50, and 300) specify the peak level, attack time, decay time, sustain level (as a percentage of peak level), and release time. There are six control inlets: the first to trigger the ADSR generator, and the others to update the values of the five parameters. The output of the abstraction is an audio signal.

This abstraction is realized as shown in figure 4.14. (You can open this patch by clicking on the `adsr` object in the patch.) The only signal objects are `line~` and `outlet~`The three `pack` objects correspond to the three message objects from the earlier figure 4.12. They take care of the attack, decay, and release segments.

The attack segment goes to a target specified as `$1` (the first creation argument of the abstraction) over `$2` milliseconds; these values may be overwritten by sending numbers to the "peak level" and "attack" inlets. The release segment is similar except simpler since the target is always zero. The hard part is

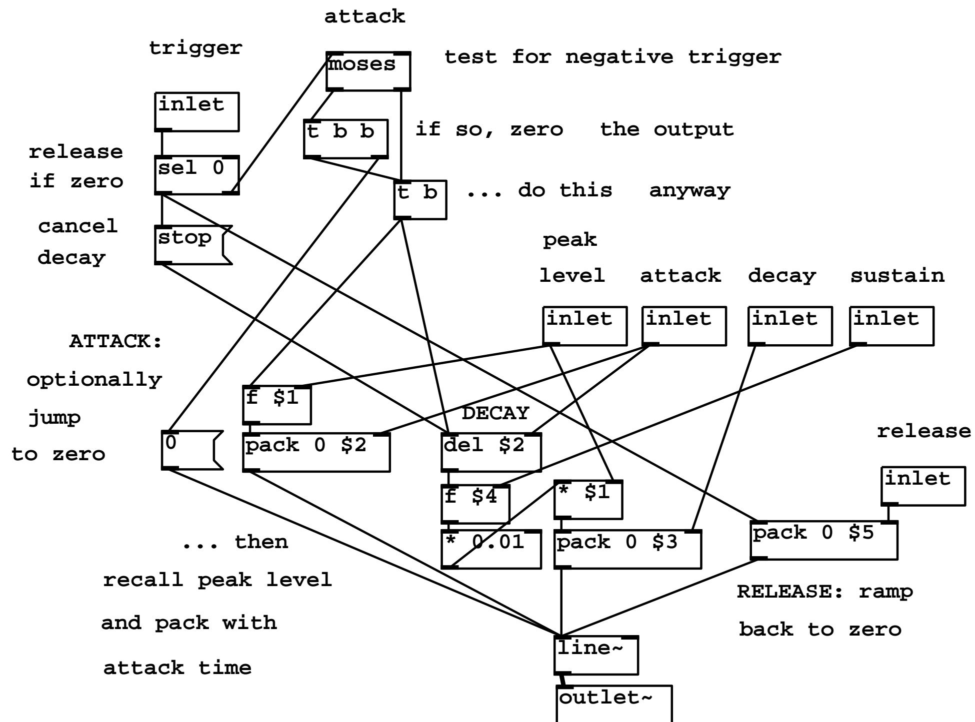


Figure 4.14: Inside the adsr abstraction.

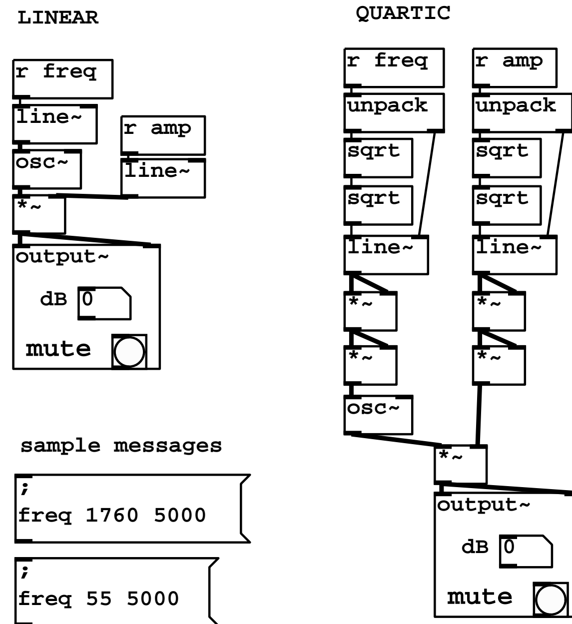


Figure 4.15: Linear and quartic transfer functions for changing amplitude and pitch.

the decay segment, which again must be set off after a delay equal to the attack time (the `del $2` object). The sustain level is calculated from the peak level and the sustain percentage (multiplying the two and dividing by 100).

The trigger inlet, if sent a number other than zero, triggers an onset (the attack and decay segments), and if sent zero, triggers the release segment. Furthermore, the ADSR generator may be reset to zero by sending a negative trigger (which also generates an onset).

4.8.2 Transfer functions for amplitude control

In section 4.2 we considered using ADSR envelopes to control amplitude, for which exponential or quartic-curve segments often give more natural-sounding results than linear ones. Patches D03.envelope.dB.pd and D04.envelope.quartic.pd (the latter is shown in figure 4.15) demonstrate the use of decibel and quartic segments. In addition to amplitude, in patch D04.envelope.quartic.pd the frequency of a sound is also controlled, using linear and quartic shapes, for comparison.

Since converting decibels to linear amplitude units is a costly operation (at least when compared to an oscillator or a ramp generator), patch D03.envelope.dB.pd uses table lookup to implement the necessary transfer function. This has the

advantage of efficiency, but the disadvantage that we must decide on the range of admissible values in advance (here from 0 to 120 dB).

For a quartic segment as in patch D04.envelope.quartic.pd no table lookup is required; we simply square the `line~` object's output signal twice in succession. To compensate for raising the output to the fourth power, the target values sent to the `line~` object must be the fourth root of the desired ones. Thus, messages to ramp the frequency or amplitude are first unpacked to separate the target and time interval, and the target's fourth root is taken (via two square roots in succession) and the two are then sent to the `line~` object. Here we have made use of one new Pd object:

`pdunpack`: unpack a list of numbers (and/or symbols) and distribute them to separate outlets. As usual the outlets appear right to left. The number and types of the outlets is determined by the creation arguments. (See also `pack`, p. 44).

The next two patches, D05.envelope.pitch.pd and D06.envelope.portamento.pd use an ADSR envelope generator to make a pitch envelope and a simple `line~` object, also controlling pitch, to make portamento. In both cases exponential segments are desirable, and they are calculated using table lookup.

4.8.3 Additive synthesis: Risset's bell

The abstraction mechanism of Pd, which we used above to make a reusable ADSR generator, is also useful for making voice banks. Here we will use abstractions to organize banks of oscillators for additive synthesis. There are many possible ways of organizing the oscillator banks besides the few we'll show here.

The simplest and most direct organization of the sinusoids is to form partials to add up to a note. The result is monophonic, in the sense that the patch will play only one note at a time, which, however, will consist of several sinusoids whose individual frequencies and amplitudes might depend both on those of the note we're playing, and also on their individual placement in a harmonic (or inharmonic) overtone series.

For example, patch D07.additive.pd (figure 4.16) uses a bank of 11 copies of an abstraction named "`partial`" (figure 4.17) in an imitation of a well-known bell instrument by Jean-Claude Risset. As described in [DJ85, p. 94], the bell sound has 11 partials, each with its own relative amplitude, frequency, and duration.

For each note, the `partial` abstraction computes a simple (quartic) amplitude envelope consisting only of an attack and a decay segment; there is no sustain or release segment. This is multiplied by a sinusoid, and the product is added into a summing bus. Two new object classes are introduced to implement the summing bus:

`catch~`: define and output a summing bus. The creation argument ("sumbus" in this example) gives the summing bus a name so that `throw~` objects below can refer to it. You may have as many summing busses (and hence `catch~` objects) as you like but they must all have different names.

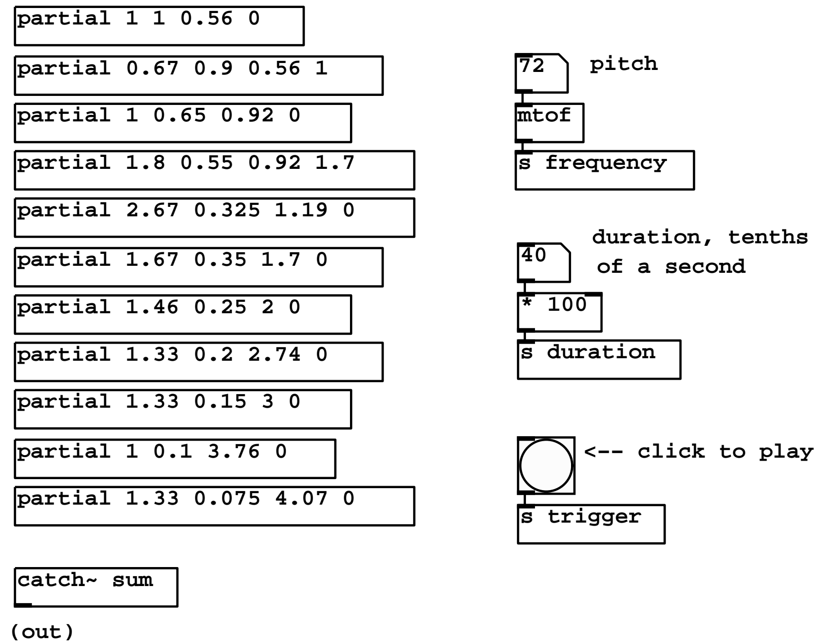


Figure 4.16: A Pd realization of Jean-Claude Risset's bell instrument. The bell sound is made by summing 11 sinusoids, each made by a copy of the `partial` abstraction.

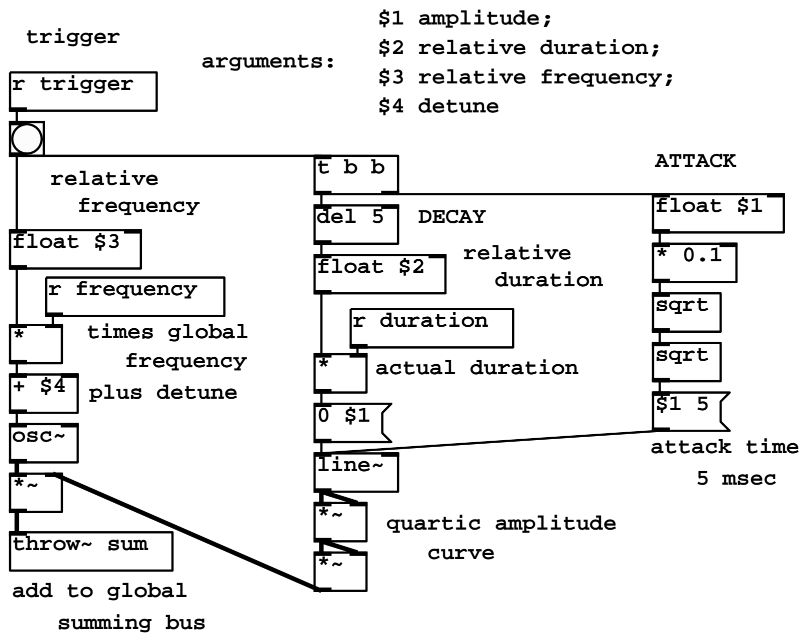


Figure 4.17: The `partial` abstraction used by Risset's bell instrument from figure 4.16.

`throw~`: add to a summing bus. The creation argument selects which summing bus to use.

The control interface is crude: number boxes control the "fundamental" frequency of the bell and its duration. Sending a "bang" message to the `s trigger` object starts a note. (The note then decays over the period of time controlled by the duration parameter; there is no separate trigger to stop the note.) There is no amplitude control except via the `output~` object.

The four arguments to each invocation of `partial` abstraction specify:

1. amplitude. The peak amplitude of the partial at its peak, at the end of the attack and the beginning of the decay of the note

2. relative duration. This is multiplied by the overall note duration (controlled in the main patch) to determine the duration of the decay portion of the sinusoid. Individual partials may thus have different decay times, so that some partials die out faster than others, under the main patch's overall control.

3. relative frequency. As with the relative duration, this controls each partial's frequency as a multiple of the overall frequency controlled in the main patch.

4. detune. A frequency in Hz. to be added to the product of the global frequency and the relative frequency.

Inside the `partial` abstraction, the amplitude is simply taken directly from the `$1` argument (multiplying by 0.1 to adjust for the high individual amplitudes); the duration is calculated from the `r duration` object, multiplying it by the `$2` argument, and the frequency is equal to $fp + d$ where f is the global frequency (from the `r frequency` object), p is the relative frequency of the partial, and d is the detune frequency.

4.8.4 Additive synthesis: spectral envelope control

The next patch example, D08.table.spectrum.pd(figure 4.18), shows a very different application of additive synthesis from the previous patch. Here the sinusoids are managed by the `spectrum-partial.pd` abstraction shown in figure 4.19. Each partial computes its own frequency as in the previous patch. Each partial also computes its own amplitude periodically (when triggered by the `r poll-table` object), using a `tabread4` object. The contents of the table (which has a nominal range of 50 dB) are converted to linear units and used as an amplitude control in the usual way.

A similar example, patch D09.shepard.tone.pd(not pictured), makes a Shepard tone using the same technique. In this example the frequencies of the sinusoids sweep over a fixed range, finally jumping from the end back to the beginning and repeating. The spectral envelope is arranged to have a peak at the middle of the pitch range and drop off to inaudibility at the edges of the range

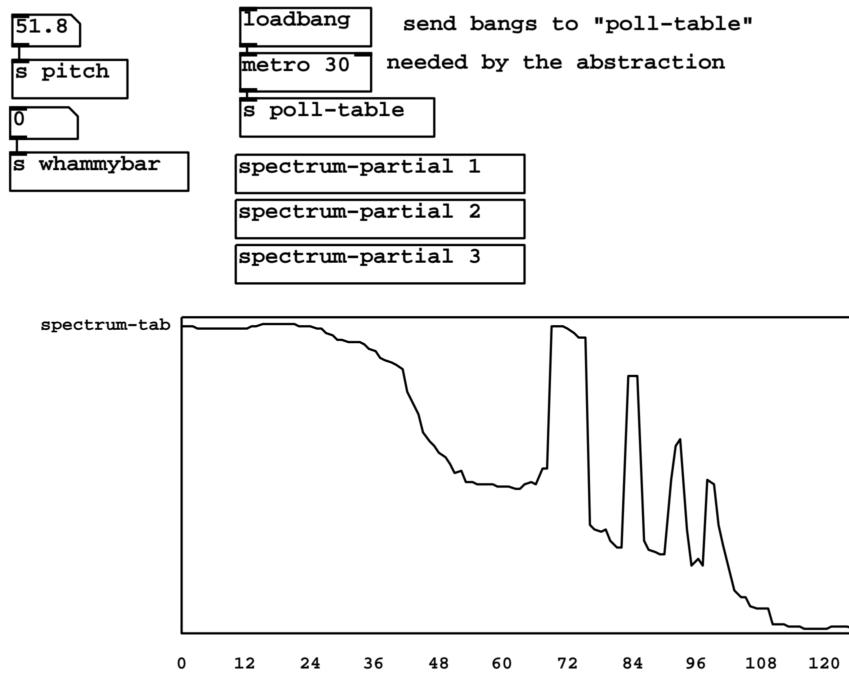


Figure 4.18: Additive synthesis for a specified spectral envelope, drawn in a table.

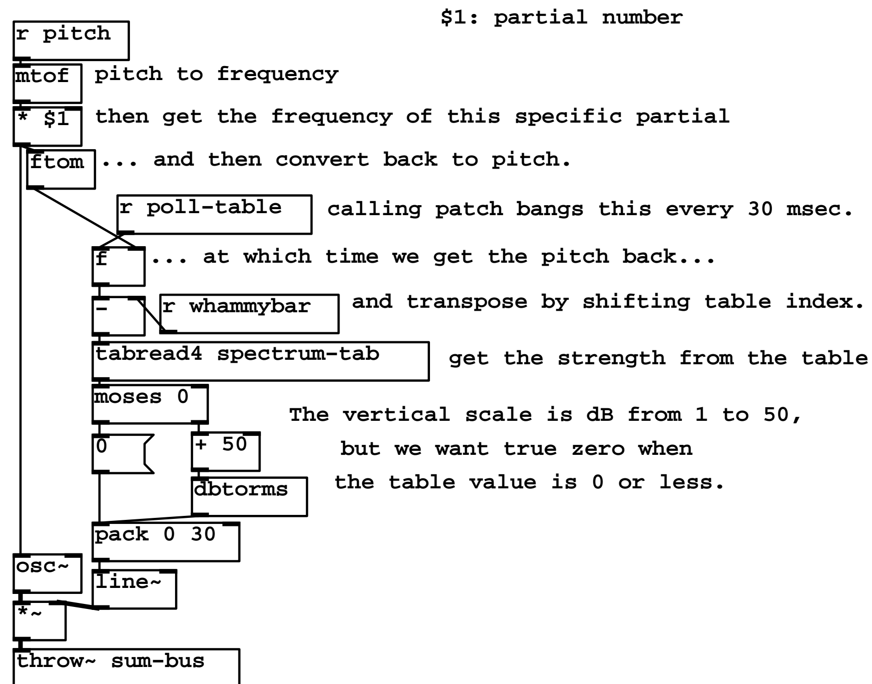


Figure 4.19: The `spectrum-partial` abstraction used in figure 4.18.

so that we hear only the continuous sweeping and not the jumping. The result is the famous auditory conundrum of an infinitely ascending or descending tone.

The general technique of synthesizing to a specified spectral envelope can be generalized in many ways: the envelope may be made to vary in time either as a result of a live analysis of another signal, or by calculating from a set of compositional rules, or by cross-fading between a collection of pre-designed spectral envelopes, or by frequency-warping the envelopes, or in many other ways.

4.8.5 Polyphonic synthesis: sampler

We move now to an example using dynamic voice allocation as described in section 4.5. In the additive synthesis examples shown previously, all the voices are always used for a fixed purpose. In the present example, we allocate voices from a bank as needed to play notes in a control stream.

Patch D11.sampler.poly.pd(figure 4.20) shows the polyphonic sampler, which uses the abstraction `sampvoice` (shown in figure 4.21). The techniques for altering the pitch and other parameters in a one-shot sampler are shown in patch D10.sampler.notes.pd(not shown) which in turn is derived from the original one-shot sampler from the previous chapter (C05.sampler.oneshot.pd, shown in figure 3.14).

The `sampvoice` objects in figure 4.20 are arranged in a different kind of summing bus from the ones before, in which each one adds its own output to the signal on its inlet, and puts the sum on its outlet. At the bottom of the eight objects, the outlet therefore holds the sum of all eight. This has the advantage of being more explicit than the `throw~` / `catch~` busses, and is preferable if visual clutter is not at issue.

The main job of the patch, though, is to distribute the "note" messages to the `sampvoice` objects. To do this we must introduce some new Pd objects:

`pdmod` : Integer modulus. For instance, 17 mod 10 gives 7, and -2 mod 10 gives 8. There is also an integer division object named `pddiv` ; dividing 17 by 10 via `pddiv` gives 1, and -2 by 10 gives -1.

`poly` : Polyphonic voice allocator. Creation arguments give the number of voices in the bank and a flag (1 if voice stealing is needed, 0 if not). The inlets are a numeric tag at left and a flag at right indicating whether to start or stop a voice with the given tag (nonzero numbers meaning "start" and zero, "stop".) The outputs are, at left, the voice number, the tag again at center, and the start/stop flag at right. In MIDI applications, the tag can be pitch and the start/stop flag can be the note's velocity.

`makenote` : Supply delayed note-off messages to match note-on messages. The inlets are a tag and start/stop flag ("pitch" and "velocity" in MIDI usage) and the desired duration in milliseconds. The tag/flag pair are repeated to the two outlets as they are received; then, after the delay, the tag is repeated with flag zero to stop the note after the desired duration.

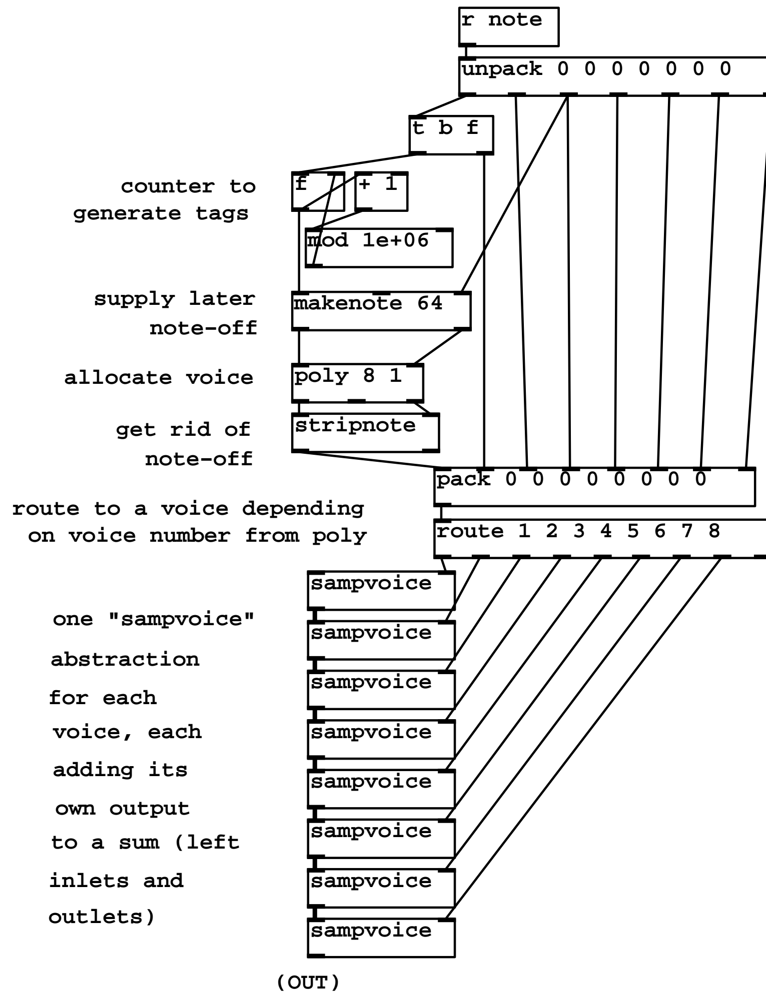


Figure 4.20: A polyphonic sampler demonstrating voice allocation and use of tags.

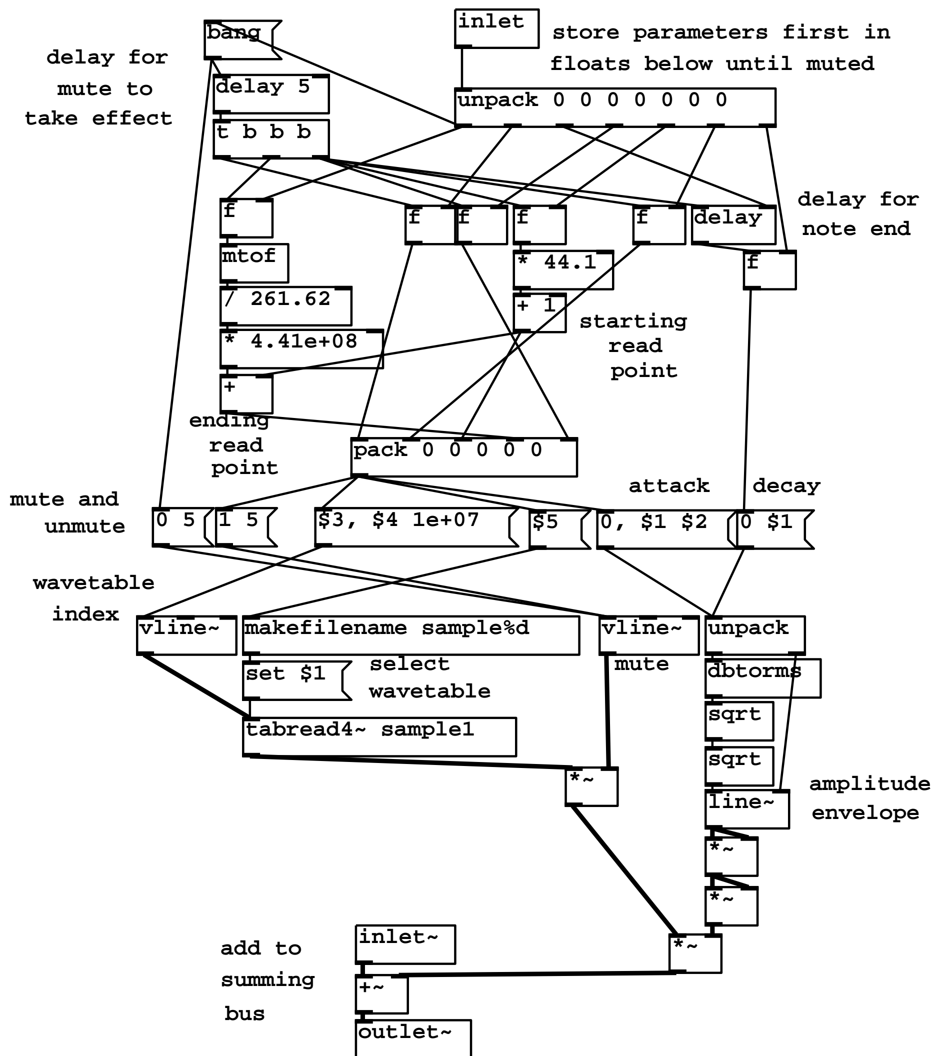


Figure 4.21: The sampvoice abstraction used in the polyphonic sampler of figure 4.20.

The "note" messages contain fields for pitch, amplitude, duration, sample number, start location in the sample, rise time, and decay time. For instance, the message,

```
60 90 1000 2 500 10 20
```

(sent to "note" via the `r note` object) means to play a note at pitch 60 (MIDI units), amplitude 90 dB, one second long, from the wavetable named "sample2", starting at a point 500 msec into the wavetable, with rise and decay times of 10 and 20 msec.

After unpacking the message into its seven components, the first job is to create a tag for the note. To do this, first the `t b f` object outputs a bang after the last of the seven parameters appear separately. The combination of the `+`, `f`, and `mod` objects act as a counter which repeats after a million steps, essentially generating a unique number corresponding to the note.

The next step is to use the `poly` to determine which voice to play which note. The `poly` object expects separate note starts and stops as input. So the tag and duration are first fed to the `makenote` object, whose output contains a flag ("velocity") at right and the tag again at left. For each tag makenote receives, two pairs of numbers are output, one to start the note, and another, after a delay equal to the note duration, to stop the note.

Having treated `poly` to this separated input, we now have to strip the note stop output, since we really only need combined "note" messages with duration fields. The `stripnote` object does this job. Finally, the voice number we have calculated is prepended to the seven parameters we started with (the `pack` object), so that the output of the `pack` object looks like this:

```
4 60 90 1000 2 500 10 20
```

where the "4" is the voice number output by the `poly` object. The voice number is used to route the note message to the desired voice using the `route` object. The desired voice (i.e., the `sampvoice` object) then gets the original list of 7 numbers shown above.

Inside the `sampvoice` object (figure 4.21), the message is then used to control the `tabreadfour~` and surrounding `line~` and `vline~` objects. The control takes place with a delay of 5 msec as in the previous sampler example. Here, however, we must store the seven parameters of the note (whereas, earlier, there were no parameters.) This is done using the six `f` objects, plus the right inlet of the rightmost `delay` object. These values are used after the delay of 5 msec. This works in tandem with the "mute" mechanism described earlier, via the `line~` object.

After the 5 msec are up, the `vline~` object in charge of generating the wavetable index is given its marching orders (and, simultaneously, the wavetable number is set for the `tabreadfour~` object and the amplitude envelope generator is set on its attack.) The wavetable index must be set discontinuously to the starting index, then ramped to an ending index over an appropriate time duration to obtain the needed transposition. The starting index in samples is just

44.1 times the starting location in milliseconds, plus one to allow for four-point table interpolation. This becomes the third number in a packed list generated by the `pack` object at the center of the voice patch.

We arbitrarily decide that the ramp will last ten thousand seconds (this is the "1e+07" appearing in the message box sent to the wavetable index generator), assuming that this is at least as long as any note we will play. Then the ending index is the starting index plus the number of samples to ramp through. At a transposition factor of one, we should move by 441 million samples in the 10 million milliseconds; and proportionally more or less depending on the transposition factor. This transposition factor is computed by the `mtof` object, dividing by 261.62 (the frequency corresponding to MIDI note 60) so that inputting 60 results in a transposition factor of one.

This and all the other necessary parameters are combined in one message via the `pack` object so that the following message boxes can easily generate the needed control messages. The only unfamiliar treatment is by the `makefilename` object, which converts numbers such as "2" to symbols such as "sample2" so that the `tabreadfour~` object's wavetable may be set.

At the bottom of the voice patch we see how the summing bus is treated from the inside; an `inlet~` object picks up the sum of all the preceding voices, the output of the current voice is added in, and the result is sent on to the next voice via the `outlet~` object.

Chapter 5

Modulation

Having taken a two-chapter detour into aspects of control and organization in electronic music, we now return to describing audio synthesis and processing techniques. So far we have considered additive and wavetable-based methods. In this chapter we will introduce three types of *modulation*, called *amplitude modulation*, *frequency modulation*, and *waveshaping*, all of which are useful for building synthetic sounds with certain, characteristic families of spectra. We will first need some terminology for speaking about spectra, which we introduce in the next section.

5.1 Taxonomy of spectra

The FOURIER SERIES (page 11) gives a description of a periodic signal as a sum of sinusoids. The frequencies of the sinusoids are in the ratio $0 : 1 : 2 : \cdots$. (The constant term in the Fourier series may be thought of as a sinusoid,

$$a_0 = a_0 cos(0 \cdot \omega n),$$

whose frequency is zero.)

The top of Figure 5.1 shows the *spectrum* of a periodic signal. The spectrum shows how the signal's power is distributed into frequencies. (This isn't a real mathematical definition; to provide one would require a background in functional analysis.)

In the case of the periodic signal, the power shown in the spectrum is concentrated on a discrete subset of the frequency axis (a discrete set consists of isolated points, only finitely many per unit). We call this a *discrete* spectrum. Furthermore, the spectrum is called *harmonic*, meaning that the frequencies where the signal's power lies are in the $0 : 1 : 2 \cdots$ ratio that arises from a periodic signal. (Note that it's not necessary for *all* of the harmonic frequencies to be present; some harmonics may have zero amplitude.) We'll only consider a signal to be harmonic if the fundamental frequency is in the range of perceptible pitches, roughly between 50 and 4000 Hz.

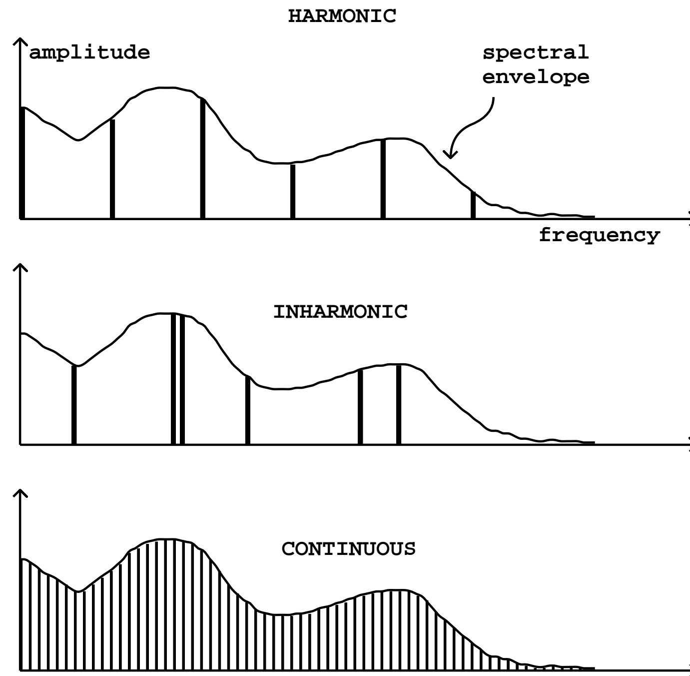


Figure 5.1: A taxonomy of timbres. The spectral envelope describes the shape of the spectrum. The sound may be discretely or continuously distributed in frequency; if discretely, it may be harmonic or inharmonic.

The graph of the spectrum shows the amplitudes of the partials of the signals. Knowing the amplitudes and phases of all the partials would allow us to reconstruct the original signal.

The second spectrum pictured in the figure is also discrete, and so the signal can again be considered as a sum of a series of partials. In this case, however, there is no fundamental frequency, i.e., no common submultiple of all the partials. We call this an *inharmonic* spectrum.

In dealing with discrete spectra, we adopt a convention whose purpose will only become clear gradually: each component sinusoid,

$$a cos(\omega n + \phi),$$

only counts as having amplitude $a/2$ as long as the angular frequency ω is nonzero. On the other hand, for a DC component, where $\omega = \phi = 0$, we give the partial the full amplitude of a.

Finally, the bottom of the figure shows a third possibility, which is that the spectrum might not be concentrated into a discrete set of frequencies, but might be spread out among all possible frequencies. This can be called a *continuous*, or *noisy* spectrum. Spectra don't have to fall into either the discrete or continuous categories; they may be mixtures of the two.

Each of the three spectra in the figure shows a continuous curve called the *spectral envelope*. In general, sounds don't have a single, well-defined spectral envelope; there may be many ways to draw a smooth-looking curve through a spectrum. On the other hand, a spectral envelope may be specified intentionally by an electronic musician. In that case, it is usually clear how to make a spectrum conform to the specified spectral envelope. For a discrete spectrum, for example, we could simply read off, from the spectral envelope, the desired amplitude of each partial and make it so.

We have been speaking of spectra here as static entities, not considering whether they change in time or not. A spectrum can be thought of as representing the momentary pitch and timbre of an audio signal.

For discrete spectra, the pitch is primarily encoded in the frequencies of the partials. Harmonic signals have a pitch determined by their fundamental frequency; for inharmonic ones, the pitch may be clear, ambiguous, or absent altogether, according to complex and incompletely understood rules. A noisy spectrum may also have a perceptible pitch if the spectral envelope contains one or more narrow peaks.

In this model the timbre (and the loudness) are mostly encoded in the spectral envelope. The distinction between continuous and discrete spectra is also a distinction in timbre. The timbre, as well as the pitch, may evolve over the life of a sound.

This way of viewing sounds is greatly oversimplied. The true behavior of audible pitch and timbre boasts many features which can't be explained in terms of this model. For instance, the timbral quality called "roughness" is sometimes thought of as being encoded in rapid changes in the spectral envelope over time. This model is useful nonetheless in discussions about how to construct discrete

or continuous spectra for a wide variety of musical purposes, as we will begin to show in the rest of this chapter.

5.2 Multiplying audio signals

We have been routinely adding audio signals together, and multiplying them by slowly-varying signals (used as amplitude envelopes for example) since chapter 1. In order to complete our understanding of the algebra of audio signals we now consider the situation where we multiply two audio signals neither of which may be assumed to change slowly. The key to understanding what happens is the:

COSINE PRODUCT FORMULA

$$\cos(a)\cos(b) = \frac{1}{2}[\cos(a+b) + \cos(a-b)].$$

To see why this is true, recall the formula for the cosine of a sum of two angles:

$$\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$$

and evaluate the right hand side of the cosine product formula; it immediately collapses to the left hand side.

We can use this formula to see what happens when we multiply two SINUSOIDS (page 1):

$$\cos(\alpha n + \phi)\cos(\beta n + \xi) =$$

$$= \frac{1}{2}[\cos\left((\alpha+\beta)n + (\phi+\xi)\right) + \cos\left((\alpha-\beta)n + (\phi-\xi)\right)].$$

In words, multiply two sinusoids and you get a result with two partials, one at the sum of the two original frequencies, and one at their difference. (If the difference $\alpha - \beta$ happens to be negative, simply switch α and β in the formula and the difference will then be positive.)

This gives us a very easy to use tool for shifting the component frequencies of a sound, which is called *ring modulation*, which is shown in its simplest form in Figure 5.2. An oscillator provides the *modulating signal*, which is simply multiplied by the input. The term "ring modulation" is used more generally to mean multiplying any two signals together, but here we'll just consider using a sinusoidal modulating signal.

Figure 5.3 shows the result of multiplying a sinusoid of angular frequency α and amplitude a, by another of angular frequency β and amplitude 1:

$$[a\cos(\alpha n)] \cdot [\cos(\beta n)]\,.$$

(For simplicity we're omitting the phase term here.) The result is shown as a spectrum. The left-hand side sinusoid appears as a single frequency, α. The product of the two sinusoids has two component frequencies. Each of the two appears at an amplitude of $a/2$.

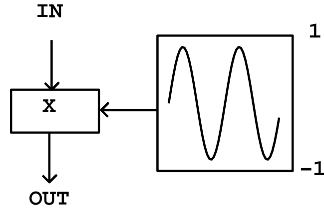


Figure 5.2: Block diagram for ring modulating an input signal with a sinusoid.

In the special case where $\alpha = \beta$, the second (difference) sideband has zero frequency and its amplitude depends on the relative phases of the two multiplicands. In this case we replace β by α in the product formula to get:

$$\cos(\alpha n + \phi)\cos(\alpha n + \xi) =$$

$$= \frac{1}{2}[\cos\left(2\alpha n + (\phi + \xi)\right) + \cos\left(\phi - \xi\right)].$$

The second term has zero frequency; its amplitude ranges from $+1/2$ to $-1/2$ as the phase difference $\phi - \xi$ varies from 0 to π radians. This situation is shown in part (c) of Figure 5.3.

We can use the distributive rule for multiplication to find out what happens when we multiply signals together which consist of more than one partial each. For example, in the situation above we can replace the signal of frequency α with a sum of several sinusoids:

$$a_1 \cos(\alpha_1 n) + \cdots + a_k \cos(\alpha_1 k).$$

Multiplying by the signal of frequency β gives partials at frequencies equal to:

$$\alpha_1 + \beta, \alpha_1 - \beta, \cdots, \alpha_n + \beta, \alpha_n - \beta.$$

As before if any frequency is negative we take its absolute value since

$$\cos(-\omega n) = \cos(\omega n),$$

so that any signal with a negative frequency is equal to one with a positive frequency.

Figure 5.4 shows the result of multiplying a complex periodic signal (with several components tuned in the ratio 0:1:2:$\cdots$) by a sinusoid. Both the spectral envelope and the component frequencies of the result transform by relatively simple rules.

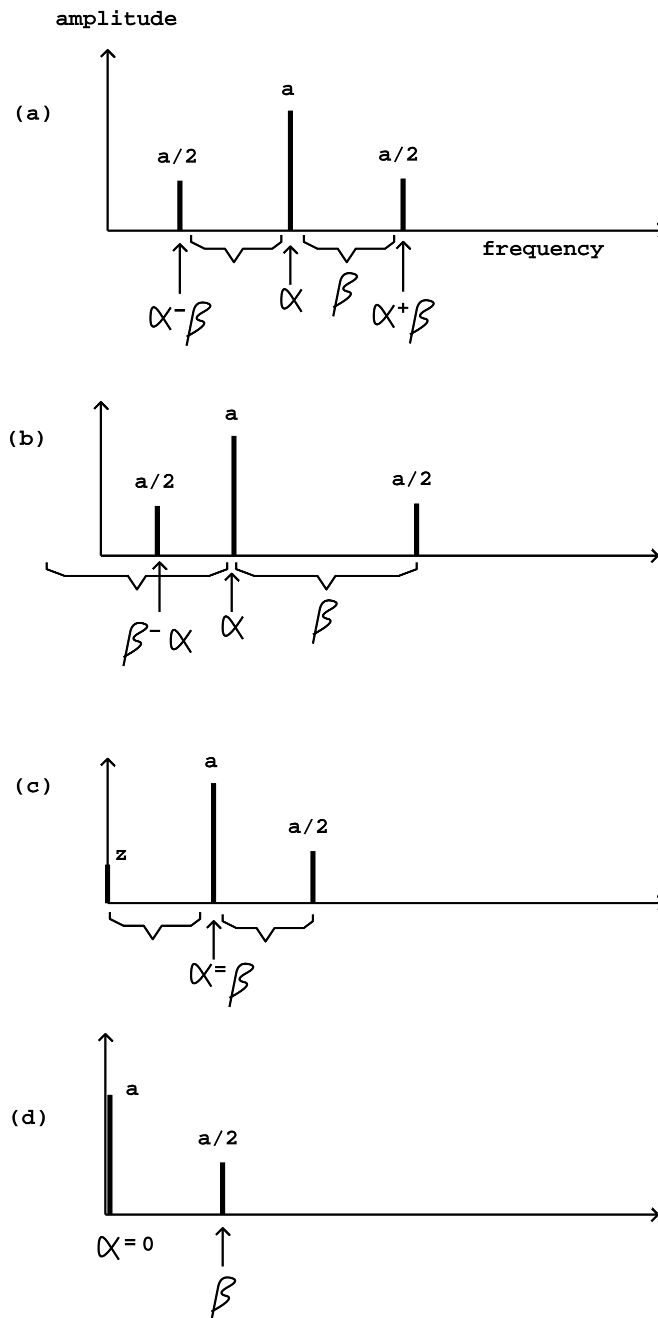


Figure 5.3: Sidebands arising from multiplying two sinusoids of frequency α and β. Part (a) shows the case where $\alpha > \beta > 0$; part (b) shows the case where $\beta > \alpha$ but $\beta < 2\alpha$, so that the lower sideband is reflected about the $f = 0$ axis. Part (c) shows the special case $\alpha = \beta$; in this one special case the amplitude of the zero-frequency sideband depends on the phases of the two sinusoids. In part (d), α is zero, so that only one sideband appears.

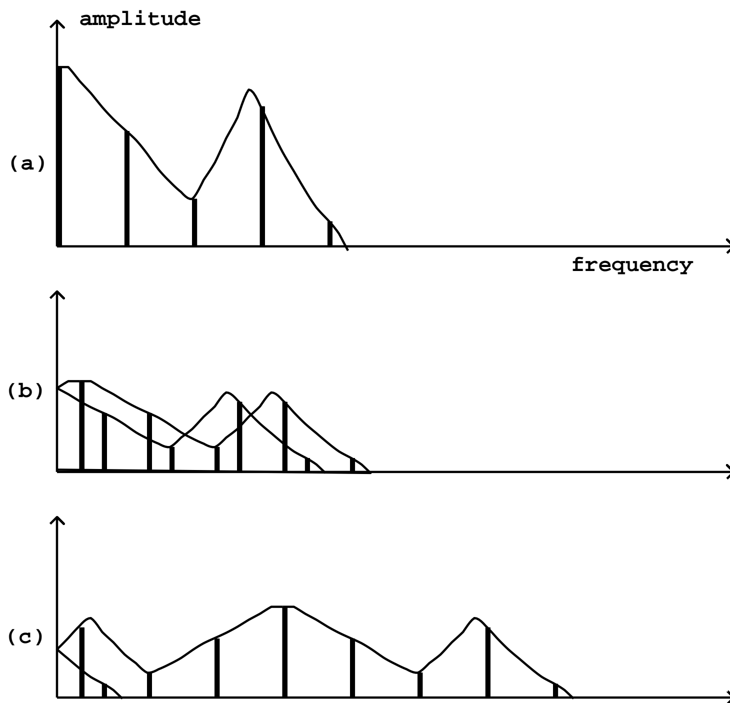


Figure 5.4: Result of ring modulation of a complex signal by a pure sinusoid: (a) the original signal's spectrum and spectral envelope; (b) modulated by a relatively low modulating frequency (1/3 of the fundamental); (c) modulated by a higher frequency, 10/3 of the fundamental.

The resulting spectrum is essentially the original spectrum combined with its reflection about the vertical axis. This combined spectrum is then shifted to the right by the modulating frequency. Finally, if any components of the shifted spectrum are still left of the vertical axis, they are reflected about it to make positive frequencies again.

In part (b) of the figure, the modulating frequency (the frequency of the sinusoid) is below the fundamental frequency of the complex signal. In this case teh amount of shifting is small, so that re-folding the spectrum at the end almost places the two halves on top of each other. The result is a spectral envelope roughly the same as the original (although half as high) and a spectrum twice as dense.

A special case, not shown, is modulation by a frequency exactly half the fundamental. In this case, pairs of partials will fall on top of each other, and will have the ratios $1/2 : 3/2 : 5/2 : \cdots$ - an odd-partial-only signal an octave below the original. This is a very simple and effective octave divider for a harmonic signal, asuming you know or can find its fundamental frequency. If you want even partials as well as odd ones (for the octave-down signal), simply mix the original signal with the modulated one.

Part (c) of the figure shows the effect of using a modulating frequency much higher than the fundamental frequency of the complex signal. Here the unfolding effect is much more clearly visible (only one partial, the leftmost one, had to be reflected to make its frequency positive.) The spectral envelope is now widely displaced from the original; this displacement is a more strongly audible effect than the relocation of partials.

In another special case, the modulating frequency may be a multiple of the fundamental of the complex periodic signal; in this case the partials all land back on other partials of the same fundamental, and the only effect is the shift in spectral envelope.

5.3 Waveshaping

Another approach to modulating a signal, called *waveshaping*, is simply to pass it through a suitably chosen nonlinear function. A block diagram for doing this is shown in Figure 5.5. The function $f()$ (called the *transfer function*) distorts the incoming waveform into a different shape. The new shape depends on the shape of the incoming wave, on the transfer function, and finally on the amplitude of the incoming signal. Since the amplitude of the input waveform affects the shape of the output waveform (and hence the timbre), this gives us an easy way to make a continuously varying family of timbres, simply by varying the input level of the transformation. For this reason, it is customary to include a leading amplitude control as part of the waveshaping operation, as shown in the block diagram.

The amplitude of the sinusoid is called the waveshaping *index*. In many situations a small index leads to relatively little distortion (hence a more nearly sinusoidal output) and a larger one gives a more distorted, hence richer, timbre.

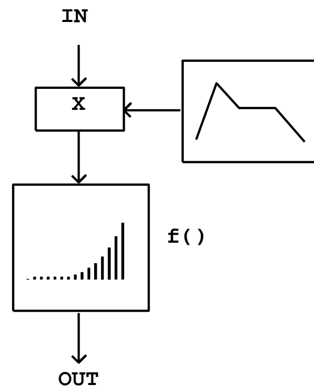


Figure 5.5: Block diagram for waveshaping an input signal using a nonlinear function $f()$. An amplitude adjustment step precedes the function lookup, to take advantage of the different effect of the wavetable lookup at different amplitudes.

Figure 5.6 shows a familiar example of waveshaping, in which the $f()$ amounts to a *clipping function*. This example shows clearly how the input amplitude—the index—can affect the output waveform. The clipping function passes its input to the output unchanged as long as it stays in the interval between -0.3 and +0.3. So when the input (in this case a sinusoid) does not exceed 0.3 in absolute value, the output is the same as the input. But when the input grows past the 0.3 limit, it is limited to 0.3; and as the amplitude of the signal increases the effect of this clipping action is progressively more severe. In the figure, the input is a decaying sinusoid. The output evolves from a nearly square waveform at the beginning to a pure sinusoid at the end. This effect will be well known to anyone who has played an instrument through an overdriven amplifier. The louder the input, the more distorted will be the output. For this reason, waveshaping is also sometimes called *distortion*.

Figure 5.7 shows a much simpler and easiest to analyse situation, in which the transfer function simply squares the input:

$$f(x) = x^2$$

For a sinusoidal input,

$$x[n] = a \cos(\omega n + \phi)$$

we get

$$f(x[n]) = \frac{a^2}{2}(1 + \cos(2\omega n + 2\phi))$$

If the amplitude a equals one, this just amounts to ring modulating the sinusoid by a sinusoid of the same frequency, whose result we described in the previous

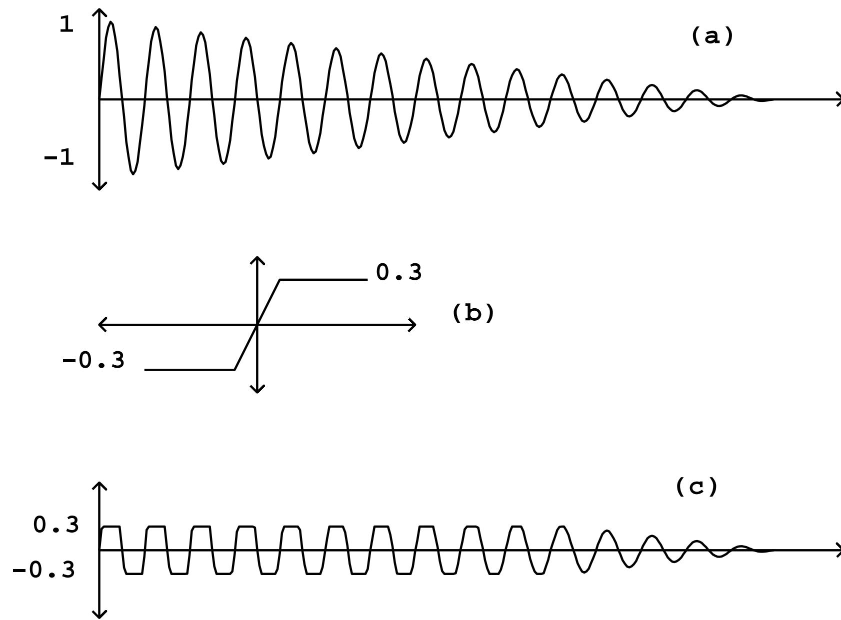


Figure 5.6: Clipping as an example of waveshaping. The input (a) is a sinusoid of varying amplitude, and the waveshaping function (b) clips its input to the interval between -0.3 and +0.3. The output is shown in (c).

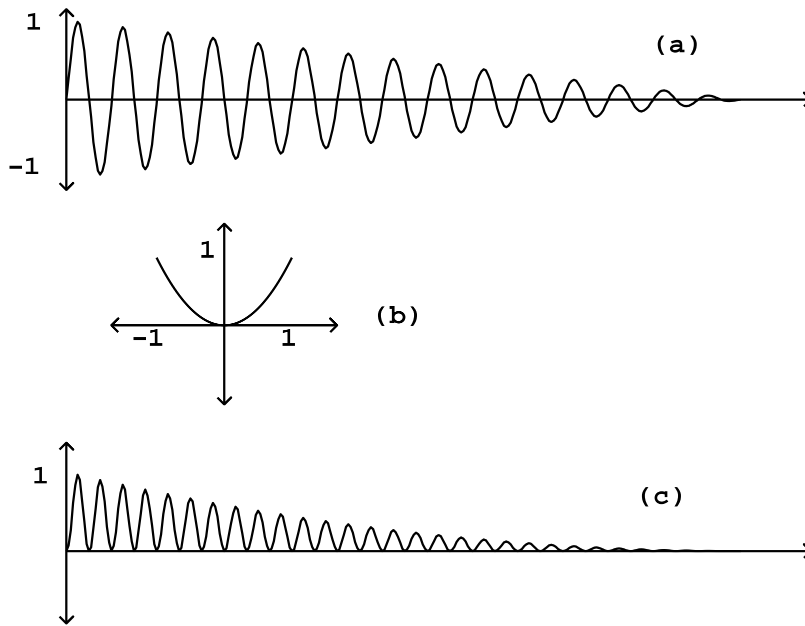


Figure 5.7: Waveshaping using a quadratic transfer function $f(x) = x^2$. The input (a) yields an output (c) at twice the frequency.

section: the output is a CD signal plus a signal at twice the original frequency. However, unlike the case of ring modulation, the amplitude of the output grows as the square of the input.

Keeping the same transfer function, we now consider the effect of sending in a combination of two sinusoids with amplitudes a and b, and angular frequencies α and β. For simplicity, we'll omit the initial phase terms. We set:

$$x[n] = a\cos(\alpha n) + b\cos(\beta n)$$

and plugging this into f gives

$$f(x[n]) = \frac{a^2}{2}(1 + \cos(2\alpha n))$$

$$+\frac{b^2}{2}(1 + \cos(2\beta n))$$

$$+ab\,(\cos((\alpha + \beta)n) + \cos((\alpha - \beta)n))$$

The first two terms are just what we would get by sending the two sinusoids through separately. The third term is the product of the two input terms, which

comes from the middle, cross term in the expansion,

$$f(x+y) = x^2 + 2xy + y^2$$

This effect, called *intermodulation*, becomes more and more dominant as the number of terms in the input increases; if there are k sinusoids in the input there are only k "straight" terms in the product, but there are $(k^2 - k)/2$ intermodulation terms.

As compared to ring modulation, which is linear in its input, waveshaping is nonlinear. While we were able to analyze linear processes by considering their action separately on all the components of the input, in this nonlinear case we also have to consider the interactions between components. The results are far more complex—sometimes sonically much richer, but, on the other hand, harder to understand or predict.

In general, we can show that a periodic input, no matter how complex, will give an output of the same periodicity. If the period is τ so that

$$x[n+\tau] = x[n]$$

then we immediately get

$$f(x[n+\tau]) = f(x[n]).$$

Combinations of periodic tones at consonant intervals give rise to distortion products at subharmonics. For instance, if two periodic signals x and y are a musical fourth apart (periods in the ratio 4:3), then the sum of the two repeats at the lower rate given by the common subharmonic. In equations we would have:

$$x[t+\tau/3] = x[t]$$
$$y[t+\tau/4] = y[t]$$

which implies

$$x[t+\tau] + y[t+\tau] = x[t] + y[t]$$

and so the distorted sum $f(x+y)$ would repeat after a period of τ:

$$f(x+y)[n+\tau] = f(x+y)[n].$$

This has been experienced by every electric guitarist who has set the amplifier to "overdrive" and played the open B and high E strings together: the distortion product is pitched the same as the low E string, two octaves below the high one.

To get a somewhat more explicit analysis of the effect of waveshaping on an incoming signal, it is sometimes useful to write the function f as a finite or infinite polynomial series:

$$f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 + \cdots$$

If the input signal $x[n]$ is a sinusoid $a\cos(\omega n)$, we can consider the action of the above terms separately:

$$f(x[n]) = f_0 + a f_1 \cos(\omega n) + a^2 f_2 \cos^2(\omega n) + a^3 f_3 \cos^3(\omega n) + \cdots$$

Since the higher order terms are multiplied by higher powers of the amplitude a, a lower value of a will weight the earlier terms more heavily, and a higher value will make the higher-order terms more prominent.

The individual terms' spectra can be found by applying the cosine product formula repeatedly:

$$1 = \cos(0)$$

$$x[n] = \cos(\omega n)$$

$$x^2[n] = \frac{1}{2} + \frac{1}{2}\cos(2\omega n)$$

$$x^3[n] = \frac{1}{4}\cos(-\omega n) + \frac{2}{4}\cos(\omega n) + \frac{1}{4}\cos(3\omega n)$$

$$x^4[n] = \frac{1}{8}\cos(-2\omega n) + \frac{3}{8}\cos(0) + \frac{3}{8}\cos(2\omega n) + \frac{1}{8}\cos(4\omega n)$$

$$x^5[n] = \frac{1}{16}\cos(-3\omega n) + \frac{4}{16}\cos(-\omega n) + \frac{6}{16}\cos(\omega n) + \frac{4}{16}\cos(3\omega n) + \frac{1}{16}\cos(5\omega n)$$

and so on. The relative weights of the components will be recognized as Pascal's triangle. They can be approximated by a Gaussian curve whose width, for the x^k term of the expansion, grows as the square root of k.

The negative-frequency terms (which have been shown separately here for clarity) are to be combined with the positive ones; the spectral envelope is folded into itself in the same way as in the ring modulation example of Figure Figure 5.4.

As long as the coefficients f_k are all positive numbers or zero, then so are all the amplitudes of the sinusoids in the expansions above. In this case all the phases stay coherent as a varies and so we get a widening of the spectrum (and possibly a drastically increasing amplitude) with increasing values of a. On the other hand, if some of the f_k are positve and others negative, the different expansions will interfere destructively; this will give a more complicated-sounding spectral evolution.

Note also that the successive expansions all contain only even or only odd partials. If the transfer function (in series form) happens to contain only even powers:

$$f(x) = f_0 + f_2 x^2 + f_4 x^4 + \cdots$$

then the result, having only even partials, will sound an octave higher than the incoming sinusoid. If only odd powers show up in the expansion of $f(x)$, then the output will contain only odd partials. Even if f can't be expressed exactly as a power series (for example, the clipping function of Figure 5.3), it is still true that if f is an even function, i.e., if

$$f(-x) = f(x)$$

you will get only even harmonics and if f is an odd function,

$$f(-x) = -f(x)$$

you get odd harmonics.

Many mathematical tricks have been proposed to use waveshaping to generate specified spectra. It turns out that you can generate pure sinusoids at any harmonic of the fundamental by using a Chebyshef polynomial as a transfer function [Leb79], and from there you can go on to build any desired static spectrum. Generating *families* of spectra by waveshaping a sinusoid of variable amplitude turns out to be trickier, although several interesting special cases have been found, one of which is developed here in chapter [?].

5.4 Frequency Modulation

If a sinusoid is given a frequency which varies slowly in time we hear it as having a varying pitch. But if the pitch changes so quickly that our ears can't track the change—for instance, if the change itself occurs at or above the fundamental frequency of the sinusoid—we hear a timbral change. The timbres so generated are rich and widely varying. The discovery by John Chowning of this possibility [Cho73] revolutionized the field of computer music. Here we develop *frequency modulation*, usually called *FM*, as a special case of waveshaping [Leb79]; the treatment here is adapted from an earlier publication [Puc01].

The FM technique, in its simplest form, is shown in figure 5.8 part (a). A frequency-modulated sinusoid is one whose frequency varies sinusoidally, at some angular frequency ω_1, about a central frequency ω_2, so that the instantaneous frequencies vary between $(1-r)\omega_2$ and $(1+r)\omega_2$, with ω_1 controlling the frequency of variation. It is customary to use a simpler, essentially equivalent formulation in which the phase of the sinusoid is modulated sinusoidally (since the instantaneous frequency is just the change of phase, and since the sample-to-sample change in a sinusoid is just another sinusoid.) The phase modulation formulation is shown in part (b) of the figure.

If the carrier and modulation frequencies don't themselves vary in time, the signal can be written as

$$x[n] = \cos(r\cos(\omega_1 n)) + \omega_2 n)$$

The parameter r, called the *index of modulation*, controls the extent of frequency variation (relative to the carrier frequency ω_2). If $r = 0$, there is no frequency variation and the expression reduces to an unmodified sinusoid:

$$x(t) = \cos(\omega_2 t)$$

We call ω_1 the modulation frequency and ω_2 the carrier frequency.

To analyse the resulting spectrum we can write,

$$x(t) = \cos(\omega_2 t) * \cos(r\cos(\omega_1 t))$$

$$- \sin(\omega_2 t) * \sin(r\cos(\omega_1 t)),$$

so we can consider it as a sum of two waveshaping generators, each operating on a sinusoid of frequency ω_1 and with a waveshaping index r, and each ring

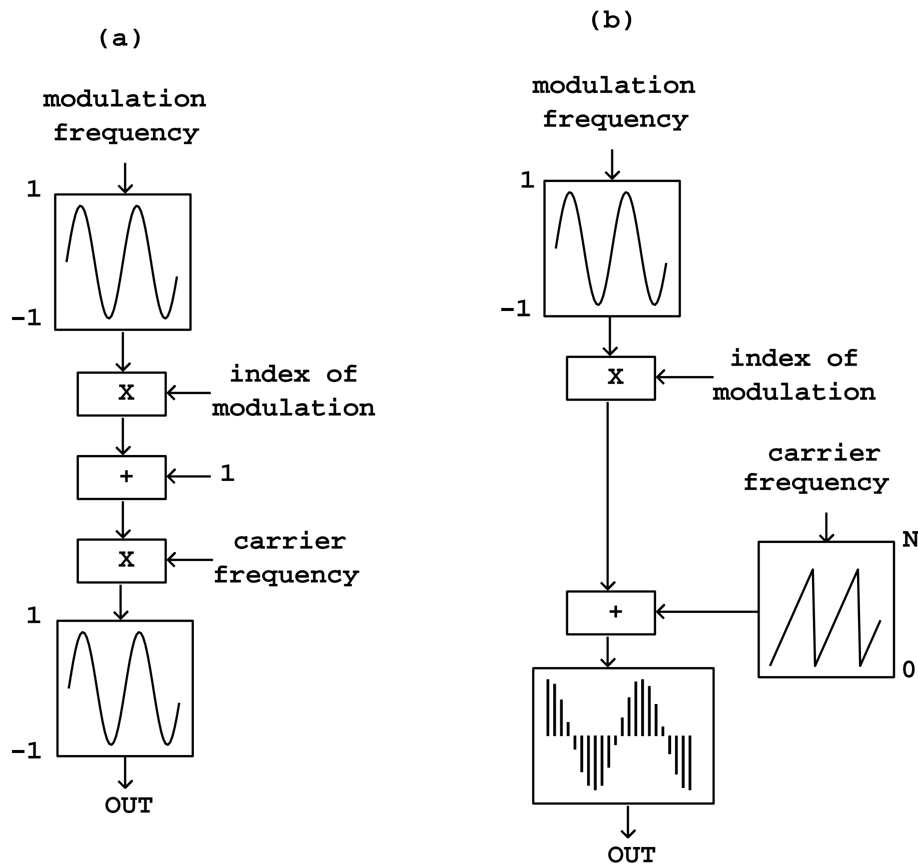


Figure 5.8: Block diagram for frequency modulation (FM) synthesis: (a) the classic form; (b) realized as phase modulation.

modulated with a sinusoid of frequency ω_2. The waveshaping function f is given by $f(x) = \cos(x)$ for the first term and by $f(x) = \sin(x)$ for the second.

Returning to Figure 5.4, we can see at a glance what the spectrum will look like. The two harmonic spectra, of the waveshaping outputs

$$\cos(r\cos(\omega_1 t))$$

and

$$\sin(r\cos(\omega_1 t))$$

have, respectively, harmonics tuned to

$$0, 2\omega_1, 4\omega_1, \cdots$$

and

$$\omega_1, 3\omega_1, 5\omega_1, \cdots$$

and each being multiplied by a sinusoid at the carrier frequency. So there willl be a spectrum centered at the carrier frequency ω_2, with sidebands at both even and odd multiples of the modulation frequency ω_1, contributed respectively by the sin and cos waveshaping terms above. The index of modulation r, as it changes, controls the relative strength of the various partials. The partials themselves are situated at the frequencies

$$\omega_2 + m\omega_1$$

where

$$m = \cdots - 2, -1, 0, 1, 2, \cdots$$

As with any situation where two periodic signals are multiplied, if there is some common supermultiple of the two periods, the resulting product will repeat at that longer period. So if the two periods are $k\tau$ and $m\tau$, where k and m are relatively prime, they both repeat after a time interval of $km\tau$. In other words, if the two have frequencies which are both multiples of some common frequency, so that $\omega_1 = k\omega$ and $\omega_2 = m\omega$, again with k and m relatively prime, the result will repeat at a frequency of the common submultiple ω. On the other hand, of no common submultiple ω can be found, or if the only submultiples are lower than any discernable pitch, then the result will be inharmonic.

Much more about FM can be found in textbooks [Moo90, p. 316] [DJ85] [Bou00] and research publications; some of the possibilities are shown in the examples.

5.5 Examples

5.5.1 Ring modulation and spectra

The first Pd example for this chapter, E01.spectrum.pd, serves to introduce a spectrum measurement tool we'll use often. We'll start with the second example,

E02.ring.modulation.pd, which shows the effect of ring modulating a harmonic spectrum (which is worked out theoretically in section 5.2 and shown in Figure 5.4). In the example we consider a signal whose harmonics (from 0 through 5) all have unit amplitude. Because signals of frequency 0 count twice as heavily (per peak amplitude) as nonzero-frequency sinusoids do, the spectral envelope peaks at DC, has a flat region from harmonics 1 through 5, and then descends to zero.

In the signal generation portion of the patch (part (a) of the figure), we sum the six partials and multiply the sum by the single, modulating oscillator. (The six signals are summed implicitly by connecting them all to the same inlet of the `*~` object.) The value of "fundamental" at the top is computed to line up well with the spectral analysis, whose result is shown in part (b) of the figure.

The spectral analysis is done using techniques which won't be described until chapter **??**, but in sum, the output shows the location of the sinusoids (assuming a discrete spectrum) on the horizontal axis and their magnitudes on the vertical one. So the presence of a peak at DC of magnitude one in the spectrum of the input signal predicts, ala figure 5.3, that there should be a peak in the output spectrum, at the modulating frequency, of height 1/2. Similarly, the two other sinusoids in the input signal, which have height 1/2 in the spectrum, give rise to two peaks each, of height 1/4, in the output. One of these four has been reflected about the left edge of the figure (the $f = 0$ axis.)

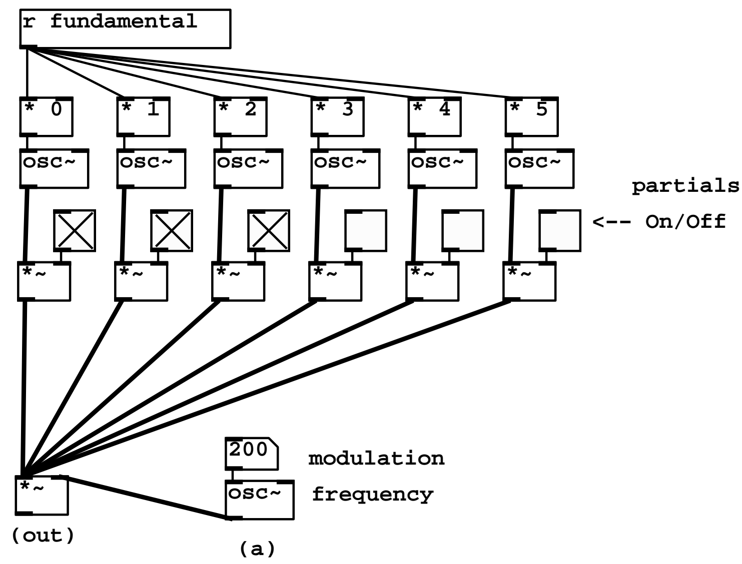
5.5.2 Octave divider and formant adder

As suggested in Section 5.2, when considering the result of modulating a complex harmonic (i.e., periodic) signal by a sinusoid, an interesting special case is to set the modulating oscillator to 1/2 the fundamental frequency, which drops the resulting sound an octave with only a relatively small deformation of the spectral envelope. Another is to modulate by a sinusoid at several times the fundamental frequency, which in effect displaces the spectral envelope without changing the fundamental frequency of the result. This is demonstrated in Patch E03.octave.divider.pd(Figure 5.10).

The subpatches `pd looper` and `pd delay` hide details. The first is a looping sampler as introduced in Chapter 2. The second is a delay of 1024 samples, which uses objects that are introduced later in chapter **??**. We will introduce one object class here.

`fiddle~`: pitch tracker. The one inlet takes a signal to analyze, and messages to change settings. Depending on its creation arguments `fiddle~`may have a variable number of outlets offering various information about the input signal. As shown here, with only one creation argument to specify window size, the third outlet attempts to report the pitch of the input, and the amplitude of that portion of the input which repeats (at least approximately) at the reported pitch. These are reported as a list of two numbers. The pitch, which is in MIDI units, is reported as zero if none could be identified.

In this patch the third outlet is unpacked into its pitch and amplitude com-



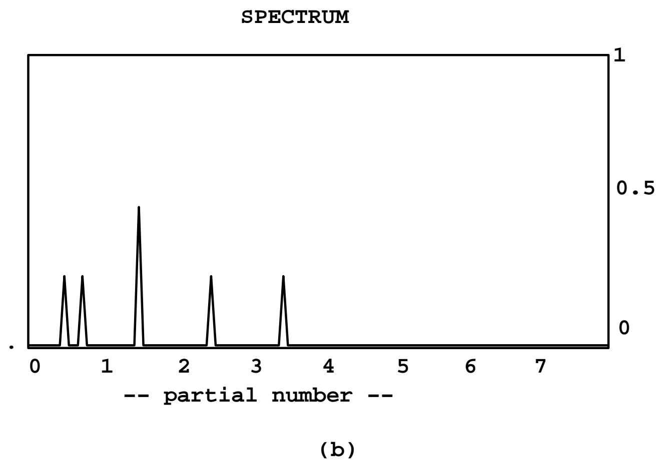


Figure 5.9: Ring modulation of a complex tone by a sinusoid: (a) its realization; (b) a measured spectrum

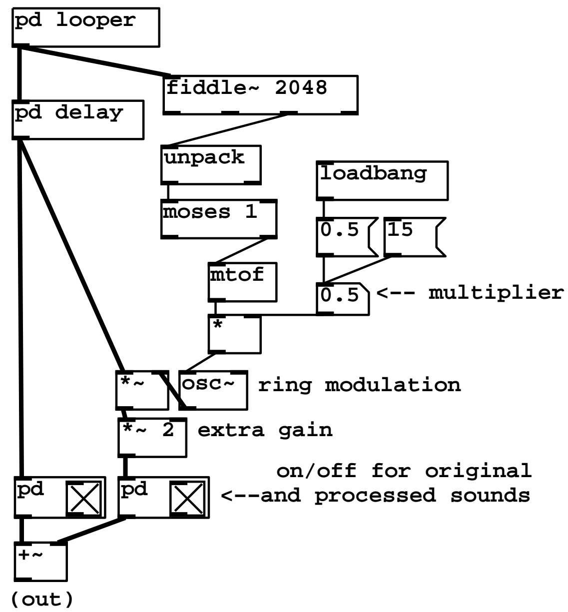


Figure 5.10: Lowering the pitch a sound by an octave by determinating its pitch and modulating at half the fundamental.

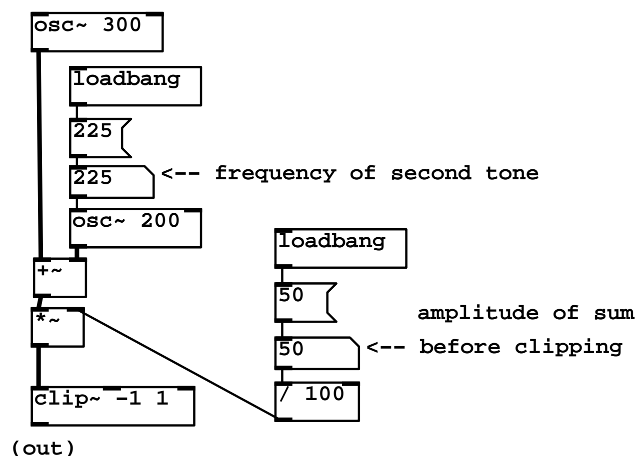


Figure 5.11: Nonlinear distortion of a sum of two sinusoids to create a difference tone.

ponents, and the pitch component is filtered by the `moses`object so that only successful pitch estimates (nonzero ones) are considered. These are converted to units of frequency by the `mtof`object. Finally, the frequency estimates are either reduced by 1/2 or else multiplied by 15, depending on the selected multiplier, to provide the modulation frequency.

5.5.3 Waveshaping and difference tones

Patch E04.difference.tone.pd(Figure 5.11) introduces waveshaping, demonstrating the nonlinearity of the process. Two sinusoids (300 and 225 Hz, or a ratio of 4 to 3) are summed and then clipped, using a new object class:

`clip~`: signal clipper. When the signal lies between the limits specified by the arguments to the `clip~` object, it is passed through unchanged; but when it falls below the lower limit or rises above the upper limit, it is replaced by the lower or upper limit, respectively. The effect of clipping a sinusoidal signal is shown graphically in Figure 5.6.

As long as the amplitude of the sum of sinusoids is less than 50 percent, the sum can't exceed one in absolute value and the `clip~` object passes the pair of sinusoids through unchanged to the output. As soon as the amplitude exceeds 50 percent, however, the nonlinearity of the `clip~` object brings forth distortion products (at frequencies $300m + 225n$ for integers m and n), all of which happening to be multiples of 75, give rise to a tone whose fundamental is 75. Seen another way, the shortest common period of the two sinudusoids is 1/75 second (which is four periods of the 300 Hx, tone and three periods of the

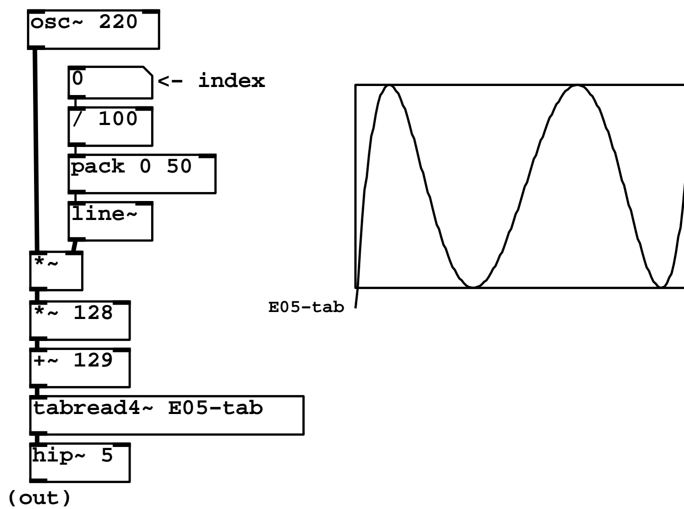


Figure 5.12: Using Chebychev polynomials as waveshaping transfer functions.

225 Hz, tone), so the result has period 1/75 second.

The 225 Hz. tone in the patch may be varied. If it is moved slightly away from 225, a beating sound results. Other values may find other common subharmonics, and still others may give rise to rich, inharmonic tones.

5.5.4 Waveshaping using Chebychev polynomials

Patch E05.chebychev.pd(Figure 5.12) demonstrates how you can use waveshaping to generate pure harmonics. We'll limit ourselves to a specific example here; for all the details see [Leb79]. In this example we would like to generate the pure fifth harmonic,

$$\cos(5\omega n)$$

by waveshaping a sinusoid

$$x[n] = \cos(\omega n)$$

We just need to find a suitable transfer function $f(x)$. Our technique is to use the formula for the waveshaping function $f(x) = x^5$ (page 125), which gives first, third and fifth harmonics; then we add a suitable multiple of x^3 to cancel the third harmonic, then a multiple of x to cancel the first harmonic:

$$16x^5 = \cos(5\omega n) + 5\cos(3\omega n) + 10\cos(\omega n)$$

$$16x^5 - 20x^3 = \cos(5\omega n) - 5\cos(\omega n)$$

$$16x^5 - 20x^3 + 5x = \cos(5\omega n)$$

and so we choose

$$f(x) = 16x^5 - 20x^3 + 5x$$

This can be done in the same way to isolate any desired harmonic; the resulting functions f are known as *Chebychev polynomials.*

To incorporate this in a waveshaping instrument, we simply build a patch that works as in Figure 5.5, computing the expression

$$x[n] = f(p[n]\cos(\omega n))$$

where $p[n]$ is a suitable *index* which may vary as a function of the sample number n.

By suitably combining Chebychev polynomials we can fix any desired superposition of components in the output waveform. But the real promise of waveshaping—that by simply changing the index we can manufacture spectra that evolve in a variety of interesting ways—is not addressed, at least directly, in the Chebychev picture.

5.5.5 Waveshaping using an exponential function

We return now to the spectra computed on Page 125, corresponding to waveshaping functions of the form $f(x) = x^k$. We note with pleasure that not only are they all in phase (so that they can be superposed with easily predictable results) but also that the spectra spread out increasingly with k. Also, in a series of the form,

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots,$$

a higher index of modulation will lend more relative weight to the higher power terms in the expansion; as we saw seen earlier, if the index of modulation is a, the terms are x^k multiplied by f_0, af_1, $a^2 f_2$, and so on.

To take the simplest possible example, suppose we wish f_0 to be the largest term for $0 < a < 1$, then for it to be overtaken by the more quickly growing af_1 term for $1 < a < 2$, which is then overtaken by the $a^2 f_2$ term for $2 < a < 3$ and so on, so that the nth term takes over at an index equal to n. To make this happen we just require that

$$a_1 = a_0, 2a_2 = a_1, 3a_3 = a_2, \cdots$$

and so fixing a_0 at 1, we get $a_1 = 1$, $a_2 = 1/2$, $a_3 = 1/6$, and in general,

$$a_k = \frac{1}{(1 \cdot 2 \cdot 3 \cdot \cdots \cdot k}$$

These are just the coefficients for the power series for the function

$$f(x) = e^x$$

where $e \approx 2.7$ is Euler's constant.

Before plugging in e^x as a transfer function it's wise to plan how we will deal with signal amplitude, since e^x grows quickly as a function of x. If we're going to plug in a sinusoid of amplitude a, the maximum output will be e^a, occuring whenever the phase is zero. A simple and natural choice is simply to divide by e^a to reduce the peak to one, giving:

$$\frac{f(a\cos(\omega n))}{e^a} = e^{a(\cos(\omega n)-1)}$$

This is realized in Patch E06.exponential.pd. Resulting spectra for $a = 0$, 4, and 16 are shown in Figure 5.13. As the waveshaping index rises, progressively less energy is present in the fundamental; the energy is increasingly spread over the partials.

5.5.6 Sinusoidal waveshaping: evenness and oddness

Another interesting class of waveshaping transfer functions is the sinusoids:

$$f(x) = \cos(x + \phi)$$

which include the cosine and sine functions (by choosing $\phi = 0$ and $\phi = -\pi/2$, respectively.) These functions, one being even and the other odd, give rise to even and odd harmonic spectra:

$$\cos(a\cos(\omega n)) = J_0(a) - 2J_2(a)\cos(2\omega n) + 2J_4(a)\cos(4\omega n) - 2J_6(a)\cos(6\omega n) \pm \cdots$$

$$\sin(a\cos(\omega n)) = 2J_1(a)\cos(\omega n) - 2J_3(a)\cos(3\omega n) + 2J_5(a)\cos(5\omega n) \mp \cdots$$

The functions $J_k(a)$ are the Bessel functions of the first kind, which engineers sometimes use to solve problems about vibrations or heat flow on discs. For other values of ϕ, we can expand the expression for f:

$$f(x) = \cos(x)\cos(\phi) - \sin(x)\sin(\phi)$$

so the result is a mix between the even and the odd harmonics, with ϕ controlling the relative amplitudes of the two. This is demonstrated in Patch E07.evenodd.pd, shown in Figure 5.14.

5.5.7 Phase modulation (also known as FM

Patch E08.phase.mod.pd, shown in Figure 5.15, shows how true frequency modulation (part a) differs from phase modulation (shown in part b). To accomplish phase modulation, the carrier oscillator is split into its phase and cosine lookup components. The signal is of the form

$$x[t] = \cos(\omega_c n + a\cos(\omega_m n))$$

where ω_c is the carrier frequency, ω_m is the modulation frequency, and a is the index of modulation—all in angular units.

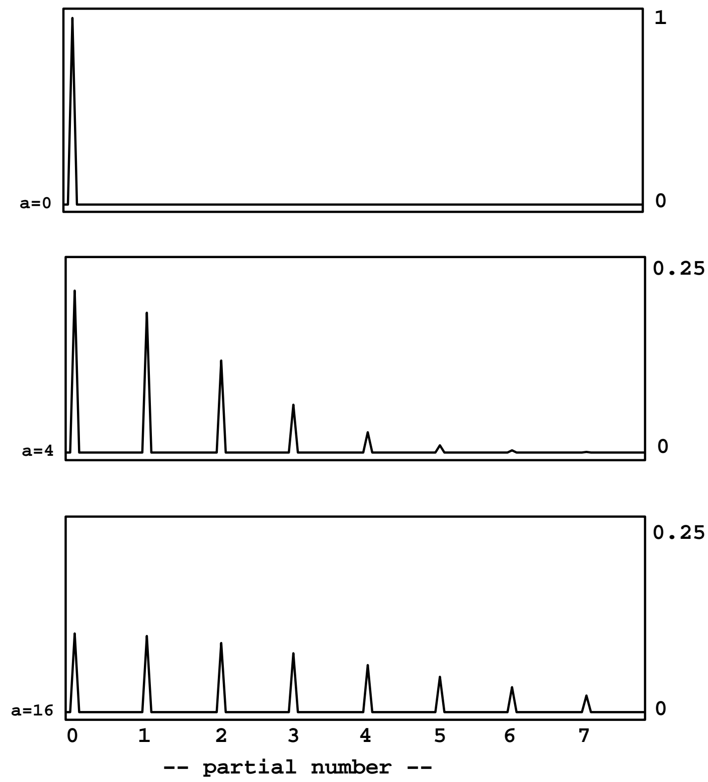


Figure 5.13: Spectra of waveshaping output using an exponential transfer function. Indices of modulation of 0, 4, and 16 are shown; note the different vertical scales.

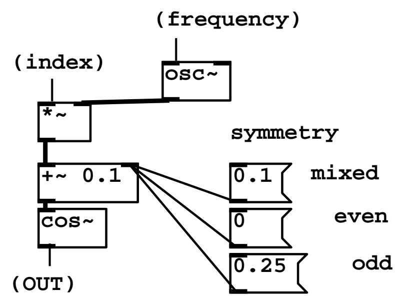


Figure 5.14: Using an additive offset to a cosine transfer function to alter the symmetry between even and odd. With no offset the symmetry is even. For odd symmetry, a quarter cycle is added to the phase. Smaller offsets give a mixture of even and odd.

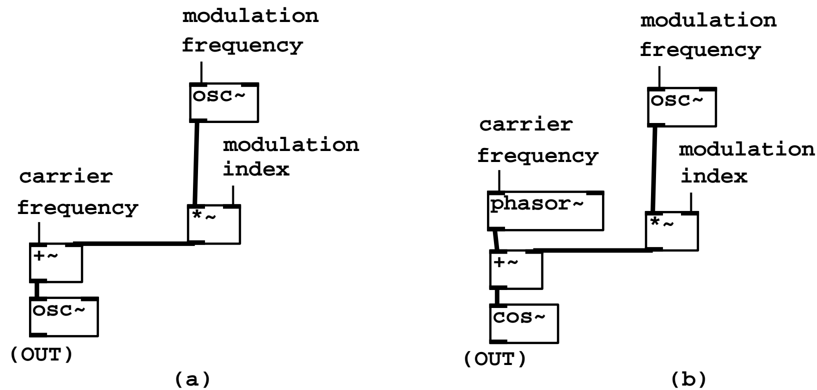


Figure 5.15: Frequency modulation (a) and phase modulation (b) compared.

We can predict the spectrum by expanding the outer cosine:

$$x[t] = \cos(\omega_c n)\cos(a\cos(\omega_m n)) - \sin(\omega_c n)\sin(a\cos(\omega_m n))$$

Plugging in the expansions from example 5.5.6 and simplifying yields:

$$x[t] = J_0(a)\cos(\omega_c n)$$

$$+J_1(a)\cos((\omega_c + \omega_m)n + \frac{\pi}{2}) + J_1(a)\cos((\omega_c - \omega_m)n + \frac{\pi}{2})$$

$$+J_2(a)\cos((\omega_c + 2\omega_m)n + \pi) + J_2(a)\cos((\omega_c - 2\omega_m)n + \pi)$$

$$+J_3(a)\cos((\omega_c + 3\omega_m)n + \frac{3\pi}{2}) + J_3(a)\cos((\omega_c - 3\omega_m)n + \frac{3\pi}{2}) + \cdots$$

So the components are centered about the carrier frequency ω_c with sidebands extending in either direction, each spaced ω_m from the next. The amplitudes are functions of the index of modulation, and don't depend on the frequencies. Figure 5.16 shows some two-operator FM spectra, measured using Patch E09.FM.spectrum.pd.

Although FM is thus just a form of ring modulated waveshaping, a confusing name change has taken place: it is FM's *carrier* frequency that appears as the ring *modulation* frequency. Keeping this in mind, we can use the strategies described in section 5.2 to generate particular combinations of frequencies. For example, if the FM carrier frequency is half the FM modulation frequency, you get a sound with odd harmonics exactly as in the octave dividing example (5.5.2).

Returning to the spectra shown in Figure 5.16, we can imagine superposing several of these (sharing a fundamental (modulating) frequency but with carriers tuned to different harmonics) to build designer spectra with energy peaks, called *formants*, at chosen locations, imitating the production of the human voice. This was used to great effect by Chowning [Cho89] in his piece, *Phoné*.

Frequency modulation need not be restricted to purely sinusoidal carrier or modulation oscillators. One well-trodden path is to effect phase modulation on the FM spectrum itself. There are then two indices of modulation (call them a and b) and two frequencies of modulation (ω_m and ω_p) and the waveform is:

$$x[n] = \cos(\omega_c n + a\cos(\omega_m n) + b\cos(\omega_p n))$$

To analyze the result, just rewrite the original FM series above, replacing $\omega_c n$ everywhere with $\omega_c n + b\cos(\omega_p n)$. The third positive sideband becomes for instance:

$$J_3(a)\cos((\omega_c + 3\omega_m)n + \frac{3\pi}{2} + b\cos(\omega_p n))$$

This is itself just another FM spectrum, with its own sidebands of frequency

$$\omega_c + 3\omega_m + k\omega_p, k = 0, \pm 1, \pm 2, \cdots$$

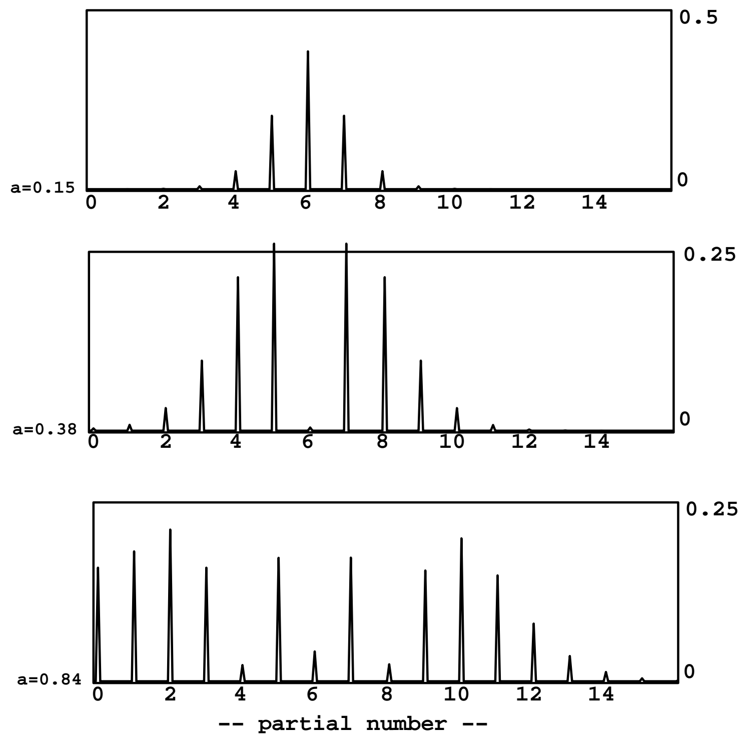


Figure 5.16: Spectra from phase modulation at three different indices. The indices are as multiples of 2π radians.

having amplitude $J_3(a)J_k(b)$ and phase $(3+k)\pi/2$ [Leb77]. Patch E10.complex.FM.pd (not shown here) illustrates this by graphing spectra from a two-modulator FM instrument.

Since early times [Sch77] researchers have sought combinations of phases, frequencies, and indices that manage to imitate familiar instrumental sounds; this became a major industry with the introduction of the Yamaha DX7 synthesizer.

Exercises

1. A sound has fundamental 440. How could it be ring modulated to give a tone at 110 Hz with only odd partials? How could you then fill in the even ones?

2. What carrier and modulation frequencies would you give a two-operator FM instrument to give frequencies of 618, 1000, and 2618 Hz? (This is a prominent feature of Chowning's *Stria* [DJ85].)

3. Suppose you wanted to make FM yet more complicated by modulating the *modulating* oscillator, as in:

$$\cos(\omega_c n + a\cos(\omega_m n + b\cos(\omega_p n)))$$

   How would the spectrum differ from that of the simple two-modulator example (section 5.5.7)?

Index

* ~, 17
cos ~, 48
delay, del, 71
env ~, 78
ftom, 20
hip ~, 46
line, 74
line ~, 19
moses, 73
mtof, 20
notein, 79
osc ~, 15
pipe, 71
receive, 20
r, 20
samphold ~, 50
select, sel, 73
send, 21
snapshot ~, 78
stripnote, 79
s, 21
tabosc4 ~, 43
tabread4 ~, 44
trigger, 79
t, 79
vline ~, 76
expr, 50
inlet, 96
inlet ~, 96
loadbang, 50
outlet, 96
outlet ~, 96
pack, 44
receive ~, 50
r ~, 50
send ~, 50
s ~, 50
tabwrite ~, 44
wrap ~, 50
catch~, 101
clip~, 132
pddiv, 107
fiddle~, 129
makenote, 107
pdmod, 107
poly, 107
throw~, 104
pdunpack, 101

abstraction (Pd), 96
additive synthesis, 12
    examples, 101, 104
ADSR envelope generator, 83
aliasing, 56
amplitude, 1
amplitude, measures of, 3
amplitude, peak, 3
amplitude, RMS, 3
arguments
    creation, 14
audio signals, digital, 1

box, 13

- GUI, 14
- message, 13
- number, 14
- object, 14

cents, 11
Chebychev polynomials, 134
class, 14
clipping, 23
clipping function, 121
continuous spectrum, 115
control, 57
control stream, 59
- numeric, 60

creation arguments, 14

dac~ object, 17
decibels, 4
delay
- compound, 70
- in Pd, 71
- on control streams, 70
- simple, 70

detection
- of events, 65

digital audio signals, 1
discrete spectrum, 113
distortion, 121
duty cycle, 36
dynamic, 5

envelope generator, 8, 83
- ADSR, 83
- resetting, 90

event, 60

foldover, 56
formant, 138
frequency modulation, 126
frequency, angular, 1
fundamental, 11

granular synthesis, 31
GUI box, 14

half step, 10
harmonic spectrum, 113
harmonics, 11

index (waveshaping), 120, 134
index of modulation, 126
inharmonic spectrum, 115
intermodulation, 124

logical time, 57

merging control streams, 70
- in Pd, 73

message box, 13
messages, 14, 71
MIDI, 10
modulation
- frequency, 126
- ring, 116

muting, 89

noisy spectrum, 115
number box, 14
numeric control stream
- in Pd, 71

Nyquist theorem, 55

object box, 14
octave, 10
oscillator, 6

partials, 115
patch, 6, 13
period, 11
polyphony, 92
power, 3
pruning control streams, 71
- in Pd, 73

real time, 57
resynchronizing control streams, 71
- in Pd, 73

ring modulation, 116

sample number, 1
sample rate, 1
sampling
- examples, 107

sawtooth wave, 24

signals, digital audio, 1
spectral envelope, 115
spectrum, 113
subpatches, 96
switch-and-ramp technique, 90

tags, 95
tasks, 92
timbre stretching, 35
time sequence, 59
transfer function, 88, 120
transient generator, 83

unit generators, 6

voice bank, 92

waveshaping, 120
wavetable lookup, 23
    noninterpolating, 24
wavetables
    transposition formula for looping, 29
    transposition formula, momentary, 30
window, 2

Bibliography

[Bal03] Mark Ballora. *Essentials of Music Technology.* Prentice Hall, Upper Saddle River, New Jersey, 2003.

[Bou00] Richard Boulanger, editor. *The Csound book.* MIT Press, Cambridge, Massachusetts, 2000.

[Cho73] John Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society*, 21(7):526–534, 1973.

[Cho89] John Chowning. Frequency modulation synthesis of the singing voice. In Max V. Mathews and John R. Pierce, editors, *Current Directions in Computer Music Research*, pages 57–64. MIT Press, Cambridge, 1989.

[DJ85] Charles Dodge and Thomas A. Jerse. *Computer music : synthesis, composition, and performance.* Schirmer, New York, 1985.

[Har87] William M. Hartmann. Digital waveform generation by fractional addressing. *Journal of the Acoustical Society of America*, 82:1883–1891, 1987.

[Leb77] Marc Lebrun. A derivation of the spectrum of fm with a complex modulating wave. *Computer Music Journal*, 1(4):51–52, 1977.

[Leb79] Marc Lebrun. Digital waveshaping synthesis. *Journal of the Audio Engineering Society*, 27(4):250–266, 1979.

[Mat69] Max V. Mathews. *The Technology of Computer Music.* MIT Press, Cambridge, Massachusetts, 1969.

[Moo90] F. Richard Moore. *Elements of Computer Music.* Prentice Hall, Englewood Cliffs, second edition, 1990.

[Puc01] Miller S. Puckette. Synthesizing sounds with specified, time-varying spectra. In *Proceedings of the International Computer Music Conference*, pages ??–??, Ann Arbor, 2001. International Computer Music Association.

[RMW02] Thomas D. Rossing, F. Richard Moore, and Paul A. Wheeler. *The Science of Sound*. Addison Wesley, San Francisco, third edition, 2002.

[Roa01] Curtis Roads. *Microsound*. MIT Press, Cambridge, Massachusetts, 2001.

[Sch77] Bill Schottstaedt. Simulation of natural instrument tones using frequency modulation with a complex modulating wave. *Computer Music Journal*, 1(4):46–50, 1977.

[Str85] John Strawn, editor. *Digital Audio Signal Processing*. William Kaufmann, Los Altos, California, 1985.