Theory and Techniques of Electronic Music

DRAFT: March 3, 2006

Miller Puckette

Copyright ©2006 Miller Puckette

All rights reserved.

Contents

1 Sinusoids, amplitude and frequency **3**
1.1 Measures of Amplitude . . . 5
1.2 Units of Amplitude . . . 6
1.3 Controlling Amplitude . . . 7
1.4 Frequency . . . 8
1.5 Synthesizing a Sinusoid . . . 9
1.6 Superposing Signals . . . 12
1.7 Periodic Signals . . . 13
1.8 About the Software Examples . . . 14
Quick Introduction to Pd . . . 17
How to find and run the examples . . . 19
1.9 Examples . . . 19
Constant amplitude scaler . . . 19
Amplitude control in decibels . . . 21
Smoothed amplitude control with an envelope generator . . . 23
Major triad . . . 24
Conversion between frequency and pitch . . . 24
More additive synthesis . . . 25
Exercises . . . 27

2 Wavetables and samplers **29**
2.1 The Wavetable Oscillator . . . 31
2.2 Sampling . . . 35
2.3 Enveloping samplers . . . 37
2.4 Timbre stretching . . . 41
2.5 Interpolation . . . 45
2.6 Examples . . . 49
2.6.1 wavetable oscillator . . . 49
2.6.2 wavetable lookup in general . . . 50
2.6.3 using a wavetable as a sampler . . . 52
2.6.4 looping samplers . . . 54
2.6.5 Overlapping sample looper . . . 56
2.6.6 Automatic read point precession . . . 58

3 Audio and control computations **61**
3.1 The sampling theorem . . . 61
3.2 Control . . . 63
3.3 Control streams . . . 65
3.4 Converting from audio signals to numeric control streams . . . 69
3.5 Control streams in block diagrams . . . 70
3.6 Event detection . . . 71
3.7 Control computation using audio signals directly . . . 73
3.8 Operations on control streams . . . 75
3.9 Control operations in Pd . . . 77
3.10 Examples . . . 79
3.10.1 Sampling and foldover . . . 79
3.10.2 Converting controls to signals . . . 81
3.10.3 Non-looping sample player . . . 82
3.10.4 Signals to controls . . . 84
3.10.5 Analog-style sequencer . . . 84
3.10.6 MIDI-style synthesizer . . . 86

4 Automation and voice management **89**
4.1 Envelope Generators . . . 89
4.2 Linear and Curved Amplitude Shapes . . . 92
4.3 Continuous and discontinuous control changes . . . 94
4.3.1 Muting . . . 95
4.3.2 Switch-and-ramp . . . 96
4.4 Polyphony . . . 98
4.5 Voice allocation . . . 98
4.6 Voice tags . . . 99
4.7 Encapsulation in Pd . . . 102
4.8 Examples . . . 103
4.8.1 ADSR envelope generator . . . 103
4.8.2 Transfer functions for amplitude control . . . 106
4.8.3 Additive synthesis: Risset's bell . . . 107
4.8.4 Additive synthesis: spectral envelope control . . . 110
4.8.5 Polyphonic synthesis: sampler . . . 113

5 Modulation **119**
5.1 Taxonomy of spectra . . . 119
5.2 Multiplying audio signals . . . 122
5.3 Waveshaping . . . 126
5.4 Frequency and phase modulation . . . 132
5.5 Examples . . . 135
5.5.1 Ring modulation and spectra . . . 135
5.5.2 Octave divider and formant adder . . . 137
5.5.3 Waveshaping and difference tones . . . 138
5.5.4 Waveshaping using Chebychev polynomials . . . 139
5.5.5 Waveshaping using an exponential function . . . 140

5.5.6 Sinusoidal waveshaping: evenness and oddness 141
5.5.7 Phase modulation and FM 143

6 Designer spectra 147
6.1 Carrier/modulator model 148
6.2 Pulse trains 151
6.2.1 Pulse trains via waveshaping 151
6.2.2 Pulse trains via wavetable stretching 152
6.2.3 Resulting spectra 152
6.3 Movable ring modulation 156
6.4 Phase-aligned formant (PAF) generator 158
6.5 Examples 163
6.5.1 Wavetable pulse train 163
6.5.2 Simple formant generator 166
6.5.3 Two-cosine carrier signal 167
6.5.4 The PAF generator 168
6.5.5 Stretched wavetables 172

7 Time shifts 173
7.1 Complex numbers 174
7.1.1 Sinusoids as geometric series 176
7.2 Time shifts and phase changes 178
7.3 Delay networks 178
7.4 Recirculating delay networks 183
7.5 Power conservation and complex delay networks 187
7.6 Artificial reverberation 192
7.6.1 Controlling reverberators 194
7.7 Variable and fractional shifts 196
7.8 Accuracy and frequency response of interpolating delay lines . . 199
7.9 Pitch shifting 200
7.10 Examples 206
7.10.1 Fixed, noninterpolating delay line 206
7.10.2 Recirculating comb filter 207
7.10.3 Variable delay line 208
7.10.4 Order of execution and lower limits on delay times 209
7.10.5 Order of execution in non-recirculating delay lines 211
7.10.6 Non-recirculating comb filter as octave doubler 213
7.10.7 Time-varying complex comb filter: shakers 214
7.10.8 Reverberator 216
7.10.9 Pitch shifter 216
7.10.10 Exercises 219

8 Filters **221**
8.1 Taxonomy of filters . . . 222
8.1.1 Low-pass and high-pass filters . . . 222
8.1.2 Band-pass and stop-band filters . . . 224
8.1.3 Equalizing filters . . . 224
8.2 Designing filters . . . 227
8.2.1 Elementary non-recirculating filter . . . 227
8.2.2 Non-recirculating filter, second form . . . 228
8.2.3 Elementary recirculating filter . . . 231
8.2.4 Compound filters . . . 231
8.2.5 Real outputs from complex filters . . . 232
8.3 Designing filters . . . 234
8.3.1 One-pole low-pass filter . . . 234
8.3.2 One-pole, one-zero high-pass filter . . . 235
8.3.3 Shelving filter . . . 236
8.3.4 Band-pass filter . . . 238
8.3.5 Peaking and band-stop filter . . . 238
8.3.6 Butterworth filters . . . 239
8.3.7 Stretching the unit circle with rational functions . . . 241
8.3.8 Butterworth band-pass filter . . . 244
8.3.9 Time-varying coefficients . . . 245
8.3.10 Impulse responses of recirculating filters . . . 247
8.3.11 All-pass filters . . . 247
8.4 Applications . . . 250
8.4.1 Subtractive synthesis . . . 250
8.4.2 Envelope following . . . 250
8.4.3 Single Sideband Modulation . . . 253
8.5 Examples . . . 255
8.5.1 Prefabricated low-, high-, and band-pass filters . . . 255
8.5.2 Prefabricated time-variable band-pass filter . . . 256
8.5.3 Envelope followers . . . 258
8.5.4 Single sideband modulation . . . 258
8.5.5 Using elementary filters directly: shelving and peaking . . . 261
8.5.6 Making and using all-pass filters . . . 261

9 Fourier analysis and resynthesis **265**
9.1 Fourier analysis of periodic signals . . . 265
9.1.1 Fourier transform as additive synthesis . . . 267
9.1.2 Periodicity of the Fourier transform . . . 267
9.2 Properties of Fourier transforms . . . 267
9.2.1 Fourier transform of DC . . . 268
9.2.2 Shifts and phase changes . . . 269
9.2.3 Fourier transform of a sinusoid . . . 271
9.3 Fourier analysis of non-periodic signals . . . 272
9.4 Fourier analysis and reconstruction of audio signals . . . 275
9.4.1 Narrow-band companding . . . 277

9.4.2 Timbre stamping (classical vocoder) 279
9.5 Phase 281
9.5.1 Phase relationships between channels 285
9.6 Phase bashing 286
9.7 Examples 288

10 Classical waveforms 297
10.1 Symmetries and Fourier series 299
10.1.1 Sawtooth waves and symmetry 300
10.2 Decomposing the classical waveforms into sawtooth and parabolic waves 302
10.3 Fourier series of the elementary waveforms 304
10.4 Predicting and controlling foldover 309
10.4.1 Oversampling 309
10.4.2 Sneaky triangle waves 310
10.4.3 Transition splicing 311
10.5 Examples 314

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 material 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 not needed. 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.

Chapter 1

Sinusoids, amplitude and frequency

Electronic music is made by synthesizing or 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*. (Some writers use the word "sample" differently, to mean "recorded sound"—but in this book we'll only use the word to mean a single number.) An example of a digital audio signal is the *S*inusoid:

Sinusoid

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

where a is the *amplitude*, ω is the *angular frequency*, and ϕ is the initial *phase*. The phase is a function of the sample number n, equal to $\phi + \omega n$. The initial phase is the phase at the zeroth sample ($n = 0$).

Strictly speaking, this is the *real Sinusoid*, distinct from the *complex Sinusoid* introduced later in chapter 7; 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. In part (a) of the figure, the horizontal axis shows successive values of n and the vertical axis shows the corresponsding values of $x[n]$. The graph is drawn in such a way as to emphasize the sampled nature of the signal. Alternatively, we could draw it more simply as a continuous curve (part b). The upper drawing is the most faithful representation of the (digital audio) Sinusoid, whereas the lower one can be considered as idealization of it.

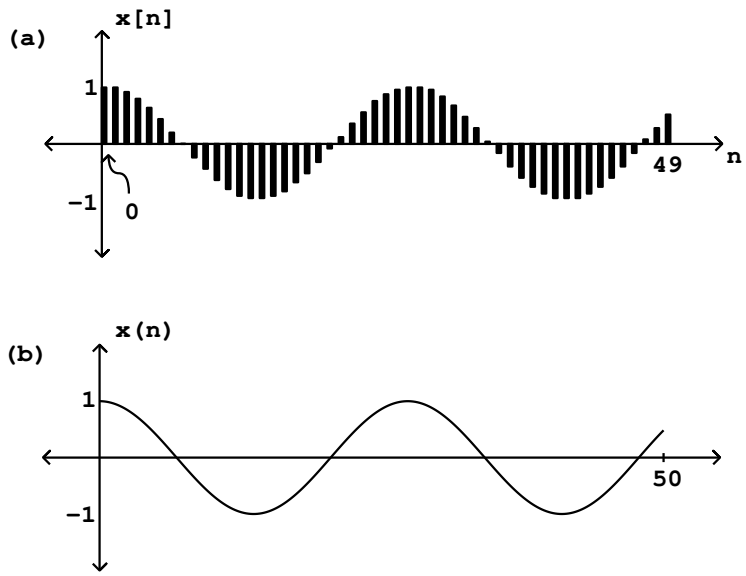


Figure 1.1: A digital audio signal, showing its discrete-time nature (part a), and idealized as a continuous function (part b). This signal is a (real-valued) Sinusoid, fifty points long, with amplitude 1, angular frequency 0.24, and initial phase zero.

Sinusoids play a key role in audio processing because, if you shift one of them left or right by any number of samples, you get another one. This makes it easy to calculate the effect of all sorts of operations on sinusoids. Our ears use this same special property to help us parse incoming sounds, which is why sinusoids, and combinations of sinusoids, 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 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 numbers. We'll casually assume here that there is ample numerical accuracy so that we can ignore round-off errors, 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, most often between -1 and 1. Modern digital audio processing software usually uses a floating-point representation for signals. This allows us to use whatever units are most 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 $P\{x[n]\}$ 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 at the moment since we're working on real-valued signals, but they will become important later when we consider complex-valued signals.) Neither the peak nor the RMS amplitude of any signal can be negative, and either one can be exactly zero only if the signal itself is zero for all n in the window.

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 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 the amplitude of a signal (either peak or RMS, as defined above), then we can define the decibel (dB) level 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. The relationship between linear amplitude and amplitude in decibels is graphed in Figure 1.2.

Still using a_0 to denote the 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

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

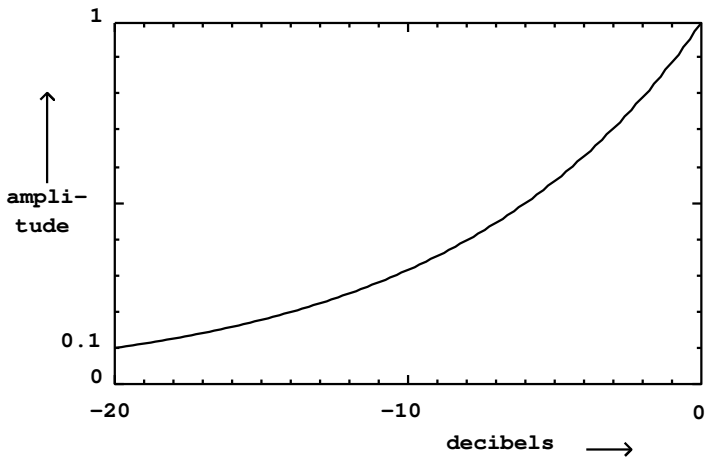


Figure 1.2: The relationship between Decibel and linear scales of amplitude. The linear amplitude 1 is assigned to 0 dB.

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 the 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 nature of human hearing (and other senses), which may partially explain why decibels are such a useful scale of amplitude[RMW02, p. 99].

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. It ranges 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].

1.3 Controlling Amplitude

Perhaps the most frequently used operation on electronic sounds is to change their amplitudes. For example, a simple strategy for synthesizing sounds is by combining Sinusoids, which can be generated by evaluating the formula on page 3, sample by sample. But the 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 factor $y \geq 0$,

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.

1.4 Frequency

Frequencies, like amplitudes, are often described on a logarithmic scale, in order to emphasize proportions between them, which usually provide a better description of the relationship between frequencies than do differences between them. 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 pitch", in which the pitch 69 is assigned to a frequency of 440 cycles per second—the A above middle C. To convert between MIDI pitch and frequency in cycles per second, apply the formulas:

Pitch/Frequency Conversion

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

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

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

MIDI itself is an old hardware protocol which has unfortunately insinuated itself into a great deal of software design. In hardware, MIDI allows only integer pitches between 0 and 127. However, the underlying scale is well defined for any "MIDI" number, even negative ones; for example a "MIDI pitch" of -4 is a decent rate of vibrato. The pitch scale cannot, however, describe frequencies less than or equal to zero cycles per second. (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

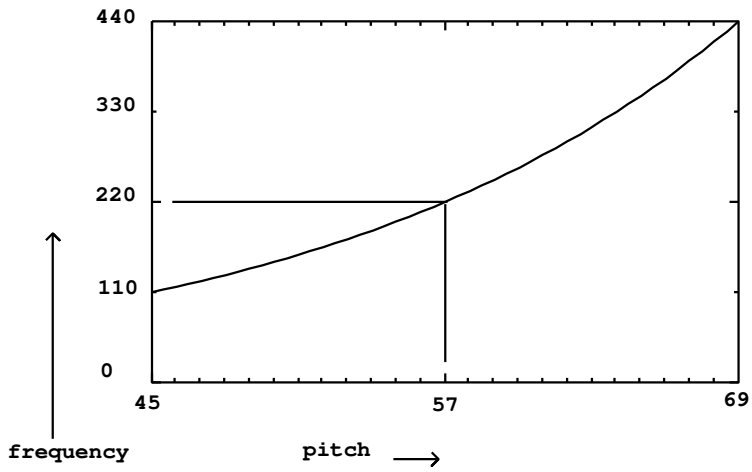


Figure 1.3: The relationship between "MIDI" pitch and frequency in cycles per second (Hertz). The span of 24 MIDI values on the horixontal axis represents two octaves, over which the output increases by a factor of four.

one hundredth of a half step. As a rule of thumb, it might take about three cents to make a discernable change in the pitch of a musical tone. At middle C this comes to a difference of about 1/2 cycle per second. A graph of frequency as a function of MIDI pitch, over a two-octave range, is shown in Figure 1.3.

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*[Mat69] 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 (page 19), we'll to choose a specific 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 made with an *oscillator*. Figure 1.4(a) shows the icon for a sinusoidal oscillator. The input is a frequency (in cycles per second), and the output is a Sinusoid of peak amplitude one.

Figure 1.4(b) shows how to multiply the output of a sinusoidal oscillator by an appropriate amplitude scaler $y[n]$ to control its amplitude. Since the

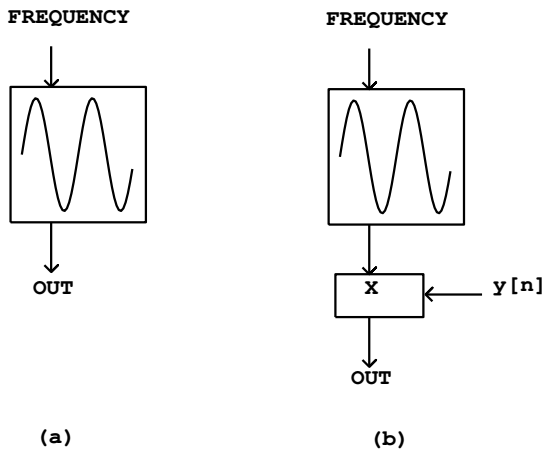


Figure 1.4: Block diagrams for (a) a sinusoidal oscillator; (b) controlling the amplitude using a multiplier and an amplitude signal $y[n]$.

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 1.5 shows how the Sinusoid of Figure 1.1 is affected by amplitude change by two different controlling signals $y[n]$. The controlling signal shown in part (a) has a discontinuity, and so therefore does the resulting amplitude-controlled sinusoid shown in (b). Parts c and d show a more gently-varying possibility for $y[n]$ and the result. Intuition suggests that the result shown in (b) won't sound like an amplitude-varying sinusoid, but instead like a sinusoid interrupted by a fairly loud "pop" after which it continues 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 that "pop") than are abruptly changing ones.

For now we can state two general rules without justifying them. First, pure Sinusoids are the class of signals most sensitive to the parasitic effects of quick amplitude change. So when you want to test an amplitude transition, if it works for Sinusoids it will probably work for other signals as well. 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 a complicated way on listening levels and the acoustic context.

Suitable amplitude control functions $y[n]$ may be obtained using an *envelope generator*. Figure 1.6 shows a network in which an envelope generator is used to control the amplitude of an oscillator. Envelope generators vary widely in

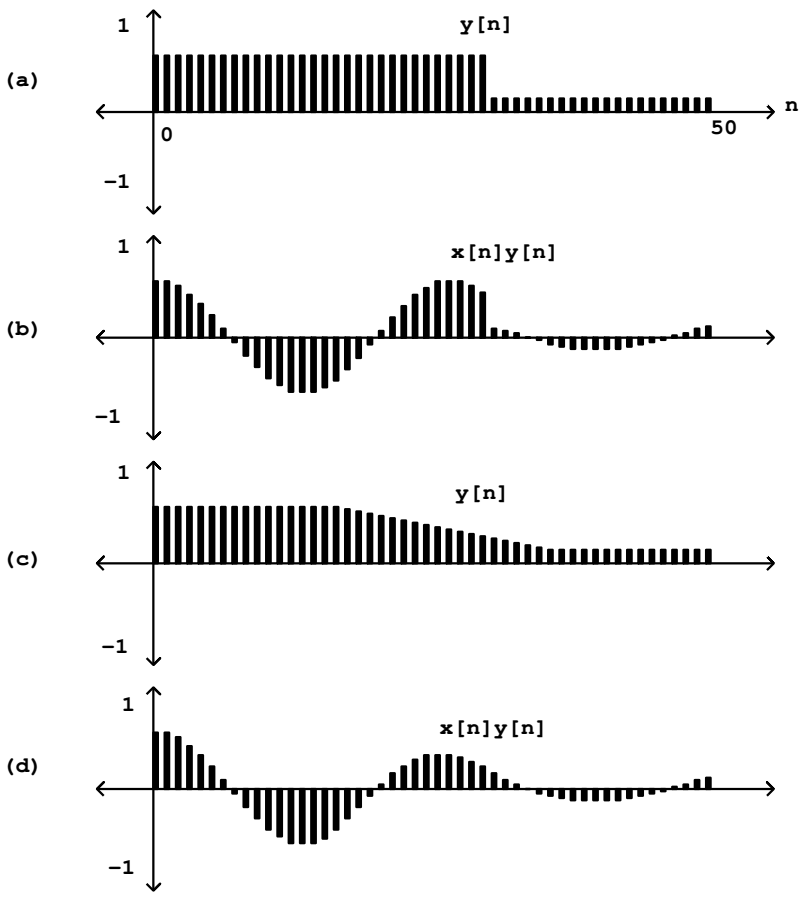


Figure 1.5: Two amplitude functions (a, c), and, in (b) and (d), the result of multiplying them by the pure Sinusoid of Figure 1.1.

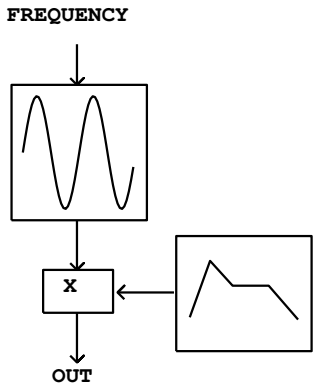


Figure 1.6: Using an envelope generator to control amplitude.

functionality from one design to another, but we will focus on the simplest kind, which generates line segments as shown in fig. 1.4(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:

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

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 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 $y[n]$,

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

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

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:

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

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

Correlation

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

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:

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

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 two uncorrelated signals both have 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.

1.7 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 τ 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 effectively ignore 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. Such a sum is called a *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 make certain technical assumptions (in effect that signals only contain frequencies up to a finite bound), we can represent any periodic signal as such a sum. This is the discrete-time variant of Fourier analysis which will reappear in chapter 9.

The angular frequencies of the Sinusoids above are all integer multiples of ω. They are called the *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 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.

A Fourier series (with only three terms) is shown in Figure 1.7. The first three graphs are those of sinusoids, whose frequencies are in a 1:2:3 ratio. The common period is marked on the horixontal axis. Each sinusoid has a different amplitude and initial phase. The sum of the three, at bottom, is not a sinusoid, but it still maintains the periodicity shared by the three component sinusoids.

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 1.8 shows a block diagram for doing this.

This is an example 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.

1.8 About the Software Examples

The examples for this book have 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.

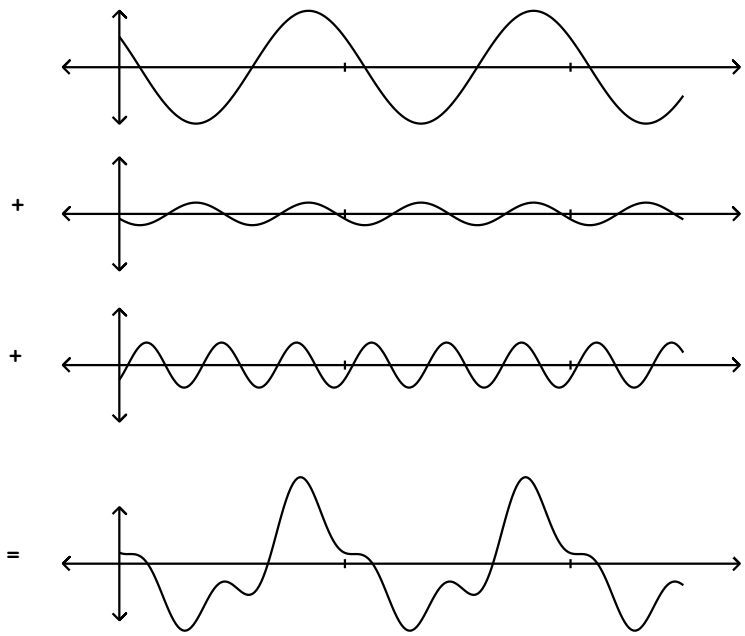


Figure 1.7: A Fourier series, showing three sinusoids and their sum. The three component sinusoids have frequencies in the ratio 1:2:3.

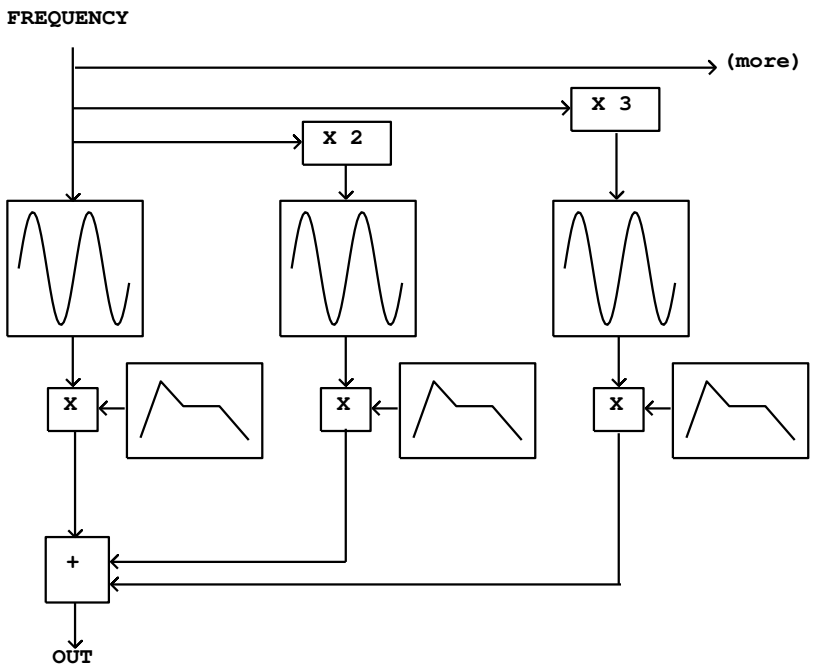


Figure 1.8: Using many oscillators to synthesize a waveform with desired harmonic amplitudes.

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.

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 1.9, 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".

- 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

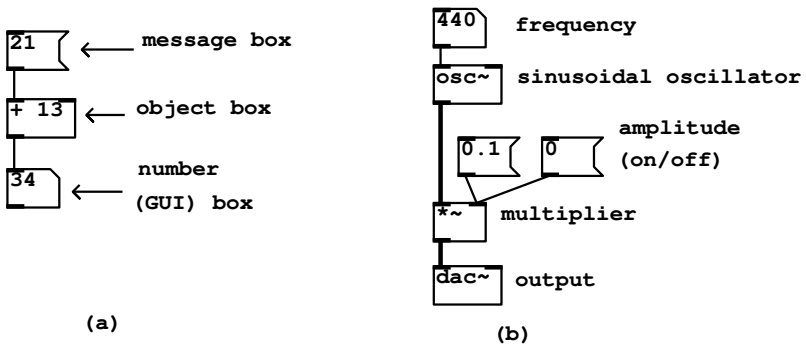


Figure 1.9: (a) three types of boxes in Pd (message, object, and GUI); (b) a simple patch to output a Sinusoid.

"+". 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. 1.9(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 1.9(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 unpredictable 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~".

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.

1.9 Examples

Constant amplitude scaler

Patch A01.sinewave.pd, shown in figure 1.10, contains essentially the simplest possible patch that makes a sound, 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) 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.

[* ~]: the multiplier. This exists in two forms. If a creation argument is

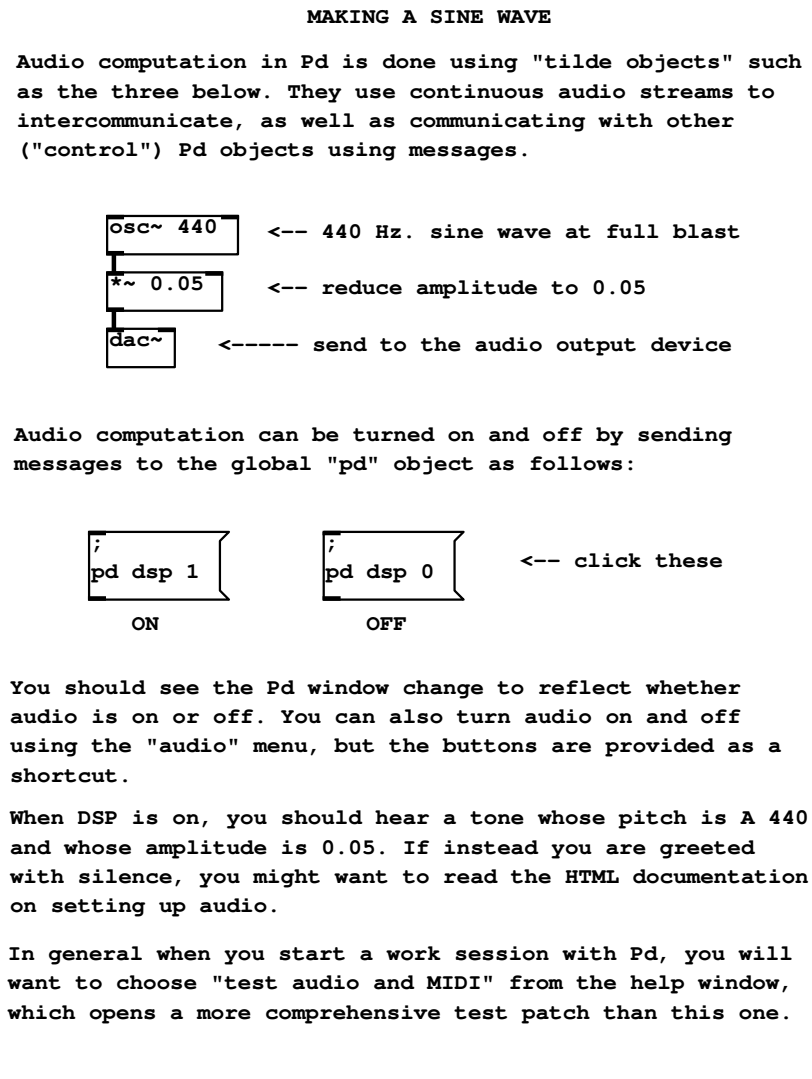


Figure 1.10: The contents of the first Pd example patch: A01.sinewave.pd.

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 ~: 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.

Amplitude control in decibels

Patch A02.amplitude.pdshows how to make a crude amplitude control; the active elements are shown in figure 1.11(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 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`

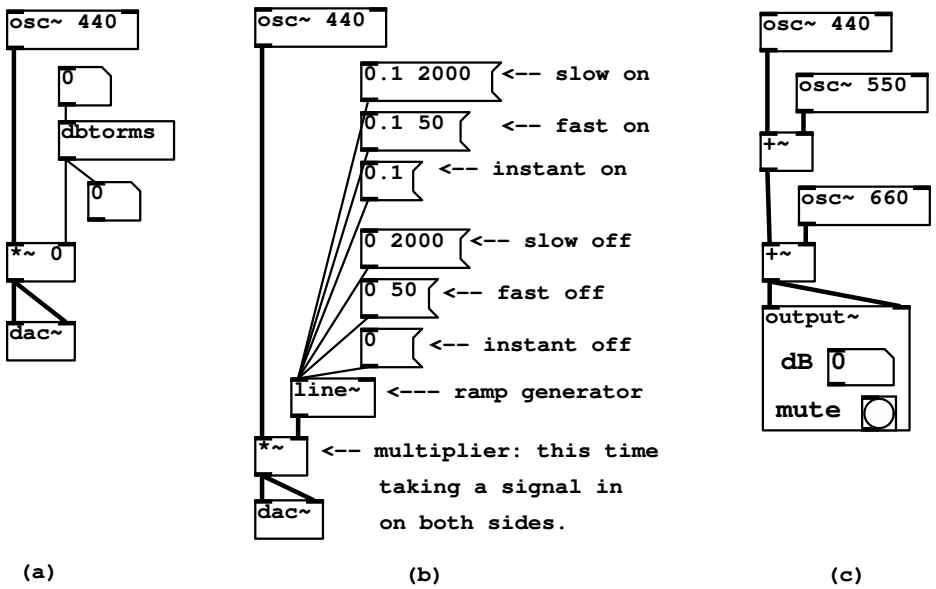


Figure 1.11: The active ingredients to three patches: (a) A02.amplitude.pd; (b) A03.line.pd; (c) A05.output.subpatch.pd.

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.

Smoothed amplitude control with an envelope generator

As figure 1.5 shows, one way to make smooth amplitude changes in a signal without clicks is to multiply by an envelope generator; one is invoked in block diagram form in figure 1.6. 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 1.11(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

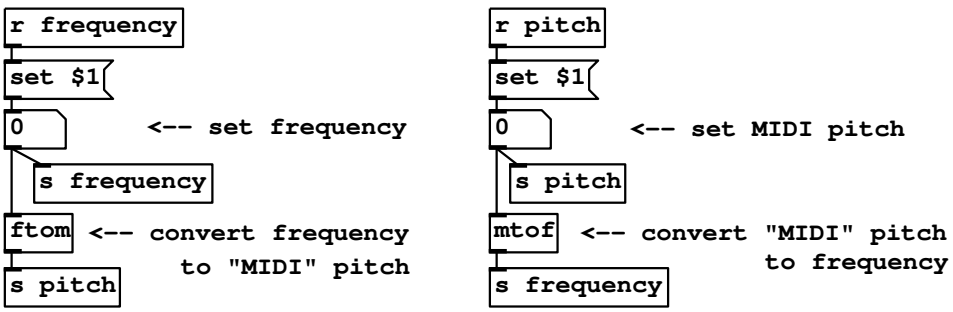


Figure 1.12: Conversion between pitch and frequency in A06.frequency.pd.

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 1.5 in action.

Major triad

Patch A05.output.subpatch.pd, whose active ingredients are shown in Figure 1.11(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.

Conversion between frequency and pitch

Patch A06.frequency.pd (figure 1.12) 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 `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.

More additive synthesis

The major triad (page 24) is one example showing how to combine several sinusoids together by summing. Here we will show two more examples. Patch A07.fusion.pd (Figure 1.13) shows four oscillators, whose frequencies are tuned in the ratio 1:2:3:4, with relative amplitudes 1, 0.1, 0.2, and 0.5. The amplitudes are set by multiplying the outputs of the oscillators (the `*~` objects below the oscillators).

Tthe second, third, and fourth oscillator are turned on and off using a *toggle switch*. This is a graphical control, like the number box introduced earlier. The toggle switch puts out values of 1 and 0 alternately when clicked on with the mouse. This value is multiplied by the sum of the second, third, and fourth oscillators, effectively turning them on and off.

Even when all four oscillators are combined (with the toggle switch in the "1" position), the result fuses into a single tone, heard at the pitch of the leftmost oscillator. In effect this patch sums a four-term Fourier series to generate a complex, periodic waveform.

Patch A08.beating.pd (Figure 1.14) shows another possiblity, in which six oscillators are tuned into three pairs of neighbors, for instalnce 330 and 330.2 Hertz. These paris slip into and out of phase with each other, so that the ampltude of the sum changes over time. This is called *beating*.

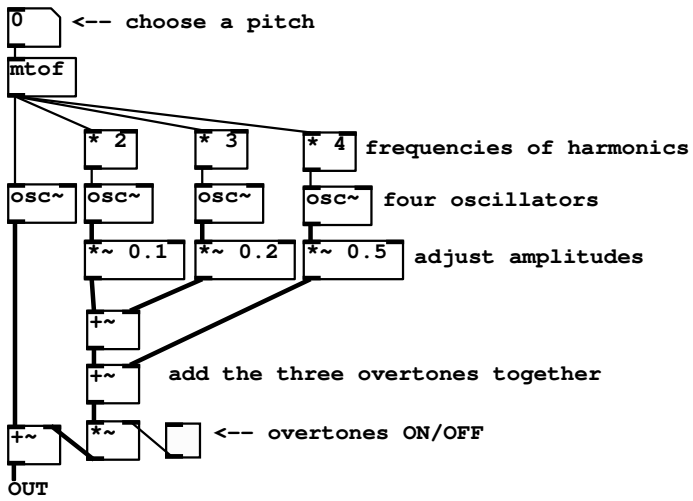


Figure 1.13: Additive synthesis using harmonically tuned oscillators.

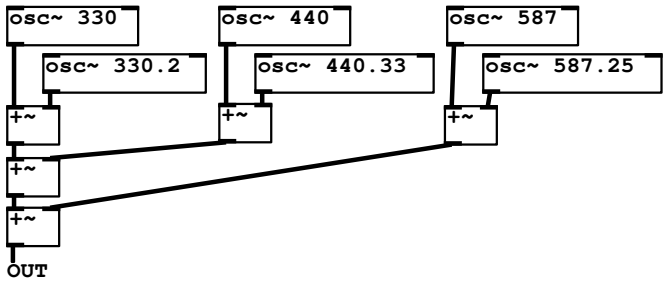


Figure 1.14: Additive synthesis: six oscillators arranged into three beating pairs.

Oscillators may be combined in other ways besides simply summing their output, and a wide range of resulting sounds is available. Patch A09.frequency.mod.pd (not shown here) demonstrates Frequency Modulation synthesis, in which one oscillator controls another's frequency. This will be more fully described in chapter 5.

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 an audio signal $x[n], n = 0, ..., M - 1$ has peak amplitude 1, what is the minimum possible RMS amplitude? What is the maximum possible?

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

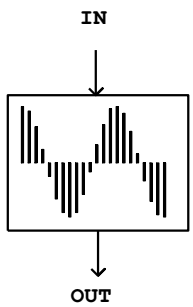


Figure 2.1: Diagram for wavetable lookup. The input is in samples, ranging approximately from 0 to the wavetable's size N, depending on the interpolation scheme.

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 cycles per second) 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 1.7). 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

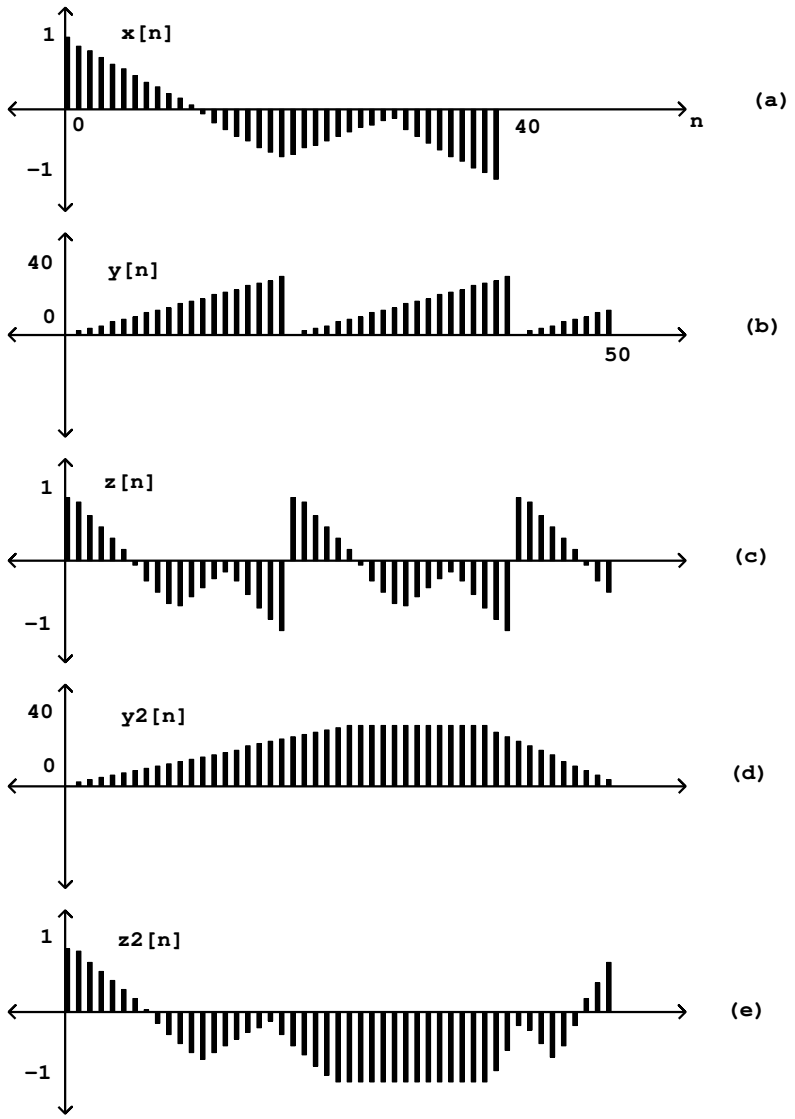


Figure 2.2: Wavetable lookup. (a): a wavetable; (b) and (d): signal inputs for lookup; (c) and (e): the corresponding outputs.

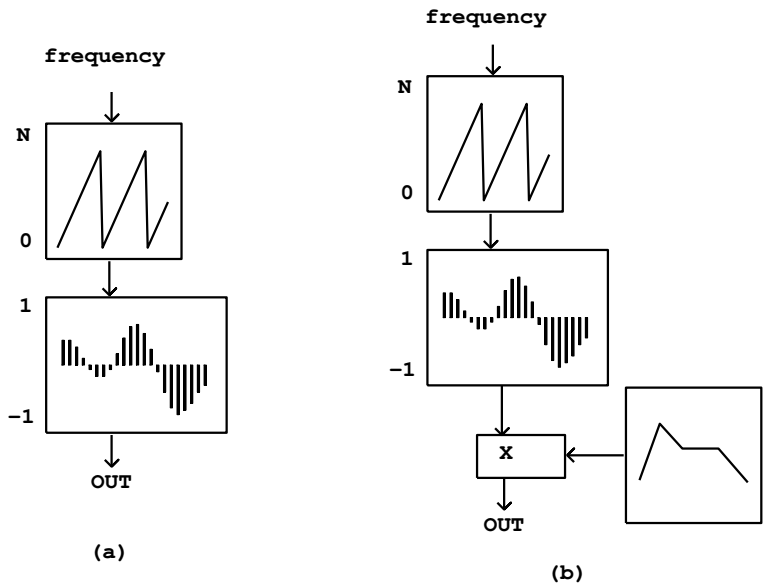


Figure 2.3: Block diagram (a) for a wavetable lookup oscillator , and (b) with amplitude control by an envelope generator.

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

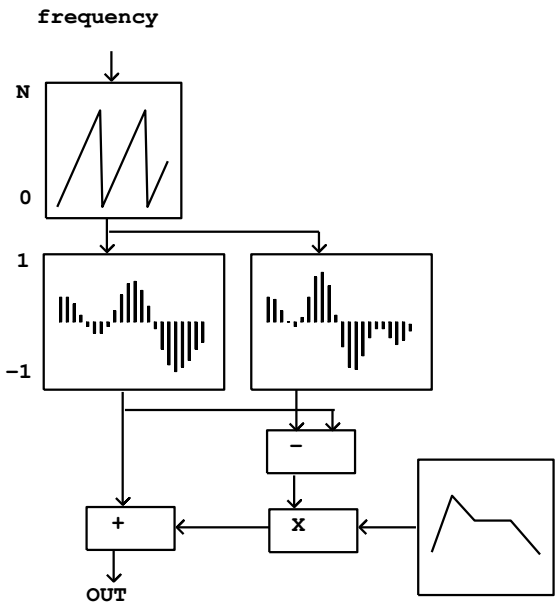


Figure 2.4: Block diagram for cross-fading between two wavetables.

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 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 5.1, 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.

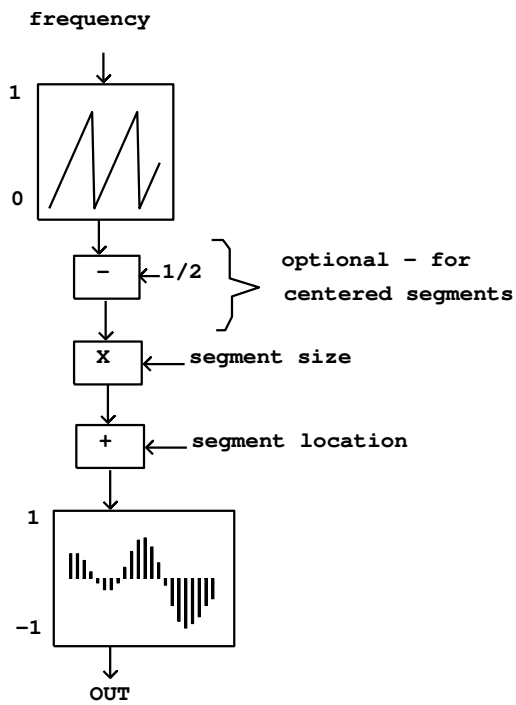


Figure 2.5: A simple looping sampler, as yet with no amplitude control. There are inputs to control the frequency and the segment size and location. The "-" operation is included if we wish the segment location to be specified as the segment's midpoint; otherwise we specify the location of the left end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

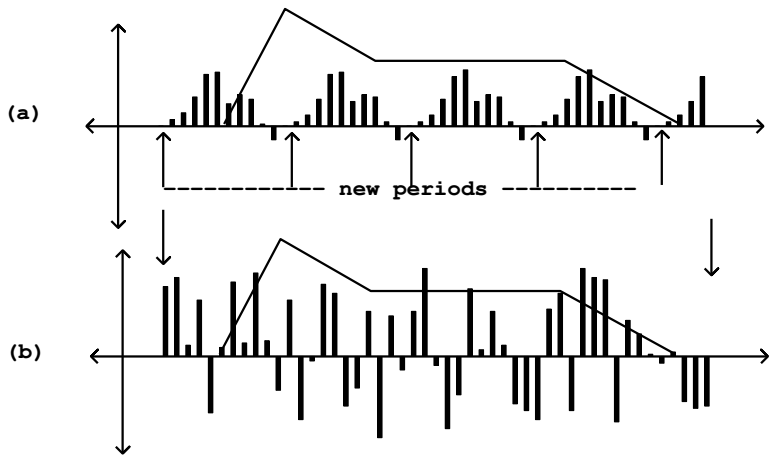


Figure 2.6: Differing envelope requirements for oscillators and samplers: (a) in an oscillator, the envelope can be chosen to conform to any desired timescale; (b) when the wavetable is a recorded sound, it's up to you to get the envelope to zero before you hit the end of the wavetable for the first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

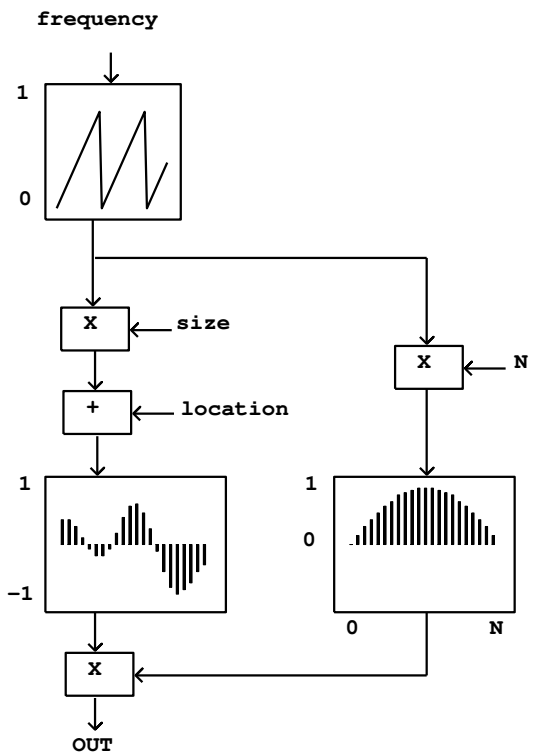


Figure 2.7: A sampler as in Figure 2.6, but with an additional wavetable lookup for enveloping.

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,

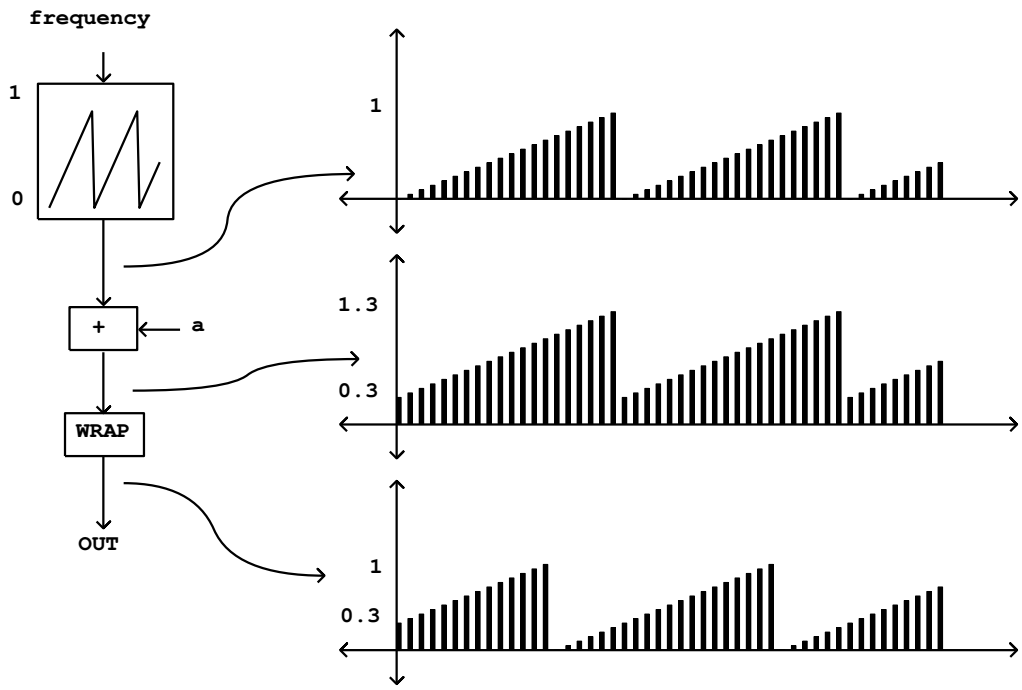


Figure 2.8: A technique for generating two or more sawtooth waves with fixed phase relationships between them. The relative phase is controlled by the parameter a.

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

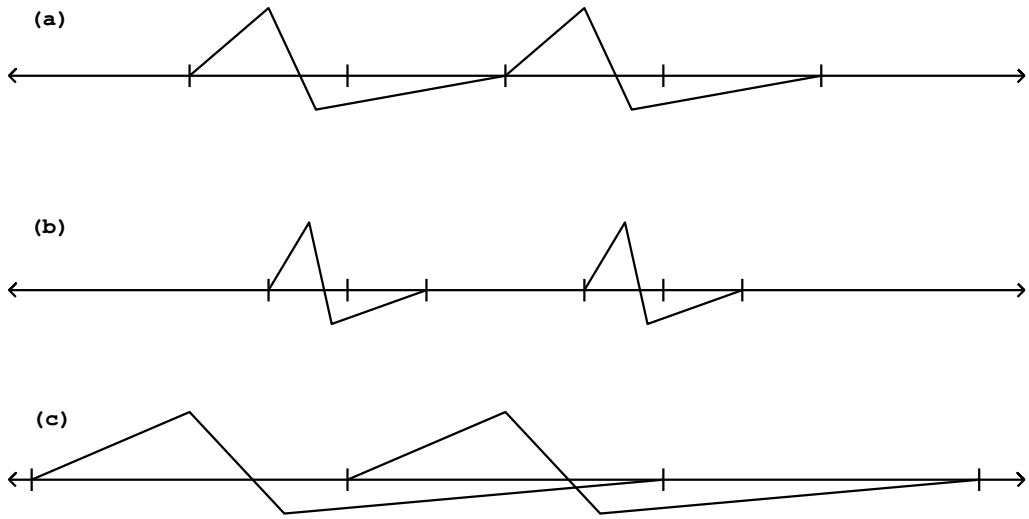


Figure 2.9: A waveform is played at a period of 20 samples: (a) at 100 percent duty cycle; (b) at 50 percent; (c) at 200 percent

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 1.7) 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:

$$a_0 + a_1 \cos(2\omega n + \phi_1) + a_2 \cos(4\omega n + \phi_2) + \cdots$$

$$= b_0 + b_1 \cos(\omega n + \theta_1) + b_2 \cos(2\omega n + \theta_2) + \cdots + b_0 + b_1 \cos(\omega n + \pi + \theta_1) + b_2 \cos(2\omega n + 2\pi + \theta_2) + \cdots$$

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

and so

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

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, ..., we get:

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

so that the partials of x_{200} are those of x_{100} shrunk, by half, to the left.

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 overtones, in frequency, 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

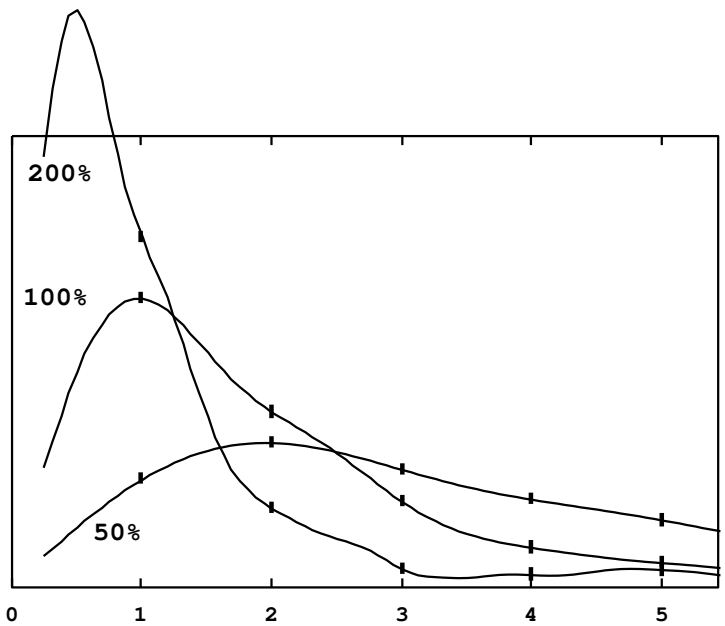


Figure 2.10: The Fourier series magnitudes for the waveforms shown in Figure 2.9. The horizontal axis is the harmonic number. We only "hear" the coefficients for integer harmonic numbers; the continuous curves are the "ideal" contour.

always be forced to behave at least reasonably well by enveloping it as in Figure 2.7.)

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 contour 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 contour (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 (con-

tinuous) 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 a residual signal all of whose energy lies in overtones of the original sinusoid. As the number of points increases, the error gets smaller in magnitude. Since the error is the difference between a sinusoid and a sequence of approximating line segments, 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

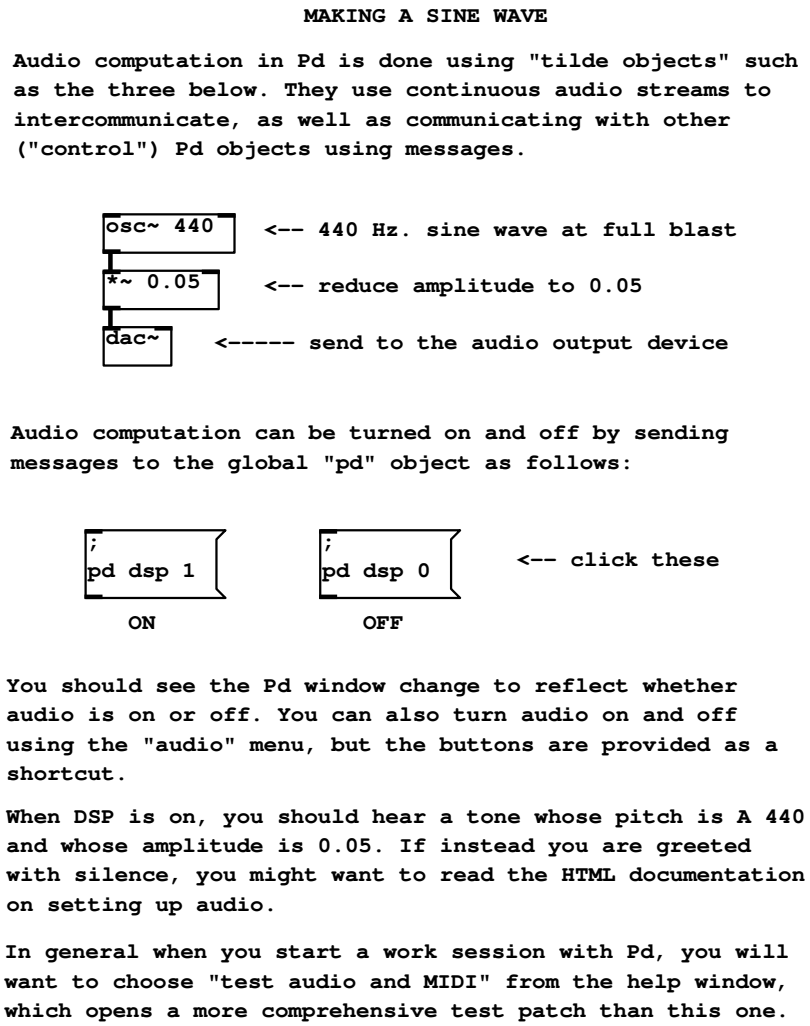


Figure 2.11: Linear interpolation of a sinusoid: (a) the original sinusoid, the interpolated sinusoid, and the best sinusoidal fit back to the interpolated version; (b) the error.

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.

various periods for 1, 2, and 4 point interpolation. (A slightly different quantity 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,

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

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

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

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

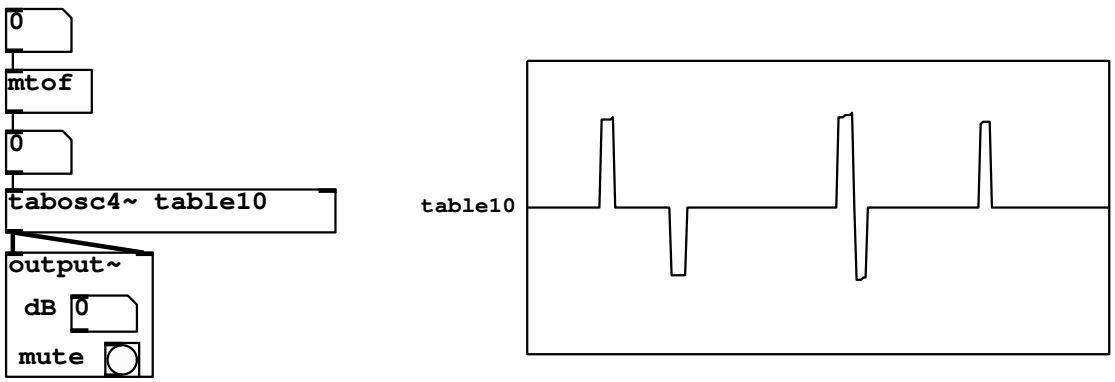


Figure 2.12: A wavetable oscillator: B01.wavetables.pd.

one. If, then, one cycle of the wavetable is arranged from 1 to N, we must 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:

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

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~`

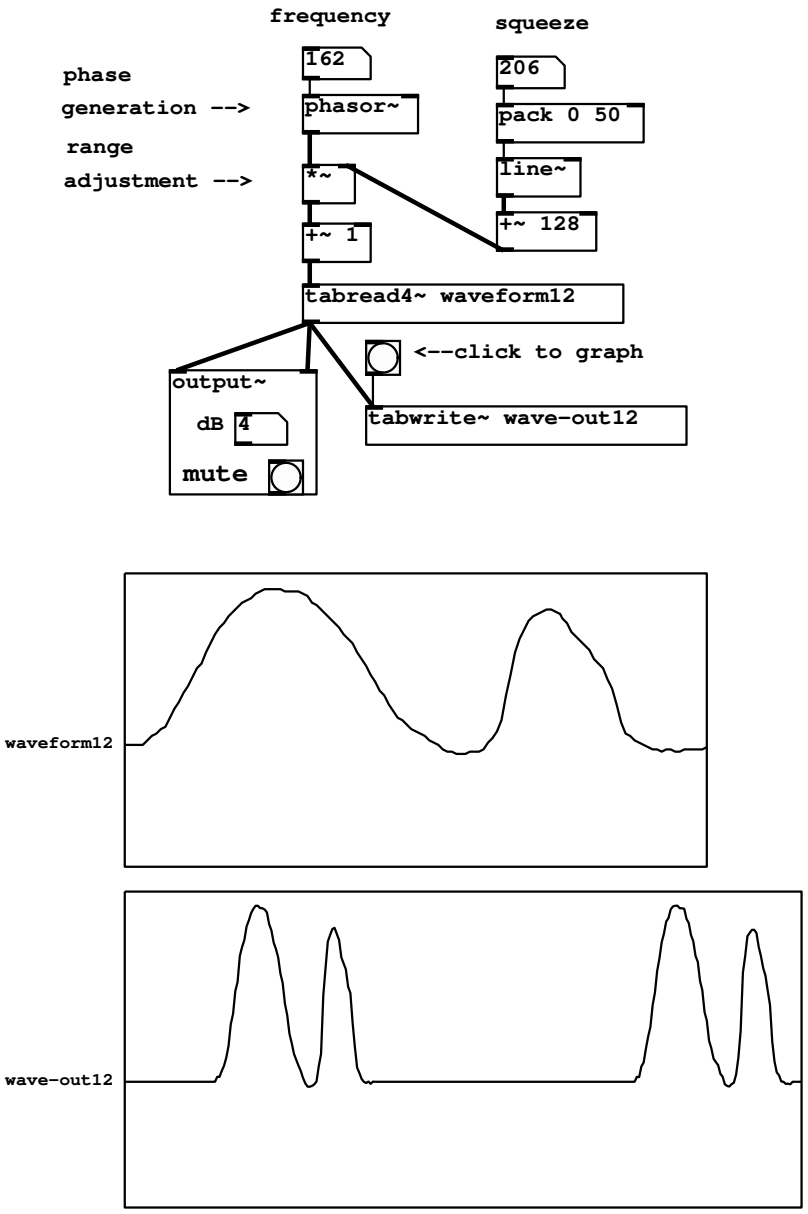


Figure 2.13: A wavetable oscillator with variable duty cycle: B03.tabread4.pd.

object below is to ramp to 206 in 50 milliseconds; in general the output of the `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 cycles per second. 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 8.

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 cycles-per-second grit. (Try it to find out what that sounds like!)

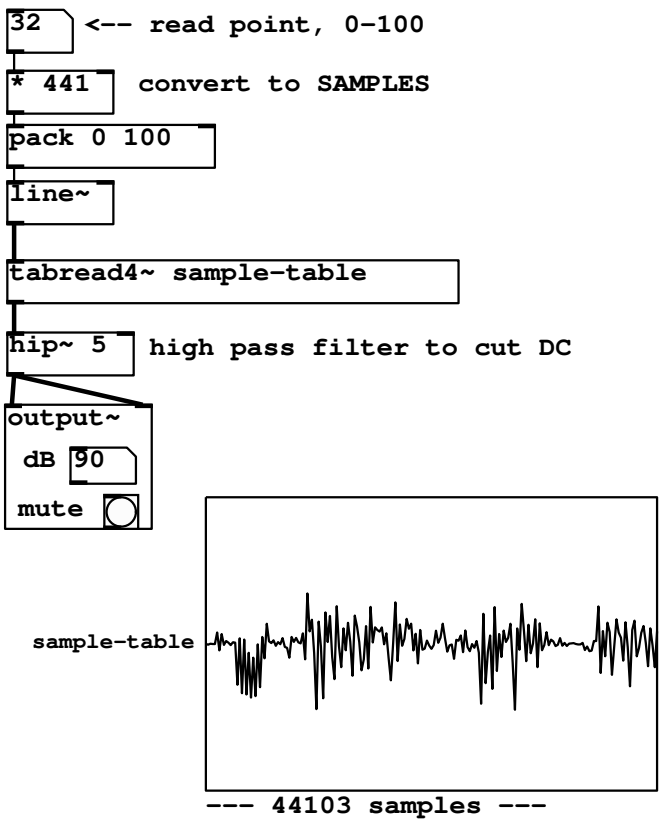


Figure 2.14: A sampler with mouse-controlled index: B04.sampler.pd.

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:

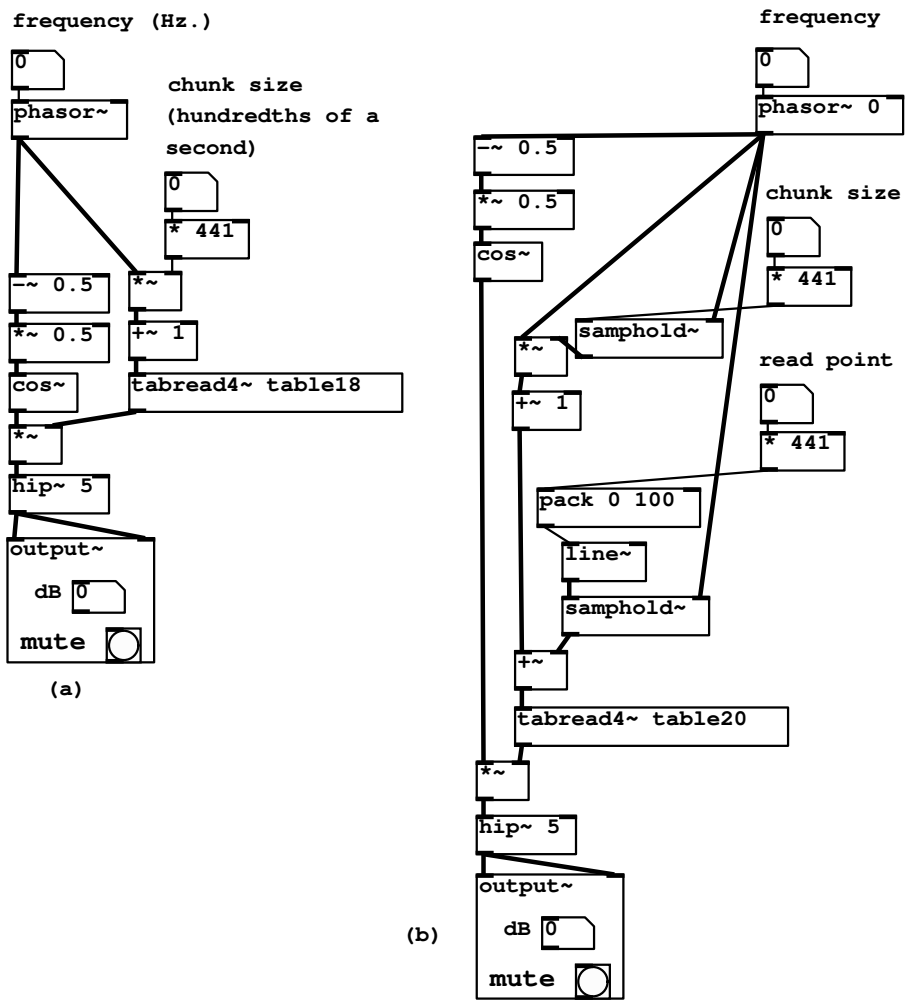


Figure 2.15: (a) a looping sampler with a synchronized envelope (B06.sampler.loop.smooth.pd); (b) the same, but with a control for read location (B08.sampler.nodoppler.pd).

[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 ~

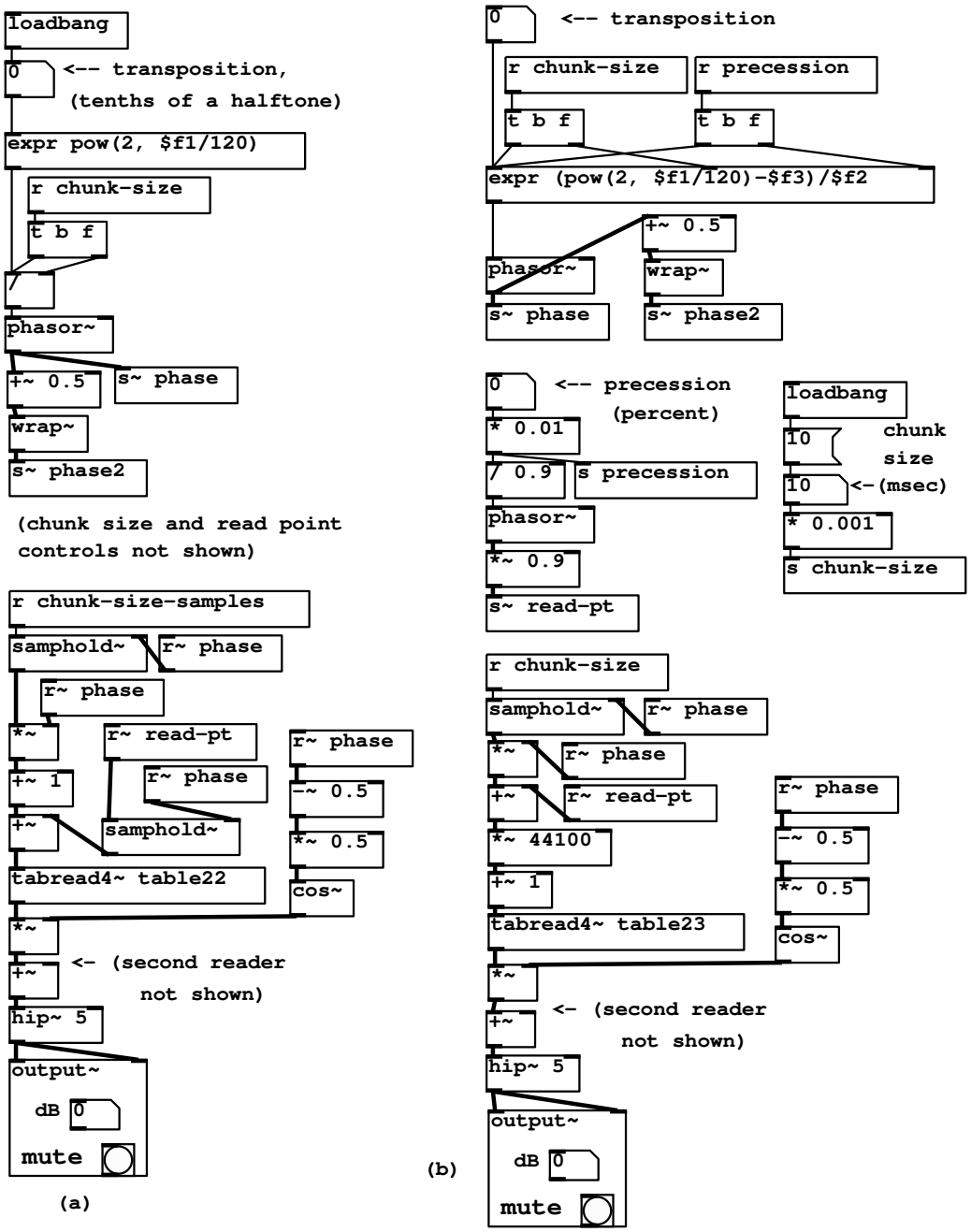


Figure 2.16: (a) two overlapped looping samplers (B10.sampler.overlap.pd); (b) the same, but with a phasor-controlled read point (B11.sampler.rockafella.pd).

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 $n = \ldots, -1, 0, 1, \ldots$. 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

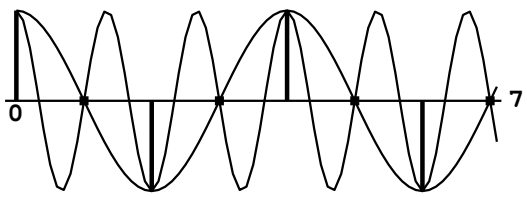


Figure 3.1: Two real sinusoids, with angular frequencies $\pi/2$ and $3\pi/2$, showing that they coincide at integers. A digital audio signal can't distinguish between the two.

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

$$\begin{aligned} cos((\pi+\omega)n+\phi) &= cos((\pi+\omega)n+\phi-2\pi n) \\ &= cos((\omega-\pi)n+\phi \\ &= cos((\pi-\omega)n-\phi), \end{aligned}$$

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 $\sim$ 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

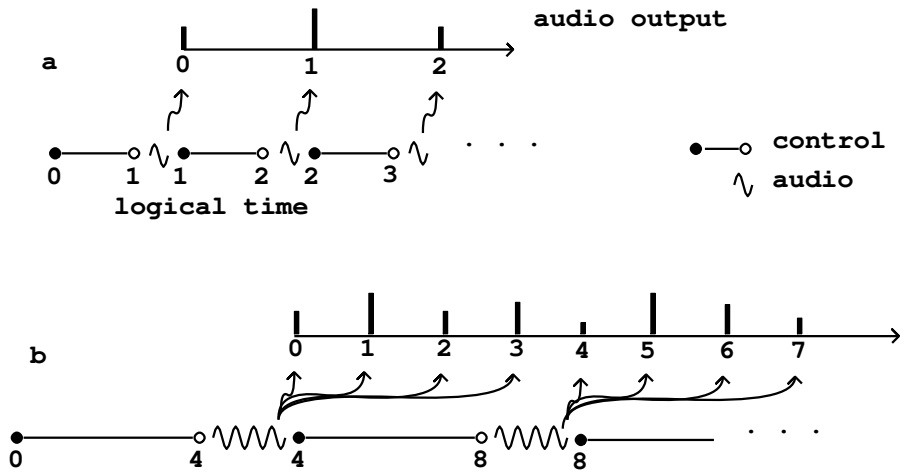


Figure 3.2: Timeline for digital audio and control computation. In (a) the block size is one sample. We compute the audio at a delay of one sample after the control computation; so sample 0 (at top of (a)) is computed at logical time 1 (logical times are shown on the number line below). In (b), the block 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.

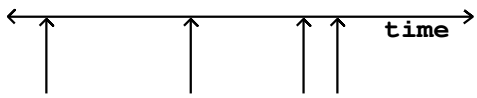


Figure 3.3: Graphical representation of a control stream as a sequence of points in time.

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

Most computer music software computes audio in blocks. This is done to increase the efficiency of individual audio operations (such as Csound's unit generators and Max/MSP and Pd's tilde objects). Each unit generator or 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 by default), the overhead is only some 30%.

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*:

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

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:

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

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:

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

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 the Nyquist frequency of the output is lower than the input square wave's frequency, and so the output is aliased to a new frequency lower than the Nyquist frequency.)

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.

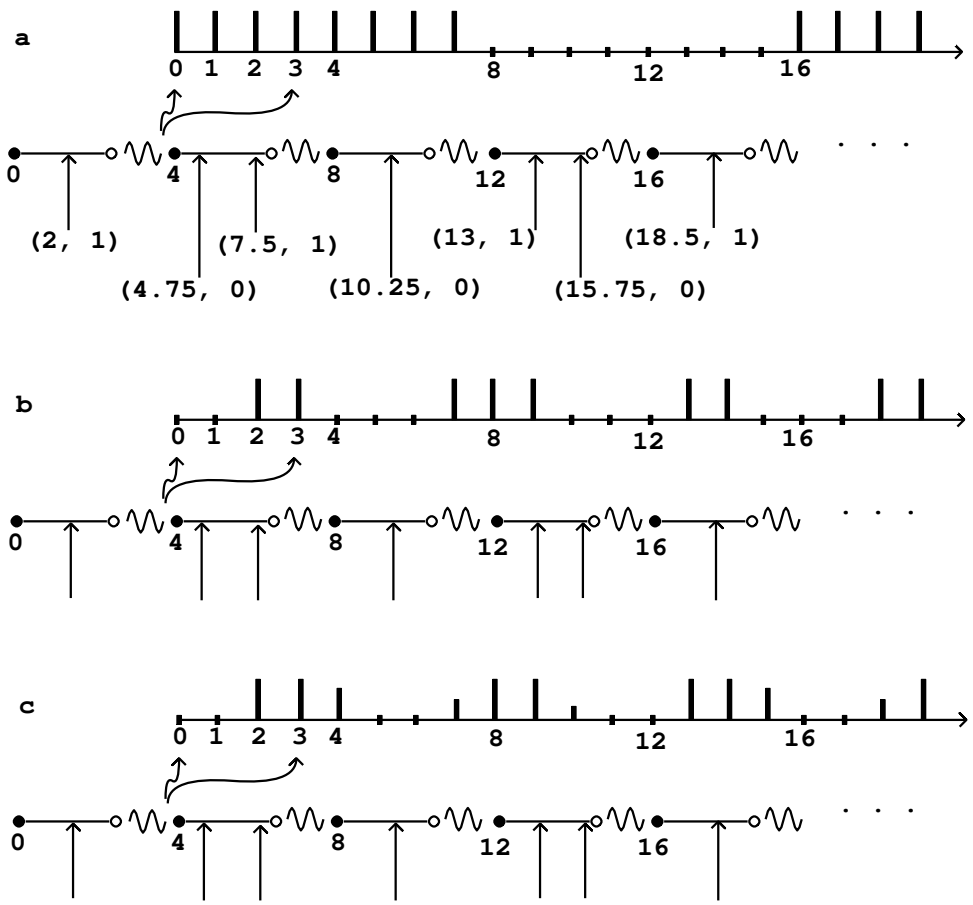


Figure 3.4: Three ways to change a control stream into an audio signal. A: as fast as possible; B: delayed to the nearest sample; C: with two-point interpolation for higher delay accuracy.

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 $0 \leq f < 1$). 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 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.

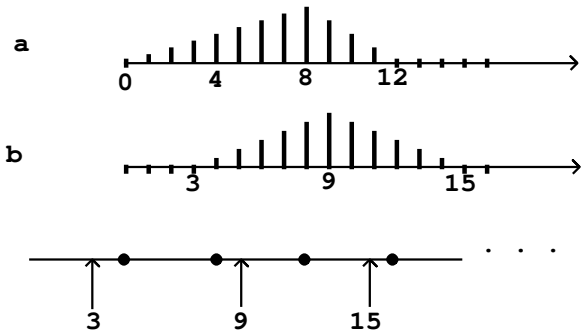


Figure 3.5: Line segment smoothing of numeric control streams: (a) aligned to block boundaries; (b) aligned to nearest sample.

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 23). 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.

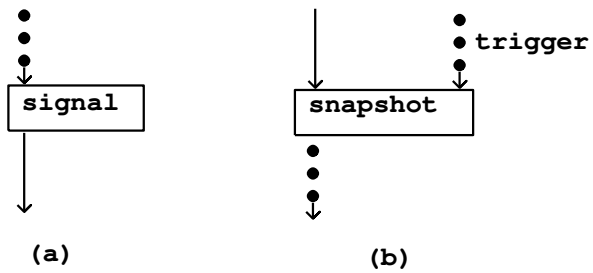


Figure 3.6: Conversion between control and audio: (a) control to signal; (b) signal to control by snapshots.

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.

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 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.

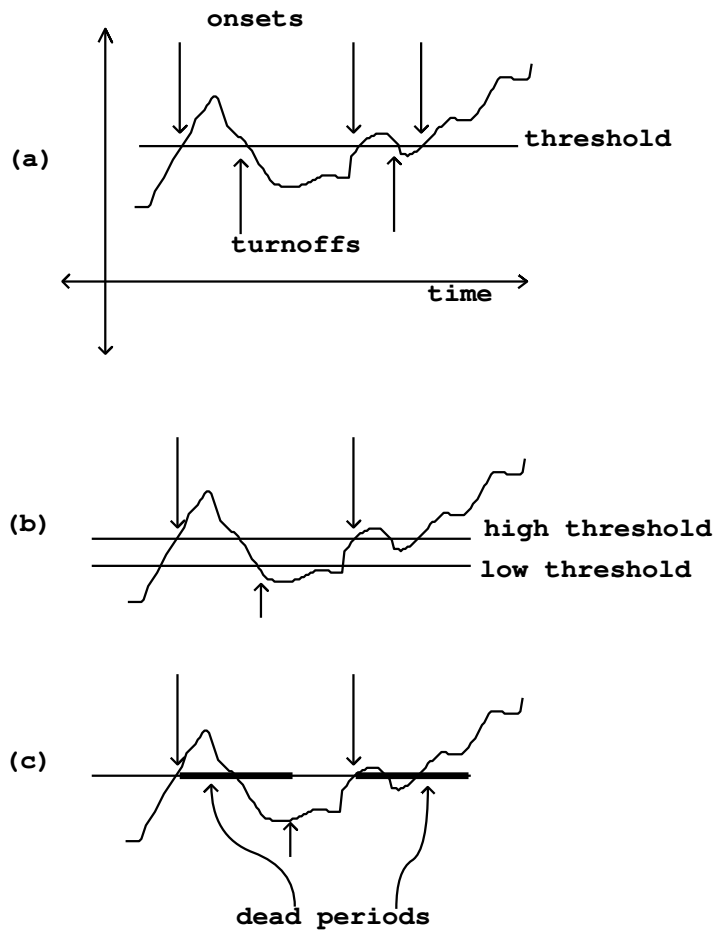


Figure 3.7: Threshold detection: (a) with no debouncing; (b) debounced using two threshold levels; (c) debounced using dead periods.

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... 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 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

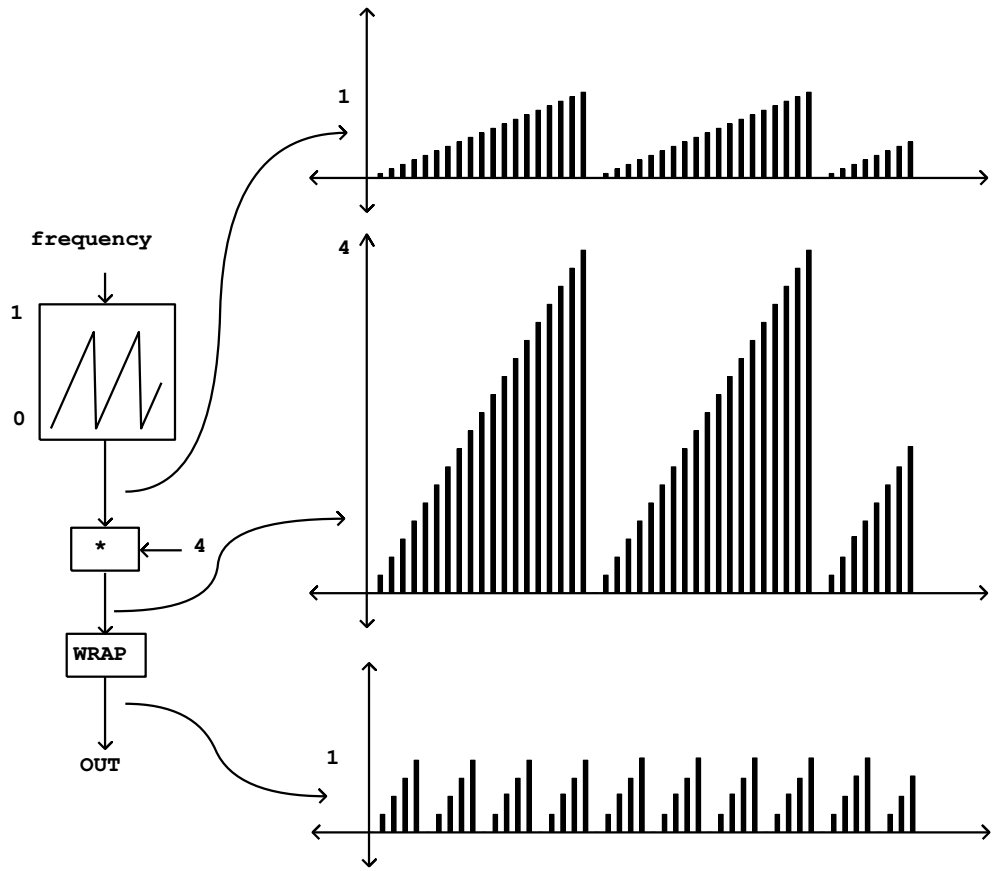


Figure 3.8: Multiplying and wrapping a sawtooth wave to generate a higher frequency.

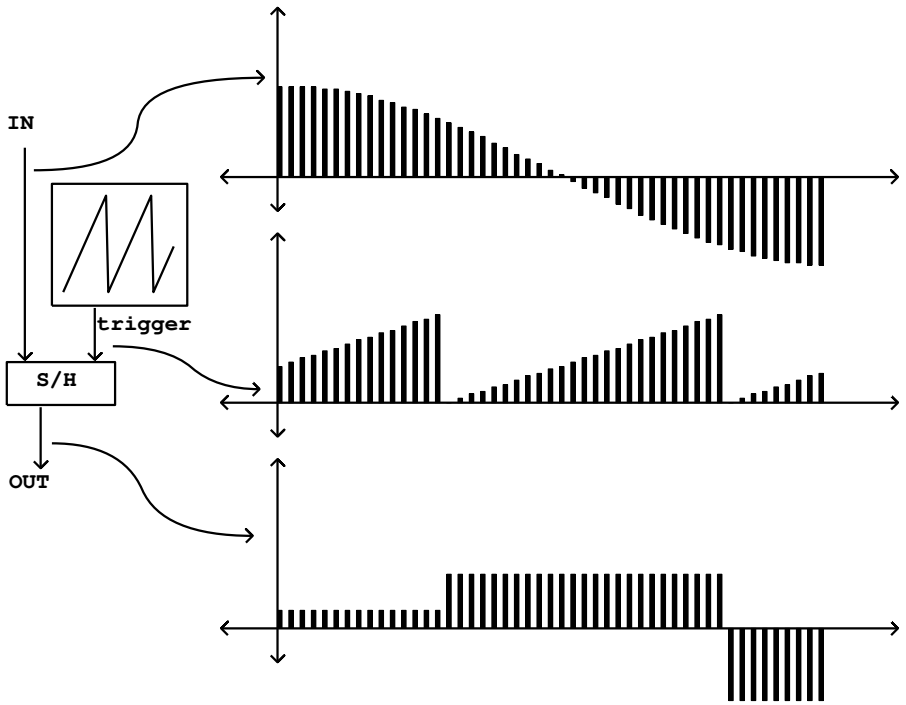


Figure 3.9: Sample and hold, using falling edges of the trigger signal.

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.

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.

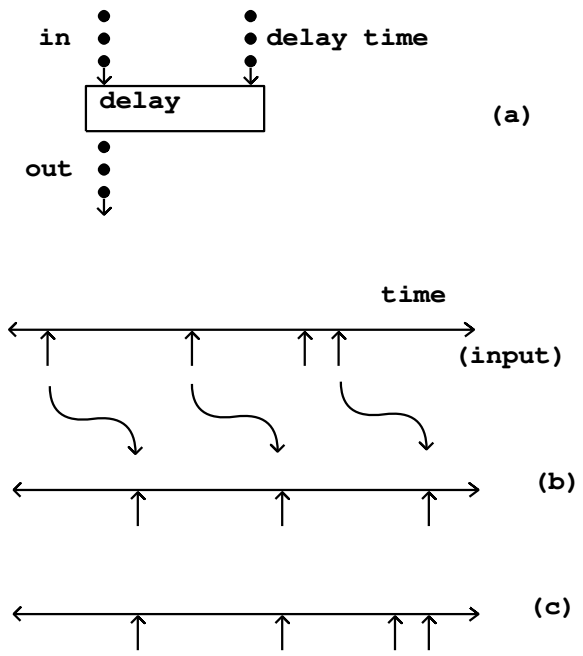


Figure 3.10: Delay as an operation on a control stream: (a) block diagram; (b) effect of a simple delay on a control stream; (c) effect of a compound delay.

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 3.11 (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 3.11 (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 1.7 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 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:

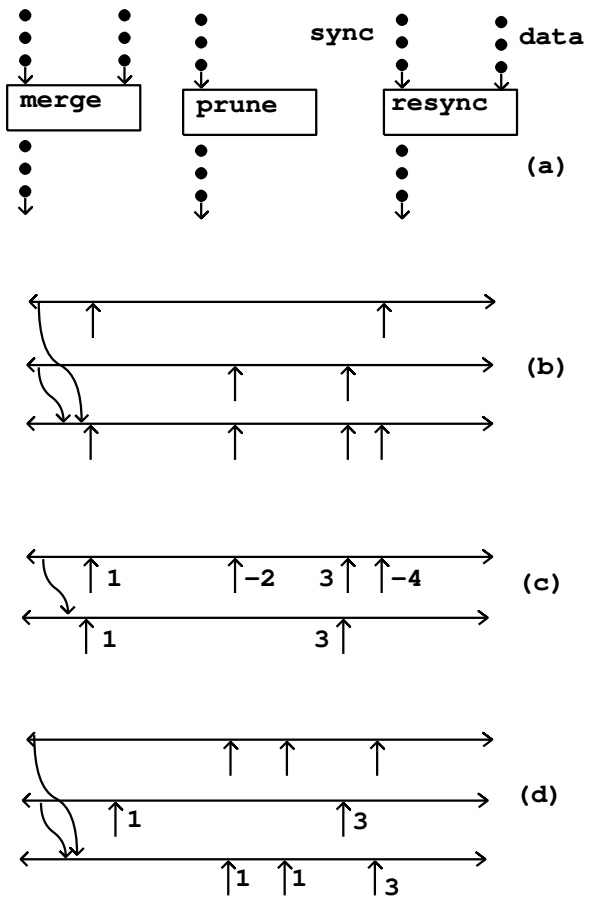


Figure 3.11: Operations on control streams (besides delay): (a) block diagrams; (b) merging; (c) pruning; (d) resynchronizing.

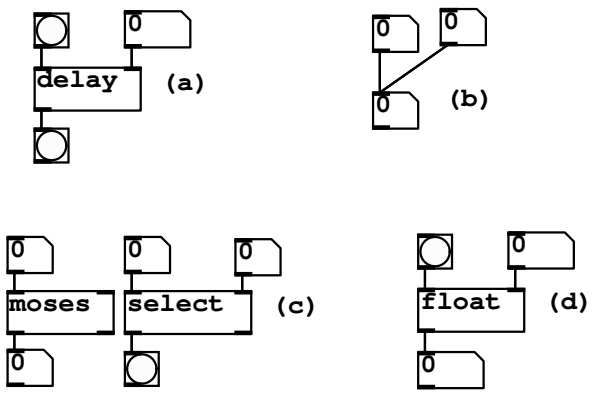


Figure 3.12: The four control operations in Pd: (a) delay; (b) merging; (c) pruning; (d) resynchronizing.

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

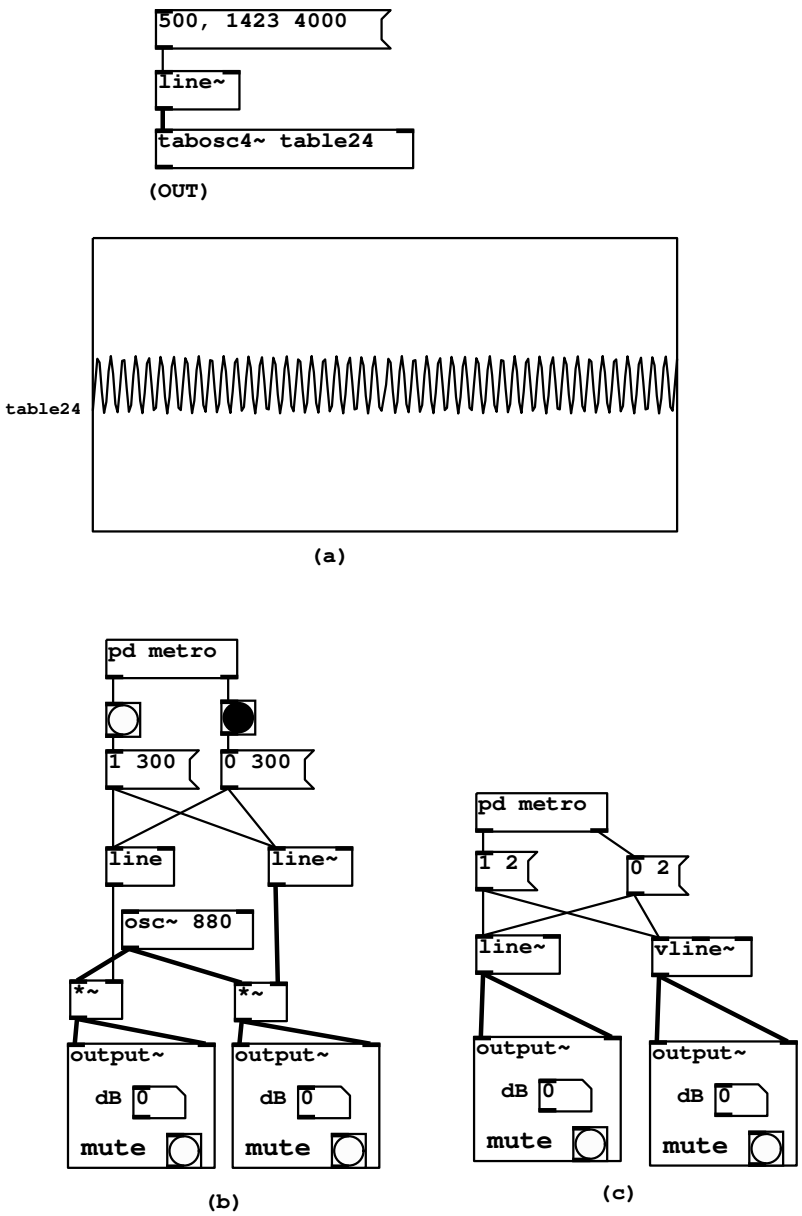


Figure 3.13: Sending an oscillator over the Nyquist frequency.

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 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 51) 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 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

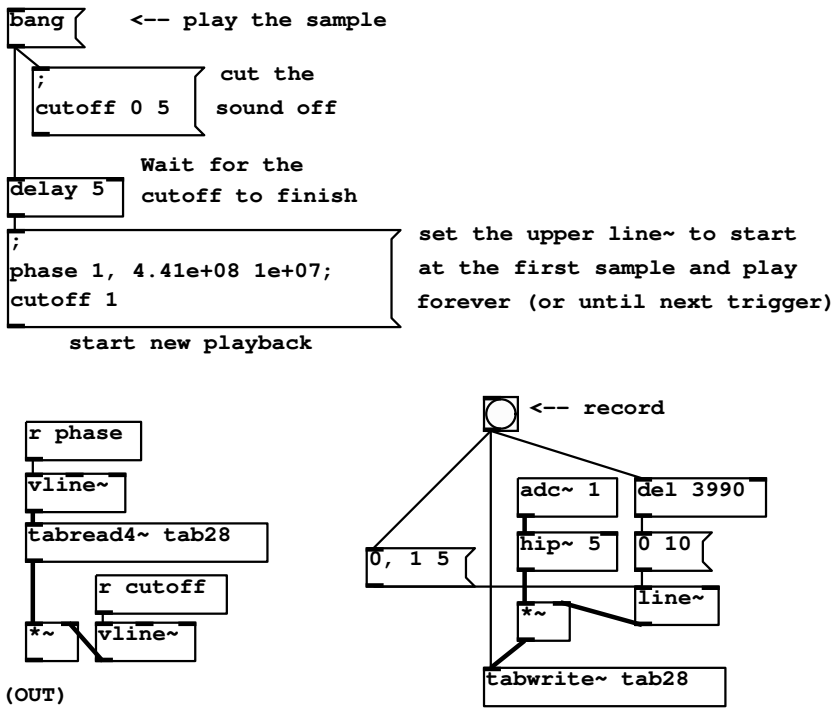


Figure 3.14: Non-looping sampler.

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 (`tabread~` instead of `tabread4~`) 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). (This multiplication can cause audible artifacts which will be discussed in chapter 5.)

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.

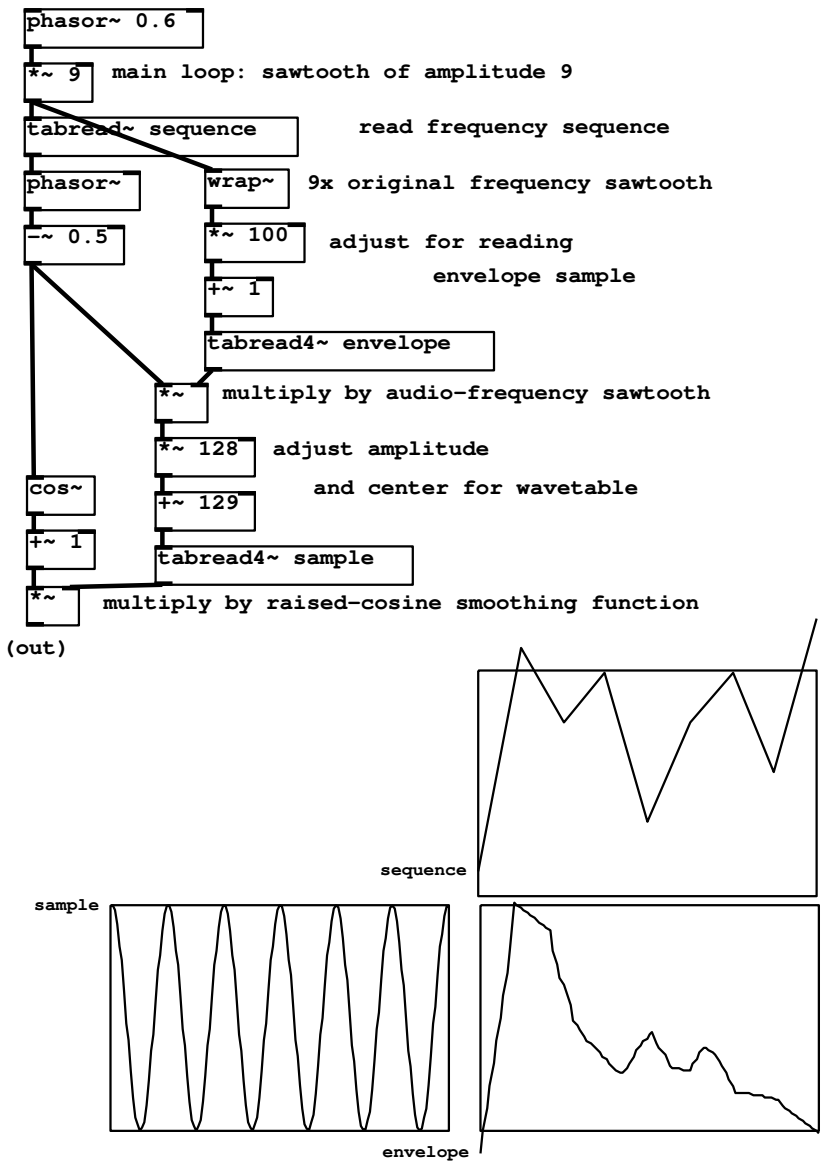


Figure 3.15: An analog-synthesizer-style sequencer.

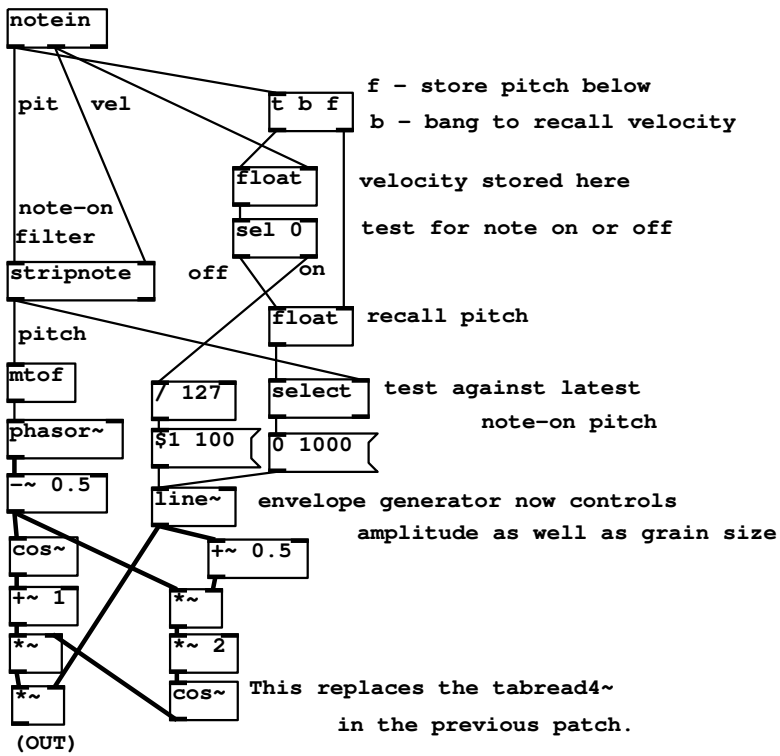


Figure 3.16: A MIDI-style monophonic synthesizer.

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 conver-

sion. 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 incoming message is converted (as faithfully as possible) into whatever format is specified by the creation argument. Although the outputs appear in Pd's standard right-to-left order, no time elapses between each of the outputs; they are deterministically ordered and yet instantaneous.

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 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 equal (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.

4. Two sawtooth waves, of unit amplitude, have frequencies 200 and 300 Hz., respectively. What is the periodicity of the sum of the two? What if you then wrapped the sum back to the range from 0 to 1? Does this result change depending on the relative phase of the two?

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 complex tone. 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 1.3 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

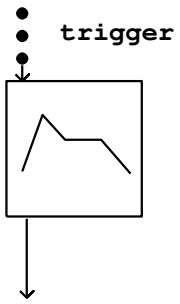


Figure 4.1: ADSR envelope as a block diagram, showing the trigger input (a control stream) and the audio output.

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.

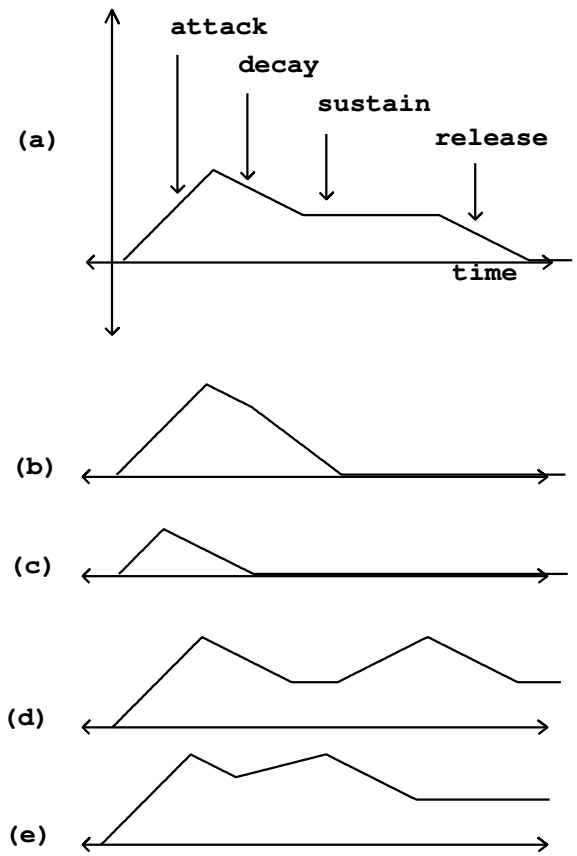


Figure 4.2: ADSR envelope output: (a) with "on" and "off" triggers separated; (b), (c) with early "off" trigger; (d), (e) re-attacked.

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:

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

$$f_2(x) = 10^{2(x-1)} \quad \text{(dB to linear)},$$

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

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:

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

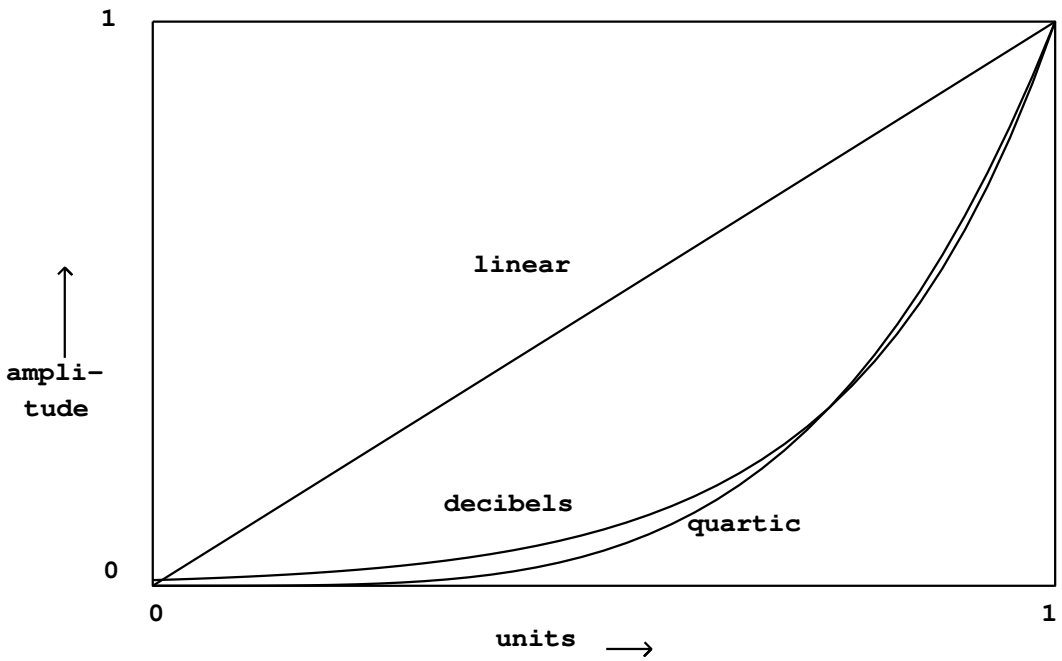


Figure 4.3: Three amplitude transfer functions. The horizontal axis is in linear, logarithmic, or fourth-root units depending on the curve.

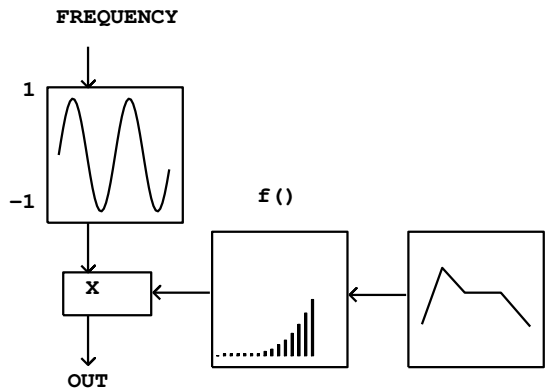


Figure 4.4: Using a transfer function to alter the shape of amplitude curves.

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;

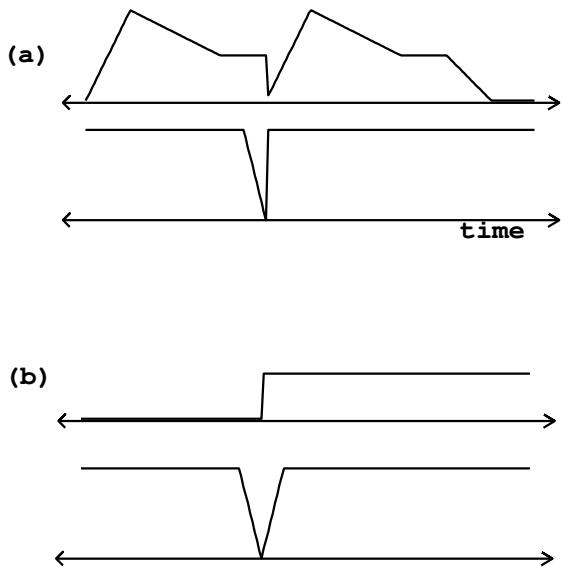


Figure 4.5: The muting technique for hiding discontinuous changes, for example of an amplitude envelope. In part (a) the envelope (upper graph) is set discontinuously to zero. The muting signal (lower graph) ramps down in advance to prepare for the change, and then is restored (discontinuously) to its 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. The discontinuity is minus the last value of the synthesis output just before it is reset to zero.

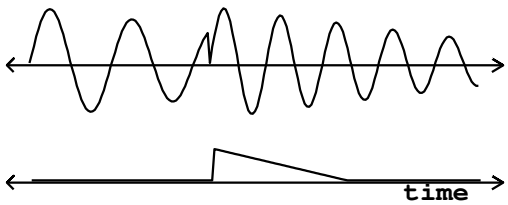


Figure 4.6: The switch-and-ramp technique for canceling out discontinuous changes. A discontinuity (upper graph) is measured and canceled out with a signal having the opposite discontinuity (lower graph), which then decays smoothly.

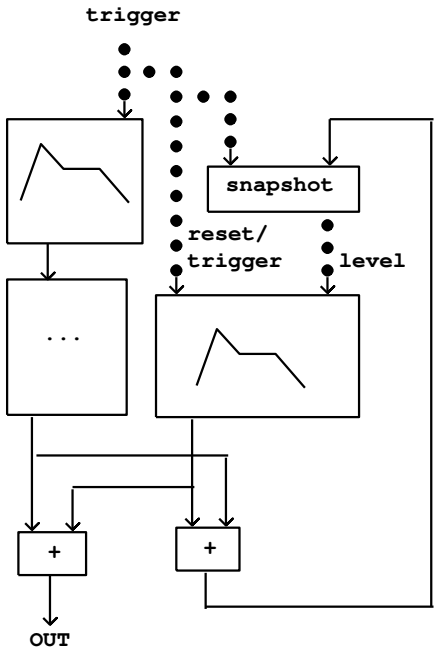


Figure 4.7: Block diagram for the switch-and-ramp technique.

To do this we measure the level the ADSR generator must now jump to. This is its own current level (which may not be zero) minus the discontinuity (or equivalently, *p*lus the synthesis output's last value). The two are added (by the `+~` object at bottom right), and then a snapshot is taken. The cancelling envelope generator (at right) is reset discontinuously to this new value, and then triggered to ramp back to zero. The `+~` object at bottom left adds the synthesizer output to the discontinuity-cancelling signal.

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

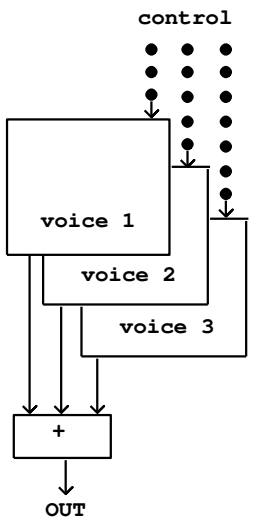


Figure 4.8: A voice bank for polyphonic synthesis.

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?

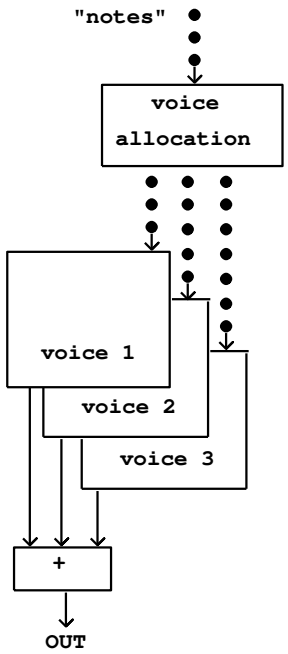


Figure 4.9: Polyphonic voice allocation

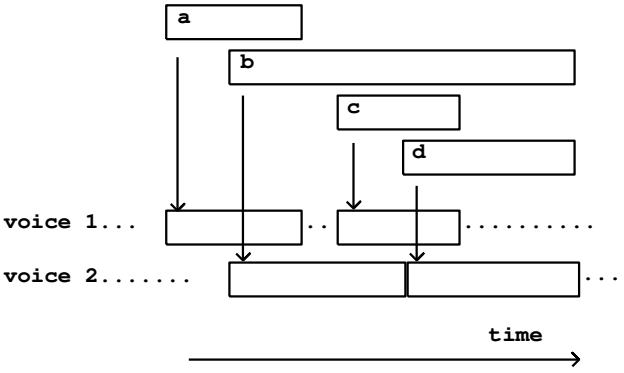


Figure 4.10: A polyphonic voice allocation algorithm, showing voice stealing.

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

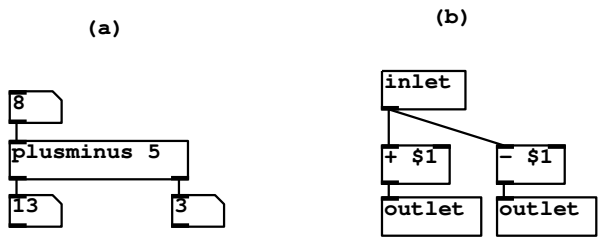


Figure 4.11: Pd's abstraction mechanism: (a) invoking the abstraction, "plusminus" with 5 as a creation argument; (b) the contents of the file, "plusminus.pd".

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 1.9(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 back 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

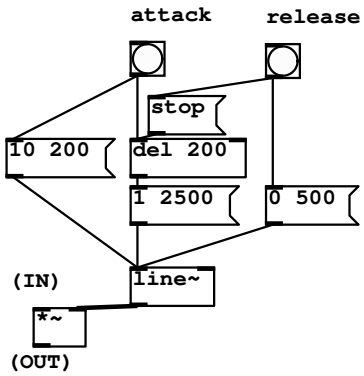


Figure 4.12: Using a `line~` object to generate an ADSR envelope.

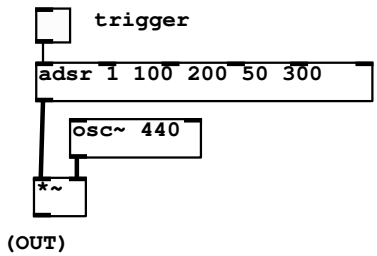


Figure 4.13: Invoking the `adsr` abstraction.

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

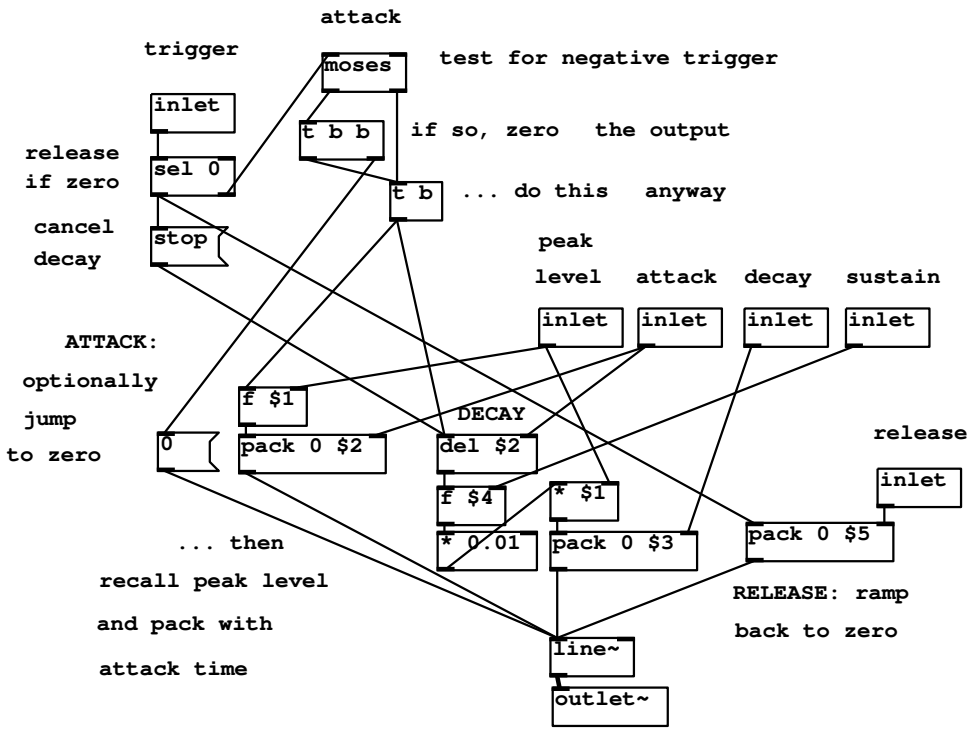


Figure 4.14: Inside the `adsr` abstraction.

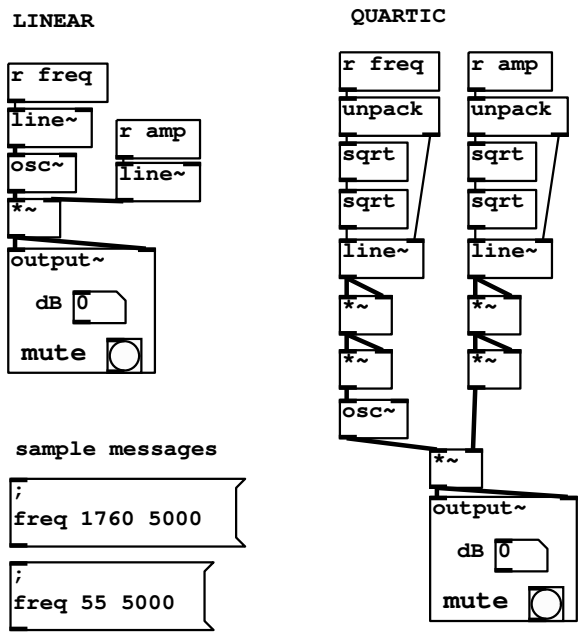


Figure 4.15: Linear and quartic transfer functions for changing amplitude and pitch.

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. 50).

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.

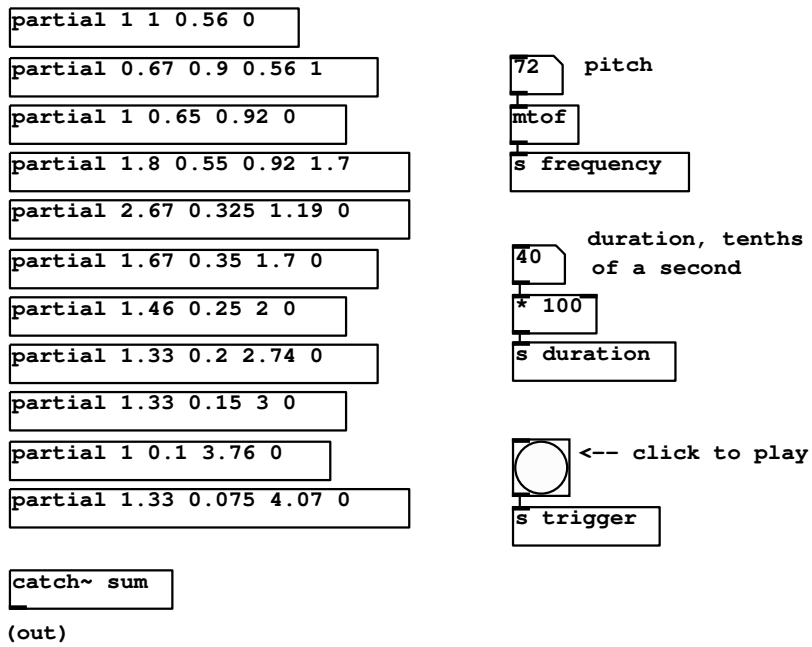


Figure 4.16: A Pd realization of Jean-Claude Risset's bell instrument. The bell sound is made by summing 11 sinusoids, each made by a copy of the `partial` abstraction.

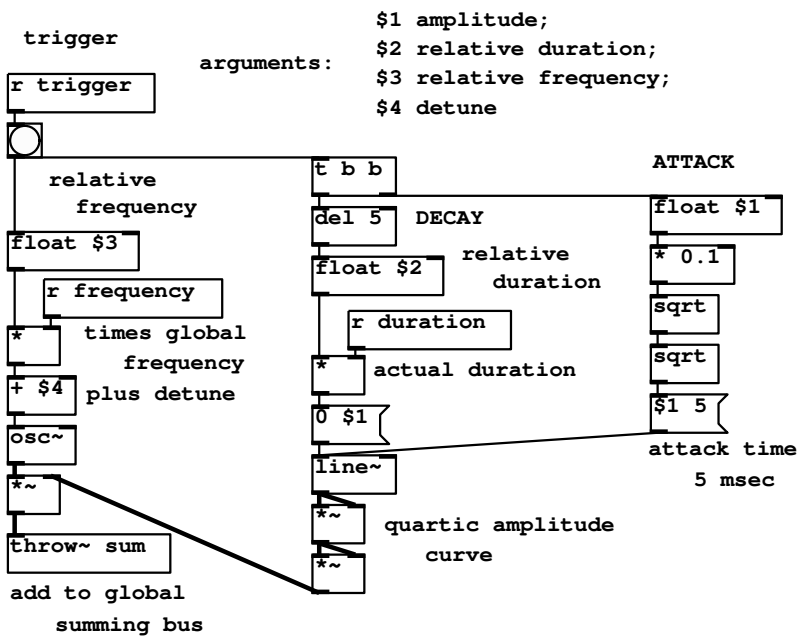


Figure 4.17: The `partial` abstraction used by Risset's bell instrument from figure 4.16.

`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

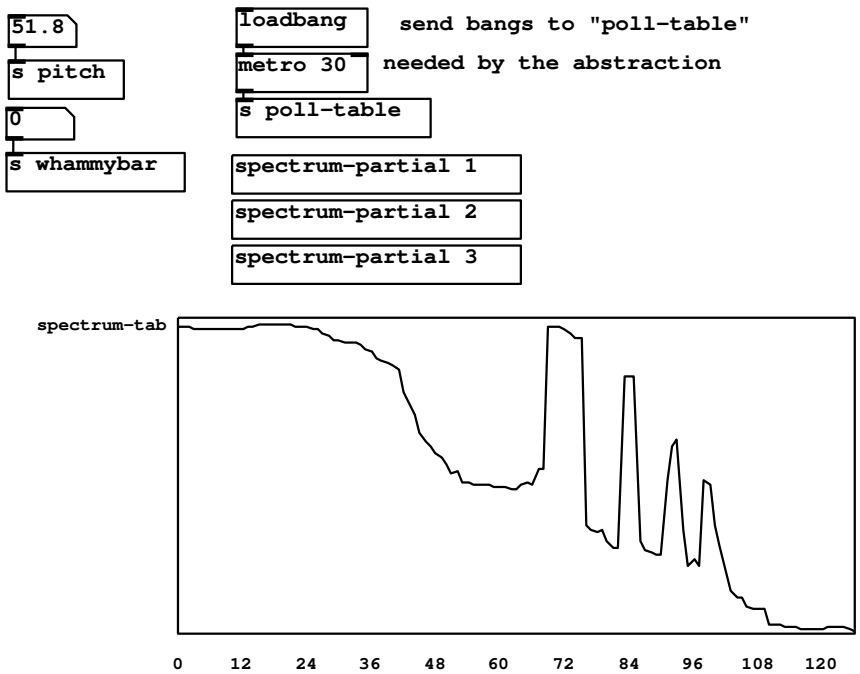


Figure 4.18: Additive synthesis for a specified spectral envelope, drawn in a table.

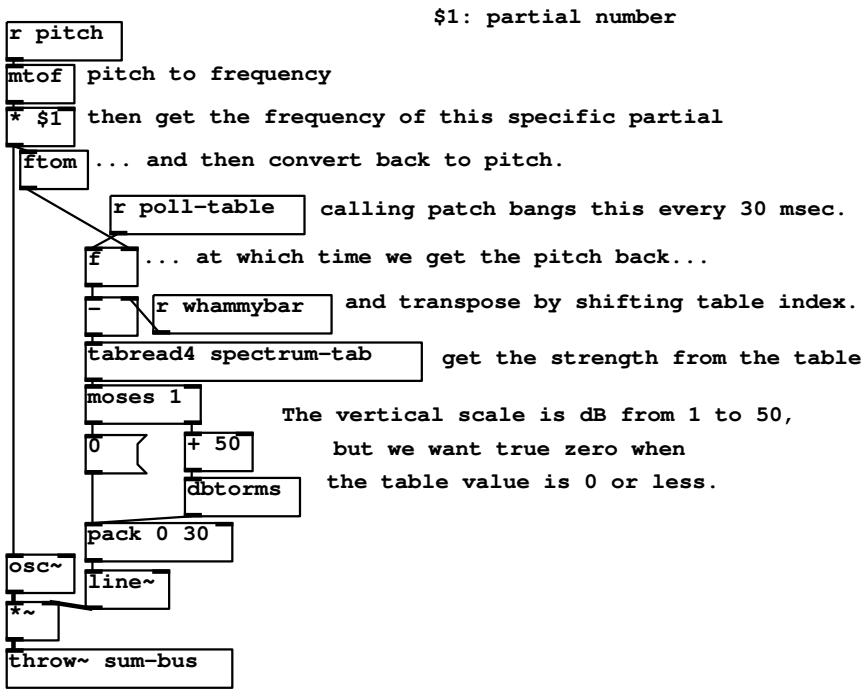


Figure 4.19: The `spectrum-partial` abstraction used in figure 4.18.

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.

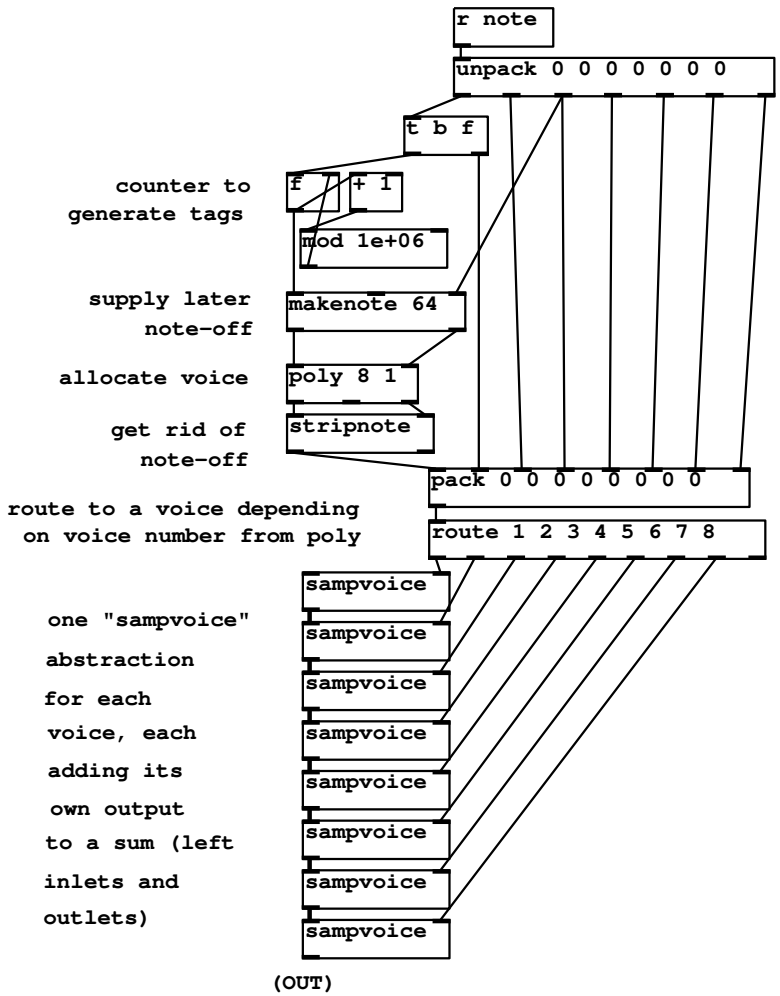


Figure 4.20: A polyphonic sampler demonstrating voice allocation and use of tags.

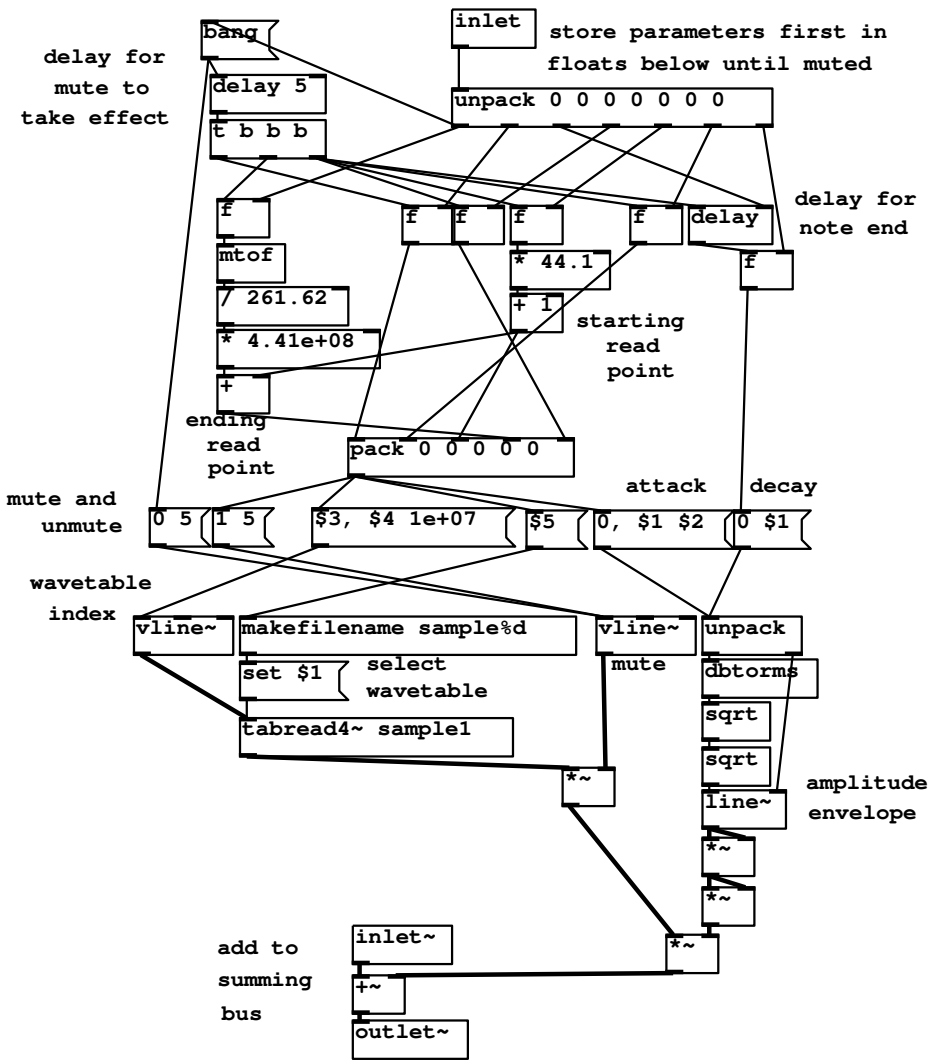


Figure 4.21: The `sampvoice` abstraction used in the polyphonic sampler of figure 4.20.

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 `tabread4~` 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 `tabread4~` 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 `tabread4~` 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

Figure 5.1 introduces a way of visualizing the *spectrum* of an audio signal. The spectrum shows how the signal's power is distributed into frequencies. (There is a real mathematical definition of the idea, but it requires much mathematical background that can't be assumed here.)

Part (a) of the figure shows the spectrum of a *harmonic signal*, which is a periodic signal whose fundamental frequency is in the range of perceptible pitches, roughly between 50 and 4000 Hz. The FOURIER SERIES (page 13) 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.)

For a harmonic 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 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.)

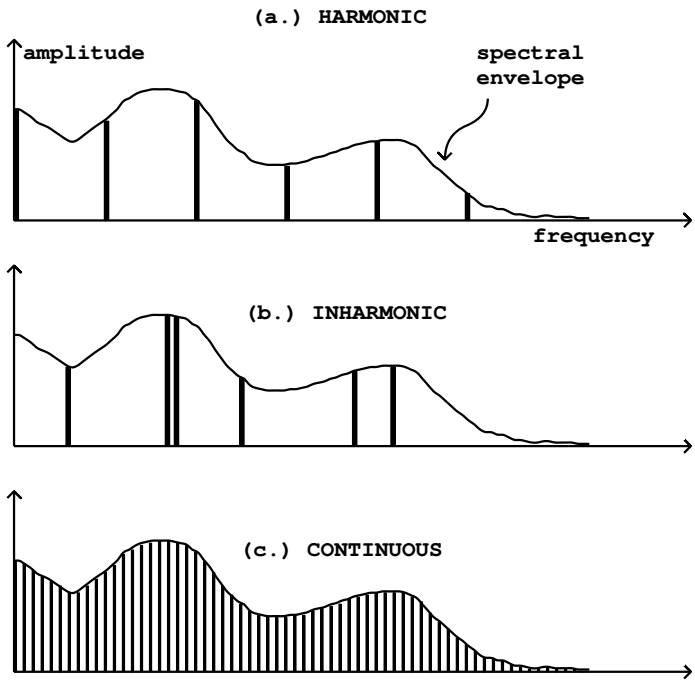


Figure 5.1: A taxonomy of timbres. The spectral envelope describes the shape of the spectrum. The sound may be discretely or continuously distributed in frequency; if discretely, it may be harmonic or inharmonic.

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.

Part (b) of the figure shows a spectrum which is also discrete, so that 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 audible common submultiple of all the partials. We call this an *inharmonic* signal. (We'll use the terms *harmonic* and *inharmonic* to describe the signals' spectra, as well as the signals themselves.)

In dealing with discrete spectra, we report each partial's RMS amplitude (section 1.1), not its peak amplitude. So 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 $\omega = \phi = 0$, the RMS amplitude is a —with no corresponding division by two. The decision to use RMS and not peak amplitude will simplify our analysis of the effects of various modulation algorithms later in this chapter. A deeper reason for this convention will become apparent in chapter 7.

Part (c) of the figure shows a third possibility, which is that the spectrum might not be concentrated into a discrete set of frequencies, but instead 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.

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.

We have been speaking of spectra here as static entities, not considering whether they change in time or not. If a signal's pitch and timbre are changing over time, we would like to think of the pitch and spectrum as descriptions of the signal's momentary behavior, which can also change over time.

This way of viewing sounds is greatly oversimplied. The true behavior of audible pitch and timbre has many aspects 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 formula holds, we can use the formula for the cosine of a sum of two angles:

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

to 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 3):

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

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.) These components are called *sidebands*.

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

Figure 5.3 shows a variety of results that may be obtained by multiplying a (modulating) sinusoid of angular frequency α and RMS amplitude a, by a (carrier) sinusoid of angular frequency β and amplitude 1:

$$[2a\cos(\alpha n)] \cdot [\cos(\beta n)].$$

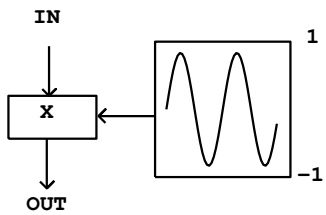


Figure 5.2: Block diagram for ring modulating an input signal with a sinusoid.

(For simplicity we're omitting the phase term here.) Each part of the figure shows both the modulation signal and tbe result in the same spectrum. The modulating signal appears as a single frequency, α, at amplitude a. The product in general has two component frequencies, each at an amplitude of $a/2$.

Parts (a) and (b) of the figure show "general" cases where α and β are nonzero and different from each other. The component frequencies of the output are $\alpha + \beta$ and $\alpha - \beta$. In part (b), since $\alpha - \beta < 0$, we get a negative frequency component. Since cosine is an even function, we have

$$\cos((\alpha - \beta)n) = \cos((\beta - \alpha)n)$$

so the negative component is exactly equivalent to one at the positive frequency $\beta - \alpha$, at the same amplitude. We still refer to the two resulting peaks as sidebands, even when they both happen to lie to the right of the original peak.

In the special case where $\alpha = \beta$, the second (difference) sideband has zero frequency. In this case phase will be significant so we rewrite the product with explicit phases, replacing β by α, to get:

$$2a\cos(\alpha n + \phi)\cos(\alpha n + \xi) =$$

$$= a\cos(2\alpha n + (\phi + \xi)) + a\cos(\phi - \xi).$$

The second term has zero frequency; its amplitude depends on the relative phase of the two sinusoids and ranges from $+a$ to $-a$ as the phase difference $\phi - \xi$ varies from 0 to π radians. This situation is shown in part (c) of Figure 5.3.

Finally, part (d) of the figure shows a carrier signal whose frequency is zero. Its value is just the constant a. In this case we get only one sideband, of amplitude $a/2$ as usual.

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, such as:

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

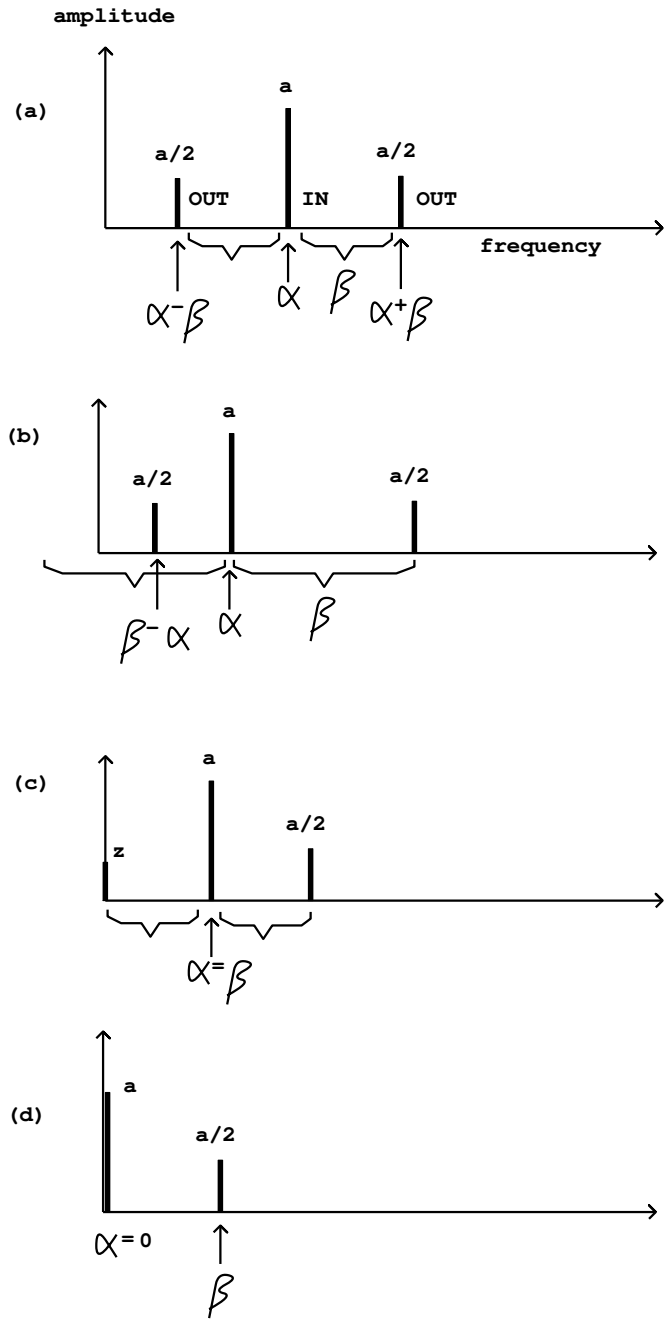


Figure 5.3: Sidebands arising from multiplying two sinusoids of frequency α and β. Part (a) shows the case where $\alpha > \beta > 0$; part (b) shows the case where $\beta > \alpha$ but $\beta < 2\alpha$, so that the lower sideband is reflected about the $f = 0$ axis. Part (c) shows the situation when $\alpha = \beta$; 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.

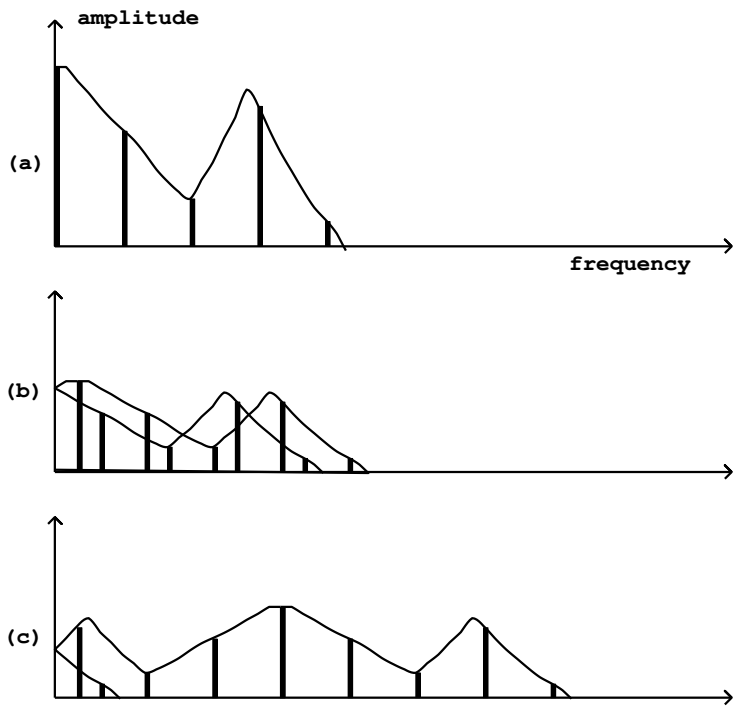


Figure 5.4: Result of ring modulation of a complex signal by a pure sinusoid: (a) the original signal's spectrum and spectral envelope; (b) modulated by a relatively low modulating frequency (1/3 of the fundamental); (c) modulated by a higher frequency, 10/3 of the fundamental.

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

$$\alpha_1 + \beta, \alpha_1 - \beta, \ldots, \alpha_k + \beta, \alpha_k - \beta.$$

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

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.

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

the shifting is by a relatively small distance, 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 often 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 also—importantly—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 5.6 shows a familiar example of waveshaping, in which $f()$ 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

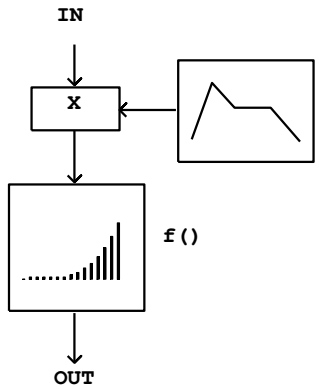


Figure 5.5: Block diagram for waveshaping an input signal using a nonlinear function $f()$. An amplitude adjustment step precedes the function lookup, to take advantage of the different effect of the wavetable lookup at different amplitudes.

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

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

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 section: the output is a DC (zero-frequency) signal plus a signal at twice the original frequency. However, in this waveshaping example, unlike the situation in 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

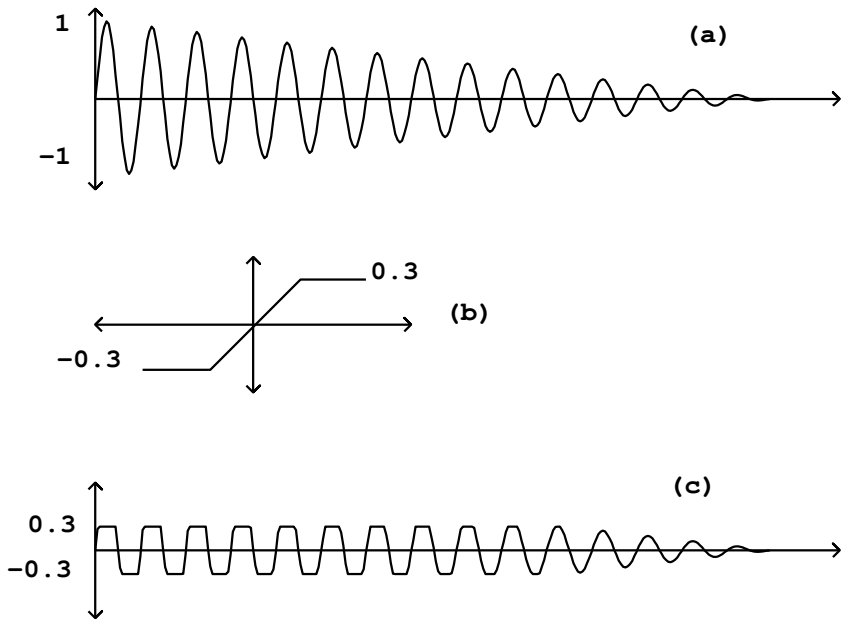


Figure 5.6: Clipping as an example of waveshaping. The input (a) is a sinusoid of varying amplitude, and the waveshaping function (b) clips its input to the interval between -0.3 and +0.3. The output is shown in (c).

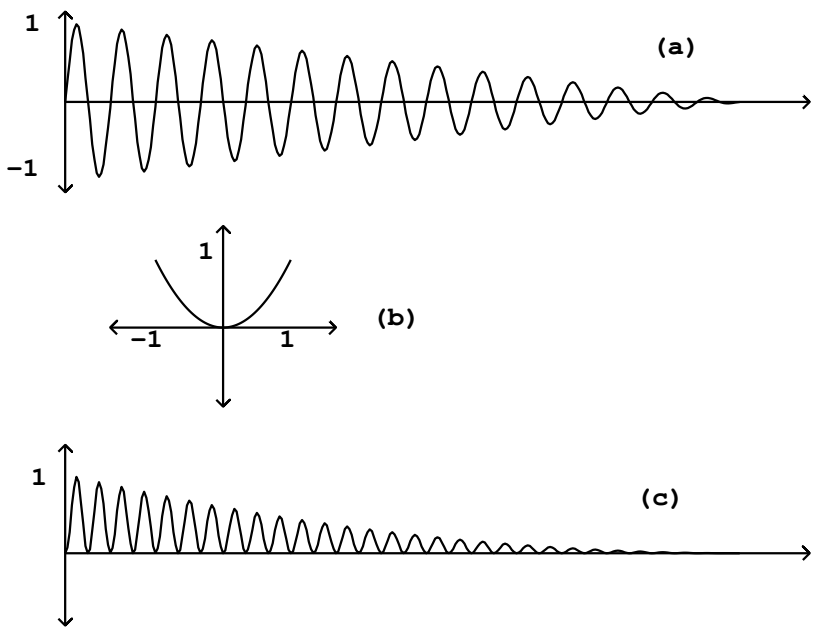


Figure 5.7: Waveshaping using a quadratic transfer function $f(x) = x^2$. The input (a) yields an output (c) at twice the frequency.

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

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

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

$$+ab\left[\cos((\alpha+\beta)n) + \cos((\alpha-\beta)n)\right]$$

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 a linear function of its input signal, 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 repeat at the same period: if the period is τ so that

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

then we immediately get

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

(In some special cases the output can repeat at a submultiple of τ, so that we get a harmonic of the input as a result; we'll see this happen later.)

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 power 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:

$$f(x[n]) = f_0 + af_1\cos(\omega n) + a^2 f_2 \cos^2(\omega n) + a^3 f_3 \cos^3(\omega n) + \cdots$$

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. Each kth row can be approximated by a Gaussian curve whose standard deviation (a measure of width) is proportional to 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 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:

$$f(x) = f_0 + f_2x^2 + f_4x^4 + \cdots$$

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 (example 5.5.4 demonstrates this.) 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 6.

5.4 Frequency and phase 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_m, about a central frequency ω_c, so that the instantaneous frequencies vary between $(1-r)\omega_c$ and $(1+r)\omega_c$, with parameters ω_m controlling the frequency of variation, and r controlling the depth of variation. The parameters ω_c, ω_m, and r are called the *carrier frequency*, the *modulation frequency*, and the *index of modulation*, respectively.

It is customary to use a simpler, essentially equivalent formulation in which the phase, instead of the frequency, of the carrier sinusoid is modulated sinusoidally. (This gives an equivalent result 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 resulting signal can be written as

$$x[n] = \cos(a\cos(\omega_m n) + \omega_c n)$$

The parameter a, which takes the place of the earlier parameter r, is also called the index of mosulation; it too controls the extent of frequency variation relative to the carrier frequency ω_c. If $r = 0$, there is no frequency variation and the expression reduces to the unmodified, carrier sinusoid:

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

To analyse the resulting spectrum we can write,

$$x[n] = \cos(\omega_c n) * \cos(a\cos(\omega_m n))$$

$$-\sin(\omega_c n) * \sin(a\cos(\omega_m n)),$$

so we can consider it as a sum of two waveshaping generators, each operating on a sinusoid of frequency ω_m and with a waveshaping index a, and each ring modulated with a sinusoid of frequency ω_c. 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

$$\cos(r\cos(\omega_m n))$$

and

$$\sin(r\cos(\omega_m n))$$

have, respectively, harmonics tuned to

$$0, 2\omega_m, 4\omega_m, \ldots$$

and

$$\omega_m, 3\omega_m, 5\omega_m, \ldots$$

and each is multiplied by a sinusoid at the carrier frequency. So there will be a spectrum centered at the carrier frequency ω_c, with sidebands at both even and odd multiples of the modulation frequency ω_m, contributed respectively by the sine and cosine 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

$$\omega_c + m\omega_m$$

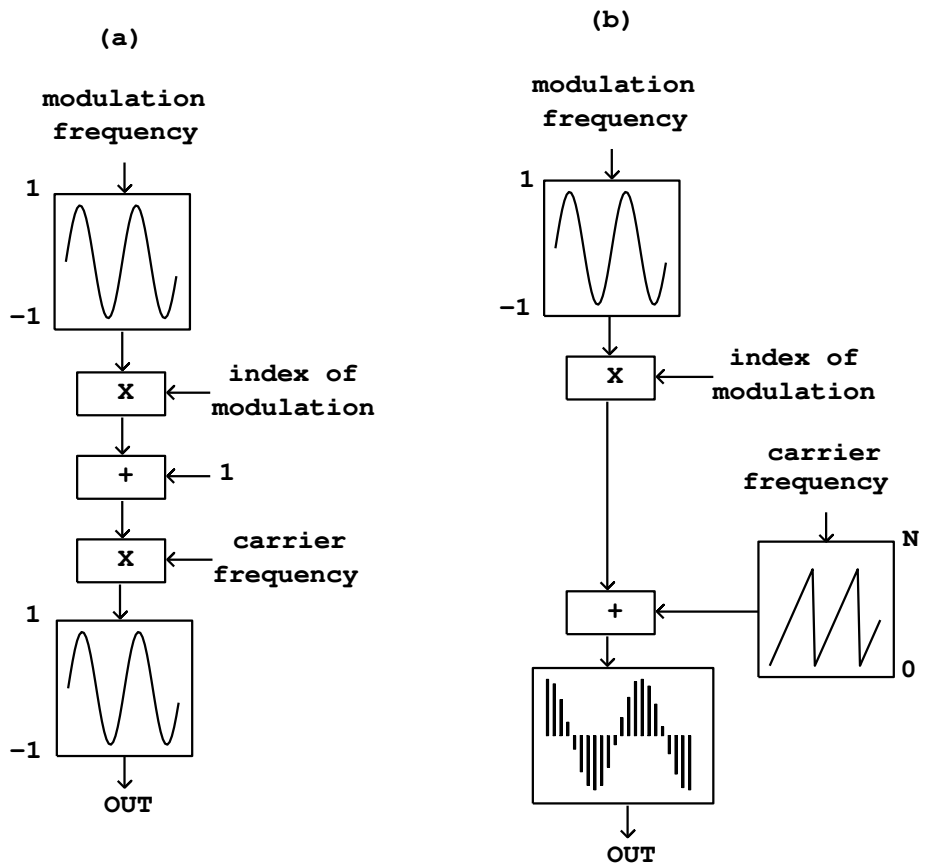


Figure 5.8: Block diagram for frequency modulation (FM) synthesis: (a) the classic form; (b) realized as phase modulation.

where

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

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 $\omega_m = k\omega$ and $\omega_c = m\omega$, 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 following examples.

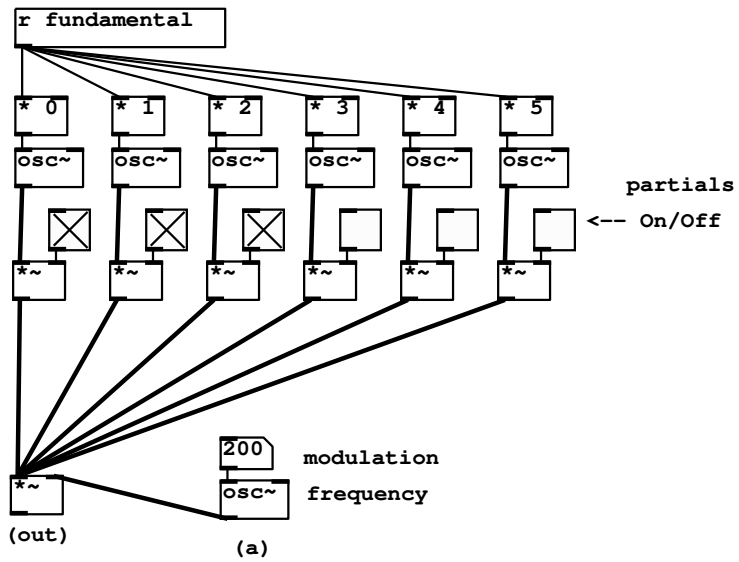
5.5 Examples

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. Here we will skip that and start with the second example, E02.ring.modulation.pd, which shows the effect of ring modulating a harmonic spectrum (which was 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. The partials may be turned on and off separately using toggle switches. When they are all on, the spectral envelope peaks at DC (because the constant signal is twice as strong as the other sinusoids), 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 9, 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.)



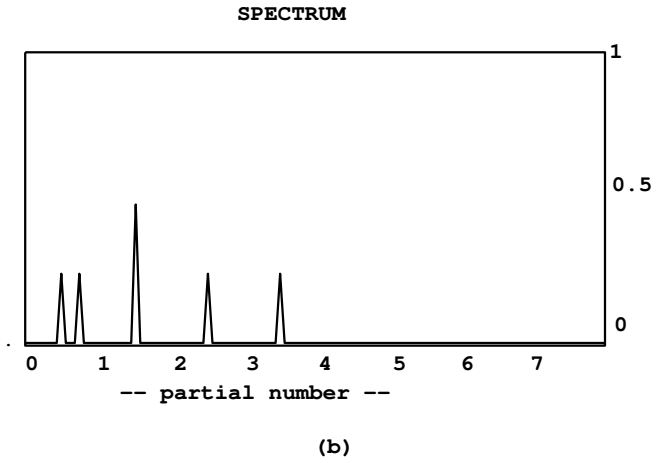


Figure 5.9: Ring modulation of a complex tone by a sinusoid: (a) its realization; (b) a measured spectrum

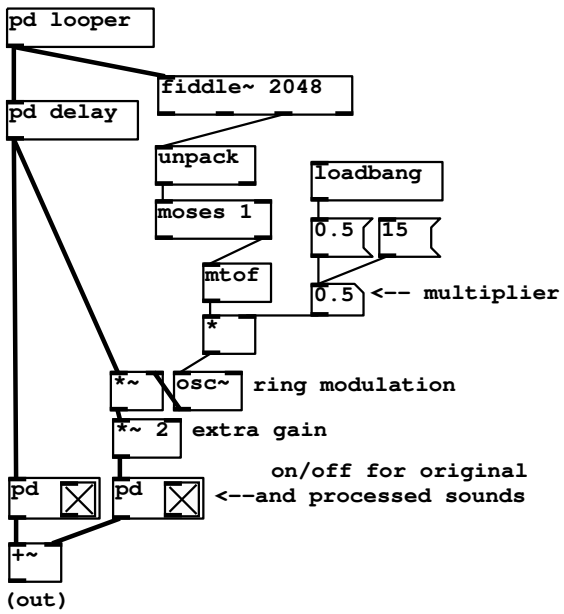


Figure 5.10: Lowering the pitch of a sound by an octave by determinating its pitch and modulating at half the fundamental.

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 signal we process here is a recorded, spoken voice.

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 7. 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

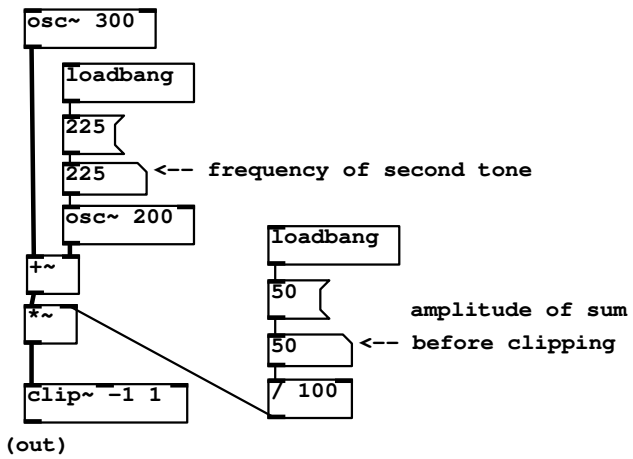


Figure 5.11: Nonlinear distortion of a sum of two sinusoids to create a difference tone.

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 components, 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. In the first case we get an octave divider, and in the second, additional high harmonics that deform the vowels.

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 was 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

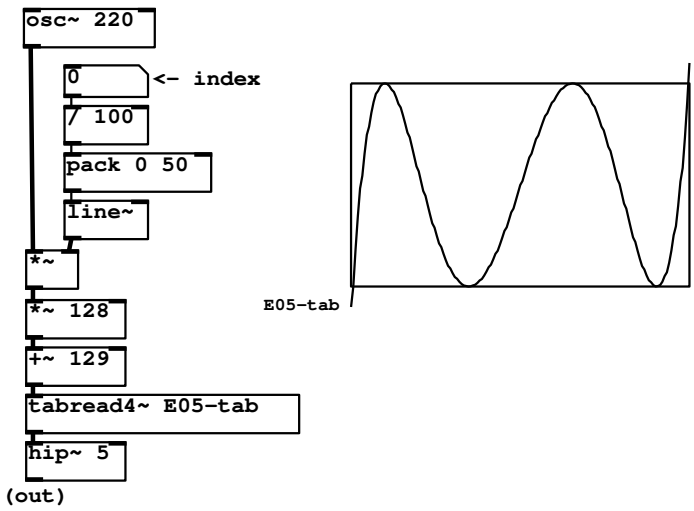


Figure 5.12: Using Chebychev polynomials as waveshaping transfer functions.

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 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 more 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 131), which gives first, third and fifth harmonics:

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

Next we add a suitable multiple of x^3 to cancel the third harmonic:

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

and then a multiple of x to cancel the first harmonic:

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

So for our waveshaping function we choose

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

This procedure allows us 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

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

where $a[n]$ is a suitable *index* which may vary as a function of the sample number n. When a happens to be one in value, out comes the pure fifth harmonic. Other values of a give varying spectra which, in general, have first and third harmonics as well as the fifth.

By suitably combining Chebychev polynomials we can fix any desired superposition of components in the output waveform (again, as long as the waveshaping index is one). 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 131, 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.

Now suppose we wish to arrange for different terms in the above expansion to dominate the result in a predictable way as a function of the index a. To choose 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

$$f_1 = f_0, 2f_2 = f_1, 3f_3 = f_2, \ldots$$

and so fixing f_0 at 1, we get $f_1 = 1$, $f_2 = 1/2$, $f_3 = 1/6$, and in general,

$$f_k = \frac{1}{1 \cdot 2 \cdot 3 \cdot ... \cdot k}$$

These are just the coefficients of 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.

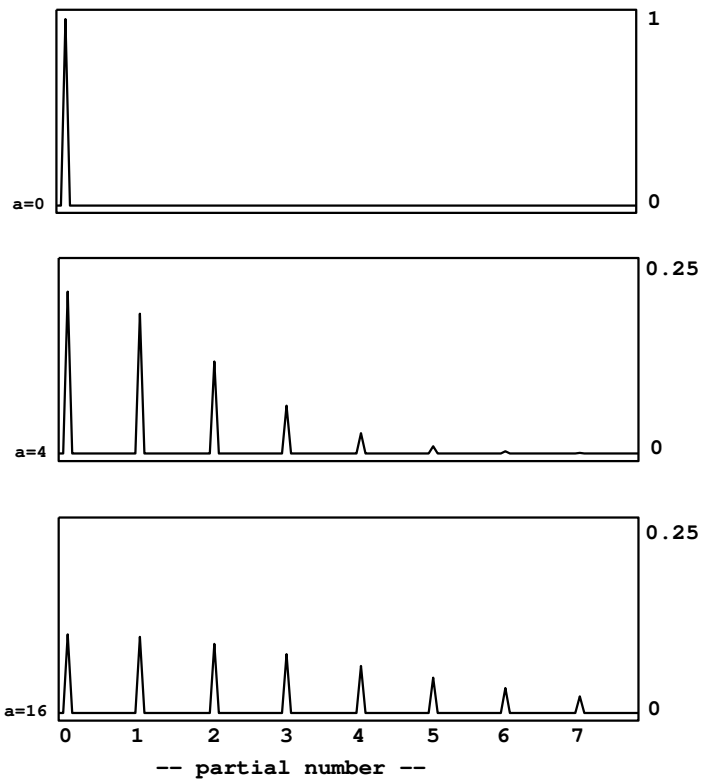


Figure 5.13: Spectra of waveshaping output using an exponential transfer function. Indices of modulation of 0, 4, and 16 are shown; note the different vertical scales.

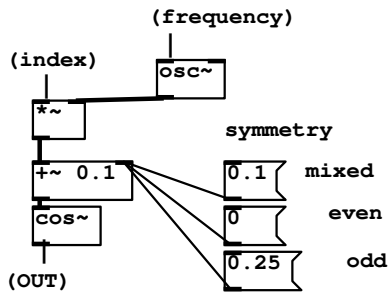


Figure 5.14: Using an additive offset to a cosine transfer function to alter the symmetry between even and odd. With no offset the symmetry is even. For odd symmetry, a quarter cycle is added to the phase. Smaller offsets give a mixture of even and odd.

5.5.7 Phase modulation and FM

Patch E08.phase.mod.pd, shown in Figure 5.15, shows how to use Pd to realize true frequency modulation (part a) and phase modulation (part b). These correspond to the block diagrams of Figure 5.8. 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.

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.

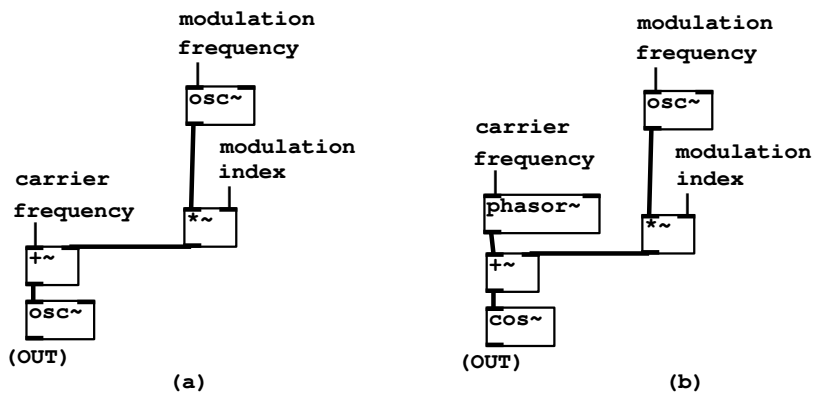


Figure 5.15: Frequency modulation (a) and phase modulation (b) compared.

Figure 5.16 shows some two-operator phase modulation spectra, measured using Patch E09.FM.spectrum.pd.

Phase modulation can thus be seen simply as a form of ring modulated waveshaping. So we can use the strategies described in section 5.2 to generate particular combinations of frequencies. For example, if the carrier frequency is half the modulation frequency, you get a sound with odd harmonics exactly as in the octave dividing example (5.5.2).

Frequency modulation need not be restricted to purely sinusoidal carrier or modulation oscillators. One well-trodden path is to effect phase modulation on the phase modulation 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

$$\omega_c + 3\omega_m + k\omega_p, k = 0, \pm 1, \pm 2, \ldots$$

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 modulation indices, for simple and compact phase modulation

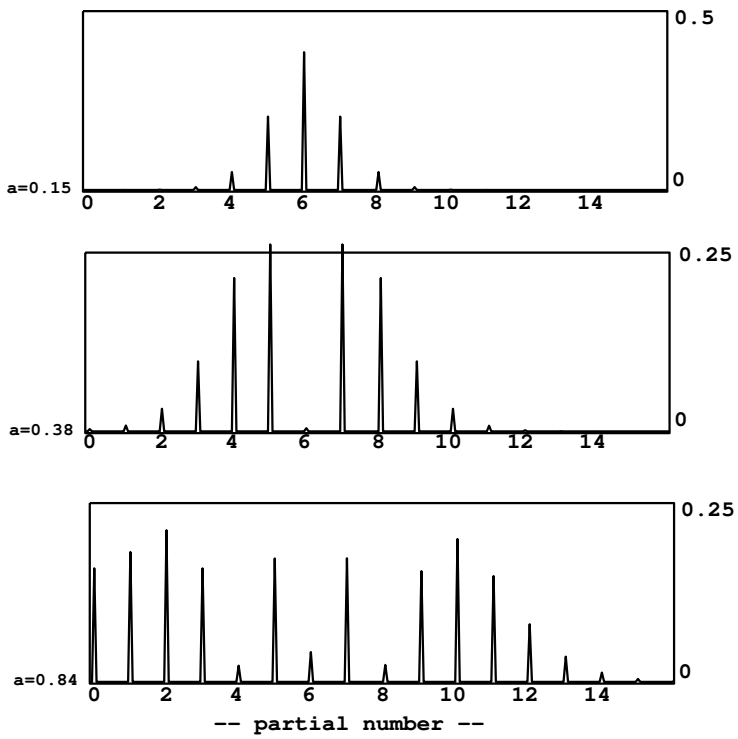


Figure 5.16: Spectra from phase modulation at three different indices. The indices are given as multiples of 2π radians.

instruments, 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 if you wanted to?

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)?

4. A sinusoid at a frequency ω is ring modulated by another sinusoid at exactly the same frequency. At what phase differences will the DC component of the result disappear?

Chapter 6

Designer spectra

As suggested at the beginning of the previous chapter, a powerful way to synthesize musical sounds is by specfying—and then realizing—specific trajectories of pitch (or more generally, frequencies of partials), along with trajectories of spectral envelope.[1] The spectral envelope is used to determine the amplitude of the individual partials, as a function of their frequencies, and is thought of as controlling the sound's (possibly time-varying) timbre.

A simple example of this would be to imitate a plucked string by constructing a sound with harmonically spaced partials in which the spectral envelope starts out rich but then dies away exponentially with higher frequencies decaying faster than lower ones, so that the timbre mellows over time. Spectral-evolution models for various acoustic instruments have been proposed [GM77] [RM69] . A more complicated example is the spoken or sung voice, in which vowels appear as spectral envelopes, dipthongs and many consonants appear as time variations in the spectral envelopes, and other consonants appear as spectrally shaped noise.

Spectral envelopes may be obtained from analysis of recorded sounds (see chapter 9 or from purely synthetic criteria. To specify a spectral envelope from scratch for every possible frequency would be tedious, and in most cases you would want to describe them in terms of their salient features. The most popular way of doing this is to specify the size and shape of the spectral envelope's peaks, which are called *formants*. Figure 6.1 shows a spectral envelope with two formants. Although the shapes of the two peaks in the spectral envelope are different, they can both be roughly described by giving the coordinates of each apex (which give the formant's *center frequency* and amplitude) and each formant's *bandwidth*. A typical measure of bandwidth would be the width of the peak at a level 3 decibels below its apex. Note that if the peak is at (or near) the $f = 0$ axis, we pretend it falls off to the left at the same rate as (in reality) it falls off to the right.

Suppose we wish to generate a harmonic sound with a specified collection

[1] This chapter loosely follows the treatment from an earlier publication [Puc01].

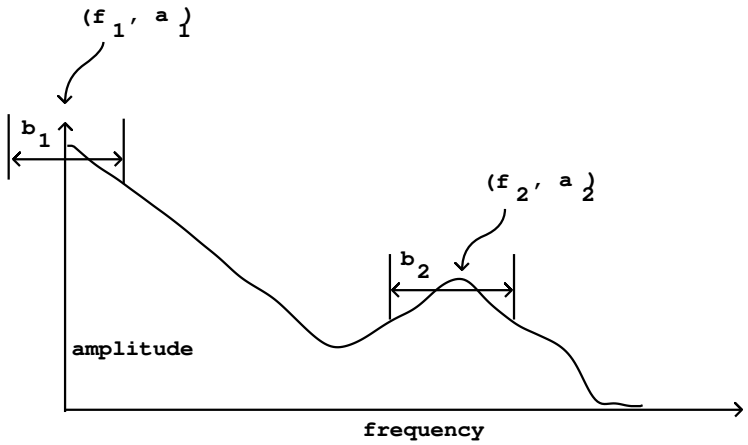


Figure 6.1: A spectral envelope showing the frequencies, amplitudes, and bandwidths of two *formants*.

of formants. Independently of the fundamental frequency desired, we wish the spectrum to have peaks with prescribed center frequencies, amplitudes, and bandwidths. Returning to the phase modulation spectra shown in Figure 5.16, we see that, at small indices of modulation at least, the result has a single, well-defined spectral peak. We can imagine adding several of these, all sharing a fundamental (modulating) frequency but with carriers tuned to different harmonics to select the various desired center frequencies, and with indices of modulation chosen to give the desired bandwidths. This was first explored by Chowning [Cho89] who arranged formants generated by phase modulation to synthesize singing voices. In this chapter we'll establish a general framework for building harmonic spectra with desired, possibly time-varying, formants.

6.1 Carrier/modulator model

In the previous chapter (see figure 5.4, page 125), we showed how to use ring modulation to modify the spectrum of a periodic signal, placing spectral peaks in specified locations. To do so we need to be able to generate periodic signals whose spectra have maxima at DC and fall off monotonically with increasing frequency. If we can make a signal with a formant at frequency zero—and no other formants besides that one—we can use ring modulation to displace the formant to any desired harmonic. The ring modulation product will be of the form

$$x[n] = \cos(\omega_c n) f(a \cos(\omega_m n))$$

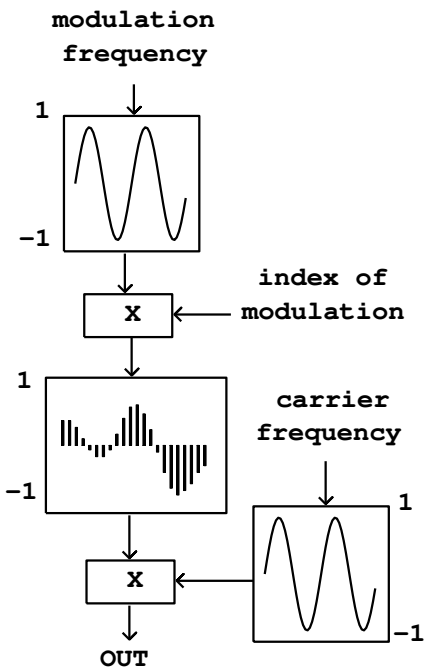


Figure 6.2: Ring modulated waveshaping for formant generation

where ω_c (the *carrier frequency*) is set to the formant center frequency and $f(a \cdot \cos(\omega_m n))$ is a signal with fundamental frequency ω_m, produced using a waveshaping function f and index a. This second term is the signal we wish to give a formant at DC with a controllable bandwidth. A block diagram for synthesizing this signal is shown in Figure 6.2.

Much earlier in section 2.4 we introduced the technique of *timbre stretching*, as part of the discussion of wavetable synthesis. This technique, which is capable of generating complex, variable timbres, can be fit into the same framework. The enveloped wavetable output for one cycle is:

$$x(\phi) = T(c\phi) * W(a\phi),$$

where ϕ, the phase, satisfies $-\pi \leq \phi \leq \pi$. Here T is a function stored in a wavetable, W is a windowing function, and c and a are the wavetable stretching and a modulation index for the enveloping wavetable. Figure 6.3 shows how to realize this in block diagram form. Comparing this to figure 2.7, we see that the only significant new feature is the addition of an index to the waveshaping function.

In this setup, as in the previous one, the first term specifies the placement of energy in the spectrum—in this case, with the parameter c acting to stretch

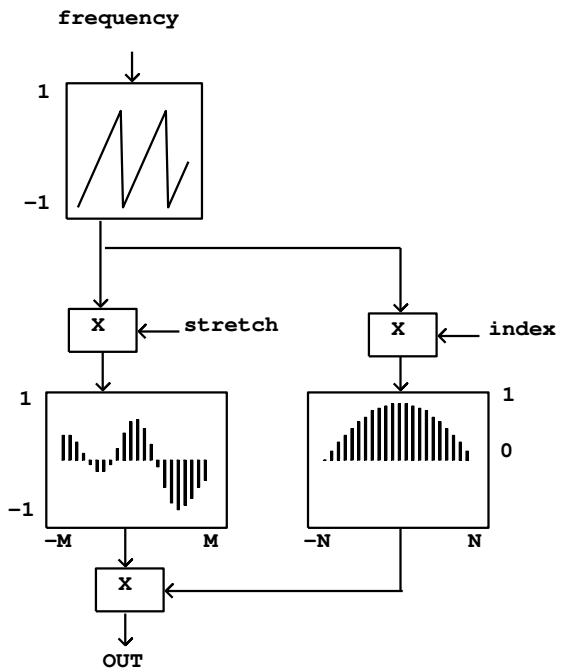


Figure 6.3: Wavetable synthesis generalized as a variable spectrum generator

out the wavetable spectrum. This is the role that was previously carried out by the choice of ring modulation carrier frequency ω_c.

Both of these (ring modulated waveshaping and stretched wavetable synthesis) can be considered as particular cases of a more general approach which is to compute functions of the form,

$$x[n] = c(\omega n)m_a(\omega n)$$

where c is a periodic function describing the carrier signal, and m_a is a periodic modulator function which depends on an index a. The modulation functions we're interested in will usually take the form of pulse trains, and the index a will control the width of the pulse; higher values of a will give narrower pulses. In the wavetable case, the modulation function must reach zero at phase wraparound points to suppress any discontinuities in the carrier function when the phase wraps around. The carrier signal will give rise to a single spectral peak (a formant) in the ring modulated waveshaping case; for wavetables, there may be a much more complicated spectrum.

In the next section we will further develop the two forms of modulating signal we've introduced here, and in the following one we'll look more closely at the carrier signal.

6.2 Pulse trains

Pulse trains may be generated either using the waveshaping formalism or the stretched wavetable one.

6.2.1 Pulse trains via waveshaping

In the ring modulated waveshaping formulation, the shape of the formant is determined by a modulation term

$$m[n] = f(a\cos(\omega_m n))$$

For small values of the index a, the modulation term varies only slightly from the constant value $f(0)$, so most of the energy is concentrated at DC. As a increases, the energy spreads out among progressively higher harmonics of the fundamental ω_m. Depending on the function f, this spread may be orderly or disorderly. An orderly spread may be desirable and then again may not, depending on whether our goal is a predictable spectrum or a wide range of different (and perhaps hard-to-predict) spectra.

A waveshaping function that gives well-behaved, simple and predictable results was already developed in section 5.5.5. Using the function

$$f(x) = e^x$$

and normalizing suitably, we get the spectra shown in figure 5.13. A slight rewriting of the waveshaping modulator for this choice of f (and taking the renormalization into account) gives:

$$\begin{aligned} m_a(\omega n) &= e^{a\cdot(\cos(\omega n)-1))} \\ &= e^{-\left[b\sin\frac{\omega}{2}\right]^2} \end{aligned}$$

where $b^2 = 2a$ so that b is proportional to the bandwidth. This can be rewritten as

$$\begin{aligned} m_a(\omega n) &= g(b\sin\frac{\omega}{2}n) \\ g(x) &= e^{-x^2} \end{aligned}$$

Except for a missing normalization factor, this is a Gaussian distribution, sometimes called a "bell curve". The amplitudes of the harmonics are given by Bessel "I" type functions.

Another good choice is the (again unnormalized) Cauchy distribution:

$$h(x) = \frac{1}{1+x^2}$$

which gives rise to a spectrum of exponentially falling harmonics:

$$h(b\sin(\omega n/2)) = G\cdot\left(\frac{1}{2} + H\cos(\omega n) + H^2\cos(2\omega n) + \cdots\right)$$

where G and H are functions of the index b (exact formulas are given in [Puc95a]).

In both this and the Gaussian case above, the bandwidth (counted in peaks, i.e., units of ω) is roughly proportional to the index b, and the amplitude of the DC term (the peak of the spectrum) is roughly proportional to $1/(1+b)$. For either waveshaping function (g or c), if b is larger than about 2, the waveshape of $m_a(\omega n)$ is approximately a (forward or backward) scan of the transfer function, and so this and the earlier example (the "wavetable formulation") both look like pulses whose widths decrease as the specified bandwidth increases.

Before considering more complicated carrier signals to go with the modulators we've seen so far, it is instructive to see what multiplication by a pure sinusoid gives us as waveforms and spectra. Figure 6.5 shows the result of multiplying two different pulse trains by a sinusoid at the sixth partial:

$$cos(6\omega n)M_a(\omega n)$$

where the index of modulation a is two in both cases. In part (a) M_a is the stretched Hanning windowing function; part (b) shows waveshaping via the unnormalized Cauchy distribution. One period of each waveform is shown.

6.2.2 Pulse trains via wavetable stretching

In the wavetable formulation, the pulse train can be made by a stretched wavetable:

$$M_a(\phi) = W(a\phi),$$

where $-\pi \leq \phi \leq \pi$ is the phase. The function W should be zero at and beyond the points $-\pi$ and π, and rise to a maximum at 0. A possible choice for the function W is

$$W(\phi) = \frac{1}{2}(\cos(\phi) + 1)$$

which is graphed in part (a) of Figure 6.4. This is known as the *Hanning window function*; it will come up again in chapter 9.

Realizing this as a repeating waveform, we get a succession of (appropriately sampled) copies of the function W, whose duty cycle is $1/a$ (parts b and c of the figure). If you don't wish the copies to overlap we require a to be at least 1. If you want overlap the best strategy is to duplicate the block diagram (Figure 6.3) out of phase, as described in Section 2.4 and realized in Section 2.6.5.

6.2.3 Resulting spectra

In both cases we see, in effect, the sixth harmonic (the carrier signal) enveloped into a *wave packet* centered at the middle of the cycle, where the phase of the sinusoid is zero. Changing the frequency of the sinusoid changes the center frequency of the formant; changing the width of the packet (the proportion of the waveform during which the sinusoid is strong) changes the bandwidth. Note

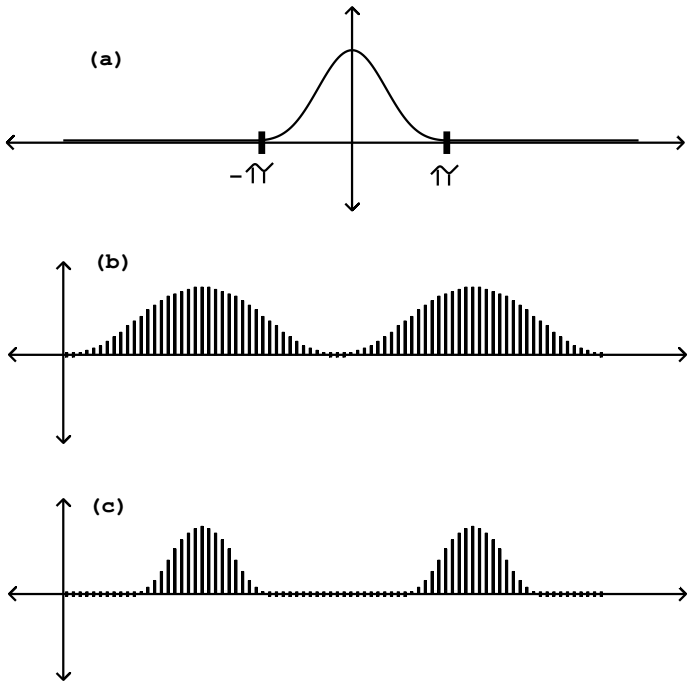


Figure 6.4: Pulse width modulation using the Hanning window function: a. the function $W(\phi) = (1 + \cos(\phi))/2$; b. the function repeated at a duty cycle of 100% (modulation index $a = 1$); c. the function at a 50% duty cycle ($a = 2$).

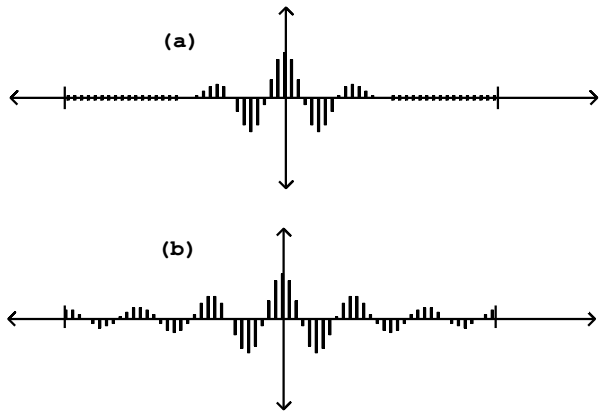


Figure 6.5: Audio signals resulting from multiplying a cosine (partial number 6) by pulse trains: (a). windowing function from the wavetable formulation; (b) waveshaping output using the Gaussian lookup function.

that the stretched Hanning window function is zero at the beginning and end of the period, unlike the waveshaping packet.

Figure 6.6 shows how the specific shape of the formant depends on the method of production. The stretched wavetable form (part (a) of the figure) behaves well in the neighborhood of the peak, but somewhat oddly starting at four partials' distance from the peak, past which we see what are called *sidelobes*: spurious extra peaks at lower amplitude than the central peak. As the analysis of Section 2.4 predicts, the entire formant, sidelobes and all, stretches or contracts inversely as the pulse train is contracted or stretched in time.

The first, strongest sidelobes on either side are about 37 dB lower in amplitude than the main peak. Further sidelobes drop off slowly when expressed in decibels; the amplitudes decrease as the square of the distance from the center peak so that the sixth sidelobe to the right, three times further than the first one from the center frequency, is about twenty decibels further down. The effect of these sidelobes is often audible as a slight buzziness in the sound.

This formant shape may be made arbitrarily fat (i.e., high bandwidth), but there is a limit on how thin it can be made, since the duty cycle of the waveform cannot exceed 100%. At this maximum duty cycle the formant strength drops to zero at two harmonics' distance from the center peak. If a still lower bandwidth is needed, waveforms may be made to overlap as described in section 2.6.5.

Parts (b) and (c) of the figure show formants generated using ring modulated waveshaping, with Gaussian and Cauchy transfer functions. The index of modulation is two in both cases (the same as for the Hanning window of part (a)), and the bandwidth is comparable to that of the Hanning example. In these examples there are no sidelobes, and moreover, the index of modulation may be

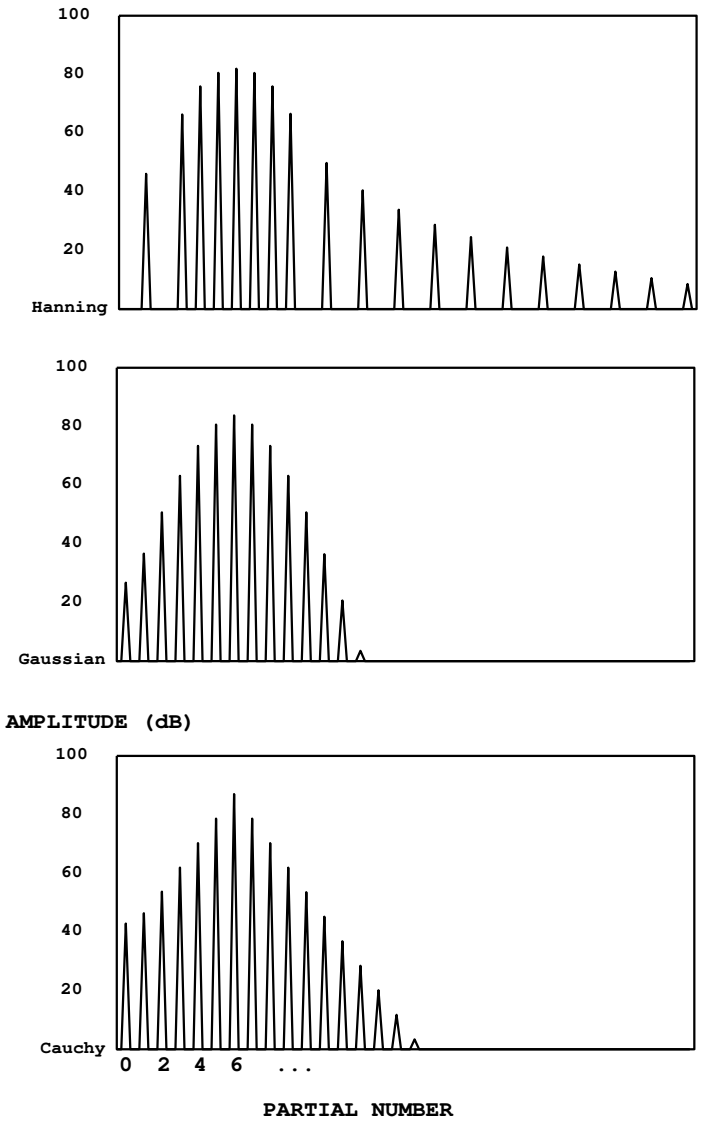


Figure 6.6: Spectra of three ring-modulated pulse trains: (a). the Hanning window function, 50% duty cycle (corresponding to an index of 2); (b), a wave-shaping pulse train using a Gaussian transfer function; (c) the same, with a Cauchy transfer function. Amplitudes are in decibels.

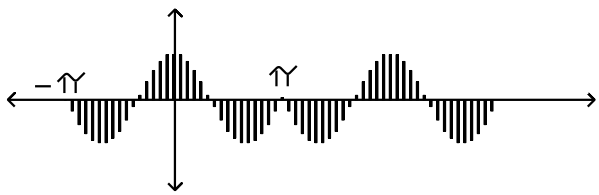


Figure 6.7: Waveform for a wavetable-based carrier signal tuned to 1.5 times the fundamental. Two periods are shown.

dropped all the way to zero, giving a pure sinusoid; there is no lower limit on bandwidth. On the other hand, since the waveform does not reach zero at the ends of a cycle, this type of pulse train cannot be used to window an arbitrary wavetable, as the Hanning pulse train could.

The Cauchy example is particularly easy to build designer spectra from, since the shape of the formant is a perfect Isosocles triangle, when graphed in decibels. On the other hand, the Gaussian example gathers more energy to the neighborhood of the formant, and drops off faster at the tails, and so has a cleaner sound.

6.3 Movable ring modulation

We turn now to the carrier signal and consider how to make it more controllable. In particular, we would like to be able to slide theh spectral energy continuously up and down. Simply sliding the frequency of the carrier oscillator will not accomplish this, since the spectra won't be harmonic except when the carrier is at an integer multiple of the fundamental frequency.

In the stretched wavetable approach we can accomplish this simply by sampling a sinusoid and transposing it to the desired "pitch". The transposed pitch isn't heard as a periodicity since the wavetable itself is read periodically at the fundamental frequency. Instead, the sinusoid is transposed as a spectral envelope.

Figure 6.7 shows a carrier signal produced in this way, tuned to produce a formant at 1.5 times the fundamental frequency. The signal has no outright discontinuity at the phase wraparound frequency, but it does have a discontinuity of slope, which, if not removed by applying a suitable modulation signal, would have very audible high-frequency components.

Using this idea we can make a complete description of how to use the block diagram of Figure 6.3 to produce a desired formant. The wavetable lookup on the left hand side would hold a sinusoid (placed symmetrically so that the phase is zero at the center of the wavetable). The right-hand-side wavetable would contain a Hanning window function (or perhaps another appropriate window-

ing function as will be developed in chapter 9. If we desire the fundamental frequency to be ω, the formant center frequency to be ω_f, and the bandwidth to be ω_b, we set the "stretch" parameter to the *center frequency quotient* ω_f/ω and the index of modulation to the *bandwidth quotient* ω_b/ω.

The output signal is simply a sample of a cosine wave at the desired center frequency, repeated at the (unrelated in general) desired period, and windowed to take out the discontinuities at period boundaries.

Although we aren't able to derive this result yet (we will need to do Fourier analysis, developed in chapter 9, it will turn out that, in the main lobe of the formant, the phases are all zero at the center of the waveform (i.e., all cosines if the waveform runs from $-\pi$ to π). This means we may superpose any number of these formants to build a more complex spectrum and the amplitudes of the partials will combine by addition. (The sidelobes don't behave quite this well: they are alternately of opposite sign and will produce cancellation patterns; but we usually can just treat them as a small, uncontrollable, residual signal.)

This method leads to an interesting generalization, which is to take a sequence of recorded wavetables, align all their component phases to those of cosines, and use them in place of the cosine function as the carrier signal. The phase alignment is necessary to allow coherent cross-fading between samples so that the spectral envelope can change smoothly. If, for example, we use successive snippets of a vocal sample as input, we get a strikingly effective vocoder; see section 9.6.

Another technique for making carrier signals that can be slid continuously up and down in frequency while maintaining a fundamental frequency is simply to cross-fade between harmonics. The carrier signal is then:

$$c(\phi) = c(\omega n) = p\cos(k\omega n) + q\cos((k+1)\omega n)$$

where $p + q = 1$ and k is an integer, all three chosen so that

$$(k + q) * \omega = \omega_f$$

so that the spectral center of mass of the two cosines is placed at ω_f. (Note that we make the amplitudes of the two cosines add to one instead of setting the total power to one; we do this because the modulator will operate phase-coherently on them.) To accomplish this we simply set k and q to be the integer and fractional part, respectively, of the center frequency quotient ω_f/ω.

The obvious way of making a control interface for this synthesis technique would be to use ramps to update ω and ω_f, and then to compute q and k as audio signals from the ramped, smoothly varying ω and ω_f. Oddly enough, despite the fact that k, p, and q are discontinuous functions of ω_f/ω, the carrier $c(\phi)$ turns out to vary continuously with ω_f/ω, and so if the desired center frequency is ramped from value to value the result is a continuous sweep in center frequency. However, more work is needed if discontinuous changes in center frequency are needed. This is not an unreasonable thing to wish for, being analogous to changing the frequency of an oscillator discontinuously.

There turns out to be a good way to accomodate this. The trick to updating k and q is to note that $c(\phi) = 1$ whenever ϕ is a multiple of 2π, regardless of the choice of p, q, and k as long as $p + q = 1$. Hence, we may make discontinuous changes in p, q, and n once per period (right when the phase is a multiple of 2π), without making discontinuities in the carrier signal.

In the specific case of FM, if we wish we can now go back and modify the original formulation:

$$p\cos(n\omega_2 t + r\cos(\omega_1 t)) +$$
$$+q\cos((n+1)\omega_2 t + r\cos(\omega_1 t)).$$

This is how to add glissandi (which are heard as dipthongs) to Chowning's original phase-modulation-based vocal synthesis technique.

6.4 Phase-aligned formant (PAF) generator

Combining the two-cosine carrier signal with the waveshaping pulse generator gives the *phase-aligned formant* generator, usually called by its acronym, PAF. The combined formula is,

$$x[n] = \underbrace{g\left(a\sin(\omega n/2)\right)}_{\text{modulator}} \underbrace{\left[p\cos(k\omega n) + q\cos((k+1)\omega n)\right]}_{\text{carrier}}$$

Here the function g may be either the Gaussian or Cauchy waveshaping functions, ω is the fundamental frequency, a is a modulation index controlling bandwidth, and k, p, and q control the formant center frequency.

Figure 6.8 shows the PAF as a block diagram. The block diagram is separated into a phase generation step, a carrier, and a modulator. The phase generation step outputs a sawtooth signal at the fundamental frequency.

The modulator is done by standard waveshaping, with a slight twist added. The formula for the modulator signals in the PAF call for an incoming sinusoid at half the fundamental frequency, i.e.,

$$\sin(\frac{\omega}{2})$$

and this nominally would require us to use a phasor tuned to half the fundamental frequency. However, since the waveshaping function is even, we may substitute the absolute value of the sinusoid:

$$\left|\sin(\frac{\omega}{2})\right|$$

which repeats at the frequency ω (the first half cycle is the same as the second one.) We can compute this simply by using a half-cycle sinusoid as a wavetable lookup function (with phase running from $-\pi/2$ to $\pi/2$), and it is this rectified sinusoid that we pass to the waveshaping function.

Although we picture the wavetable function over both negative and positive values (reaching from -10 to 10), in actuality we're only using the positive side for

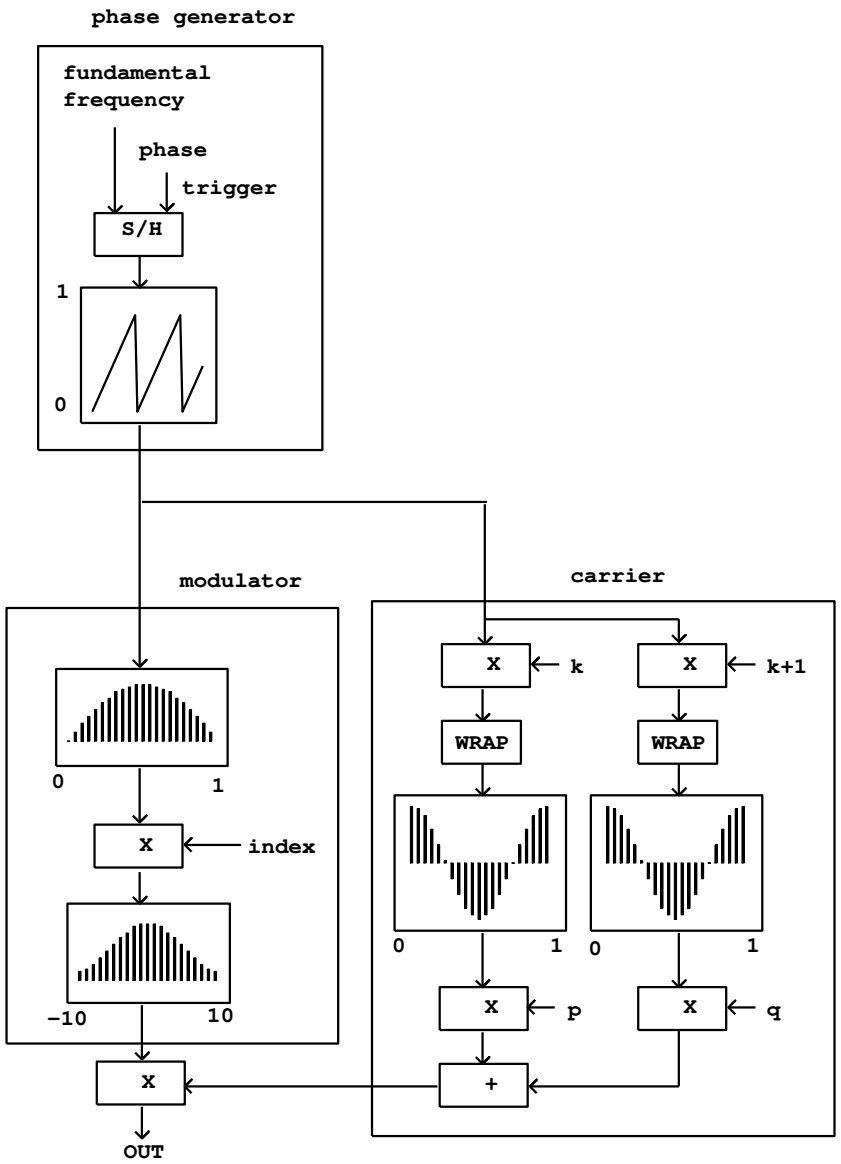


Figure 6.8: The PAF generator as a block diagram.

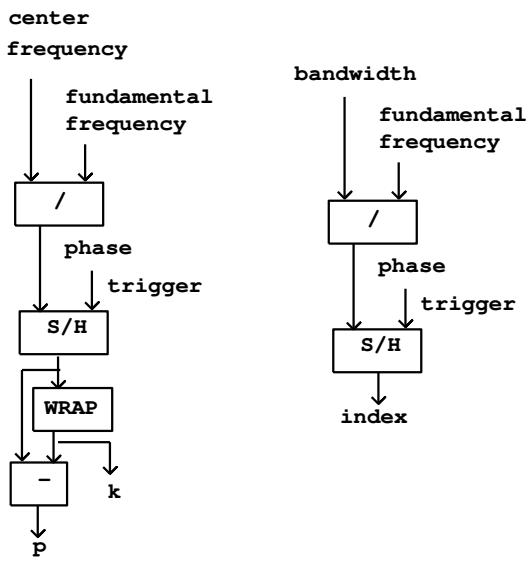


Figure 6.9: Calculation of the time-varying parameters a (the waveshaping index), k, and q for use in the block diagram of Figure 6.8.

lookup, ranging from 0 to b, the index of modulation. If the index of modulation exceeds the input range of the table (here set to stop at 10 as an example), the table lookup address should be clipped. The table should extend far enough into the tail of the waveshaping function so that the effect of clipping is inaudible.

The carrier signal is a weighted sum of two cosines, whose frequencies are increased by multiplication (by k and $k+1$, respectively) and wrapping. In this way all the lookup phases are controlled by the same sawtooth oscillator.

The quantities k, q, and the wavetable index b (labeled "index" in the figure), are calculated as shown in Figure 6.9. They are functions of the specified fundamental frequency, the formant center frequency, and the bandwidth, which are the original parameters of the algorithm. The quantity p, not shown in the figure, is just $1-q$.

As described in the previous section, the quantities k, p, and q may only change at phase wraparound points, that is to say, at periods of $2\pi/\omega$. Since the calculation of k, etc., depends on the value of the parameter ω, it follows that ω itself may only be updated when the phase is a multiple of 2π; otherwise, a change in ω could send the center frequency $(k+q)\omega$ to an incorrect value for a (very audible) fraction of a period. In effect, all the parameter calculations should be synchronized to the phase of the original oscillator.

Having the oscillator's phase control the updating of its own frequency is the first example we have seen of *feedback*, which in general means using any

process's output as one of its inputs. When processing digital audio signals at a fixed sample rate (as we're doing), it is never possible to have the process's *current* output as an input, since at the time we would need it we haven't yet calculated it. The best we can hope for is to use the previous sample of output—in effect, adding one sample of delay. In block environments (such as Max, Pd, and Csound) the situation becomes more complicated, but we will delay discussing that until the next chapter (and avoid dealing with the problem in the examples at the end of this one).

The amplitude of the central peak in the spectrum of the PAF generator is roughly $1/(1+b)$; in other words, close to unity when the index b is smaller than one, and falling off inversely with larger values of b. For values of b less than about ten, the loudness of the output does not vary greatly, since the introduction of other partials, even at lower amplitudes, offsets the decrease of the center partial's amplitude. However, if using the PAF to generate formants with specified peak amplitudes, the output should be multiplied by $1+b$ (or even, if necessary, a better approximation of the correction factor, whose exact value depends on the waveshaping function). This amplitude correction should be ramped, not sampled-and-held.

Since the expansion of the waveshaping (modulator) signal consists of all cosine terms (i.e., since they all have initial phase zero), as do the two components of the carrier, it follows from the cosine product formula that the components of the result are all cosines as well. This means that any number of PAF generators, if they are made to share the same oscillator for phase generation, will all be in phase and combining them gives the sum of the individual spectra. So we can make a multiple-formant version as shown in figure 6.10.

Figure 6.11 shows a possible output of a pair of formants generated this way; the first formant is centered halfway between partials 3 and 4, and the second at partial 12, with lower amplitude and bandwidth. The Cauchy waveshaping function was used, which generates linearly sloped spectra (viewed in dB). The two superpose additively, so that the spectral envelope curves smoothly from one formant to the other. The lower formant also adds to its own reflection about the vertical axis, so that it appars slightly curved upward there.

The PAF generator can be altered if desired to make inharmonic spectra by sliding the partials upward or downward in frequency. To do this, add a second oscillator to the phase of both carrier cosines, but not to the phase of the modulation portion of the diagram, nor to the controlling phase of the sample-and-hold units. It turns out that the sample-and-hold strategy for smooth parameter updates still works; and furthermore, mutiple PAF generators sharing the same phase generation portion will still be in phase with each other.

Our technique for superposing spectra does not work as predictably for phase modulation as it does for the PAF generator; the partials of the phase modulation output have complicated phase relationships and they seem difficult to combine coherently. In general, phase modulation will give more complicated patterns of spectral evolution, whereas PAF is easier to predict and turn to specific desired effects.

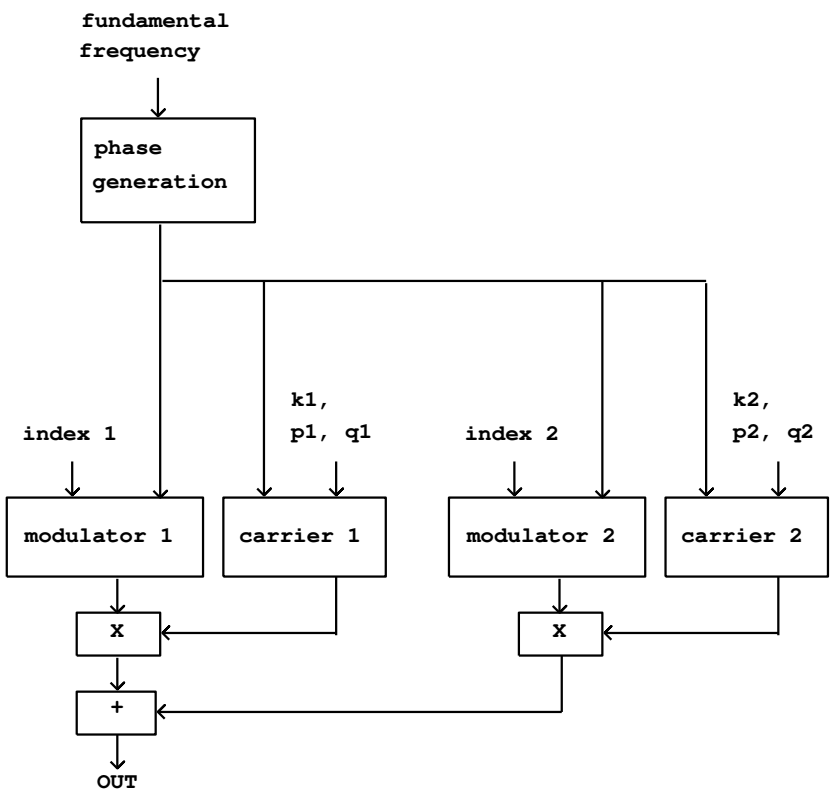


Figure 6.10: Block diagram for making a spectrum with two formants using the PAF generator.

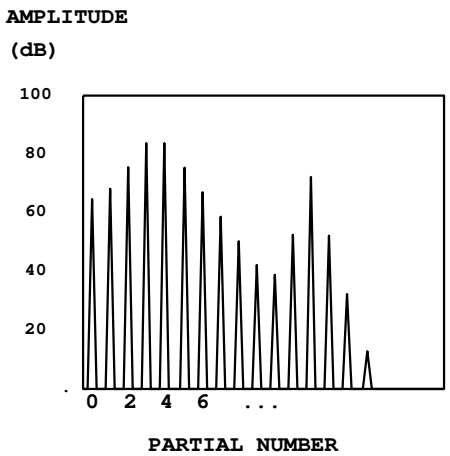


Figure 6.11: Spectrum from a two-formant PAF generator.

6.5 Examples

6.5.1 Wavetable pulse train

Patch F01.pulse.pd (Figure 6.12) generates a variable-width pulse train using stretched wavetable lookup. Figure 6.13 shows two intermediate products of the patch and its output. The patch carries out the job in the simplest possible way, which places the pulse at phase π instead of phase zero; this will be fixed in later patch examples by adding 0.5 to the phase and wrapping.

The initial phase is adjusted to run from -0.5 to 0.5 and then scaled by a multiplier of at least one, resulting in the signal of Figure 6.13 part (a); this corresponds to the output of the `*~`object, five from the bottom in the patch shown. The graph in part (b) shows the result of clipping the sawtooth wave back to the interval between -0.5 and 0.5, using the `clip~` object. If the scaling multiplier were at its minimum (one), the sawtooth would only range from -0.5 to 0.5 anyway and the clipping would have no effect. For any value of the scaling multiplier greater than one, the clipping output sits at the value -0.5, then ramps to 0.5, then sits at 0.5. The higher the multiplier, the faster the waveform ramps and the more time it spends clipped at the bottom and top.

The `cos~` object then converts this waveform into a pulse. Inputs of both -0.5 and 0.5 go to -1 (they are one cycle apart); at the midpoint of the waveform, the input is 0 and the output is thus 1. The output therefore sits at -1, traces a full cycle of the cosine function, then comes back to rest at -1. The proportion of time the waveform spends tracing the cosine function is one divided by the multiplier; so it's 100% for a multiplier of 1, 50% for 2, and so on. Finally, the pulse output is adjusted to range from 0 to 1 in value; this is graphed in part (c) of the figure.

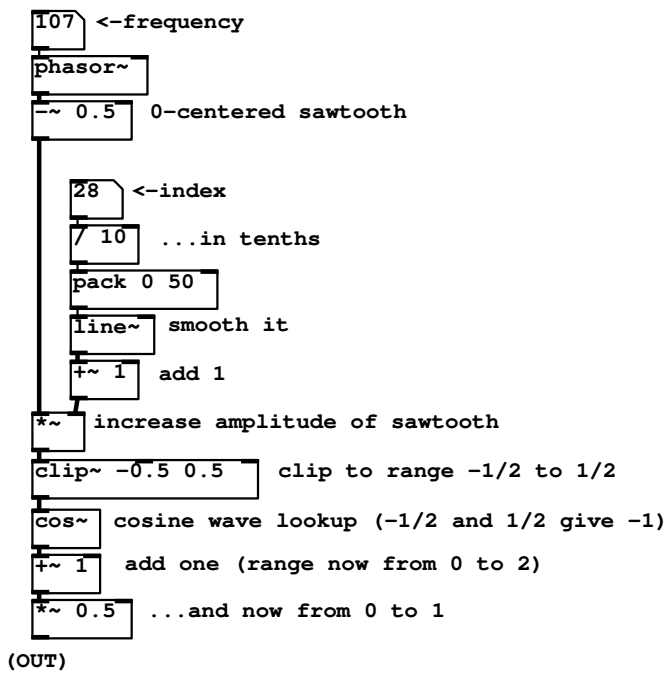


Figure 6.12: Example patch F01.pulse.pd, which synthesizes a pulse train using stretched wavetable lookup.

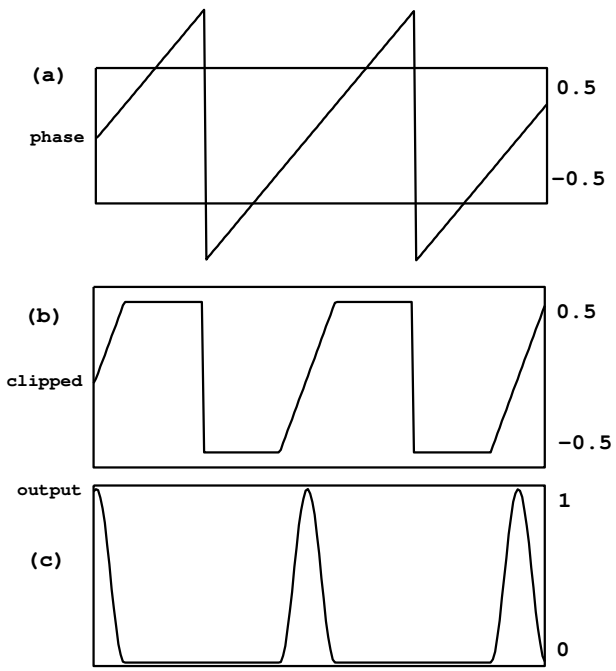


Figure 6.13: Intermediate audio signals from Figure 6.12: (a) the result of multiplying the phasor by the "index"; (b) the same, clipped to lie between -0.5 and 0.5; (c) the output.

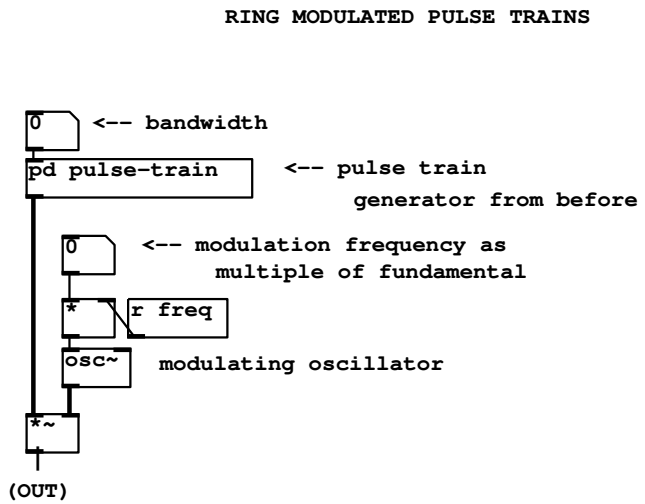


Figure 6.14: Excerpts from patch example F05.ring.modulation.pd combining ring modulation with a stretched wavetable pulse generator

6.5.2 Simple formant generator

The next three patch examples demonstrate the sound of the varying pulse width, graph its spectrum, and contrast the waveshaping pulse generator. Skipping to patch example F05.ring.modulation.pd (Figure 6.14), we show the simplest way of combining the pulse generator with a ring modulating oscillator to make a formant. The pulse train from the previous example is contained in the `pd pulse-train` subpatch. It is multiplied by an oscillator whose frequency is controlled as a multiple of the fundamental frequency. If the multiple is an integer, a harmonic sound results. No attempt is made to control the relative phases of the components of the pulse train and of the carrier sinusoid.

The next patch example, F06.packets.pd(Figure 6.15), shows how to combine the stretched wavetable pulse train with a sampled sinusoid to realize movable formants, as described in section 6.3. The pulse generator is as before, but now the carrier signal is a broken sinusoid. Since its phase is the fundamental phase times the center frequency quotient, the sample-to-sample phase increment is the same as for a sinusoid at the center frequency. However, when the phase wraps around, the carrier phase jumps to a different place in the cycle, as was illustrated in Figure 6.7. Although the relative bandwidth must be at least one, the relative center frequency (the quotient between center frequency and fundamental) may be as low as zero if desired.

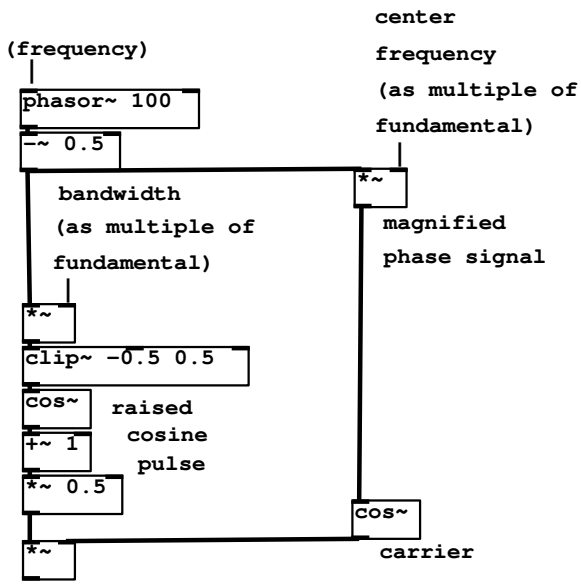


Figure 6.15: Using stretched wavetable synthesis to make a formant with movable center frequency.

6.5.3 Two-cosine carrier signal

Patch F08.two.cosines.pd (figure 6.16) shows how to make a carrier signal that cross-fades between harmonics to make continuously variable center frequencies. The center frequency quotient (the center frequency divided by the fundamental) appears as the output of a `line~` object. This is separated into its fractional part (using the `wrap~` object) and its integer part (by subtracting the fractional part from the original). These are labelled as q and k to agree with the treatment in section 6.3.

The phase—a sawtooth wave at the fundamental frequency—is multiplied by both k and $k+1$ (the latter by adding the original sawtooth into the former), and the cosines of both are taken; they are therefore at k and $k+1$ times the fundamental frequency and have no discontinuities at phase wrapping points. The next several objects in the patch compute the weighted sum $pc_1 + qc_2$, where c_1, c_2 are the two sinusoids and $p = 1 - q$, by evaluating an equivalent expression, $c_1 + q(c_2 - c_1)$. This gives us the desired movable-frequency carrier signal.

Patch F09.declickit.pd (not shown here) shows how, by adding a `samphold~` object after the `line~` object controlling center frequency, you can avoid discontinuities in the output signal even if the desired center frequency changes discontinuously. In the particular example shown, the frequency quotient (i.e., carrier divided by fundamental) alternates between 4 and 13.5. At ramp times

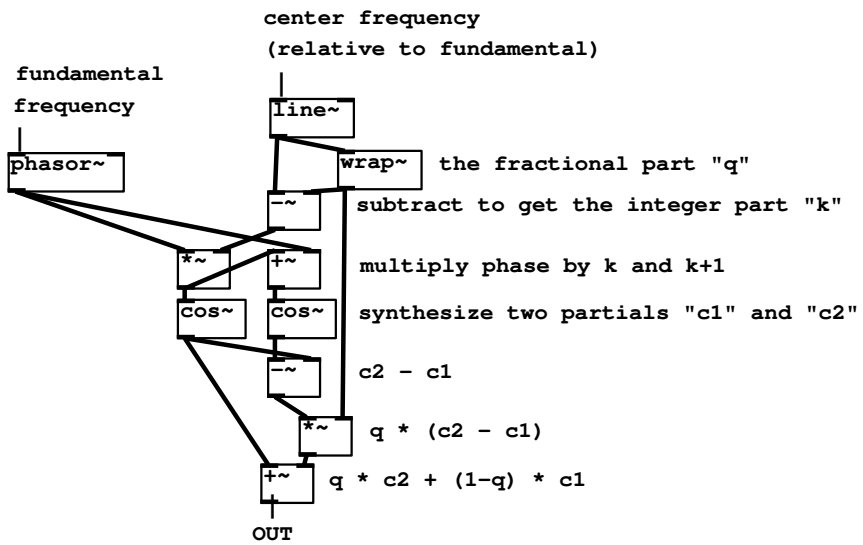


Figure 6.16: Cross-fading between sinusoids to make movable center frequencies.

below about 20 msec there are audible artifacts when using the `line~` object alone which disappear when the `samphold~` object is added. (A disadvantage of sample-and-holding the frequency quotient is that, for very low fundamental frequencies, the changes can be heard as discrete steps. So in situations where the fundamental frequency is low and the center frequency need not change very quickly, it may be better to omit the sample-and-hold step.)

The next two patch examples demonstrate using the crossfading-oscillators carrier as part of the classic two-modulator phase modulation technique. The same modulating oscillator is added separately to the phases of the two cosines. The resulting spectra can be made to travel up and down in frequency, but because of the complicated phase relationships between neighboring peaks in the phase modulation spectrum, no matter how you align two such spectra you can never avoid getting phase cancellations where they overlap.

6.5.4 The PAF generator

Patch F12.paf.pd (Figure 6.17) is a realization of the PAF generator, described in section 6.4. The control inputs specify the fundamental frequency, the center frequency, and the bandwidth, all in "MIDI" units. The first steps taken in the realization are to divide center frequency by fundamental (to get the center frequency quotient) and the bandwidth by fundamental to get the index of modulation for the waveshaper. The center frequency quotient is sampled-and-held so that it is only updated at periods of the fundamental.

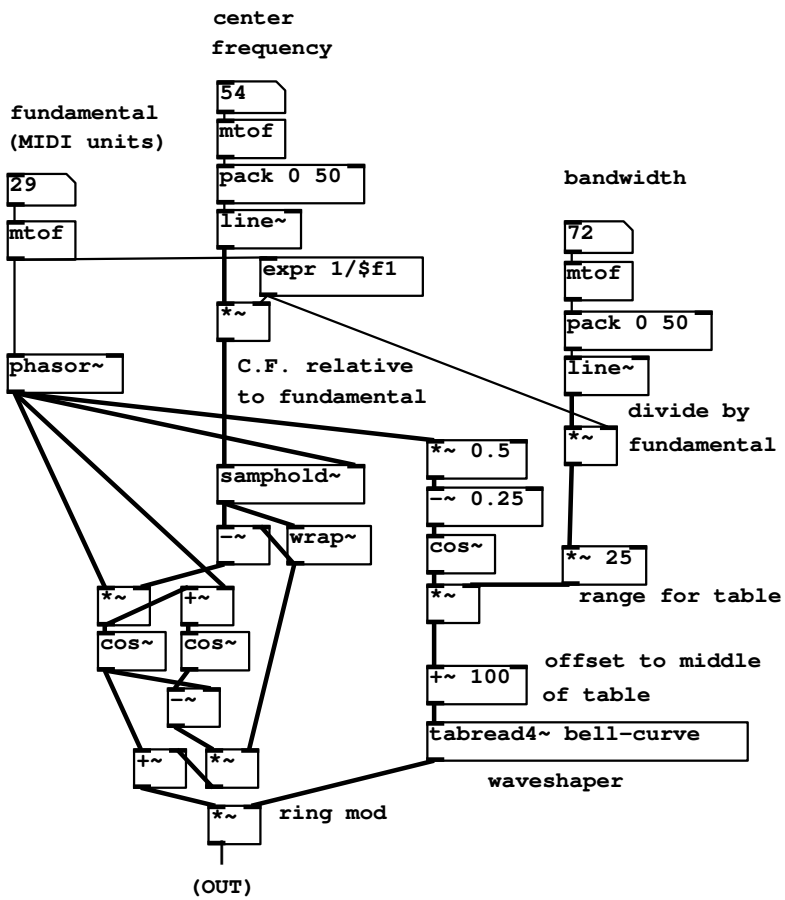


Figure 6.17: The phase-aligned formant (PAF) synthesis algorithm.

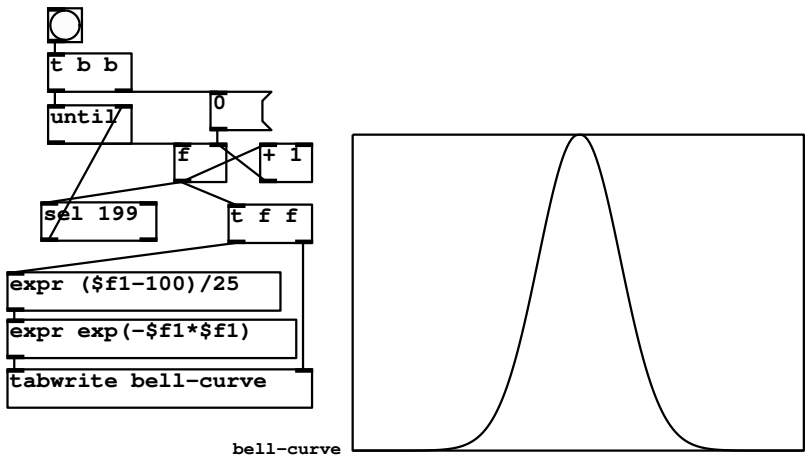


Figure 6.18: Filling in the wavetable for Figure 6.17.

The one oscillator (the `phasor~` object) runs at the fundamental frequency. This is used both to control a `samphold~` object which synchronizes updates to the center frequency quotient (labeled "C.F. relative to fundamental" in the figure), and to compute phases for both `cos~` objects which operate as shown earlier in Figure 6.16.

The waveshaping portion of the patch uses a half period of a sinusoid as a lookup function (to compensate for the frequency doubling because of the symmetry of the lookup function). To get a half-cycle of the sine function we multiply the phase by 0.5 and subtract 0.25, so that the adjusted phase runs from -0.25 to +0.25, once each period. This scans the positive half of the cycle defined by the `cos~` object.

The amplitude of the half-sinusoid is then adjusted by an index of modulation (which is just the quotient of bandwidth to fundamental frequency). The table ("bell-curve") has a Gaussian (unnormalized as before) sampled from -4 to 4 over 200 points (25 points per unit), so the center of the table, at point 100, corresponds to the central peak of the bell curve. Outside the interval from -4 to 4 the Gaussian is negligibly small.

Figure 6.18 shows how the Gaussian wavetable is prepared. One new control object is needed:

[until] : When the left, "start" inlet is banged, output sequential bangs (with no elapsed time between them) iteratively, until the right, "stop" inlet is banged. The stopping `bang` message must originate somehow from the `until` object's outlet; otherwise, the outlet will send `bang` messages forever, freezing out any other object which could break the loop.

As used here, a loop driven by an `until` object counts from 0 to 199, inclusive. The loop count is maintained by the "`f`" and "`+ 1`" objects, each of which

feeds the other. But since the "`+ 1`" object's output goes to the right inlet of the "`f`", its result (one greater) will only emerge from the "`f`" the next time it is banged by "`until`". So each bang from "`until`" increments the value by one.

The way the loop is started up matters: the upper "`t b b`" object (short for "trigger bang bang") must first send the number zero to the "`f`", thus initializing it, and then set the `until` object sending bangs, incrementing the value, until stopped. To stop it when the value reaches 199, a `select` object matches that value and, when it sees the match, bangs the "stop" inlet of the `until` object.

Meanwhile, for every number from 0 to 199 which comes out of the "`f`" object, we create an ordered pair of messages to the `tabwrite` object. First, at right, the index itself, from 0 to 199. Then for the left inlet, the first `expr` object adjusts the index to range from -4 to 4 (which previously ranged from 0 to 199) and the second one evaluates the Gaussian function.

In this patch we have not fully addressed the issue of updating the carrier/fundamental quotient at the appropriate times. Whenever the carrier frequency is changed the sample-and-hold step properly delays the update of the quotient. But if, instead or in addition, the fundamental itself changes abruptly, then for a fraction of a period the `phasor~` object's frequency and the quotient are out of sync. Pd does not allow the `samphold~` output to be connected back into the `phasor~` input without the inclusion of an explicit delay (see the next chapter) and there is no simple way to modify the patch to solve this problem.

Assuming that we *did* somehow clock the `phasor~` object's input synchronously with its own wraparound points, we would then have to do the same for the bandwidth/fundamental quotient on the right side of the patch as well. In this scenario there is no problem updating that value continuously.

A practical solution to this updating problem has been simply to rewrite the entire patch as its own Pd object; this also turns out to use much less CPU time than the pictured patch, and is the more practical solution overall—as long as you don't want to experiment with making embellishments or other changes to the algorithm.

Such embellishments might include: adding an inharmonic upward or downward shift in the partials; allowing to switch between smooth and sampled-and-held center frequency updates; adding separate gain controls for even and odd partials; introducing gravel by irregularly modulating the phase; allowing mixtures of two or more waveshaping functions; making more percussive attacks by aligning the phase of the oscillator with the timing of an amplitude envelope generator.

One final detail about amplitude is in order: since the amplitude of the strongest partial decreases roughly as $1/(1+b)$ where b is the index of modulation, it is sometimes (but not always) desirable to correct the amplitude of the output by multiplying by $1+b$. This is only an option if b is smoothly updated (as in this patch example), not if it is sampled-and-held. One situation in which this is appropriate is in simulating plucked strings (by setting center frequency to the fundamental, starting with a high index of modulation and dropping it exponentially); it would be appropriate to hear the fundamental dropping, not rising, in amplitude as the string decays.

6.5.5 Stretched wavetables

Instead of using waveshaping, fomant synthesis is also possible using stretched wavetables, as shown in Patch F14.wave.packet.pd (not shown here). THe technique is essentially that of Patch B10.sampler.overlap.pd (described in section 2.6.5), with a cosine lookup instead of the more general wavetable, but with the addition of a control to set the duty cycle of the amplitude envelopes. The units are adjusted to be compatible with those of the previous example.

Chapter 7

Time shifts

At 5:00 some afternoon, put on your favorite recording of the Ramones' string quarter number 5. The next Saturday, play the same recording at 5:00:01, one second later in the day. The two playings should sound the same, with any differences more likely to be of context or environment than to be intrinsic to the sound. Shifting the whole thing one second (or, if you like, a few days and a second) has no physical effect on the sound.

But now suppose you played it at 5:00 and 5:00:01 on the same day (on two different playback systems, since the music lasts much longer than one second.) Now the sound is much different. Moreover, the difference, whatever it is, clearly resides in neither of the two individual sounds, but rather in the *interference* between the two. This interference can be perceived in at least four different ways:

Canons: combining two copies of a signal with a time shift sufficient for the signal to change appreciably, we might hear the two as separate musical streams, in effect comparing the signal to its earlier self. If the signal is a melody, the time shift might be comparable to the length of one or several notes.

Echos: At time shifts between about 30 milliseconds and about a second, the later copy of the signal can sound like an echo of the first one. An echo may reduce the intelligibility of the signal (especially if it consists of speech), but usually won't change the overall "shape" of melodies or phrases.

Filtering: At time shifts below about 30 milliseconds, the copies are too close together in time to be perceived separately, and the dominant effect is that some frequencies are enhanced and others suppressed. This changes the spectral envelope of the sound.

Altered room quality: If the second copy is altered to be much quieter than the first, and moreover if we add a third, or even a thousand more delayed copies at reduced amplitudes, they can mimic the echos that arise in a room or other acoustic space.

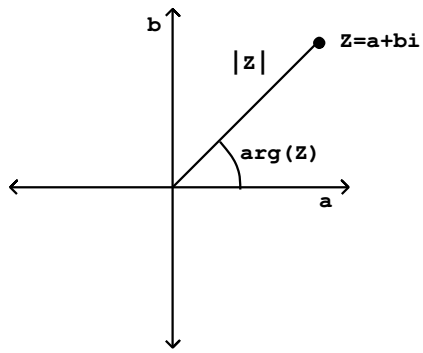


Figure 7.1: A number, Z, in the complex plane. The axes are for the real part a and the imaginary part b.

The overall quality of a given arrangement of delayed copies of a signal may combine two or more of these affects.

Mathematically, the effect of a time shoft on a signal can be described as a phase change of each of the signal's sinusoidal components. The phase shift of each component is different depending on its frequency (as well as on the amount of time shift). In the rest of this chapter we will often consider superpositions of sinusoids at different phases. Heretofore we have been content to use real-valued sinusoids in our analysis, but in this and later chapters the formulas will become more complicated and we will need more powerful mathematical tools to manage them. So in a preliminary section of this chapter we will develop the additional background needed.

7.1 Complex numbers

Complex numbers are written as:

$$Z = a + bi$$

where a and b are real numbers and $i = \sqrt{-1}$. (In this book we'll use capital letters to denote complex numbers and lowercase for real numbers.) Since a complex number has two real components, we use a Cartesian plane (in place of a number line) to graph it, as shown in Figure 7.1. The quantities a and b are called the *real* and *imaginary parts* of Z, written as:

$$a = \mathrm{re}(Z),$$

$$b = \mathrm{im}(Z)$$

If Z is a complex number, its *magnitude*, written as $|Z|$, is just the distance in the plane from the origin to the point (a, b):

$$|Z| = \sqrt{(a^2 + b^2)}$$

and its *argument*, written as $\angle(Z)$, is the angle from the positive a axis to the point (a, b):

$$\angle(Z) = \arctan\left(\frac{b}{a}\right)$$

If we know the magnitude and argument of a complex number (say they are r and θ, for instance) we can reconstruct the real and imaginary parts:

$$a = r\cos(\theta)$$

$$b = r\sin(\theta)$$

A complex number may be written in terms of its real and imaginary parts a and b (this is called *rectangular form*), or alternatively in *polar form*, in terms of r and θ:

$$Z = r \cdot [\cos(\theta) + i\sin(\theta)]$$

The rectangular and polar formulations are equivalent, and the equations above show how to compute a and b from r and θ and vice versa.

The main reason we use complex numbers in electronic music is because they magically encode sums of angles. We frequently have to add angles together in order to talk about the changing phase of an audio signal as time progresses (or as it is shifted in time, as in this chapter). It turns out that, if you multiply two complex numbers, the argument of the product is the sum of the arguments of the two factors. To see how this happens, we'll multiply two numbers Z_1 and Z_2, written in polar form:

$$Z_1 = r_1 \cdot [\cos(\theta_1) + i\sin(\theta_1)]$$

$$Z_2 = r_2 \cdot [\cos(\theta_2) + i\sin(\theta_2)]$$

giving:

$$\begin{aligned} Z_1 Z_2 = r_1 r_2 \cdot [&\cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2) \\ &+ i\,(\sin(\theta_1)\cos(\theta_2) + \cos(\theta_1)\sin(\theta_2))] \end{aligned}$$

Here the minus sign in front of the $\sin(\theta_1)\sin(\theta_2)$ term comes from multiplying i by itself, which gives -1. We can spot the cosine and sine summation formulas in the above expression, and so it simplifies to:

$$Z_1 Z_2 = r_1 r_2 \cdot [\cos(\theta_1 + \theta_2) + i\sin(\theta_1 + \theta_2)]$$

And so, by inspection, it follows that the product $Z_1 Z_2$ has magnitude $r_1 r_2$ and argument $\theta_1 + \theta_2$.

We can use this property of complex numbers to add and subtract angles (by multiplying and dividing complex numbers with the appropriate arguments) and then to take the cosine and sine of the result by extracting the real and imaginary parts of the result.

7.1.1 Sinusoids as geometric series

Recall the formula for a (real-valued) sinusoid from page 3:

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

This is a sequence of cosines of angles (called phases) which increase arithmetically with the sample number n. The cosines are all adjusted by the factor a. We can now re-write this as the real part of a much simpler and easier to manipulate sequence of complex numbers, by using the properties of their arguments and magnitudes.

Suppose that our complex number Z happens to have magnitude one, so that it can be written as:

$$Z = \cos(\omega) + i\sin(\omega)$$

Then for any integer n, the number Z^n must have magnitude one as well (because magnitudes multiply) and argument $n\theta$ (because arguments add). So,

$$Z^n = \cos(n\omega) + i\sin(n\omega)$$

This is also true for negative values of n, so for example,

$$\frac{1}{Z} = Z^{-1} = cos(\omega) - i\sin(\omega)$$

Figure 7.2 shows graphically how the powers of Z wrap around the unit circle, which is the set of all complex numbers of magnitude one. They form a geometric sequence:

$$\ldots, Z^0, Z^1, Z^2, \ldots$$

and taking the real part of each term we get a real sinusoid with initial phase zero and amplitude one:

$$\ldots, \cos(0), \cos(\omega), \cos(2\omega), \ldots$$

The sequence of complex numbers is much easier to manipulate algebraically than the sequence of cosines.

Furthermore, suppose we multiply the elements of the sequence by some (complex) constant A with magnitude a and argument ϕ. This gives

$$\ldots, A, AZ, AZ^2, \ldots$$

The magnitudes are all a and the argument of the nth term is $\phi + n\omega$, so the sequence is equal to

$$a \cdot [\cos(\phi) + i\sin(\phi)],$$

$$a \cdot [\cos(\omega + \phi) + i\sin(\omega + \phi)],$$

$$\cdots, a \cdot [\cos(n\omega + \phi) + i\sin(n\omega + \phi)], \ldots$$

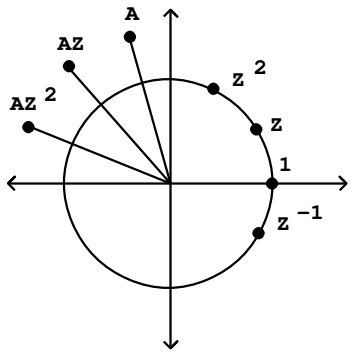


Figure 7.2: The powers of a complex number Z with $|Z| = 1$, and the same sequence multiplied by a constant A.

and so the real part is just the real-valued sinusoid:

$$\mathrm{re}(AZ^n) = a \cdot [\cos(n\omega + \phi)]$$

The complex amplitude A encodes both the amplitude (equal to its magnitude a) and the inital phase (its argument ϕ); the unit-magnitude complex number Z controls the frequency which is just its argument ω.

Figure 7.2 also shows the sequence $A, AZ, AZ^2, \ldots$; in effect this is the same sequence as $1, Z, Z^2, \ldots$, but amplified and rotated according to the amplitude and initial phase. In a complex sinusoid of this form, A is called the *complex amplitude*.

Using complex numbers to represent the amplitudes and phases of sinusoids can clarify manipulations that otherwise might seem unmotivated. For instance, in Section **??** we looked at the sum of two sinusoids with the same frequency. In the language of this chapter, we let the two sinusoids be written as:

$$X[n] = AZ^n, \; Y[n] = BZ^n$$

where A and B encode the phases and amplitudes of the two signals. The sum is then equal to:

$$X[n] + Y[n] = (A + B)Z^n$$

which is a sinusoid whose amplitude equals $|A + B|$ and whose phase equals $\angle(A + B)$. This is clearly a much easier way to manipulate amplitudes and phases than using series of sines and cosines. Eventually, of course, we will take the real part of the result; this can usually be left to the very last step of the calculation.

7.2 Time shifts and phase changes

Starting from any (real or complex) signal $X[n]$, we can make other signals by time shifting the signal X by a (positive or negative) integer d:

$$Y[n] = X[n-d]$$

so that the dth sample of Y is the 0th sample of X and so on. If the integer d is positive (or zero), then Y is a delayed copy of X. If d is negative, then Y anticipates X; this can be done to a recorded sound but isn't practical as a real-time operation.

Time shifting is a linear operation (considered as a function of the input signal X); if you time shift a sum $X_1 + X_2$ you get the same result as if you time shift them separately and add afterward.

Time shifting has the further property that, if you time shift a sinusoid of frequency ω, the result is another sinusoid of the same frequency—time shifting never introduces frequencies that weren't present in the signal before it was shifted. This property, called *time invariance*, makes it easy to analyze the effects of time shifts—and linear combinations of them—by considering separately what the operations do on individual sinusoids.

Furthermore, the effect of a time shift on a sinusoid is simple: it just changes the phase. If we use a complex sinusoid, the effect is even simpler. If for instance

$$X[n] = AZ^n$$

then

$$Y[n] = X[n-d] = AZ^{(n-d)} = Z^{-d}AZ^n = Z^{-d}X[n]$$

so time shifting a complex sinusoid by d samples is the same thing as scaling it by Z^{-d}—it's just an amplitude change by a particular complex number. Since $|Z| = 1$ for a sinusoid, the amplitude change does not change the magnitude of the sinusoid, only its phase.

The phase change is equal to $-d\omega$, where $\omega = \angle(Z)$ is the angular frequency of the sinusoid. This is exactly what we should expect since the sinusoid advances ω radians per sample and it is offset (i.e., delayed) by d samples.

7.3 Delay networks

If we consider our digital audio samples $X[n]$ to correspond to successive moments in time, then time shifting the signal by d samples corresponds to a *delay* of d/R time units, where R is the sample rate. (If d is negative, then we are saying that the output predicts the input; this isn't practical in systems, such as Pd, that schedule computations in order of time.)

Figure 7.3 shows one example of a *linear delay network*: an assembly of delay units, possibly with amplitude scaling operations, combined using addition and subtraction. The output is a linear function of the input, in the sense that adding two signals at the input is the same as processing each one separately

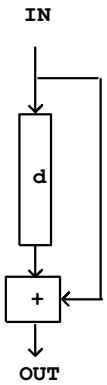


Figure 7.3: A delay network. Here we add the incoming signal to a delayed copy of itself.

and adding the results. Moreover, they are time invariant, i.e., they create no new frequencies in the output that weren't present in the input.

In general there are two ways of thinking about delay networks. We can think in the *time domain*, in which we draw waveforms as functions of time (or of the index n), and consider delays as time shifts. Alternatively we may think in the *frequency domain*, in which we dose the input with a sinusoid (so that its output is a sinusoid at the same frequency) and report the amplitude and/or phase change brought by the network, as a function of the frequency (encoded in the complex number Z). We'll now look at the delay network of Figure 7.3 in each of the two ways in turn.

Figure 7.4 shows the network's behavior in the time domain. We invent some sort of suitable test function as input (it's a rectangular pulse eight samples wide in this example) and graph the input and output as functions of the sample number n. This particular delay network adds the input to a delayed copy of itself.

A frequently used test function is an *impulse*, which is a pulse lasting only one sample. The utility of this is that, if we know the output of the network for an impulse, we can find the output for any other digital audio signal—because any signal $x[n]$ is a sum of impulses, one of height $x[0]$, the next one occurring one sample later and having height $x[1]$, and so on. Later, when the networks get more complicated, we will move to using impulses as input signals to show their time-domain behavior.

On the other hand, we can analyze the same network in the frequency domain by considering a (complex-valued) test signal,

$$X[n] = Z^n$$

where Z has unit magnitude and argument ω. We already know that the output

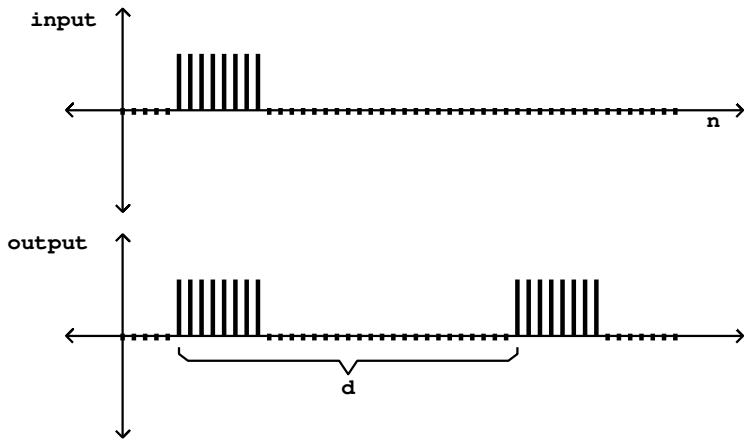


Figure 7.4: The time domain view ot the delay network of Figure 7.3. The output is the sum of the input and its time shifted copy.

is another complex sinusoid with the same frequency, that is,

$$HZ^N$$

for some complex number H (which we want to find.) So we write the output directly as the sum of the input and its delayed copy:

$$Z^n + Z^{-d}Z^n = (1 + Z^{-d})Z^n$$

and find by inspection that:

$$H = 1 + Z^{-d}$$

We can understand the frequency-domain behavior of this delay network by studying how the complex number H varies as a function of the angluar frequency ω. We are especially interested in its argument and magnitude—which tell us the relative phase and amplitude of the sinusoid that comes out. We will work this example out in detail to show how the arithmetic of complex numbers can predict what happens when sinusoids are combined additively.

Figure 7.5 is a graph, in the complex plane, showing how the quantities 1 and Z^{-d} combine additively. To add complex numbers we add their real and complex parts separately. So the complex number 1 (real part 1, imaginary part 0) is added coordinate-wise to the complex number Z^{-d} (real part $\cos(-d\omega)$, imaginary part $\sin(-d\omega)$). This is shown graphically by making a parallelogram, with corners at the origin and at the two points to be added, and whose fourth corner is the sum H.

As the figure shows, the result can be understood by symmetrizing it about the real axis: instead of 1 and Z^{-d}, it's easier to sum the quantities $Z^{d/2}$ and

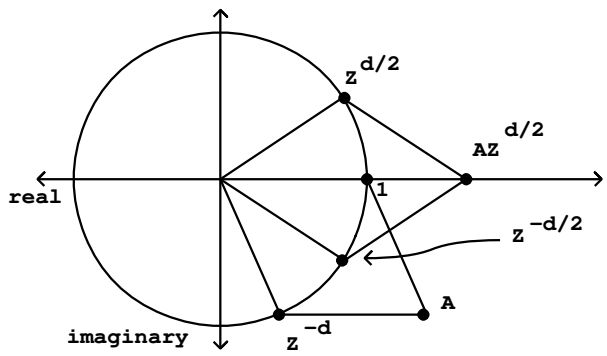


Figure 7.5: Analysis, in the complex plane, of the frequency-domain behavior of the delay network of Figure 7.3. The complex number Z encodes the frequency of the input. The delay line output is the input times $Z^{-}d$. The total (complex) gain is H. We find the magnitude and argument of H by symmetrizing the sum, rotating it by $d/2$ times the angular frequency of the input.

$Z^{-d/2}$, because they are symmetric about the real (horizontal) axis. (Strictly speaking, we haven't defined the quantities $Z^{d/2}$ and $Z^{-d/2}$; we use those expressions to denote unit complex numbers whose arguments are half those of Z^d and Z^{-d}.) We rewrite the gain as:

$$H = Z^{-d/2}(Z^{d/2} + Z^{-d/2})$$

The first term is a phase shift of $-d\omega/2$. The second term is best understood in rectangular form:

$$\begin{aligned} & Z^{d/2} + Z^{-d/2} \\ = & (\cos(\omega d/2) + i\sin(\omega d/2)) + (\cos(\omega d/2) - i\sin(\omega d/2)) \\ = & 2\cos(\omega d/2) \end{aligned}$$

This real-valued quantity may be either positive or negative; its absolute value gives the magnitude of the output:

$$|H| = 2|\cos(\omega d/2)|$$

The quantity $|H|$ is called the *gain* of the delay network at the angular frequency ω, and is graphed in Figure 7.6. The frequency-dependent gain of a delay network (that is, the gain as a function of frequency) is called the network's *frequency response*.

Since the network has greater gain at some frequencies than at others, it may be considered as a *filter*, that can be used to separate certain components of a sound from others. Because of the shape of this particular gain expression as

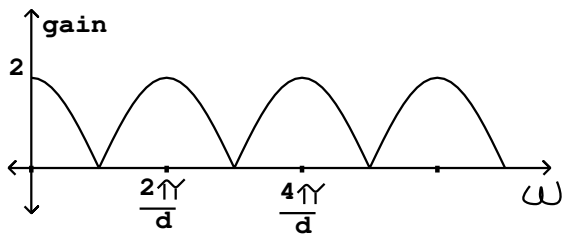


Figure 7.6: Gain of the delay network of Figure 7.3, shown as a function of angular frequency ω.

a function of ω, this kind of delay network is called a (non-recirculating) *comb filter*.

The output of the network is a sum of two sinusoids of equal amplitude, and whose phases differ by ωd. The resulting output amplitude can therefore be checked against the prediction of Section **??**—and they agree. The result also agrees with common sense: if the angular frequency ω is set so that an integer number of periods fit into d samples, i.e., if ω is a multiple of $2\pi/d$, the output of the delay is exactly the same as the original signal, and so the two combine to make an output with twice the original amplitude. If the delay is half the period, on the other hand (so that $\omega = \pi/d$) the delay output is out of phase and cancels the input exactly.

This particular delay network has an interesting application: if we have a periodic (or nearly periodic) incoming signal, whose fundamental frequency is ω radians per sample, we can tune the comb filter so that the peaks in the gain are aligned at even harmonics and the odd ones fall where the gain is zero. To do this we choose $d = \pi/\omega$, i.e., set the delay time to exactly one half period of the incoming signal. In this way we get a new signal whose harmonics are $2\omega, 4\omega, 6\omega, \ldots$, and so it now has a new fundamental frequency at twice the original one. Except for a factor of two, the amplitudes of the remaining harmonics still follow the spectral envelope of the original sound. So we have a tool now for raising the pitch of an incoming sound by an octave without changing its spectral envelope. This octave doubler is the reverse of the octave divider introduced back in Chapter 5.

The time domain and frequency domain pictures are complementary ways of looking at the same delay network. When the delays inside the network are smaller than the ear's ability to resolve events in time—less than about 20 milliseconds—the time domain picture becomes less relevant to our understanding of the delay network, and we turn mostly to the frequency-domain picture. On the other hand, when delays are greater than about 50 milliseconds, the peaks and valleys of plots showing gain versus frequency (such as that of Figure 7.6) become crowded so closely together that the frequency-domain view

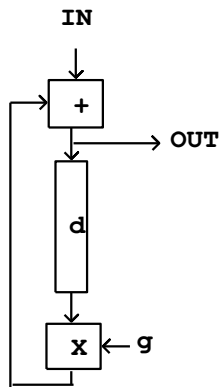


Figure 7.7: Block diagram for a recirculating comb filter. Here d is the delay time in samples and g is the feedback coefficient.

becomes less important. Both are nonetheless valid over the entire range of possible delay times.

7.4 Recirculating delay networks

It is sometimes desirable to connect the outputs of one or more delays in a network back into their own or each others' inputs. Instead of getting one or several echos of the original sound as in the simple example above, we can potentially get an infinite number of echos, each one feeding back into the network to engender yet others.

The simplest example of a recirculating network is the *recirculating comb filter* whose block diagram is shown in Figure 7.7. As with the earlier, simple comb filter, the input signal is sent down a delay line whose length is d samples. Unlike the simple comb filter, the delay line's output is also inserted back in its input; the delay's input is now the sum of the original input and the delay output. The output is multiplied by a number g before feeding it back into its input.

The time domain behavior of the recirculating comb filter is shown in Figure 7.8. Here we consider the effect of sending an impulse into the network. We get back the original impulse, plus a series of echos, each in turn d samples after the previous one, and multiplied each time by the gain g. In general, a delay network's output given an impulse as input is called the network's *impulse response*.

Note that we have chosen a gain g that is less than one in absolute value. If we chose a gain greater than one (or less than -1), each echo would have a larger magnitude than the previous one. Instead of falling exponentially as they

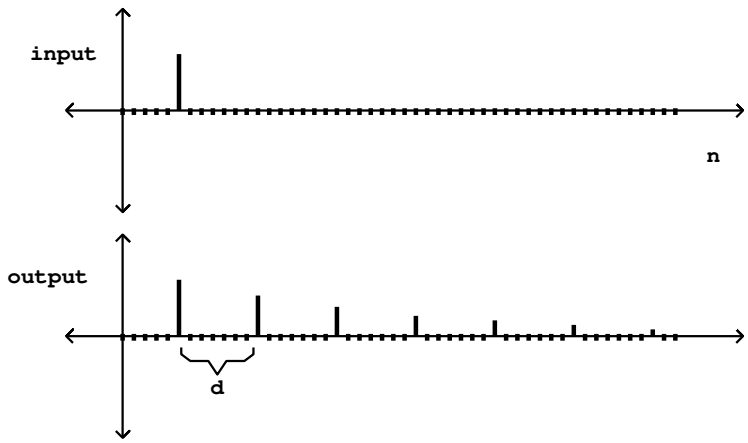


Figure 7.8: Time-domain analysis of the recirculating comb filter, using an impulse as input.

do in the figure, they would grow exponentially. A recirculating network whose output eventually falls toward zero after its input terminates is called *stable*; one whose output grows without bound is called *unstable*.

We can also analyse the recirculating comb filter in the frequency domain. The situation is now complicated enough that it is almost prohibitively hard to analyze using real sinusoids, and so we get the first real payoff for having introduced complex numbers, which greatly simplify the analysis.

If, as before, we feed the input with the signal,

$$X[n] = Z^n$$

with $|Z| = 1$, we can write the output as

$$Y[n] = (1 + gZ^{-d} + g^2 Z^{-2d} + \cdots)X[n]$$

Here the terms in the sum come from the series of discrete echos. It follows that the amplitude of the output is:

$$H = 1 + gZ^{-d} + \left(gZ^{-d}\right)^2 + \cdots$$

This is a geometric series; we can sum it using the standard technique: first multiply both sides by gZ^{-d} to give:

$$gZ^{-d}H = gZ^{-d} + \left(gZ^{-d}\right)^2 + \left(gZ^{-d}\right)^3 + \cdots$$

and subtract from the original equation to give:

$$H - gZ^{-d}H = 1,$$

and, solving for H,

$$H = \frac{1}{1 - gZ^{-d}}$$

A faster (but slightly less intuitive) method to get the same result is to examine the recirculating network itself to yield an equation for H, as follows. Since we named the input $X[n]$ and the output $Y[n]$, the signal going into the delay line is $X[n] + Y[n]$, and passing this through the delay line and multiplier gives

$$(X[n] + Y[n])gZ^{-d}$$

This is just the output signal again, so:

$$Y[n] = (X[n] + Y[n])gZ^{-d}$$

and dividing by $X[n]$ and using $H = Y[n]/X[n]$ gives:

$$H = (1 + H)gZ^{-d}$$

This is equivalent to the earlier equation for H.

Now we would like to make a graph of the frequency response (the gain as a function of frequency) as we did for non-recirculating comb filters in Figure 7.6. This again requires that we make a preliminary picture in the complex plane. We would like to estimate the magnitude of H equal to:

$$|H| = \frac{1}{|1 - gZ^{-d}|}$$

where we used the multiplicative property of magnitudes to conclude that the magnitude of a (complex) reciprocal is the reciprocal of a (real) magnitude. Figure 7.9 shows the situation graphically. The gain $|H|$ is the reciprocal of the length of the segment reaching from the point 1 to the point gZ^{-d}. Figure 7.10 shows a graph of the frequency response $|H|$ as a function of the angular frequency $\omega = \angle(Z)$.

Figure 7.9 can be used to analyze how the frequency response $|H(\omega)|$ should behave qualitatively as a function of g. The height and bandwidth of the peaks both depend on g. The maximum value that $|H|$ can attain is when

$$Z^{-d} = 1$$

This occurs at the frequencies $\omega = 2\pi/d, 4\pi/d, \ldots$ as in the simple comb filter above. At these maxima it reaches

$$|H| = \frac{1}{1 - g}$$

The next important question is the bandwidth of the peaks in the frequency response. So we would like to find a particular frequency, $\angle(W)$, giving rise to a value of $|H|$ that is, say, 3 decibels below the maximum. To do this, we return

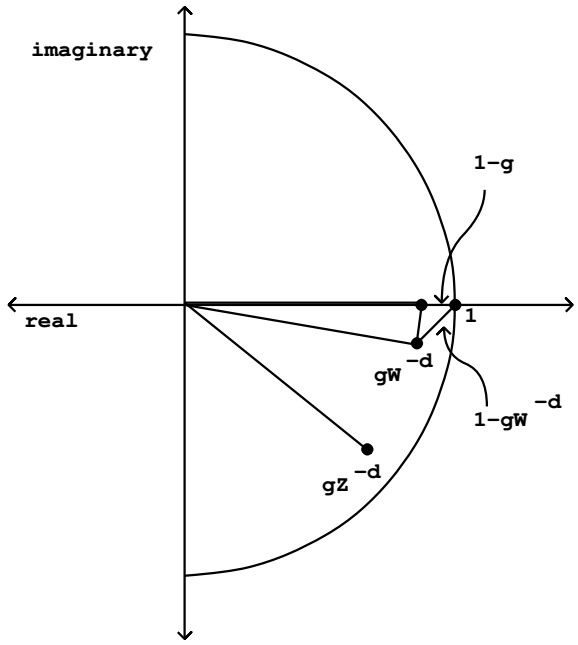


Figure 7.9: Diagram in the complex plane for approximating the output gain $|H|$ of the recirculating comb filters at three different frequencies: 0, and the arguments of two unit complex numbers W and Z; W is chosen to give a gain about 3 dB below the peak.

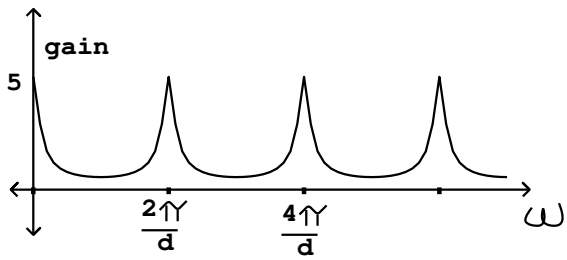


Figure 7.10: Frequency response of the recirculating comb filter with $g = 0.8$. The peak gain is $1/(1-g) = 5$. Peaks are much narrower than for the simple comb filter.

to Figure 7.9, and try to find W so that the distance from the point 1 to the point

$$1 - W^{-d}$$

is about $\sqrt{2}$ times the distance from 1 to g (since $\sqrt{2}$:1 is a ratio of approximately 3 decibels).

We do this by arranging for the imaginary part of W^{-d} to be roughly $1 - g$ or its negative, making a nearly isosocles right triangle between the points 1, $1-g$, and W^{-d}. (Here we're supposing that g is at least 2/3 or so; otherwise this approximation isn't very good). The hypoteneuse of a right isococles triangle is always $\sqrt{2}$ times the leg, and so the gain drops by that factor compared to its maximum.

We now make another approximation, that the imaginary part of W^{-d} is approximately the angle in radians it cuts from the real axis:

$$\pm(1 - g) \approx \mathrm{im}(W^{-d}) \approx \angle(W^{-d})$$

So the region of each peak reaching within 3 decibels of the maximum value is about

$$(1 - g)/d$$

(in radians) to either side of the peak. The bandwidth narrows (and the filter peaks become sharper) as g approaches its maximum value of 1. .

As with the non-recirculating comb filter of section 7.3, the teeth of the comb are closer together for larger valuse of the delay d. On the other hand, a delay of $d = 1$ (the shortest possible) gets only one tooth (at zero frequency) below the Nyquist frequency π (the next tooth, at 2π, corresponds again to a frequency of zero by foldover).

So the recirculating comb filter with $d = 1$ is just a low-pass filter. Delay networks with one-sample delays will be the basis for designing many other kinds of digital filters in chapter 8.

7.5 Power conservation and complex delay networks

The same techniques will work to analyze any delay network, although for more complicated networks it becomes harder to characterize the results, or to design the network to have specific, desired properties. Another point of view can sometimes be usefully brought to the situation, particularly when flat frequency responses are needed, either in their own right or else to ensure that a complex, recirculating network remains stable at feedback gains close to one.

The central fact we will use is that if any delay network, with either one or many inputs and outputs, is constructed so that its output power (averaged over time) always equals its input power, that network has to have a flat frequency response. This is almost a tautology; if you put in a sinusoid at any frequency on one of the inputs, you will get sinusoids of the same frequency at the outputs,

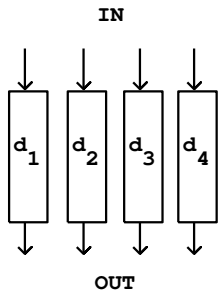


Figure 7.11: First fundamental building block for unitary delay networks: delay lines in parallel.

and the sum of the power on all the outputs will equal the power of the input, so the gain, suitably defined, is exactly one.

In order to work with power-conserving delay networks we will need an explicit definition of "total average power". If there is only one signal (call it $x[n]$), the average power is given by:

$$P(x[n]) = \left[|x[0]|^2 + |x[1]|^2 + \cdots + |x[N-1]|^2\right]/N$$

where N is a large enough number so that any fluctuations in amplitude get averaged out. This definition works as well for complex-valued signals as for real-valued ones. The average total power for several digital audio signals is just the sum of the individual signal's powers:

$$P(x_1[n] + \cdots + x_r[n]) = P(x_1[n]) + \cdots + P(x_r[n])$$

where r is the number of signals to be combined. With this definition, since the individual signals' power is perserved by delaying them, so is the power of the sum.

It turns out that a wide range of interesting delay networks has the property that the total power output equals the total power input; they are called *unitary*. First, we can put any number of delays in parallel, as shown in Figure 7.11. Whatever the total power of the inputs, the total power of the outputs has to be the same.

A second family of power-preserving transformations is composed of rotations and reflections of the signals $x_1[n]$, ... , $x_r[n]$, considering them, at each fixed time point n, as r numbers, or as a point in an r-dimensional space. The rotation or reflection must be one that leaves the origin $(0, \ldots, 0)$ fixed.

For each sample number n, the total contribution to the average signal power is proportional to

$$|x_1|^2 + \cdots + |x_r|^2$$

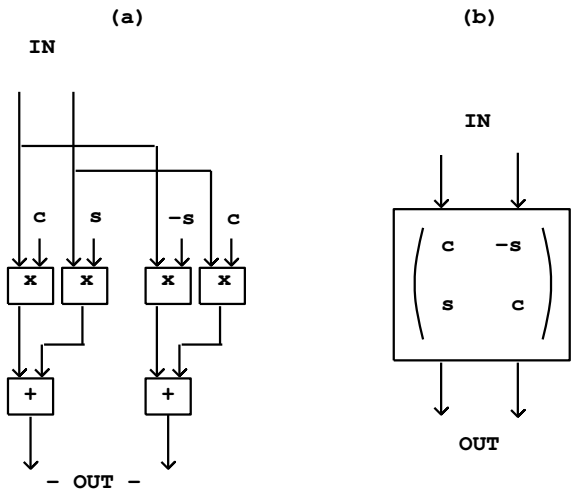


Figure 7.12: Second fundamental building block for unitary delay networks: rotating two digital audio signals. Part (a) shows the transformation explicitly; (b) shows it as a matrix operation.

This is just the Pythagorean distance from the origin to the point $(x_1, \ldots, x_r)$. Since rotations and reflections are distance-preserving transformations, the distance from the origin before transforming must equal the distance from the origin afterward. This shows that the total power of the transformed signals must equal the total power of the original ones.

Figure 7.12 shows a rotation matrix operating on two signals. In part (a) the transformation is shown explicitly. If the input signals are $x_1[n]$ and $x_2[n]$, the outputs are:

$$y_1[n] = cx_1[n] + sx_2[n]$$

$$y_2[n] = -sx_1[n] + cx_2[n]$$

where c, s are given by

$$c = \cos(\theta)$$

$$s = \sin(\theta)$$

for an *angle of rotation* θ. Considered as points on the Cartesian plane, the point (y_1, y_2) is just the point (x_1, x_2) rotated counter-clockwise by the angle θ. The two points are thus at the same distance from the origin:

$$|y_1|^2 + |y_2|^2 = |x_1|^2 + |x_2|^2$$

and so the two output signals have the same total power as the two input signals.

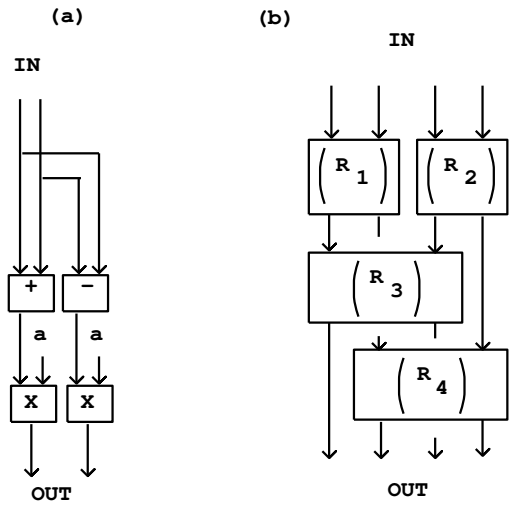


Figure 7.13: Details about rotation (and reflection) matrix operations: (a) rotation by the angle $\theta = \pi/4$, so that $a = \cos(\theta) = \sin(\theta) = \sqrt{1/2} \approx 0.7071$; (b) combining two-dimensional rotations to make higher-dimensional ones.

For an alternative description of rotation in two dimensions, consider complex numbers $X = x_1 + x_2 i$ and $Y = y_1 + y_2 i$. The above transformation amounts to setting

$$Y = XZ$$

where Z is a complex number with unit magnitude and argument θ. Since $|Z| = 1$, it follows that $|X| = |Y|$.

If we perform a rotation on a pair of signals and then invert one (but not the other) of them, the result is a *reflection*. This also preserves total signal power, since we can invert any or all of a collection of signals without changing the total power. In two dimensions, a reflection appears as a transformation of the form

$$y_1[n] = cx_1[n] + sx_2[n]$$

$$y_2[n] = sx_1[n] - cx_2[n]$$

Special and useful rotation and reflection matrices are obtained by setting the $\theta = \pm\pi/4$, so that $s = c = \sqrt{1/2}$. This allows us to simplify the computation as shown in Figure 7.13 (part a) because each signal need only be multiplied by the one quantity $c = s$.

Any rotation or reflection of more than two input signals may be accomplished by repeatedly rotating and/or reflecting them in pairs. For example, in part (b) of Figure 7.13, four signals are combined in pairs, in two succesive stages, so that in the end every signal input feeds into all the outputs. We could

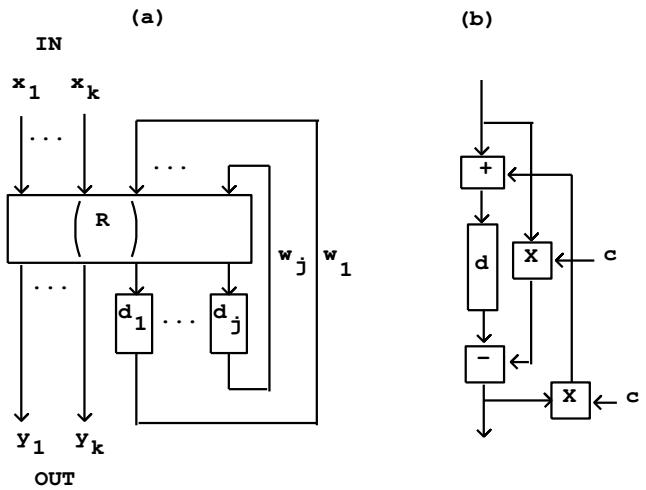


Figure 7.14: Flat frequency response in recirculating networks: (a) in general, using a rotation matrix R; (b). the "allpass" configuration.

do the same with eight signals (using three stages) and so on. Furthermore, if we use the special angle $\pi/4$, all the input signals will contribute equally to each of the outputs.

Any combination of delays and rotation matrices, applied in succession to a collection of audio signals, will result in a flat frequency response, since each individual operation does.

This already allows us to generate an infinitude of flat-response delay networks, but so far, none of them are recirculating. A third operation, shown in Figure 7.14, allows us to make recirculating networks that still enjoy flat frequency responses.

Part (a) of the figure shows the general layout. The transformation R is assumed to be any combination of delays and mixing matrices that are power preserving in the aggregate. The input signals $x_1, \ldots x_k$ are collectively labeled as a compound signal X, and similarly the output signals $y_1, \ldots y_k$ are shown as Y. Some other signals $w_1, \ldots w_j$ (where j is not necessarily equal to k) appear at the output of the transformation R and are fed back to its input.

If R is indeed power preserving, the total input power (the power of the signals X plus that of the signals W) must equal the output power (the power of the signals Y plus W), and subtracting the power of W from the equality, we find that the input and the output power are equal.

If we let $j = k = 1$ so that X, Y, and W each represent a single signal, and let the transformation R be a rotation by θ, followed by a delay of d samples on the W output, the result is the well-known allpass filter. With some juggling, and letting $c = \cos(\theta)$, we arrive at the network shown in part (b) of the figure.

Allpass filters have many applications, some of which we will visit later in this book.

7.6 Artificial reverberation

Reverberation in real, natural spaces arises from a complicated pattern of sound reflections off the walls and other objects that define the space. It is a great oversimplification to imitate this process by recirculating, discrete delay networks. Nonetheless, this approach of modeling reverberation using recirculating delay lines can, with much work, be made to give good results.

The central idea is to idealize any room (or other reverberant space) as a parallel collection of delay lines that model the memory of the air inside the room. At each point on the walls of the room, many straight-line paths terminate, each corresponding to a possible line of sight ending at that point; the sound then reflects into many other paths, each one originating at that point, and leading eventually to some other point on a wall.

Although the wall (and the air we passed through to get to the wall) absorbs some of the sound, a fraction of the incident power is reflected. If this fraction is close to one, the room reverberates for a long time; if it is exactly one, the reverberation lasts forever. If at any frequency the walls reflect more energy overall than they receive, the sound will feed back unstably; this never happens in real rooms (conservation of energy prevents it), but can happen in an artificial reverberator if it is not designed correctly.

To make an artificial reverberator using a delay network, we must fill two two competing demands simultaneously. First, the delay lines must be long enough to prevent *coloration* in the output as a result of from comb filtering. (Even if we move beyond the simple comb filter of section 7.4, the frequency response tends to have peaks and valleys whose spacing varies inversely with total delay time.) On the other hand, we should not hear individual echos; the *echo density* should ideally be at least one thousand per second.

To accomplish both of these aims, we assemble some number of delay lines and connect their outputs back to their inputs. The feedback path—the connection from the outputs back to the inputs of the delays—should have an aggregate gain that varies gently as a function of frequency, and never exceeds one for any frequency. A good starting point is to give the feedback path a flat frequency response and a gain slightly less than one; this is done using rotation matrices.

Ideally this is all we should need to do, but in reality we will not always want to use the thousands of delay lines it would take to model the paths between every possible pair of points on the walls. In practice we usually use between four and sixteen delay lines to model the room. This simplification sometimes reduces the echo density below what we would wish, so we might use more delay lines at the input of the recirculating network to increase the density.

Figure 7.15 shows a class of reverberator designs that use this principle. The incoming sound, shown as two separate signals in this example, is first thickened by progressively delaying one of the two signals and then intermixing them using

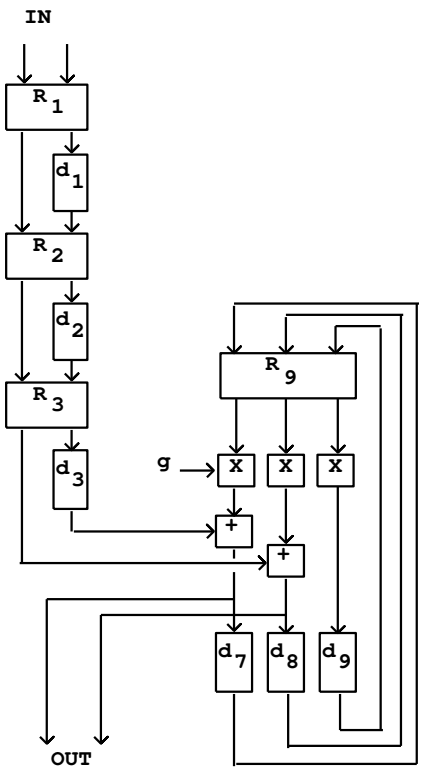


Figure 7.15: Reverberator design using power-preserving transformations and recirculating delays.

a rotation matrix. At each stage the number of echoes of the original signal is doubled; typically we would use between 6 and 8 stages to make between 64 and 256 echos, all with a total delay of between 30 and 80 milliseconds. The figure shows three such stages.

Next comes the recirculating part of the reverberator. After the original thickening, if any, the input signal is fed into a bank of parallel delay lines, and their outputs are again mixed using a rotation matrix. The mixed outputs are attenuated by a gain $g \leq 1$, and fed back into the delay lines to make a recirculating network.

The value g controls the reverberation time. If the average length of the recirculating delay lines is d, then any incoming sound is attenuated by a factor of g after a time delay of d. The gain after a time t is thus approximately

$$g^{t/d}$$

which in decibels gives:

$$20\frac{t}{d}\log_{10}(g).$$

The usual measure of reverberation time (RT) is the time at which the gain drops by sixty decibels:

$$20\frac{\mathrm{RT}}{d}\log_{10}(g) = -60$$

$$\mathrm{RT} = \frac{-3d}{\log_{10}(g)}$$

If g is one, this formula gives ∞, since the logarithm of one is zero.

The framework shown above is the basis for most modern reverberator designs. Many extensions of this underlying design have been proposed. The most important next step is to introduce filters in the recirculation path so that high frequencies can be made to decay more rapidly than low ones; this is readily accomplished with a very simple low-pass filter, but we will not work this out here, having not yet developed the needed filter theory.

In general, to use this framework to design a reverberator involves making many complicated choices of delay times, gains, and filter coefficients. Mountains of literature have been published on this topic; Barry Blesser has published a good overview [Ble01]. Much more is known about reverbertor design and tuning that has not been published; detailed reverberator designs are often kept secret for commercial reasons. In general, the design process involves painstaking and lengthy tuning by trial, error, and critical listening.

7.6.1 Controlling reverberators

Artificial reverberation is used almost universally in recording or sound reinforcement to sweeten the overall sound. However, and more interestingly, reverberation may be used as a sound source in its own right. The special case of infinite reverberation (i.e., reverberation with the gain set so that the reverberation time is infinite) is very useful as a way of grabbing live sounds and extending them in time.

To make this work in practice it is necessary to open the input of the reverberator only for a short period of time, during which the input sound is not varying too rapidly. If an infinite reverberator's input is left open the sound will collect and quickly become an indecipherable mass. To "infinitely reverberate" a note of a live instrument, it is best to wait until after the attack portion of the note and then allow perhaps 1/2 second of the note's steady state to enter the reverberator. It is possible to build chords from a monophonic instrument by repeatedly opening the input at different moments of stable pitch.

Figure 7.16 shows how this can be done in practice. The two most important controls are an input gain to the reverberator, and its feedback gain. To capture a sound, we set the feedback gain to one (infinite reverberation time) and momentarily open the input at time t_1. To add other sounds to an already held one, we simply reopen the input gain at the appropriate moments (at time

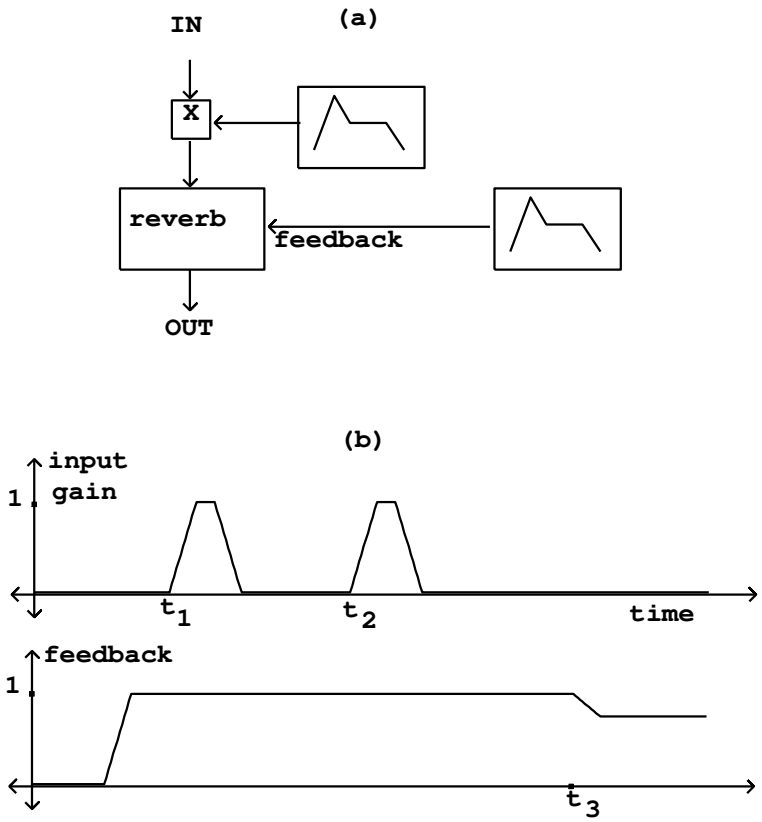


Figure 7.16: Controlling a reverberator to capture sounds pointilistically: (a) the network; (b) examples of how to control the input gain and feedback to capture two sounds at times t_1 and t_2, and to hold them until a later time t_3.

t_2 in the figure, for example.) Finally, we can erase the recirculating sound, thus both fading the output and emptying the reverberator, by setting the feedback gain to a value less than one (as at time t_3). The further we reduce the feedback gain, the faster the output will decay.

7.7 Variable and fractional shifts

Like any audio synthesis or processing technique, delay networks become much more powerful and interesting if their characteristics can be made to change over time. The gain parameters (such as g in the recirculating comb filter) may be controlled by envelope generators, varying them while avoiding clicks or other artifacts. The delay times (such as d before) are not so easy to vary smoothly for two reasons.

First, we have only defined time shifts for integer values of d, since for fractional values of d an expression such as $x[n-d]$ is not determined if $x[n]$ is only defined for integer values of n. To make fractional delays we will have to introduce some suitable interpolation scheme. And if we wish to vary d smoothly over time, it will not give good results simply to hop from one integer to the next.

Second, even once we have achieved perfectly smoothly changing delay times, the artifacts caused by varying delay time become noticeable even at very small relative rates of change; while in most cases you may ramp an amplitude control between any two values over 30 milliseconds without trouble, changing a delay by only one sample out of every hundred makes a very noticeable shift in pitch—so much so, that one frequently will vary a delay deliberately in order to hear the artifacts, and only incidentally getting from one specific delay time value to another one.

The first matter (fractional delays) can be dealt with using an interpolation scheme, in exactly the same way as for wavetable lookup (Section 2.5). For example, suppose we want to estimate a delay of $d = 1.5$ samples. For each n we want to estimate a value for $x[n-1.5]$. We could do this using standard four-point interpoation, putting a cubic polynomial through the four "known" points (0, x[n]), (1, x[n-1]), (2, x[n-2]), (3, x[n-3]), and then evaluating the polynomial at the point 1.5. Doing this repeatedly for each value of n gives the delayed signal.

This four-point interpolation scheme can be used for any delay of at least one sample. Delays of less than one sample can't be calculated this way because we need two input points more recent than the desired delay. This was possible in the above example, but for a delay of 0.5 samples, for instance, we would need the value of $x[n+1]$, which is in the future.

The accuracy of the estimate could be further improved by using higher-order interpolation schemes. However, there is a trade-off between quality and computational efficiency. Furthermore, if we move to higher-order interpolation schemes, the minimum possible delay will increase, causing trouble in some situations.

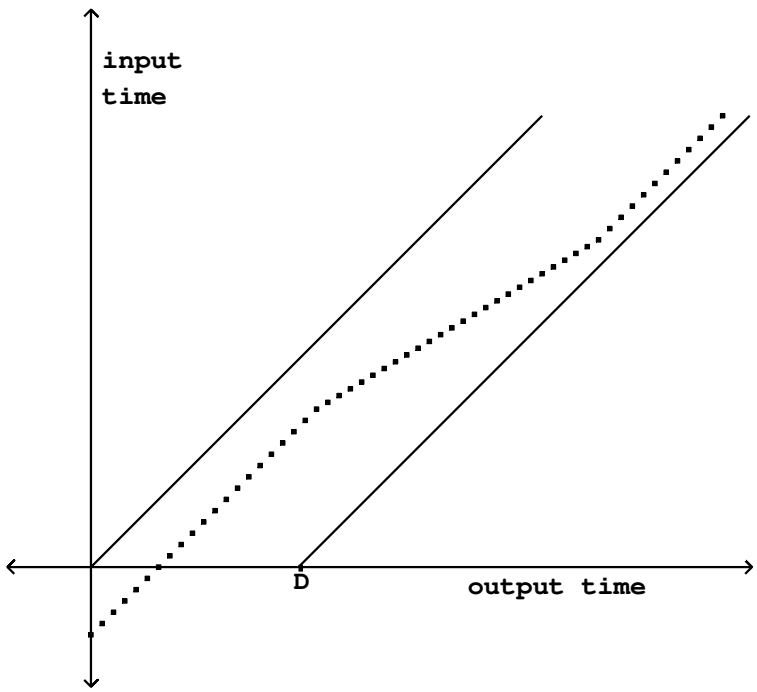


Figure 7.17: A variable length delay line, whose output is the input from some previous time. The output samples can't be newer than the input samples, nor older than the length D of the delay line. The slope of the input/output relationship controls the momentary transposition of the output.

The second matter to consider is the artifacts—whether wanted or unwanted—that arise from changing delay lines. In general, a discontinuous change in delay time will give rise to a discontinuous change in the output signal, since it is in effect interrupted at one point and made to jump to another. If the input is a sinusoid, the result is a discontinuous phase change.

If it is desired to change the delay line occasionally between fixed delay times (for instance, at the beginnings of musical notes), then we can use the techniques for managing sporadic discontinuities that were introduced in section 4.3. In effect these techniques all work by muting the output in one way or another. On the other hand, if it is desired that the delay time change continuously—while we are listening to the output—then we must directly address the question of artifacts that result from the changes.

Figure 7.17 shows the relationship between input and output time in a variable delay line. The delay line is assumed to have a fixed maximum length D. At each sample of output (corresponding to a point on the horizontal axis), we output one (possibly interpolated) sample of the delay line's input. The vertical

axis shows which sample (integer or fractional) to use from the input signal. Letting n denote the output sample number, the vertical axis shows the quantity $n - d[n]$, where $d[n]$ is the (time-varying) delay in samples. If we denote the input sample location by:

$$y[n] = n - d[n]$$

then the output of the delay line is:

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

where the signal x is evaluated at the point $y[n]$, interpolating appropriately in case $y[n]$ is not an integer. This is exactly the formula for wavetable lookup (page 29). We can use all the properties of wavetable lookup of recorded sounds to predict the behavior of variable delay lines.

There remains one difference between delay lines and wavetables: the material in the delay line is constantly being refreshed. Not only can we not read into the future, but, if the the delay line is D samples in length, we can't read further than D samples into the past either:

$$0 < d[n] < D$$

or, negating this and adding n to each side,

$$n > y[n] > n - D.$$

This last relationship appears as the region between the two diagonal lines in Figure 7.17; the function $y[n]$ must stay within this strip.

Returning to Section 2.2, the Momentary Transposition Formulas for Wavetables predict that the sound emerging from the delay line will be transposed by a factor $t[n]$ given by:

$$t[n] = y[n] - y[n-1] = 1 - (d[n] - d[n-1])$$

If $d[n]$ does not change with n, the transposition factor is 1 and the sound emerges from the delay line at the same speed as it went in. But if the delay time is increasing as a function of n, the resulting sound is transposed downward, and if $d[n]$ decreases, upward.

This is called the *Doppler effect*, and it occurs in nature as well. The air that sound travels through can sometimes be thought of as a delay line. Changing the length of the delay line corresponds to moving the listener toward or away from a stationary sound source; the Doppler effect from the changing path length works precisely the same in the delay line as it would be in the physical air.

Returning to Figure 7.17, we can predict that there is no pitch shift at the beginning, but then when the slope of the path decreases the pitch will drop for an interval of time before going back to the original pitch (when the slope returns to one). The delay time can be manipulated to give any desired transposition, but the greater the transposition, the less long we can maintain it before we run off the bottom or the top of the diagonal region.

7.8 Accuracy and frequency response of interpolating delay lines

Since they are in effect doing wavetable lookup, variable delay lines introduce distortion to the signals they operate on. Moreover, a subtler effect comes out in the case the delay line is not changing in length: the frequency response, in real situations, is never perfectly flat for a delay line whose length is not an integer.

If the delay time is changing from sample to sample, the distortion results of section 2.5 apply. To use them, we suppose that the delay line input can be broken down into sinusoids and consider separately what happens to each individual sinusoid. We can use table 2.5 to predict the RMS level of all the combined distortion products for an interpolated variable delay line.

We'll assume here that we want to use four-point interpolation. For sinusoids with periods longer than 32 samples (that is, for frequencies below 1/16 of the Nyquist frequency) the distortion is 96 dB or better—unlikely ever to be noticeable. At a 44 kHz. sample rate, these periods would correspond to frequencies up to about 1400 Hz. At higher frequencies the quality degrades, and above 1/4 the Nyquist frequency the distortion products, which are only down about 50 dB, should start to become readily audible.

The situation for a complex tone will depend primarily on the amplitudes and frequencies of its higher partials. Suppose, for instance, that a tone's partials above 5000 Hz. are at least 20 dB less than its strongest partial, and that above 10000 Hz they are down 60 dB. Then as a rough estimate, the distortion products from the range 5000-10000 will each be limited to about -68 dB and those from above 10000 Hz. will be limited to about -75 dB (because the worst figure in the table is about -15 dB and this must be added to the strength of the partial involved.)

If the high-frequency content of the input signal does turn out to give unacceptable distortion products, in general it is more effective to increase the sample rate than the number of points of interpolation. For periods greater than 4 samples, doubling the period (such as by doubling the sample rate) decreases distortion by about 24 dB.

The 4-point interpolating delay line's frequency response is nearly flat up to half the Nyquist frequency, but thereafter it can dive quickly. Suppose (to pick the worst case) that the delay is set halfway between two integers, say 1.5. Cubic interpolation gives:

$$x[1.5] = \frac{-x[0] + 9x[1] + 9x[2] - x[3]}{8}$$

Now let $x[n]$ be a (real-valued) unit-amplitude sinusoid with angular frequency ω, whose phase is zero at 1.5:

$$x[n] = \cos(\omega \cdot (n - 1.5))$$

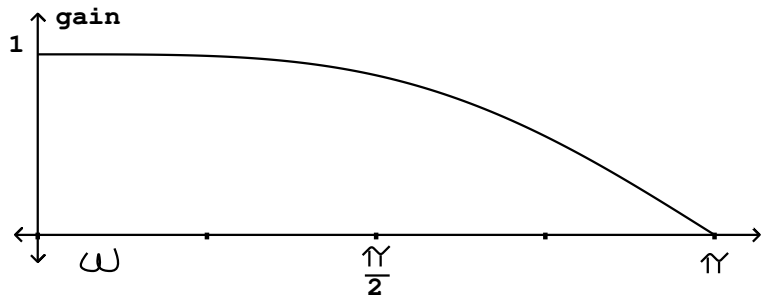


Figure 7.18: Gain of a four-point interpolating delay line with a delay halfway between two integers. The DC gain is one.

and compute $x[1.5]$ using the above formula:

$$x[1.5] = \frac{9\cos(\omega/2) - \cos(3\omega/2)}{4}$$

This is the peak value of the sinusoid that comes back out of the delay line, and since the peak amplitude going in was one, this shows the frequency response of the delay line. This is graphed in Figure 7.18. At half the Nyquist frequency ($\omega = \pi/2$) the gain is about -1 dB, which is a barely perceptible drop in amplitude. At the Nyquist frequency itself, however, the gain is zero.

As with the results for distortion, the frequency response improves radically with a doubling of sample rate. If we run our delay at a sample rate of 88200 Hz. instead of the standard 44100, we will get only about 1 dB of rolloff all the way up to 20000 Hz.

7.9 Pitch shifting

A favorite use of variable delay lines is to alter the pitch of an incoming sound using the Doppler effect. It may be desirable to alter the pitch in a variable way (either randomly or periodically, for example), or alternatively, to maintain a fixed musical interval of transposition over a length of time.

Returning to Figure 7.17, we see that, using a single variable delay line, we can maintain any desired pitch shift for a limited interval of time, but if we wish to sustain a fixed transposition we will always eventually land outside the diagonal strip of admissible delay times. In the simplest scenario, we simply vary the transposition up and down so as to remain in the strip.

This works, for example, if we wish to apply vibrato to a sound as shown in figure 7.19. Here the delay function is

$$d[n] = a_0 + a\cos(\omega n)$$

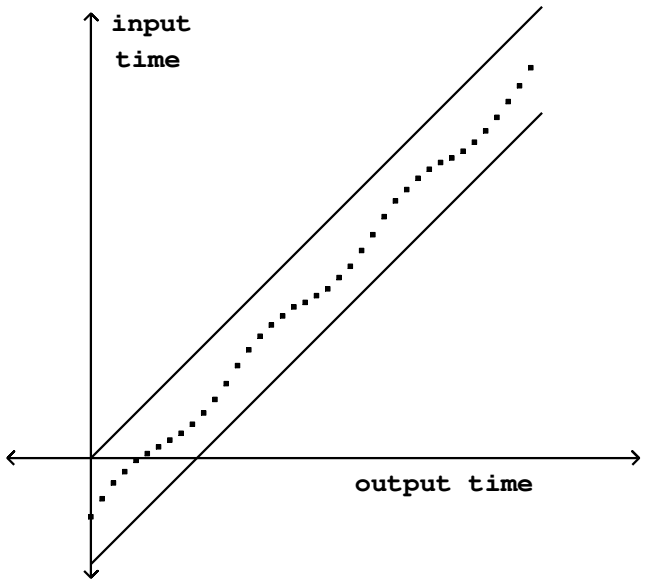


Figure 7.19: Vibrato using a variable delay line. Since the pitch shift alternates between upward and downward, it is possible to maintain it without drifting outside the strip of admissible delay.

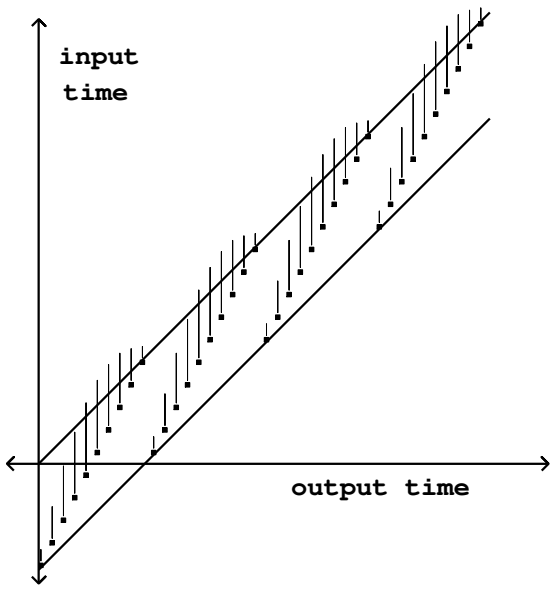


Figure 7.20: Piecewise linear delay functions to maintain a constant transposition (except at the points of discontinuity). The outputs are enveloped as suggested by the bars above each point, to smooth the output at the points of discontinuity in delay time.

where a_0 is the average delay, a is the amplitude of variation about the average delay, and ω is an angular frequency. The momentary transposition, which depends on the sample number n, is approximately

$$t = 1 + a\omega \cos(\omega n)$$

This ranges in value between $1 - a\omega$ and $1 + a\omega$.

Suppose, on the other hand, that we wish to maintain a constant transposition over a longer interval of time. In this case we caon't maintain the transposition forever, but it is still possible to maintain it over fixed intervals of time broken by discontinuous changes, as shown in Figure 7.20.

The delay time is the output of a suitably normalized sawtooth function, and the output of the variable delay line is enveloped as suggested in the figure to avoid discontinuities.

This is accomplished as shown in Figure7.21. The output of the sawtooth generator is used in two ways. First it is adjusted to run batween the bounds d_0 and $d_0 + w$, and this adjusted sawtooth controls the delay time, in samples. The initial delay d_0 should be at least enough to make the variable delay feasible; for four-point interpolation this must be at least one sample. Larger values of

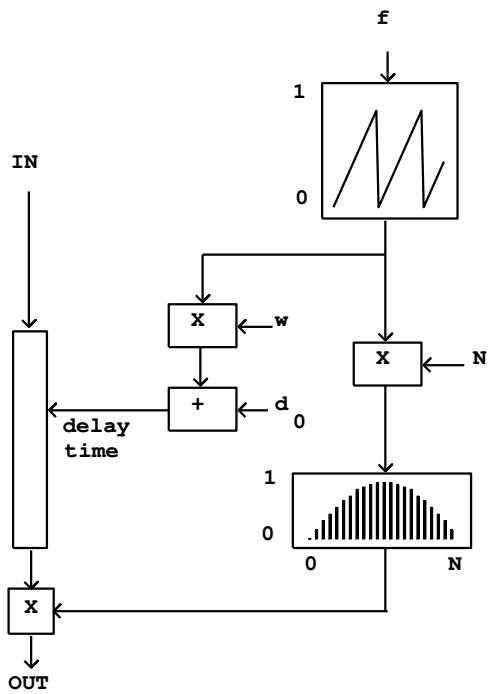


Figure 7.21: Using a variable delay line as a pitch shifter. The sawtooth wave creates a smoothly increasing or decreasing delay time. The output of the delay line is enveloped to avoid discontinuities. Another copy of the same diagram should run 180 degrees (π radians) out of phase with this one.

d_0 add a constant, additional delay to the output; this is usually offered as a control in a pitch shifter since it is essentially free. The quantity w is sometimes called the *window size*.

The sawtooth output is also used to envelope the output in exatly the same way as in the enveloped wavetable sampler of Figure 2.7. The envelope is zero at the points where the sawtooth wraps around, and in between, rises smoothly to a maximum value of 1 (for unit gain).

If the frequency of the sawtooth wave is f (in cycles per second), then its value sweeps from 0 to 1 every R/f samples (where R is the sample rate). The difference between successive samples is thus f/R. If we let $x[n]$ denote the output of the sawtooth oscillator, then

$$x[n+1] - x[n] = \frac{f}{R}$$

(except at the wraparound points). If we adjust the output range of the wavetable oscillator to the value w (as is done in the figure) we get a new slope:

$$w \cdot x[n+1] - w \cdot x[n] = \frac{wf}{R}$$

Adding the constant d_0 has no effect on this slope. The Momentary Transposition is then calculated as:

$$t = 1 - \frac{wf}{R}$$

To complete the design of the pitch shifter we must add the other copy halfway out of phase. This gives rise to a delay reading pattern as shown in Figure 7.22.

The pitch shifter can transpose either upward (using negative sawtooth frequencies, as in the figure) or downward, using positive ones. Pitch shift is usuappy controlled by changing f with w fixed. To get a desired transposition interval t, set

$$f = \frac{(t-1)R}{w}$$

The window size w should be chosen small enough, if possible, so that the two delayed copies ($w/2$ samples apart) do not sound as distinct echoes. However, very small values of w will force f upward; values of f greater than about 5 result in very audible modulation. So if very large transpositions are required, the value of w may need to be increased. Typical values range from 30 to 100 milliseconds (about $R/30$ to $R/10$ samples).

Although the frequency may be changed at will, even discontinuously, w must be handled more carefully. The most common choice is to mute the output while changing w discontinuously; alternatively, w may be ramped continuously but this causes hard-to-predict Doppler shifts.

The choice of envelope is usually one half cycle of a sinusoid. If we assume on average that the two delay outputs have neither positive nor negative correlation, the signal power from the two delay lines, after enveloping, will add to a constant (since the sum of squares of the two envelopes is one).

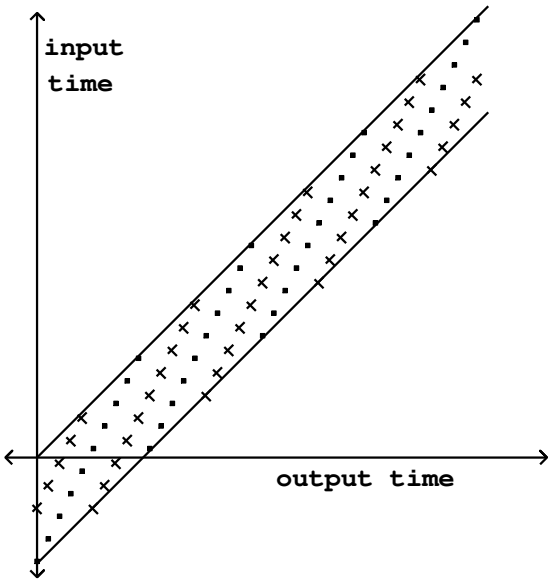


Figure 7.22: The pitch shifter's delay reading pattern using two delay lines, so that one is at maximum amplitude exactly when the other is switching.

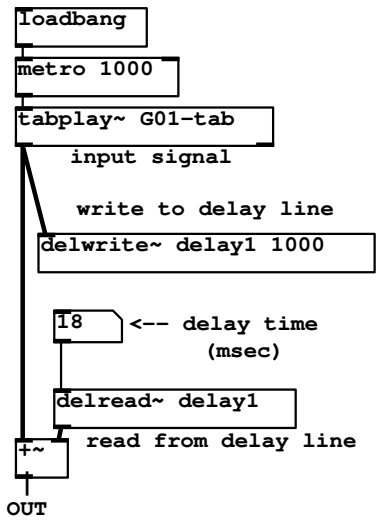


Figure 7.23: Example patch G01.delay.pd, showing a noninterpolating delay with a delay time controlled in milliseconds.

Many variations exist on this pitch shifting algorithm. One widely used variant is to use a single delay line, with no enveloping at all. In this situation it is necessary to choose the point at which the delay time jumps, and the point it jumps to, so that the output stays continuous. For example, one could wait for the output signal to pass through zero (a "zero crossing") and jump discontinuously to another one. Using only one delay line has the advantage that the signal output sounds more "present". A disadvantage is that, since the delay time is a function of input signal value, the output is no longer a linear function of the input.

7.10 Examples

7.10.1 Fixed, noninterpolating delay line

Patch G01.delay.pd (Figure 7.23) applies a simple delay line to an input signal. Two new objects are needed:

`delwrite~`: define and write to a delay line. The first creation argument gives the name of the delay line (and two delay lines may not share the same name). The second creation argument is the length of the delay line in milliseconds. The inlet takes an audio signal and writes it continuously into the delay line.

`delread~`: read from (or "tap") a delay line. The first creation argument gives the name of the delay line (which should agree with the name of the corre-

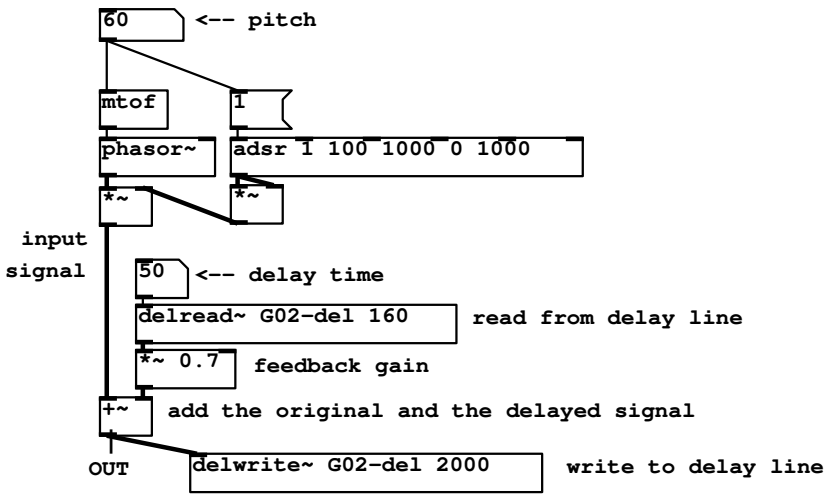


Figure 7.24: Recirculating delay (still noninterpolating).

sponding `delwrite~` object; this is how Pd knows which `delwrite~` to associate with the `delread~` object). The second (optional) creation argument gives the delay time in milliseconds. This may not be negative and also may not exceed the length of the delay line as specified to the `delwrite~` object. Incoming numbers (messages) may be used to change the delay time dynamically. However, this will make a discontinuous change in the output, which should therefore be muted if the delay time changes.

This example simply pairs one `delwrite~` and one `delread~` object to make a simple, noninterpolating delay. The input signal is a looped recording. The delayed and the non-delayed signal are added to make a non-recirculating comb filter. At delay times below about 10 milliseconds, the filtering effect is most prominent, and above that, a discrete echo becomes audible. There is no muting protection on the delay output, so clicks are possible when the delay time changes.

7.10.2 Recirculating comb filter

Patch G02.delay.loop.pd (Figure 7.24) shows how to make a recirculating delay network. The delay is again accomplished with a `delwrite~`/`delread~` pair. The output of the `delread~` object is multiplied by a feedback gain of 0.7 and fed into the `delwrite~` object. An input (supplied by the `phasor~` and associated objects) is added into the `delwrite~` input; this sum becomes the output of the network. This is the recirculating comb filter of Section 7.4.

The network of tilde objects does not have any cycles, in the sense of objects feeding either directly or indirectly (via connections through other ob-

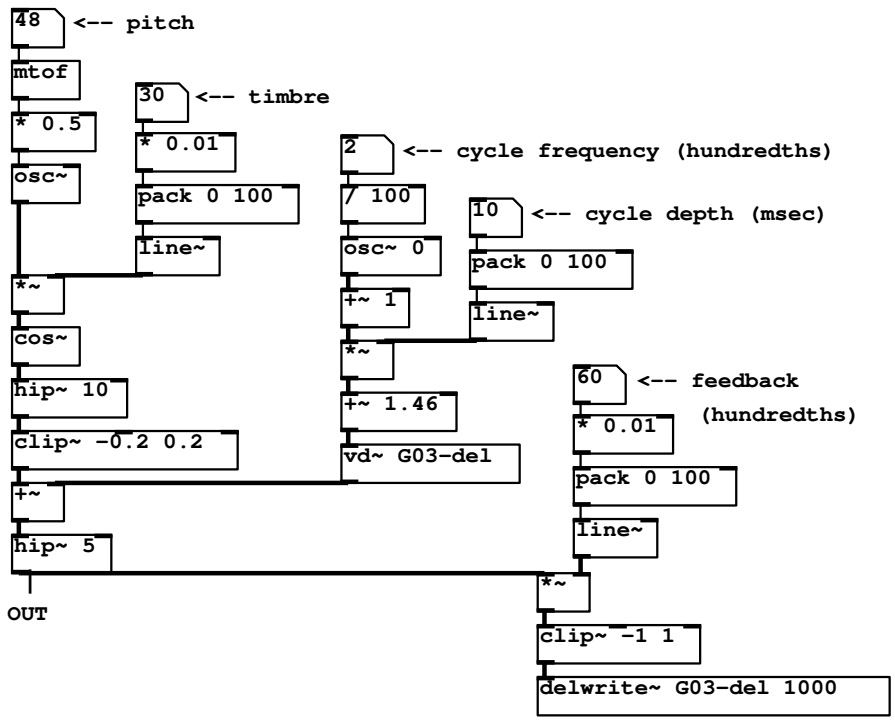


Figure 7.25: The flanger: an interpolating, variable delay line.

jects) to themselves. The feedback in the network occurs implicitly between the `delwrite~`and `delread~` objects.

7.10.3 Variable delay line

The next example, Patch G03.delay.variable.pd (Figure 7.25), is another recirculating comb filter, this time using a variable-length delay line. One new object is introduced here:

`vd~`: Read from a delay line, with a time-varying delay time. As with the `delread~` object, this reads from a delay line whose name is specified as a creation argument. Instead of using a second argument and/or control messages to specify the delay time, in the case of the `vd~` object the delay in milliseconds is specified by an incoming audio signal. The delay line is read using four-point (cubic) interpolation; the minimum achievable delay is one sample.

Here the objects on the left side, from the top down to the `clip~ -0.2 0.2` object, form a waveshaping network; the index is set by the "timbre" control, and the waveshaping output varies between a near sinusoid and a bright, buzzy sound. The output is added to the output of the `vd~` object. The sum is then

high pass filtered (the `hip~` object at lower left), multiplied by a feedback gain, clipped, and written into the delay line at bottom right. There is a control at right to set the feedback gain; here, in contrast with the previous example, it is possible to specify a gain greater than one in order to get unstable feedback. For this reason the second `clip~` object is inserted within the delay loop (just above the `delwrite~` object) so that the signal cannot exceed 1 in absolute value.

The length of the delay is controlled by the signal input to the `vd~` object. An oscillator with variable frequency and gain, in the center of the figure, provides the delay time. The oscillator is added to one to make it nonnegative before multiplying it by the "cycle depth" control, which effectively sets the range of delay times. The minimum possible delay time of 1.46 milliseconds is added so that the true range of delay times is between the minimum and the same plus twice the "depth". The reason for this minimum delay time is taken up in the discussion of the next example.

Comb filters with variable delay times are sometimes called *flangers*. As the delay time changes the peaks in the frequency response move up and down in frequency, so that the timbre of the output changes constantly in a characteristic way.

7.10.4 Order of execution and lower limits on delay times

When using delays (as well as other state-sharing tilde objects in Pd), the order in which the `delwrite~` and `delread~` operations are done can affect the outcome of the computation. Although the tilde objects in a patch may have a complicated topology of audio connections, in reality Pd executes them all in a sequential order, one after the other, to compute each block of audio output. This linear order is guaranteed to be compatible with the audio interconnections, in the sense that no tilde object's computation is done until all its inputs, for that same block, have been computed.

Figure 7.26 shows two examples of tilde object topologies and their translation into a sequence of computation. In part (a) there are four tilde objects, and because of the connections, the object `a` must produce its output before either of `b` or `c` can run; and both of those in turn are used in the computation of `d` . So the possible orderings of these four objects are `a-b-c-d` and `a-c-b-d`. These two orderings will have exactly the same result unless the computation of `b` and `c` somehow affect each other's output (as delays might).

Part (b) of the figure shows a cycle of tilde objects. This network cannot be sorted into a compatible sequential order, since each of `a` and `b` requires the other's output to be computed first. In general, a sequential ordering of the tilde objects is possible if and only if there are no cycles anywhere in the network of tilde objects and their audio signal interconnections. Pd reports an error when such a cycle appears. (Note that the situation for *control* interconnections between objects is more complicated and flexible; see the Pd documentation for details.)

To see the effect of the order of computation on a `delwrite~`/`delread~` pair, we can write explicitly the input and output signals in the two possible orders,

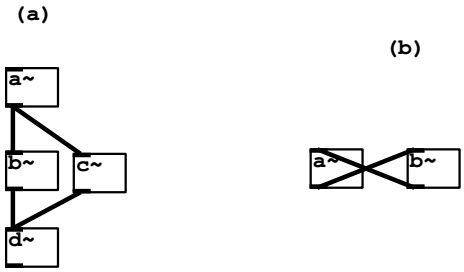


Figure 7.26: Order of execution of tilde objects in Pd: (a), an acyclic network. The objects may be executed in either the order a-b-c-d or a-c-b-d. In part (b), there is a cycle, and there is thus no compatible linear ordering of the objects because each one would need to be run before the other.

with the minimum possible delay. If the write operation comes first, at a block starting at sample number n, the operation can be written as:

$$x[n], \ldots, x[n+B-1] \longrightarrow \boxed{\texttt{delwrite\~{}}}$$

where B is the block size (as in Chapter 3). Having put those particular samples into the delay line, a following `delread~` is able to read the same values out:

$$\boxed{\texttt{delread\~{}}} \longrightarrow x[n], \ldots, x[n+B-1]$$

On the other hand, suppose the `delread~` object comes before the `delwrite~`. Then the samples $x[n], \ldots, x[n+B-1]$ have not yet been stored in the delay line, so the most recent samples that may be read belong to the previous block:

$$\boxed{\texttt{delread\~{}}} \longrightarrow x[n-B], \ldots, x[n-1]$$

$$x[n], \ldots, x[n+B-1] \longrightarrow \boxed{\texttt{delwrite\~{}}}$$

Here the minimum delay we can possibly obtain is the block size B. So the minimum delay is either 0 or B, depending on the order in which the `delread~` and `delwrite~`objects are sorted into a sequence of execution.

Looking back at the patches of Figures 7.24 and 7.25, which both feature recirculating delays, the `delread~` object must be placed earlier in the sequence than the `delwrite~` object. This is true of any design in which a delay's output is fed back into its input. The minimum possible delay is B samples. For a (typical) sample rate of 44100 Hz. and block size of 64 samples, this comes to 1.45 milliseconds. This might not sound at first like a very important restriction. But if you are trying to tune a recirculating comb filter to a specific pitch, the highest you can get only comes to about 690 Hz. To get shorter recirculating delays you must increase the sample rate or decrease the block size.

Patch G04.control.blocksize.pd (Figure 7.27) shows how the block size can be controlled in Pd using a new object:

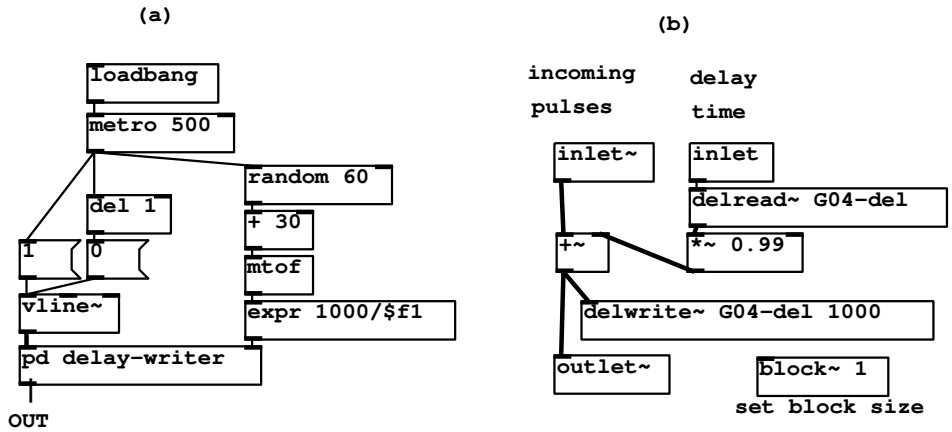


Figure 7.27: A patch using block size control to lower the loop delay below the normal 64 samples: (a). the main patch; (b). the "delay-writer" subpatch with a `block~` object and a recirculating delay network.

`block~`, `switch~`: Set the local block size of the patch window the object sits in. Block sizes must be powers of two. The `switch~` object, in addition, can be used to turn audio computation within the window on and off, using control messages. Additional creation arguments can set the local sample rate and specify overlapping computations (demonstrated in Chapter 9).

In part (a) of the figure (the main patch), a rectangular pulse is sent to the `pd delay-writer` subpatch, whose output is then returned to the main patch. Part (b) of the figure shows the contents of the subpatch, which sends the pulses into a recirculating delay. The `block~` object specifies that, in this subpatch, signal computation uses a block size (B) of only one. So the minimum achievable delay is one sample instead of the default 64.

Putting a pulse (or other excitation signal) into a recirculating comb filter to make a pitch is sometimes called *Karplus-Strong synthesis*, having been described in a paper by them [KS83], although the idea seems to be older. It made its way for example into Paul Lansky's 1979 piece, *Six Fantasies on a poem by Thomas Campion*.

7.10.5 Order of execution in non-recirculating delay lines

In nonrecirculating delay networks, it should be possible to place the operation of writing into the delay line earlier in the sequence than that of reading it. There should thus be no lower limit on the length of the delay line (except whatever is imposed by the interpolation scheme; see section 7.7). In languages such as Csound, the sequence of unit generator computation is (mostly) explicit, so this is easy to specify. In the graphical patching environments, however, the order is

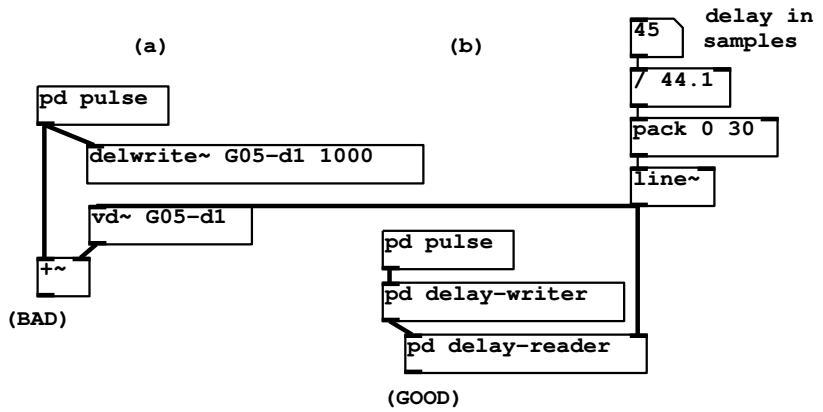


Figure 7.28: Using subpatches to ensure that delay lines are written before they are read in non-recirculating networks: (a) the `delwrite~` and `vd~` objects might be executed in either the "right" or the "wrong" order; (b) the `delwrite~` object is inside the `pd delay-writer` subpatch and the `vd~` object is inside the `pd delay-reader` one. Because of the audio connection between the two subpatches, the order of execution of the read/write pair is forced to be the correct one.

implicit and another approach must be taken to ensuring that, for example, a `delwrite~` object is computed before the corresponding `delread~` object. One way of accomplishing this is shown in Patch G05.execution.order.pd (Figure 7.28).

In part (a) of the figure, the connections in the patch do not determine which order the two delay operations appear in the sorted sequence of tilde object computation; the `delwrite~` object could be computed either before or after the `vd~` object. If we wish to make sure the writing operation happens before the reading operation, we can proceed as in part (b) of the figure, and put the two operations in subpatches, connecting the two via audio signals so that the first subpatch must be computed before the second one. (Audio computation in subpatches is done atomically, in the sense that the entire subpatch contents are considered as the audio computation for the subpatch as a whole. So everything in the one subpatch happens before anything in the second one.)

In this example, the "right" and the "wrong" way to make the comb filter have audibly different results. For delays less than 64 samples, the right hand side of the patch (using subpatches) gives the correct result, but the left hand side doesn't produce delays below the 64 sample block size.

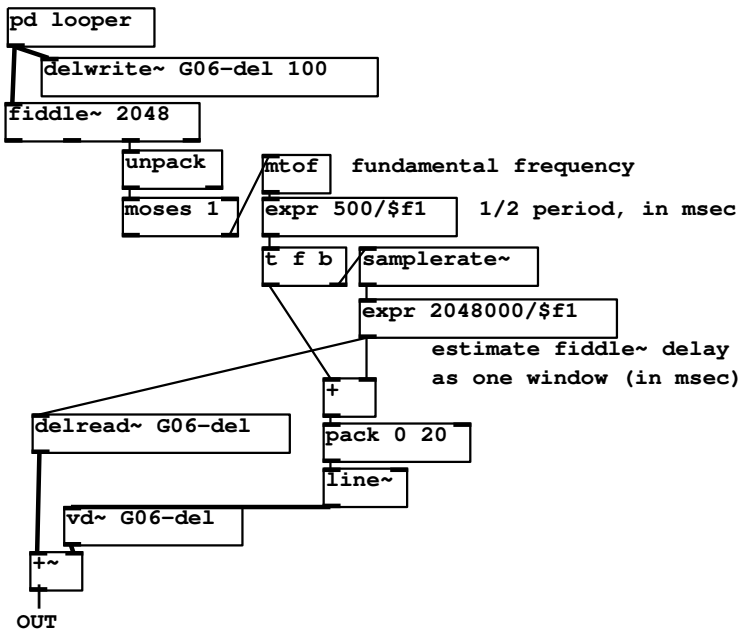


Figure 7.29: An "octave doubler" uses pitch information (obtained using a `fiddle~` object) to tune a comb filter to remove the odd harmonics in an incoming sound.

7.10.6 Non-recirculating comb filter as octave doubler

In patch G06.octave.doubler.pd (Figure 7.29) we revisit the idea of pitch-based octave shifting introduced earlier in patch E03.octave.divider.pd. There, knowing the periodicity of an incoming sound allowed us to tune a ring modulator to introduce subharmonics. Here we realize the octave doubler described in Section 7.3. Using a variable, non-recirculating comb filter we filter out odd harmonics, leaving only the even ones, which sound an octave higher. As before, the spectral envelope of the sound is roughly preserved by the operation, so we can avoid the "chipmunk" effect we would have got by using speed change to do the transposition.

The comb filtering is done by combining two delayed copies of the incoming signal (from the `pd looper` subpatch at top). The fixed one (`delread~`) is set to a delay of 1/2 the window size of the pitch following algorithm. Whereas in the earlier example this was hidden in another subpatch, we can now show this explicitly. The delay in milliseconds is estimated as equal to the 2048-sample analysis window used by the `fiddle~` object; in milliseconds this comes to $1000 \cdot 2048/R$ where R is the sample rate.

The variable delay is the same, plus 1/2 of the measured period of the

incoming sound, or $1000/(2f)$ milliseconds where f is the frequency in cycles per second. The sum of this and the fixed delay time is then smoothed using a `line~` object to make the input signal for the variable delay line.

Since the difference between the two delays is $1/(2f)$, the resonant frequencies of the resulting comb filter are $2f, 4f, 6f \cdots$; the frequency response (section 7.3) is zero at the frequencies $f, 3f, \ldots$, so the resulting sound contains only the partials at multiples of $2f$—an octave above the original. Seen another way, the incoming sound is output twice, a half-cycle apart; odd harmonics are thereby shifted 180 degrees (π radians) and cancel; even harmonics are in phase with their delayed copies and remain in the sum.

Both this and the octave divider may be altered to make shifts of 3 or 4 to one in frequency, and they may also be combined to make compound shifts such as a music fifth (a ratio of 3:2) by shifting down an octave and then back up a factor of three. (You should do the down-shifting before the up-shifting for best results.)

7.10.7 Time-varying complex comb filter: shakers

Patch G07.shaker.pd(Figure 7.30) shows a different way of extending the idea of a comb filter. Here we combine the input signal at four different time shifts (instead of two, as in the original non-recirculating comb filter), each at a different positive and/or negative gain. To do this, we insert the input signal into a delay line and tap it at three different points; the fourth "tap" is the original, undelayed signal.

As a way of thinking about the frequency response of a four-tap comb filter, we consider first what happens when two of the four gains are close to zero. Then we end up with a simple non-recirculating comb filter, with the slight complication that the gains of the two delayed copies may be different. If they are both of the same sign, we get the same peaks and valleys as predicted in Section 7.3, only with the valleys between peaks possibly being more shallow. If they are opposite in sign, the valleys become peaks and the peaks become valleys.

Depending on which two taps we supposed were nonzero, the peaks and valleys are spaced by different amounts; the delay times are chosen so that 6 different delay times can arise in this way, ranging between 6 and 30 milliseconds. In the general case in which all the gains are non-zero, we can imagine the frequency response function varying continuously between these extremes, giving a succession of complicated patterns. This has the effect of raising and lowering the amplitudes of the partials of the incoming signal, all independently of the others, in a complicated pattern, to give a steadily time-varying timbre.

The right-hand side of the patch takes care of changing the gains of the input signal and its three time-shifted copies. Each time the `metro` object fires, a counter is incremented (the `f`, `+ 1`, and `mod 4` objects). This controls which of the amplitudes will be changed. The amplitude itself is computed by making a random number and normalizing it to lie between -0.7 and 1.3 in value. The random value and the index are packed (along with a third value, a time interval)

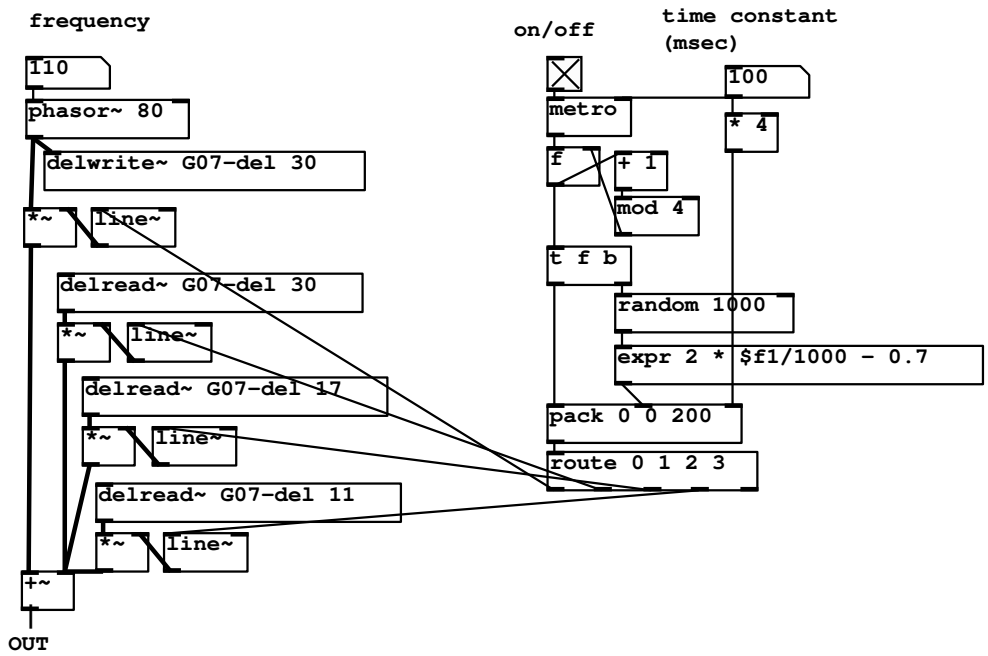


Figure 7.30: A "shaker", a four-tap comb filter with randomly varying gains on the taps.

and this triple goes to the `route` object. The first element of the triple (the counter) selects which output to send the other two values to; as a result, one of the four possible `line~` objects gets a message to ramp to a new value.

If the time variation is done quickly enough, there is also a modulation effect on the original signal; in this situation the straight line segments used in this example should be replaced by modulating signals with more controllable frequency content; we will revisit this in the next chapter.

7.10.8 Reverberator

Patch G08.reverb.pd(Figure 7.31) shows a simple artificial reverberator, essentially a realization of the design shown in Figure 7.15. Four delay lines are fed by the input and by their own recirculated output. The delay outputs are intermixed using rotation matrices, built up from elementary rotations of $\pi/4$ as in Figure 7.13 part (a).

The normalizing multiplication (by $\sqrt{1/2}$ at each stage) is absorbed into the feedback gain, which therefore cannot exceed 1/2. At a feedback gain of exactly 1/2, all the energy leaving the delay lines is reinserted into them, so the reverberation lasts perpetually.

Figure 7.32 shows the interior of the `reverb-echo` abstraction used in the reverberator. The two inputs are mixed (using the same rotation matrix and again leaving the renormalization for later). One channel is then delayed. The delay times are selected to grow roughly exponentially; this ensures a smmoth and spread-out pattern of echos.

Many extensions of this idea are possible of which we'll only name a few. It is natural, first, to put low-pass filters at the end of the delay lines, to mimic the typically faster decay of high frequencies than low ones. It is also common to use more than four recirculating delays; one reverberator in the Pd distribution uses sixteen. Finally, it is common to allow separate control of the amplitudes of the early echos (heard directly) and that of the recirculating signal; parameters such as these are thought to control sonic qualities described as "presence", "warmth", "clarity", and so on.

7.10.9 Pitch shifter

Patch G09.pitchshift.pd(Figure 7.33) shows a realization of the pitch shifter described in Section 7.9. A delay line (defined and written elsewhere in the patch) is read using two `vd~` objects. The delay times vary between a minimum delay (provided as the "delay" control) and the minimum plus a window size (the "window" control.)

The desired pitch shift in half-tones (h) is first converted into a transposition factor

$$t = 2^{h/12} = e^{\log(2)/12 \cdot h} \approx e^{0.05776h}$$

(called "speed change" in the patch). The computation labeled "tape head rotation speed" is the same as the formula for f given on page 204. Here the

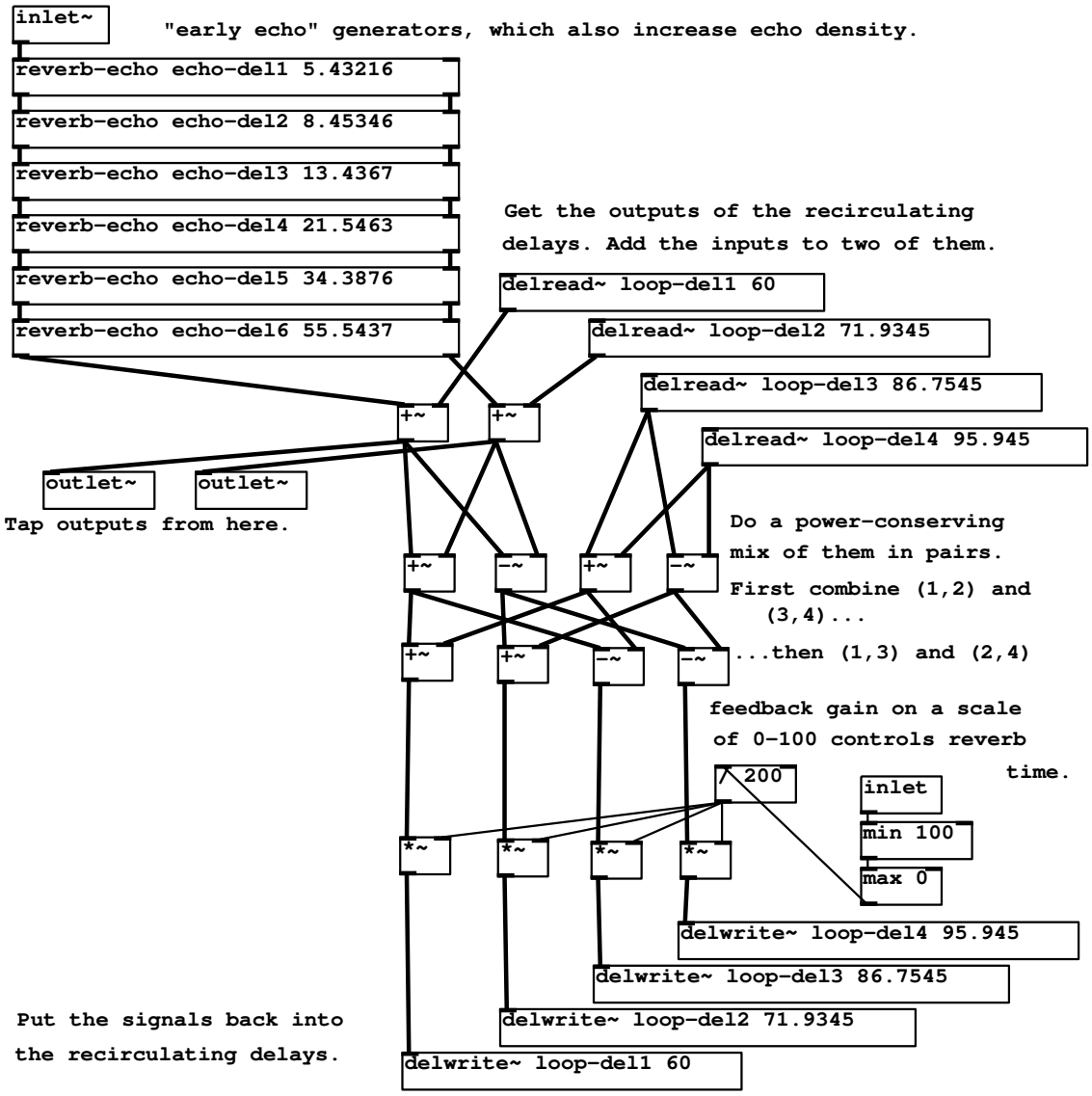


Figure 7.31: An artificial reverberator.

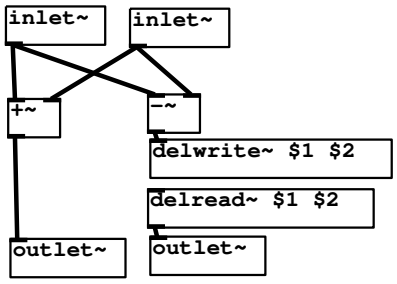


Figure 7.32: The echo generator used in the reverberator.

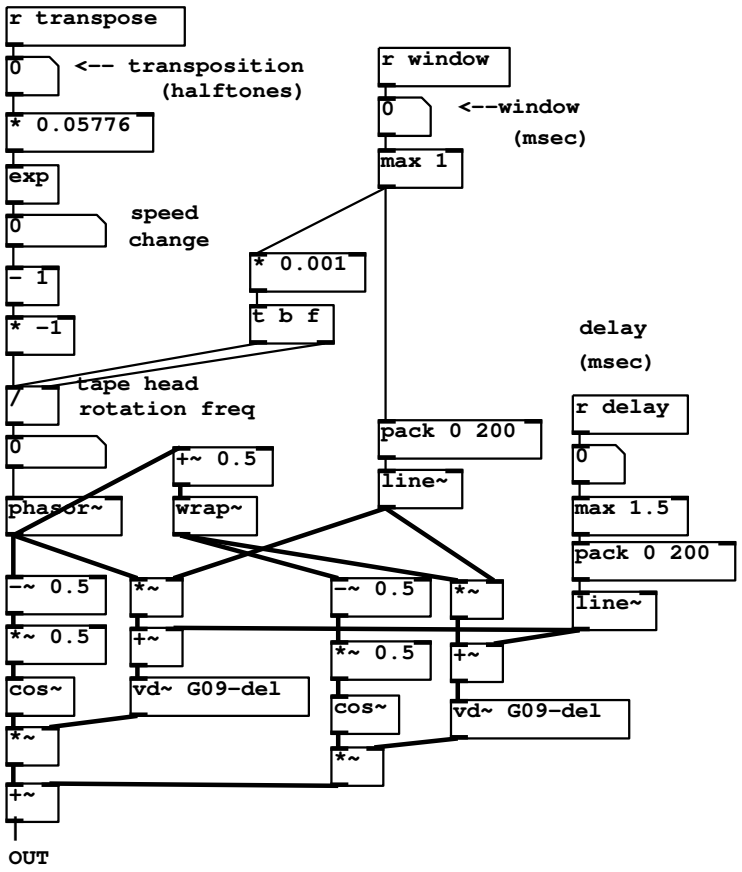


Figure 7.33: A pitch shifter using two variable taps into a delay line.

positive interval (seven half steps) gives rise to a transposition factor greater than one, and therefore to a negative value for f.

Once f is calculated, the production of the two phased sawtooth signals and the corresponding envelopes parallels exactly that of Section 2.6.5 (the overlapping sample looper). The minimum delay is added to each of the two sawtooth signals to make delay inputs for the `vd~` objects, whose outputs are multiplied by the corresponding envelopes and summed.

7.10.10 Exercises

1. A complex number has magnitude one and argument $\pi/4$. What are its real and imaginary parts?

2. A complex number has magnitude one and real part 1/2. What is its imaginary part? (There are two possible values.)

3. What delay time would you give a comb filter so that its first frequency response peak is at 440 Hz.? If the sample rate is 44100 Hz., what frequency would correspond to the nearest integer delay?

4. Suppose you made a variation on the non-recirculating comb filter so that the delayed signal was subtracted from the original instead of adding. What would the new frequency response be?

5. Derive an explicit formula for the frequency response of a recirculating comb filter with delay time d and feedback gain g, as a function of angular frequency ω.

$$\text{(answer :}\left[(1 - g\cos(2\pi\omega d))^2 + (g\sin(2\pi\omega d))^2\right]^{-1/2}\text{)}$$

6. If you want to make a 6-Hz. vibrato with a sinusoidally varying delay line, and if you want the vibrato to change the frequency by 5%, how big a delay variation would you need? How would this change if the same depth of vibrato was desired at 12 Hz.?

Chapter 8

Filters

In the previous chapter we saw that a delay network can have a non-uniform frequency response, in other words, a gain that varies as a function of frequency. Delay networks also typically affect the phase of incoming signals in ways that depend on frequency. If the delay times used are short, the frequency-dependent gain and phase change become the best way of describing the effect of the delay network on its input. Delay networks which are designed specifically to effect particular amplitude or phase changes on their inputs are called *filters*.

In block diagrams, filters are shown as in figure 8.1, part (a). The curve shown within the block gives a qualitative representation of the filter's frequency response (as defined in section 7.3). This frequency response may vary with time, and depending on the design of the filter, one of more controls (or additional audio inputs) might be used to change it.

Suppose, still following the procedure of section 7.3, we put a unit-amplitude, complex-valued sinusoid with angular frequency ω into a filter. We expect to get out a sinusoid of the same frequency and some amplitude, which depends on ω. This gives us a complex-valued function $H(\omega)$, which is called the *transfer function* of the filter.

The frequency response is the gain as a function of the frequency ω. It is is equal to the absolute value of the transfer function. A filter's frequency response is customarily graphed as in Figure 8.1, part (b). An incoming unit-amplitude sinusoid of frequency ω comes out of the filter with amplitude $|H(\omega)|$.

It is sometimes also useful to know the phase response of the filter, equal to $\angle(H(\omega))$. For a fixed frequency ω, the filter's output phase will be $\angle(H(\omega))$ radians ahead of its input phase.

The design and use of filters is a huge subject, because the wide range of uses a filter might be put to encourages a wide variety of filter design processes. In some uses a filter must exactly follow a prescribed frequency response, in others it is important to minimize computation time, in others the phase response is important, and in still others the filter must behave well when its parameters change quickly with time.

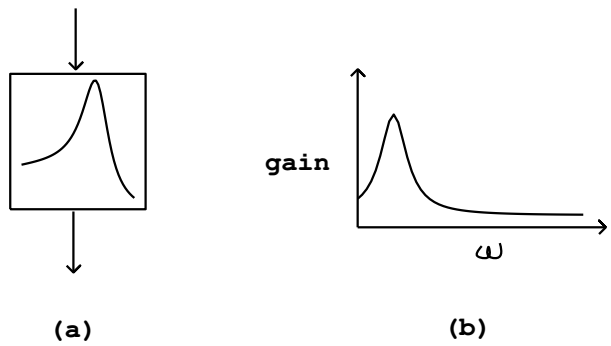


Figure 8.1: Representations of a filter. (a) in a block diagram; (b) a graph of its frequency response.

8.1 Taxonomy of filters

Over the history of electronic music the technology for building filters have changed constantly, but certain kinds of filters reappear throughout. In this section we will give some nomenclature for describing filters of several generic, recurring types. Later we'll develop some basic strategies for making filters with desired characteristics, and finally we'll discuss some common applications of filters in computer music.

8.1.1 Low-pass and high-pass filters

By far the most frequent purpose for using a filter is extracting either the low-frequency or the high-frequency portion of an audio signal, attenuating the rest. This is accomplished using a *low-pass* or *high-pass* filter.

Ideally, a low-pass or high-pass filter would have a frequency response of 1 up to (or down to) a specified cutoff frequency and zero past it; but such filters cannot be realized in practice. Instead, we try to find realizable approximations to this ideal response. The more design effort and computation time we put into it, the closer we can get.

Figure 8.2 shows the frequency response of a low-pass filter. The frequency spectrum is divided into three bands, labeled on the horizontal axis. The *passband* is the region (frequency band) where the filter should pass its input through to its output with unit gain. For a low-pass filter (as shown), the passband reaches from a frequency of zero up to a certain frequency limit. For a high-pass filter, the passband would appear on the right-hand side of the graph and would extend from the frequency limit up to the highest frequency possible. Any realizable filter's passband will be only approximately flat; the deviation from flatness is called the *ripple*, and is often specified by giving the ratio between

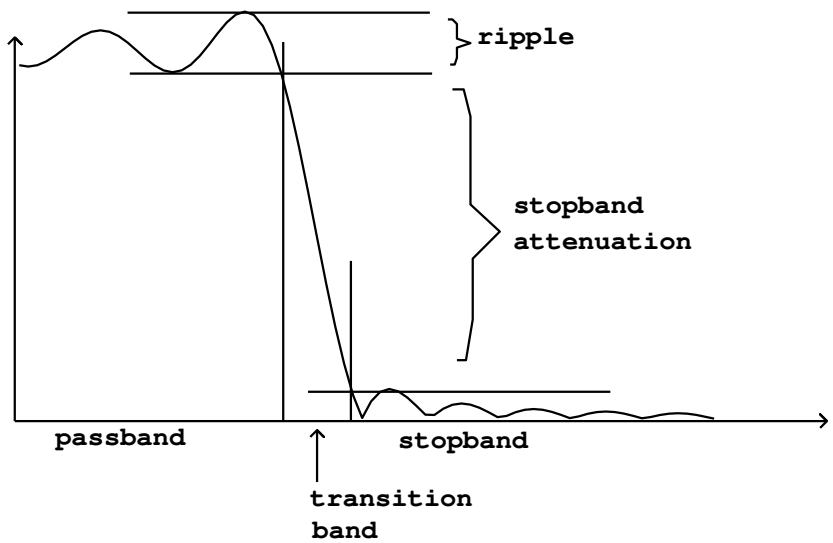


Figure 8.2: Terminology for describing the frequency response of low-pass and high-pass filters. The horizontal axis is frequency and the vertical axis is gain. A low-pass filter is shown; a high-pass filter has the same features switched from right to left.

the highest and lowest gain in the passband, expressed in decibels. The ideal low-pass or high-pass filter would have a ripple of 0 dB.

The *stopband* of a low-pass or high-pass filter is the region of the spectrum (the frequency range) over which the filter is intended not to transmit its input. The *stopband attenuation* is the difference, in decibels, between the lowest gain in the passband and the highest gain in the stopband. Ideally this would be infinite; the higher the better.

Finally, a realisable filter, whose frequency response is always a continuous function of frequenecy, always needs a region, or frequency band, over which the gain drops from the passband gain to the stopband gain; this is called the *transition band*. The thinner this band can be made, the more nearly ideal the filter.

8.1.2 Band-pass and stop-band filters

A *band-pass filter* admits frequencies within a given band, rejecting frequencies below it and above it. Much the same terminology as before may be used to describe a band-pass filter, as shown in figure 8.3. A stop-band filter does the reverse, rejecting frequencies withing the band and letting through frequencies outside it.

In practice, a simpler language is often used for describing bandpass filters, as shown in Figure 8.4. Here there are only two controls: a *center frequency* and a *bandwidth*. The passband is considered to be the region where the filter's output has at least half the power ($\sqrt(1/2)$ times the gain) of its peak. The bandwidth is the width, in frequency units, of the passband. The center frequency is the point of maximum gain, which is approximately the midpoint of the passband.

8.1.3 Equalizing filters

In some situations, such as *equalization*, the goal isn't to pass signals of certain frequencies while stopping others altogether, but to make controllable adjustments, boosting or attenuating signals, over a frequency range, by a desired gain. Two other filter types are useful for this. First, a *shelving filter* (figure 8.5) is used for selectively boosting or reducing either the low or high end of the frequency range. Below a selectable crossover frequency, the filter tends toward a low-frequency gain, and above it it tends toward a different, high-frequency one. The crossover frequency, low-frequency gain, and high-frequency gain can all be adjusted independently.

Second, a *peaking filter* (Figure 8.6) is capable of boosting or attenuating signals within a range of frequencies. The center frequency and bandwidth (which together control the range of frequencies affected), and the in-band and out-of-band gains are separately adjustible.

Parametric equalizers often employ two shelving filters (one each to adjust the low and high ends of the spectrum) and two or three peaking filters to adjust bands in between.

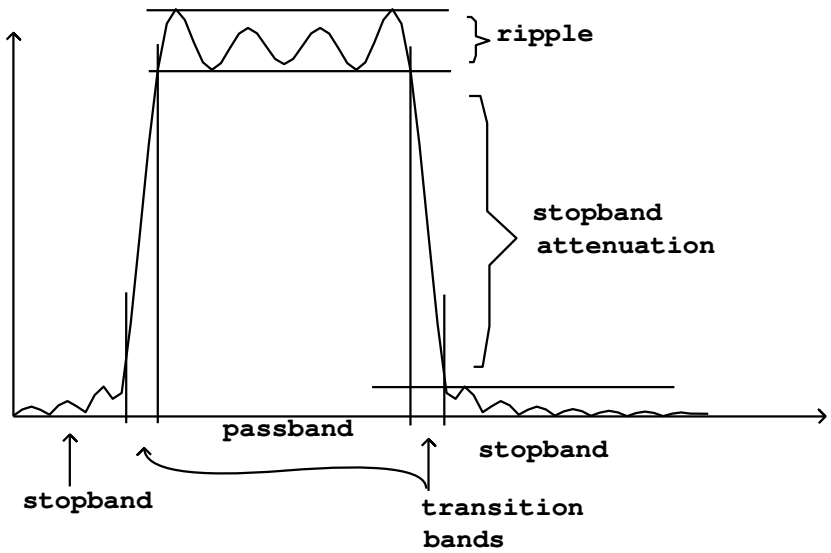


Figure 8.3: Terminology for describing the frequency response of band-pass and stop-band filters. The horizontal axis is frequency and the vertical axis is gain. A band-pass filter is shown; a stop-band filter would have a contiguous stopband surrounded by two passbands.

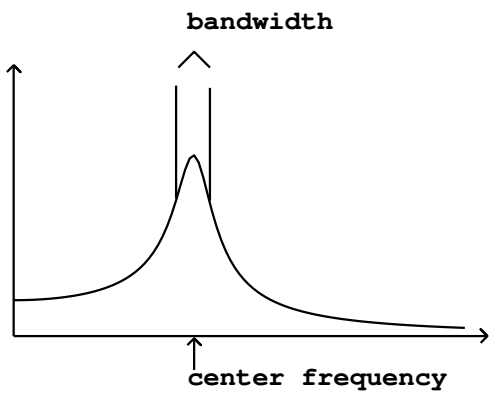


Figure 8.4: A simplified view of a band-pass filter, showing bandwidth and center frequency.

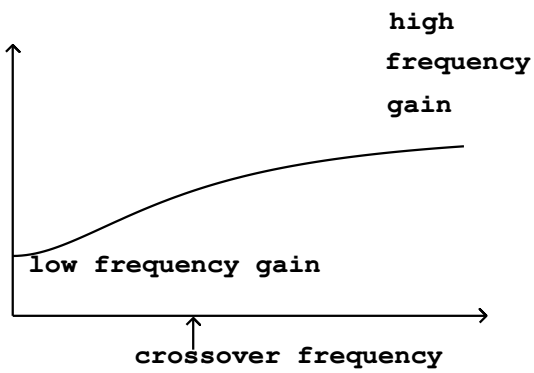


Figure 8.5: A shelving filter, showing low and high frequency gain, and crossover frequency.

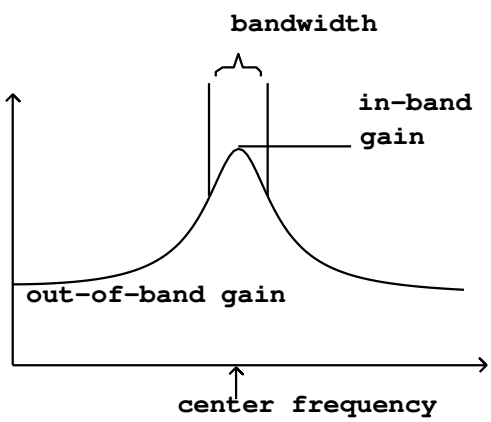


Figure 8.6: A peaking filter, with controllable center frequency, bandwidth, and in-band and out-of-band gains.

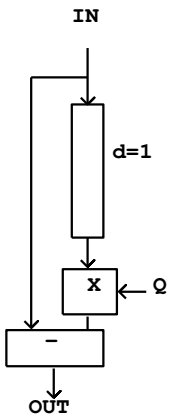


Figure 8.7: A delay network with a single-sample delay and a complex gain Q. This is the non-recirculating elementary filter, first form. Compare the simpler non-recirculating comb filter shown in Figure 7.3, which corresponds to choosing $Q = -1$ here.

8.2 Designing filters

We saw in chapter 7 how to predict the frequency and phase response of delay networks. The art of filter design lies in finding a delay network whose transfer function (which controls the frequency and phase response) has a desired shape. We will devalop an approach to designing such delay networks by using the two types of comb filters developed in chapter 7: recirculating and non-recirculating. Here we will be interested in the special case where the delay is only one sample in length. In this situation, the frequency responses shown in Figures 7.6 and 7.10 no longer look like combs; the second peak recedes all the way to the sample rate, 2π radians, when $d = 1$. Since only frequencies between 0 and the Nyquist frequency (π radians) are audible, in effect there is only one peak when $d = 1$.

In the comb filters shown in Chapter 7, the peaks are situated at DC (zero frequency), and we will now want to be able to place them at other, nonzero frequencies. We will be able to do this by using delay networks—comb filters—with complex-valued gains.

8.2.1 Elementary non-recirculating filter

We generalize the non-recirculating comb filter to the design shown in figure 8.7, called the *non-recirculating elementary filter*, of the first form.

To find the frequency response, as in Chapter 7 we feed the delay network a complex sinusoid $1, Z, Z^2, \ldots$ whose frequency is $\omega = \arg(Z)$, so that as before,

$Z = e^{i\omega}$. The nth sample of the input is Z^n and that of the output is

$$(1 - QZ^{-1})Z^n$$

so the transfer function is

$$H(Z) = 1 - QZ^{-1} = 1 - Qe^{i\omega}.$$

This can be represented graphically as shown in Figure 8.8. Suppose we write the coefficient Q in polar form:

$$Q = qe^{i\alpha}$$

Then the gain of the filter is the distance from the point Q to the point Z in the complex plane. Analytically we can see this because

$$|1 - QZ^{-1}| = |Z||1 - QZ^{-1}| = |Q - A|$$

Graphically, the number QZ^{-1} is just the number Q rotated backwards (clockwise) by the angular frequency ω of the incoming sinusoid. The value $|1-QZ^{-1}|$ is the distance from QZ^{-1} to 1 in the complex plane, which is equal to the distance from Q to Z.

As the frequency of the input sweeps from 0 to 2π, the point Z travels couterclockwise around the unit circle. At the point where $\omega = \alpha$, the distance is at a minimum, equal to $q - 1$. The maximum occurs which Z is at the opposite point of the circle. Figure 8.9 shows the transfer function for three different values of q.

8.2.2 Non-recirculating filter, second form

Occasionally we will use a variant of the filter above, as shown in Figure 8.10, called the elementary non-recirculating filter, second form. Instead of multiplying the delay output by Q we multiply the direct signal by its *complex conjugate* $\overline{A}$. (If

$$A = a + bi = re^{i\alpha}$$

is any complex number, its complex conjugate is defined as:

$$\overline{A} = a - bi = re^{-i\alpha}$$

Graphically this flips the entire complex plane across the real axis.) The transfer function of the new filter is

$$H(Z) = |\overline{Q} - Z^{-1}|$$

This gives rise to the same frequency response as the first form since

$$|\overline{Q} - Z^{-1}| = |Q - \overline{Z^{-1}}| = |Q - Z|$$

Here we use the fact that $\overline{Q} = Z^{-1}$, for all unit complex numbers $Q = e^{i\omega}$, since

$$\overline{Q} = e^{-i\omega} = Q^{-1}$$

Although the two forms of the elementary non-recirculating filter have the same frequency response, their phase responses are different, and we will occasionally use the second form for its phase response.

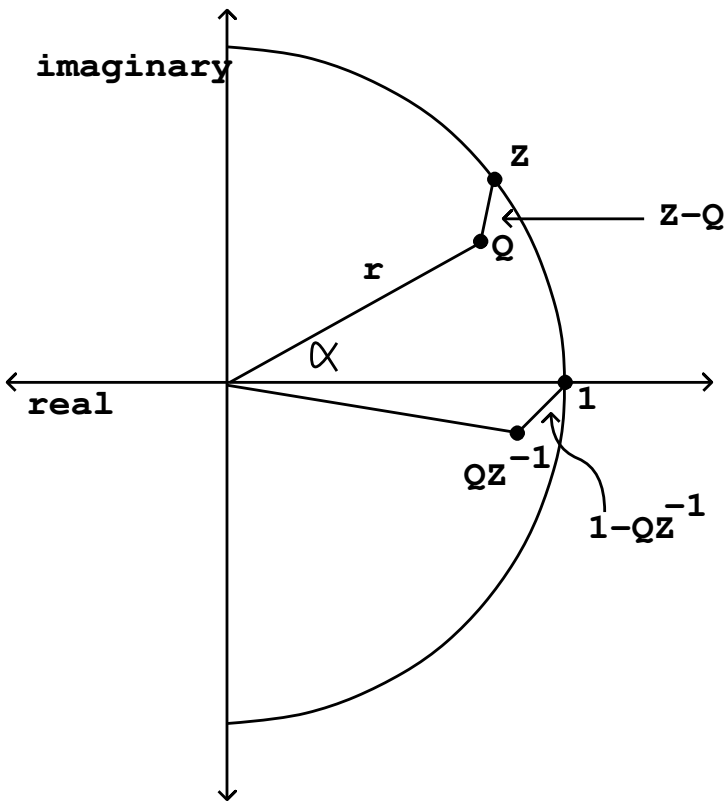


Figure 8.8: Diagram for calculating the frequency response of the non-recirculating elementary filter (Figure 8.7). The frequency response is given by the length of the segment connecting Z to A in the complex plane.

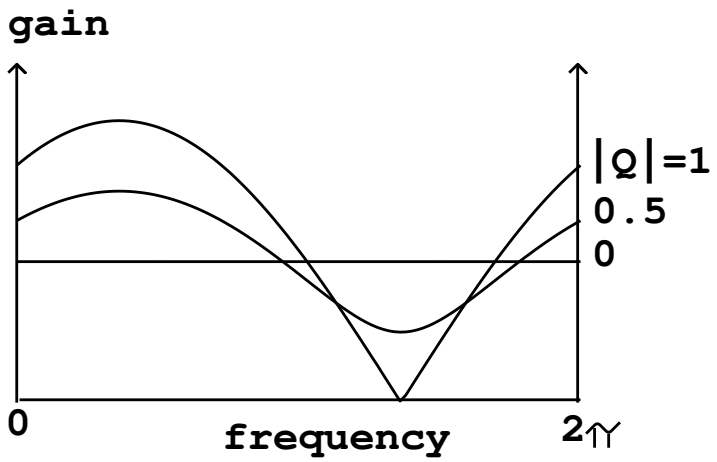


Figure 8.9: Frequency response of the elementary non-recirculating filter Figure 8.7. Three values of Q are used, all with the same argument (-2 radians), but with varying absolute value.

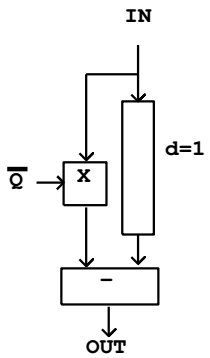


Figure 8.10: The elementary non-recirculating filter, second form.

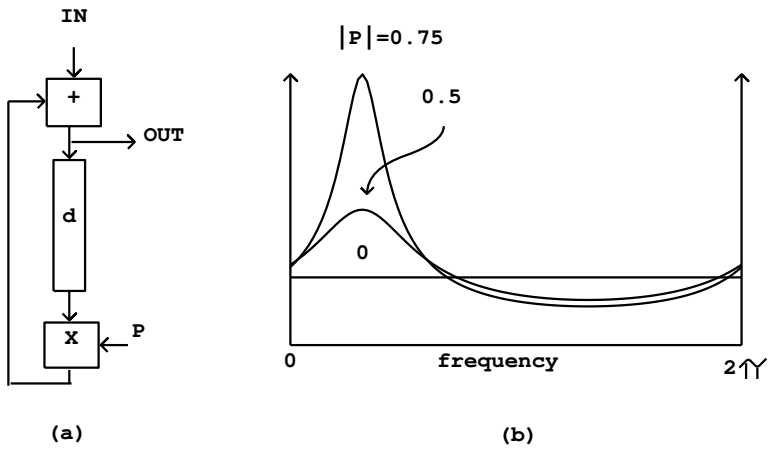


Figure 8.11: The elementary recirculating filter: (a) block diagram; (b) frequency response.

8.2.3 Elementary recirculating filter

The simplest recirculating filter is the recirculating comb filter of Figure 7.7 with a complex-valued feedback gain P as shown in Figure 8.11, part (a). By the same analysis as before, feeding this network a sinusoid whose nth sample is Z^n gives an output of:

$$\frac{1}{1 - PZ^{-1}} Z^n$$

so the transfer function is

$$H(Z) = \frac{1}{1 - PZ^{-1}}$$

The recirculating filter is stable when $|P| < 1$; when, instead, $|P| > 1$ the output grows exponentially as the delayed sample recirculates.

The transfer function is thus just the inverse of that of the non-recirculating filter (first form). If you put the two in series, using the same value of P, the output would theoretically be exactly equal to the input. (This analysis only demonstrated that for sinusoidal inputs; that it follows for other signals as well won't be evident until we have the background developed in chapter 9.)

8.2.4 Compound filters

We can use the recirculating and non-recirculating filters developed here to create a *compound filter*by putting several elementary ones in series. If the parameters of the non-recirculating ones (of the first type) are $Q_1, \ldots, Q_j$ and

those of the recirculating ones are $P_1, \ldots, P_k$, then putting them all in series, in any order, will give the transfer function:

$$H(Z) = \frac{(1 - Q_1 Z^{-1}) \cdots (1 - Q_j Z^{-1})}{(1 - P_1 Z^{-1}) \cdots (1 - P_k Z^{-1})}$$

The frequency response of the resulting compound filter is the product of those of the elementary ones. (One could also combine elementary filters by adding their outputs, or making more complicated networks of them; but for most purposes the series configuration is the easiest one to deal with.)

8.2.5 Real outputs from complex filters

In most applications, we start with a real-valued signal to filter and we need a real-valued output, but in general, a compound filter with a transfer function as above will give a complex-valued output. However, we can construct filters with non-real-valued coefficients which nonetheless give real-valued outputs, so that the analysis that we carry out using complex numbers can be used to predict, explain, and control real-valued output signals. We do this by pairing each elementary filter (with coefficient A, say) with another having as its coefficient the complex conjugate $\overline{A}$.

For example, putting two non-recirculating filters, with coefficients Q and $\overline{Q}$, in series gives a transfer function equal to:

$$H(Z) = (1 - QZ^{-1}) \cdot (1 - \overline{Q}Z^{-1})$$

which has the property that:

$$H(\overline{Z}) = \overline{H(Z)}$$

Now if we put any real-valued sinusoid:

$$X_n = 2\,\mathrm{re}(AZ^n) = AZ^n + \overline{AZ}^n$$

we get out:

$$AH(Z)Z^n + \overline{AH(Z)\overline{Z}}^n$$

Here we're using two properties of complex conjugates. First, you can add and multiply them at will:

$$\overline{A + B} = \overline{A} + \overline{B}$$

$$\overline{AB} = \overline{A}\,\overline{B}$$

and second, anything plus its complex conjugate is real, and is in fact twice its real part:

$$A + \overline{A} = 2\,\mathrm{re}(A)$$

The above result for two conjugate filters extends to any compound filter; in general, we always get a real-valued output from a real-valued input if we arrange

that each coefficient Q_i and P_i in the compound filter is either real-valued, or in a pair with its complex conjugate.

When pairing recirculating elementary filters, it is possible to avoid computing one of each pair, as long as the input is real-valued (and so, the output is as well.) Supposing the input is a real sinusoid of the form:

$$\left(AZ^n + \overline{A}Z^{-n}\right)$$

we apply a single recirculating filter with coefficient P. Letting $a[n]$ denote the real part of the output, we have:

$$\begin{aligned} a[n] &= 2\mathrm{re}\left[\frac{1}{1-PZ^{-1}}AZ^n + \frac{1}{1-PZ}\overline{A}Z^{-n}\right] \\ &= 2\mathrm{re}\left[\frac{1}{1-PZ^{-1}}AZ^n + \frac{1}{1-\overline{P}Z^{-1}}AZ^n\right] \\ &= 2\mathrm{re}\left[\frac{2-2\,\mathrm{re}(P)Z^{-1}}{(1-PZ^{-1})(1-\overline{P}Z^{-1})}AZ^n\right] \\ &= \mathrm{re}\left[\frac{2-2\,\mathrm{re}(P)Z^{-1}}{(1-PZ^{-1})(1-\overline{P}Z^{-1})}AZ^n + \frac{2-2\,\mathrm{re}(\overline{P})Z^{-1}}{(1-PZ^{-1})(1-\overline{P}Z^{-1})}\overline{A}Z^{-n}\right] \end{aligned}$$

(In the second step we used the fact that you can conjugate all or part of an expression, without changing the result, if you're just going to take the real part anyway. The fourth step did the same thing backward.) Comparing the input to the output, we see that the effect of passing a real signal through a complex one-pole filter, then taking the real part, is equivalent to passing the signal through a two-pole, one-zero filter with transfer function equal to:

$$H_{\mathrm{re}}(Z) = \frac{1-\mathrm{re}(P)Z^{-1}}{(1-PZ^{-1})(1-\overline{P}Z^{-1})}$$

With this definition, we can rewrite:

$$a[n] = H_{\mathrm{re}}(Z)AZ^n + H_{\mathrm{re}}(\overline{Z})AZ^{-n}$$

A similar calculation shows that taking the imaginary part gives the output:

$$b[n] = H_{\mathrm{im}}(Z)AZ^n + H_{\mathrm{im}}(\overline{Z})AZ^{-n}$$

with the transfer function:

$$H_{\mathrm{im}}(Z) = \frac{\mathrm{im}(P)Z^{-1}}{(1-PZ^{-1})(1-\overline{P}Z^{-1})}$$

So taking either the real or imaginary part of a one-pole filter output gives filters with two poles places at conjugates. We can combine the two in a particular way to give the simplest possible numerator of one:

$$H_{\mathrm{re}}(Z) + \frac{\mathrm{im}(P)}{\mathrm{re}(P)}H_{\mathrm{im}}(Z) = \frac{1}{(1-PZ^{-1})(1-\overline{P}Z^{-1})}$$

This is the transfer function for two conjugate recirculating filters in series, and so we have shown that we can just run the signal through one of the stages and combine the real and imaginary part to get the same result. This technique (called *partial fractions*) may be repeated for any number of stages in series, as long as we compute the appropriate combination of real and imaginary parts of the output of each stage to form the (real) input of the next stage. No similar shortcut seems to exist for non-recirculating filters; in that case it is necessary to compute each member of each complex-conjugate pair explicitly.

8.3 Designing filters

The frequency response of a series of elementary recirculating and non-recirculating filters can be estimated graphically by plotting all the coefficients $Q_1, \ldots, Q_j$ and $P_1, \ldots, P_k$ on the complex plane and reasoning as in Figure 8.8. The overall frequency response is the product of all the distances from the point Z to each of the Q_i, divided by the product of the distances to each of the P_i.

One customarily marks each of the Q_i with an "o" (this is called a "zero") and each of the P_i with an "x" (called a "pole"); their names are borrowed from the field of complex analysis. A plot showing the poles and zeros associated with a filter is unimaginatively called a *pole-zero plot*.

When Z is close to a zero the frequency response tends to dip, and when it is close to a pole, the frequency response tends to rise. The effect of a pole or a zero is more pronounced, and also more local, if it is close to the unit circle that Z is constrained to lie on. Poles must lie within the unit circle for a stable filter. Zeros may lie on or outside it, but any zero Q outside the unit circle may be replaced by one within it, at the point $1/\overline{Q}$, to give a constant multiple of the same frequency response. Except in special cases we will keep the zeros inside the circle as well as the poles.

In the rest of thie section we will show how to construct several of the filter types most widely used in electronic music. The theory of digital filter design is vast, and we will only give an introduction here. A deeper treatment is available online from Julius Smith at ccrma.stanford.edu. See also [Ste96] for a fuller treatment of filtering theory in the context and language of Digital Signal Processing.

8.3.1 One-pole low-pass filter

The one-pole low-pass filter has a single pole located at a positive real number p, as pictured in Figure 8.12. This is just a recirculating comb filter with delay length $d = 1$, and the analysis of section 7.4 applies. The maximum gain occurs at a frequency of zero, corresponding to the point on the circle closest to the point p. The gain there is $1/(1-p)$. If we move a distance of $1-p$ units up or down from the real (horizontal) axis, the distance increases by a factor of about $\sqrt{2}$, and so we expect the half-power point to occur at an angular frequency of about $1-p$.

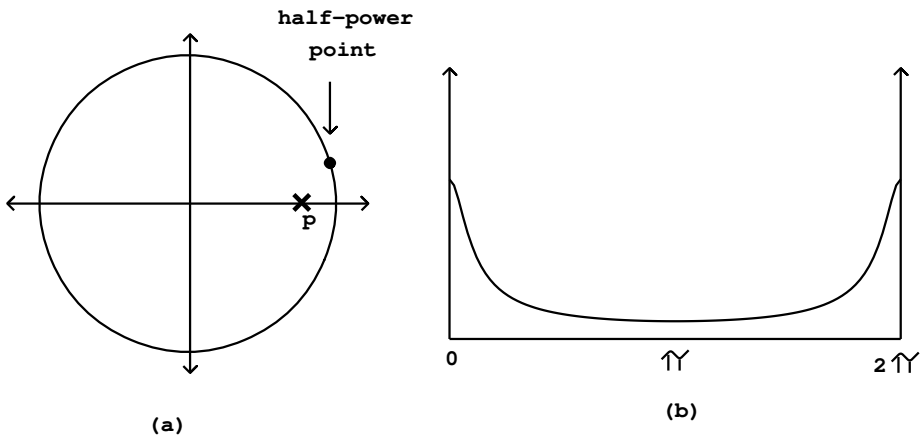


Figure 8.12: One-pole lowpass filter: (a) pole-zero diagram; (b) frequency response.

This calculation is often made in reverse: if we wish the half-power point to lie at a given angular frequency ω, we set $p = 1 - \omega$. This approximation only works well if the value of ω is well under $\pi/2$, as it usually is in practice.

It is customary to normalize the one-pole lowpass filter, multiplying it by the constant factor $1 - p$ in order to give a gain of 1 at zero frequency; nonzero freqencies will then get a gain less than one.

The frequency response is graphed in Figure 8.12 part (b). The audible frequencies only reach to the middle of the graph; the right-hand side of the frequency response curve all lies above the Nyquist frequency π.

The one-pole lowpass filter is often used to smooth noisy signals to seek slow-moving trends in them. For instance, if you use a physical controller and care about changes on the order of 1/10 second or so, you can smooth the control by using a low-pass filter whose half-power point is 20 or 30 cycles per second.

8.3.2 One-pole, one-zero high-pass filter

Sometimes an audio signal carries an unwanted constant offset, or in other words, a zero-frequency component. For example, the waveshaping spectra of Section 5.3 almost always contain a constant component. This is inaudible, but, since it specifes electrical power that is converted to heat in your speakers, its presence reduces the level of loudness you can reach without distortion. Another name for a constant signal component is "DC", meaning "direct current".

An easy and practical way to remove the zero-frequency component from an

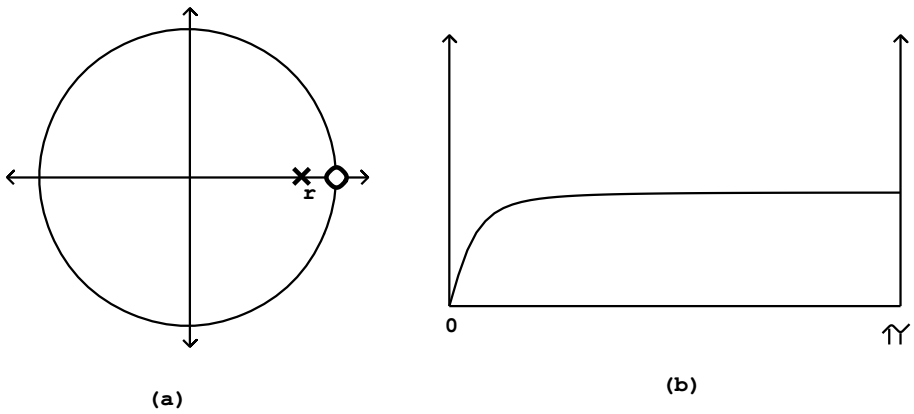


Figure 8.13: One-pole, one-zero high-pass filter: (a) pole-zero diagram; (b) frequency response (from zero to Nyquist frequency).

audio signal is to use a one-pole lowpass filter to extract it, and then subtract the result from the signal. The resulting transfer function is one minus the transfer function of the lowpass, giving:

$$H(Z) = 1 - \frac{1-p}{1-pZ^{-1}} = p\frac{1-Z^{-1}}{1-pZ^{-1}}$$

The factor of $1-p$ in the numerator of the low-pass transfer function is the normalization factor needed so that the gain is one at zero frequency.

By examining the right-hand side of the equation (comparing it to the general formula for compound filters), we see that there is still a pole at the real number p, and there is now also a zero at the point 1. The pole-zero plot is as shown in Figure 8.13 part (a), and the frequency response in part (b). (From here on, we will only plot frequency responses to the Nyquist frequency π; in the previous example we plotted it all the way up to the sample rate, 2π.)

8.3.3 Shelving filter

Generalizing the one-zero, one-pole filter above, supose we place the zero at a point q, a real number close to, but less than, one. The pole, at the point p, is similarly situated, and might be either greater than or less than q, i.e., to the right or left, respectively, but with both q and p within the unit circle. This situation is diagrammed in Figure 8.14.

At points of the circle far from p and q, the effects of the pole and the zero are nearly inverse (the distances to themare nearly equal), so the filter largely passes those frequencies unaltered. In the neighborhood of p and q, on the other hand, the filter will have a gain greater or less than one depending on which of

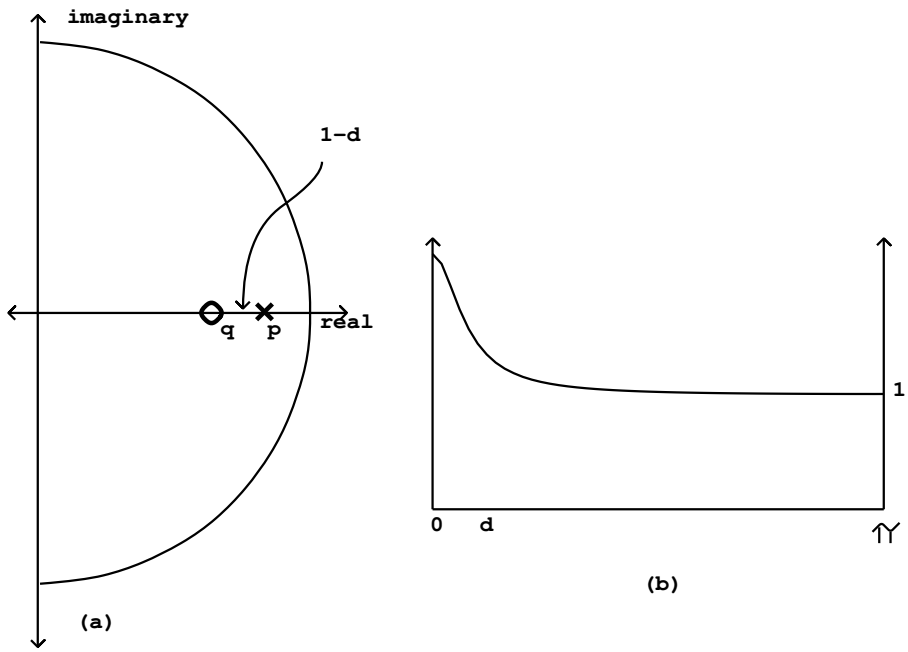


Figure 8.14: One-pole, one-zero shelving filter: (a) pole-zero diagram; (b) frequency response.

p or q is closer to the circle. This configuration therefore acts as a low-frequency shelving filter. (To make a high-frequency shelving filter we do the same thing, only placing p and q close to -1 instead of 1.)

To find the parameters of a desired shelving filter, start with a desired transition frequency ω (in angular units) and a desired low-frequency gain g. First we choose an average distance d, as pictured in the figure, from the pole and the zero to theedge of the circle. For small values of d, the region of influence (the crossover frequency) is about d radians.

Then put the pole at $p = 1 - d\sqrt{g}$ and the zero at $q = 1 - d/\sqrt{g}$. The gain at zero frequency is then

$$\frac{1-q}{1-p} = g$$

as desired. For example, in the figure, d is 0.25 radians and g is 2.

8.3.4 Band-pass filter

Taking the filter types shown above, which all have real-valued poles and zeros, we now transform them to operate on bands located off the real axis. The low-pass, high-pass, and shelving filters will then become band-pass, band-stop, and peaking filters. First we will develop the band-pass filter. Suppose we want a center frequency at ω radians, and a bandwidth of β. We take the low-pass filter with cutoff frequency β; its pole is located, for small calues of β, roughly at $p - 1 - \beta$. Now rotate this value by ω radians in the complex plane, by multiplying by the complex number $\cos\omega + i\sin\omega$. The new pole is at:

$$p_1 = (1-\beta)(\cos\omega + i\sin\omega)$$

To get a real-valued output, this must be paired with another pole:

$$p_2 = \overline{p_1} = (1-\beta)(\cos\omega - i\sin\omega)$$

The resulting pole-zero plot is as shown in Figure 8.15.

The peak is approximately (not exactly) at the desired center frequency ω and the 3-dB points approximately β radians above and below it. It is often desirable to normalize the filter to have a peak gain near unity; this is done by multiplying the input or output by the product of the distances of the two poles to the peak on the circle, or (very approximately):

$$\beta * (\beta + 2\omega)$$

For some applications it is desirable to add a zero at the points 1 and -1, so that the gain drops to zero at angular frequencies 0 and π.

8.3.5 Peaking and band-stop filter

In the same way, a peaking filter is obtained from a shelving filter by rotating the pole and the zero, and by providing a conjugate pole and zero, as shown in

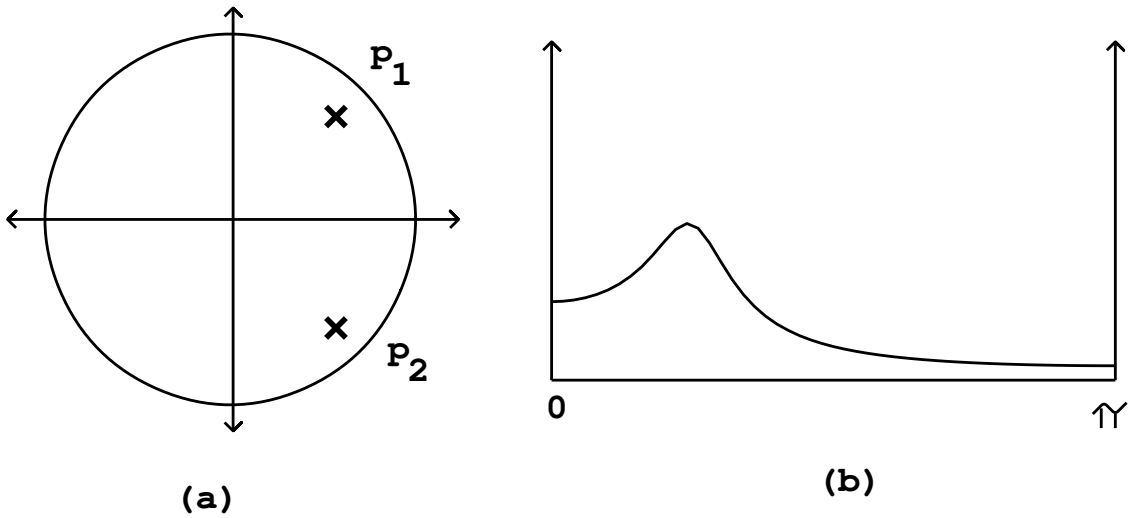


Figure 8.15: Two-pole band-pass filter: (a) pole-zero diagram; (b) frequency response.

Figure 8.16. If the desired center frequency is ω, and the radius of the pole and zero (as for the shelving filter) are p and q, then we place the the upper pole and zero at

$$p(\cos\omega + i\sin\omega),\ q(\cos\omega + i\sin\omega)$$

As a special case, placing the zero on the unit circle gives a band-stop filter; in this case the gain at the center frequency is zero. This is analogous to the one-pole, one-zero high-pass filter above.

8.3.6 Butterworth filters

A filter with one real pole and one real zero can be configured as a shelving filter, as a high-pass filter (putting the zero at the point 1) or as a low-pass filter (putting the zero at -1). The frequency responses of these filters are quite blunt; in other words, the transition regions are wide. It is often desirable to get a sharper filter, either shelving, low- or high-pass, whose two bands are flatter and separated by a narrower transition region.

A procedure borrowed from the analog filtering world transforms real, one-pole, one-zero filters to corresponding *Butterworth filters*, which have narrower transition regions. This procedure is described clearly and elegantly in the last chapter of [Ste96]. Since it involves passing from the discrete-time to the continuous-time domain, the derivation uses calculus; it also requires using notions of complex exponentiation and roots of unity which we are avoiding here.

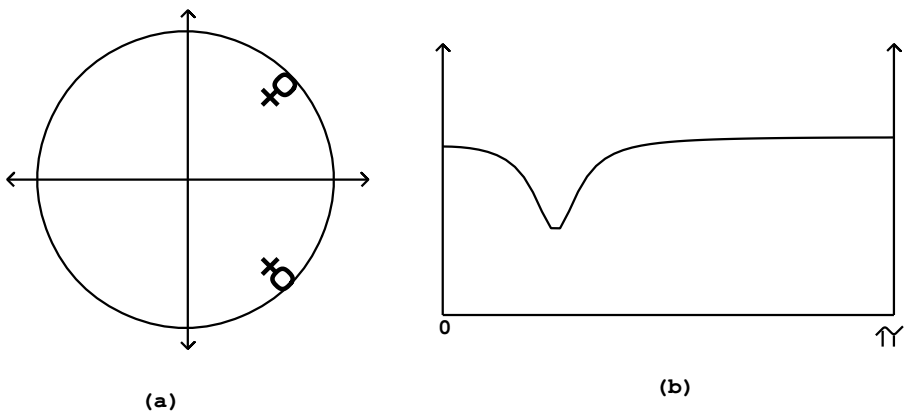


Figure 8.16: A peaking filter: (a) pole-zero diagram; (b) frequency response. Here the filter is set to attenuate by 6 decibels at the center frequency.

To make a Butterworth filter out of a high-pass, low-pass, or shelving filter, suppose that either the pole or the zero is given by the expression

$$\frac{1-r^2}{(1+r)^2}$$

where r is a parameter ranging from 1 to inf. If $r = 0$ this is the point 1, and if $r = \text{inf}$ it's -1.

Then, for reasons which will remain mysterious, we replace the point (whether pole or zero) by n points given by:

$$\frac{(1-r^2)-(2r\sin(\alpha))i}{1+r^2+2r\cos(\alpha))}$$

where α ranges over the values:

$$\frac{\pi}{2}(\frac{1}{n}-1), \frac{\pi}{2}(\frac{3}{n}-1), \ldots, \frac{\pi}{2}(\frac{2n-1}{n}-1)$$

In other words, α takes on n equally spaced angles between $-\pi/2$ and $\pi/2$. The points are arranged in the complex plane as shown in Figure 8.17. They lie on a circle through the original real-valued point, which cuts the unit circle at right angles.

A good choice for a nominal cutoff or shelving frequency defined by these circular collections of poles or zeros is simply the spot where the circle intersects the unit circle, corresponding to $\alpha = \pi/2$. This gives the point

$$\frac{(1-r^2)-2ri}{1+r^2}$$

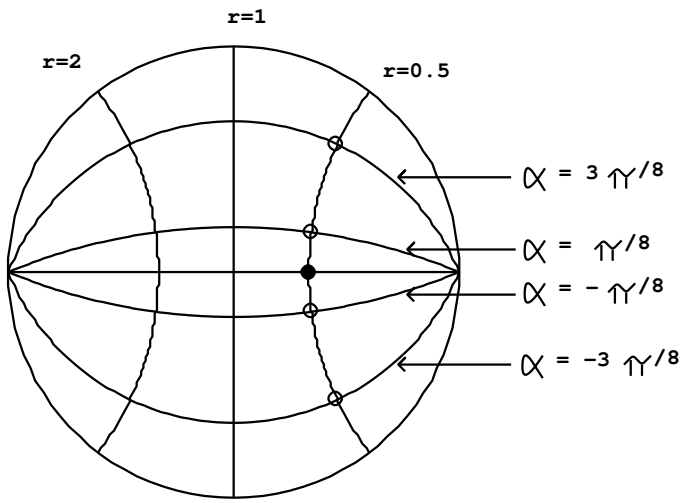


Figure 8.17: Replacing a real-valued pole or zero with an array as for a Butterworth filter. In this example we get four new poles or zeros as shown, lying along the circle where $r = 0.5$.

which, after some algebra, gives an angular frequency equal to

$$\beta = 2\arctan(r)$$

Figure 8.18, part (a), shows a pole-zero diagram and frequency response for a Butterworth low-pass filter with three poles and three zeros. Part (b) shows the frequency response of the low-pass filter and three other filters obtained by choosing different values of β (and hence r) for the zeros, while leaving the poles stationary. As the zeros progress from $\beta = \pi$ to $\beta = 0$, the filter, which starts as a low-pass filter, becomes a shelving filter and then a high-pass one.

8.3.7 Stretching the unit circle with rational functions

In section 8.3.4 we saw a simple way to turn a low-pass filter into a band-pass one. It is tempting to apply the same method to turn our Butterworth low-pass filter into a higher-quality band-pass filter; but to try to preserve the higher quality of the Butterworth filter we have to be more careful in the design of the transformation we use. In this section we will prepare the way to making the Butterworth band-pass filter by introducing the idea of rational transformations of the complex plane which preserve the unit circle.

This discussion is adapted from [PB87], pp. 201-206 (I'm grateful to Julius Smith for this pointer). There the tansformation is carried out in continuous time, but here we have adapted the method to operate in discrete time, in order to make teh discussion self-contained.

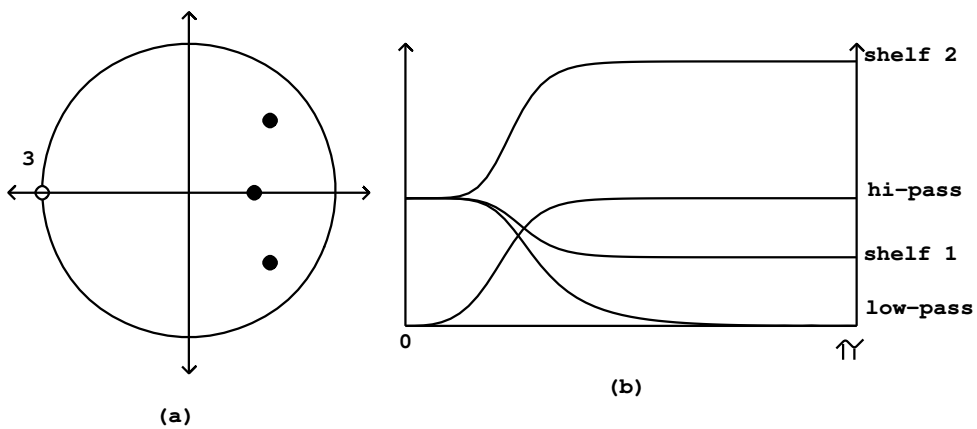


Figure 8.18: (a). Pole-zero plot for a Butterworth low-pass filter with three poles and three zeros. The poles are chosen for a cutoff frequency $\beta = \pi/4$. (b). Frequency responses for four filters with the same pole configuration, with different placements of zeros (but leaving the poles fixed). The low-pass filter results from setting $\beta = \pi$ for the zeros; the two shelving filters correspond to $\beta = 3\pi/10$ and $\beta = 2\pi/10$, and finally the high-pass filter is obtained setting $\beta = 0$. The high-pass filter is normalized for unit gain at the Nyquist frequency, and the others for unit gain at DC.

The idea is to start with any filter with a transfer function as before:

$$H(Z) = \frac{(1 - Q_1 Z^{-1}) \cdots (1 - Q_j Z^{-1})}{(1 - P_1 Z^{-1}) \cdots (1 - P_k Z^{-1})}$$

whose frequency response (the gain at a frequency ω) is given by:

$$|H(\cos(\omega) + i \sin(\omega))|$$

Now suppose we can find a rational function, $R(Z)$, which distorts the unit circle in some desirable way. For R to be a rational function means that it can be written as a quotient of two polynomials (so, for example, the transfer function H is a rational function). That R sends points on the unit circle to other points on the unit circle is just the condition that $|R(Z)| = 1$ whenever $Z = 1$. It can easily be checked that any function of the form,

$$\frac{R(Z) = \phi A_n Z^n + A_{n-1} Z^{n-1} + \cdots + A_0}{\overline{A_0} Z^n + \overline{A_1} Z^{n-1} + \cdots + \overline{A_n}}$$

where the leading factor ϕ has unit magnitude (i.e., $|\phi| = 1$), and as before we use $\overline{A}$ to denote the complex conjugate of a complex number A. The same reasoning as in section 8.2.2 confirms that $|R(Z)| = 1$ whenever $Z = 1$.

Once we have a suitable rational function R, we simply fabricate a new rational function,

$$J(Z) = H(R(Z))$$

The gain of the new filter J at the frequency ω is then equal to

$$|J(\cos(\omega) + i \sin(\omega))| = |H(\cos(\phi) + i \sin(\phi))|$$

where we choose ϕ so that:

$$\cos(\phi) + i \sin(\phi) = R(\cos(\omega) + i \sin(\omega))$$

For example, suppose we start with a one-zero, one-pole low-pass filter:

$$\frac{H(Z) = -1 - Z^{-1}}{g - Z^{-1}}$$

and apply the function

$$\frac{R(Z) = -Z^2 = -1 \cdot Z^2 + 0 \cdot Z + 0}{0 \cdot Z^2 + 0 \cdot Z + 1}$$

Geometrically, this choice of R stretches the unit circle uniformly to twice its circumference and wraps it around itself twice. The points 1 and -1 are both sent to the point -1, and the points i and $-i$ are sent to the point 1. The resulting transfer function is

$$J(Z) = \frac{-1 - Z^{-2}}{g - Z^{-2}} = -\frac{(1 - iZ^{-1})(-1 - iZ^{-1})}{(\sqrt{g} - Z^{-1})(-\sqrt{g} - Z^{-1})}$$

The pole-zero plots of H and J are shown in Figure 8.19. From a low-pass filter we ended up with a band-pass filter. The points i and $-i$ which R sends to 1 (where the original filter's gain is highest) become points of highest gain for the new filter.

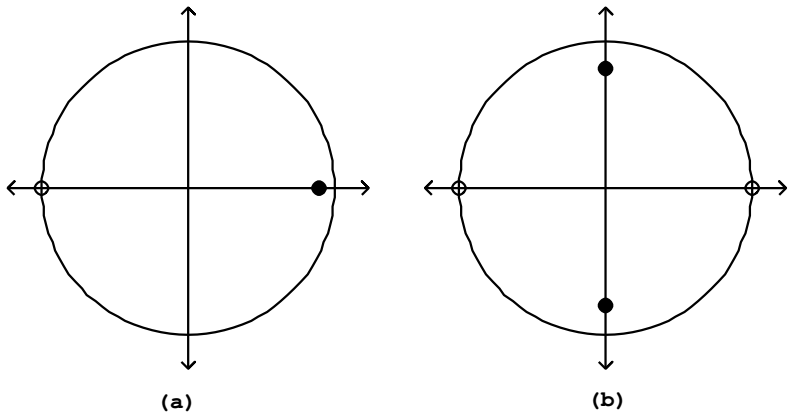


Figure 8.19: (a). A pole-zero plot for a one-pole, one-zero low-pass filter; (b). a plot for the resulting filter after the transformation $R(Z) = Z^2$. THe result is a band-pass filter with center frequency $\pi/2$.

8.3.8 Butterworth band-pass filter

We can apply the above to transform the Butterworth filter into a high-quality band-pass filter with center frequency $\pi/2$. A further transformation can then be applied to shift the center frequency to any desired value ω between 0 and π. The transformation will be of the form,

$$\frac{S(Z) = aZ + b}{bZ + a}$$

where a and b are real numbers and not both are zero. This is a particular case of the general form given above for unit-circle-preserving rational functions. We can check moreover that $S(1) - 1$ and $S(-1) = -1$, and that the top and bottom halves of the unit circle are transformed symmetrically; if Z goes to W then $\overline{Z}$ goes to $\overline{W}$. The qualitative effect of the transformation S is to slide points of the unit circle, non-uniformly, toward 1 or -1.

In particular, we wish to choose S so that:

$$S(\cos(\omega) + i\sin(\omega)) = i$$

If we do that, keep $R = -Z^2$ as before, and let H be the transfer function for a low-pass Butterworth filter, then the combined filter with transfer function $H(R(S(Z)))$ will be a band-pass filter with center frequency ω. Solving for a and b gives:

$$a = \cos(\frac{\pi}{4} - \frac{omega}{2}),\ b = \sin(\frac{\pi}{4} - \frac{omega}{2})$$

The new transfer function, $H(R(S(Z)))$, will have $2n$ poles and $2n$ zeros (if n is the degree of the Butterworth filter H).

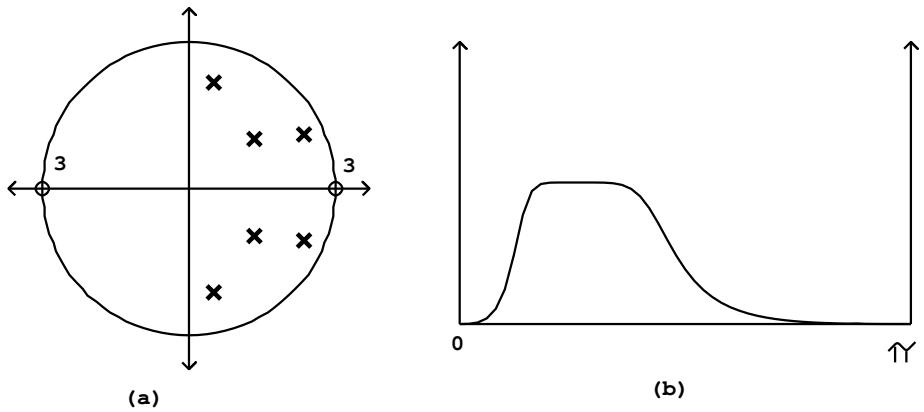


Figure 8.20: Butterworth band-pass filter: (a) pole-zero diagram; (b) frequency response. The center frequency is $\pi/4$. The bandwidth depends both on center frequency and on the bandwidth of the original Butterworth low-pass filter used.

Knowing the transfer function is good, but even better is knowing the locations of all the poles and zeros of the new filter, which we need to be able to compute it using elementary filters. If Z is a pole of the transfer function $J(Z) = H(R(S(Z)))$, that is, if $J(Z) = \inf$, then $R(S(Z))$ must be a pole of H. The same goes for zeros. To find a pole or zero of J we set $R(S(Z)) = W$, where W is a pole or zero of H, and solve for Z. This gives:

$$-\left[\frac{aZ+b}{bZ+a}\right]^2 = W$$

$$\frac{aZ+b}{bZ+a} = \pm\sqrt{-W}$$

$$Z = \frac{\pm a\sqrt{-W} - b}{\mp b\sqrt{-W} + a}$$

(Here a and b are as given above and we have used the fact that $a^2 + b^2 = 1$). A sample pole-zero plot and frequency response of J are shown in Figure 8.20.

8.3.9 Time-varying coefficients

In some recursive filter designs, changing the coefficients of the filter can inject energy into the system. A physical analogue is a child on a swing set. The child oscillates back and forth at the resonant frequency of the system, and pushing or pulling it injects or extracts energy smoothly. However, if you decide to shorten the chain or move the swingset itself, you may inject an unpredictable amount

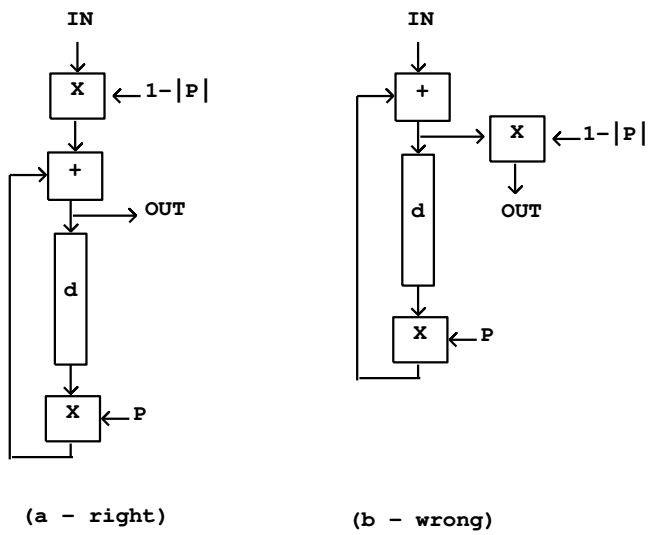


Figure 8.21: Normalizing a recirculating elementary filter: (a) correctly, by multiplying in the normalization factor at the input; (b) incorrectly, multiplying at the output.

of energy into the system. This corresponds to changing the coefficients in a resonant recirculating filter.

Happily, the simple one-zero and one-pole filters used here don't have this difficulty; if the feedback or feed-forward gain is changed smoothly (in the sense of an amplitude envelope) the output will behave smoothly as well.

The only subtlety arises when trying to normalize a recursive filter's output when the feedback gain is close to one. For example, suppose we have a one-pole low-pass filter with gain 0.99 (for a cutoff frequency of 0.02 radians, or 47 Hz. at the usual sample rate). To normalize this for unit DC gain we multiply by 0.01. Suppose now we wish to double the cutoff frequency by changing the gain imperceptibly to 0.98. This is fine except that the normalizing factor suddenly doubles. If we multiply the filter's output by the normalizing factor, the output will suddenly, although perhaps only momentarily, jump by a factor of two.

The trick is to normalize at the *input* of the filter, not the output. Figure 8.21 part (a) shows a complex recirculating filter, with feedback gain P, normalized at the input by $1 - |P|$ so that the peak gain is one. Part (b) shows the wrong way to do it, multiplying at the output.

Things get more complicated when several elementary recirculating filters are put in series, since the correct normalizing factor is in general a function of all the coefficients. One possible approach, if such a filter is required to change

rapidly, is to normalize each input separately as if it were acting alone, then multiplying the output, finally, by whaever further correction is needed.

8.3.10 Impulse responses of recirculating filters

In Section 7.4 we derived the impulse response of a recirculating comb filter, of which the one-pole low-pass filter is a special case. In Figure 8.22 we show the result for two low-pass filters and one complex one-pole resonant filter. All are elementary recirculating filters as introduced in section 8.2.3. Each is normalized to have unit maximum gain.

In the case of a low-pass filter, the impulse response gets longer (and lower) as the pole gets closer to one. Suppose the pole is at a point $1 - 1/n$ (so that the cutoff frequency is $1/n$ radians). The normalizing factor is also $1/n$. After n points, the output diminishes by a factor of

$$\left(1 - \frac{1}{n}\right)^n \approx \frac{1}{e}$$

where e is Euler's constant, about 2.718. The filter can be said to have a *settling time* of n samples. In the figure, $n = 5$ for part (a) and $n = 10$ for part (b). In general, the settling time (in samples) is approximately one over the cutoff frequency (in angular units).

The situation gets more interesting when we look at a resonant one-pole filter, that is, one whose pole lies off the real axis. In part (c) of the figure, the pole P has absolute value 0.9 (as in part (b)), but its argument is set to $2\pi/10$ radians. We get the same settling time as in part (b), but the output rings at the resonant frequency (and so at a period of 10 samples in this example).

A natural question to ask is, how many periods of ringing do we get before the filter decays to strength $1/e$? If the pole of a resonant has modulus $1 - 1/n$ as above, we have seen in section 8.2.3 that the bandwidth (call it b) is about $1/n$, and we have seen here that the settling time is about n. The resonant frequency (call it ω) is the argument of the pole, and the period in samples is $2\pi/\omega$. The number of periods that make up the settling time is thus:

$$\frac{n}{2\pi/\omega} = \frac{1}{2\pi}\frac{\omega}{b} = \frac{q}{2\pi}$$

where q is the *quality* of the filter, defined as the center frequency divided by bandwidth. Resonant filters are often specified in terms of the center frequency and "q", instead of bandwidth.

8.3.11 All-pass filters

Sometimes a filter is applied to get a desired phase change, rather than to alter the amplitudes of the frequency components of a sound. In this situation we would need a way to design a filter with a constant, unit frequency response but which changes the phase of an incoming sinusoid in a way that depends

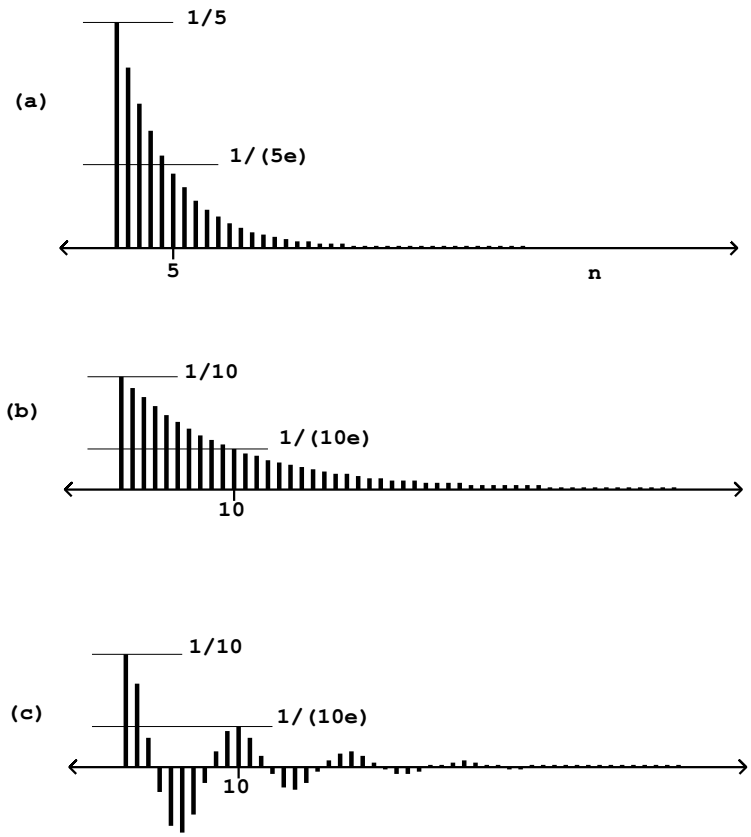


Figure 8.22: The impulse response of three elementary recirculating (one-pole) filters, normalized for peak gain 1: (a) low-pass with $P = 0.8$; (b) low-pass with $P = 0.9$; (c) band-pass, with $|P| = 0.9$ and a center frequency of $2\pi/10$.

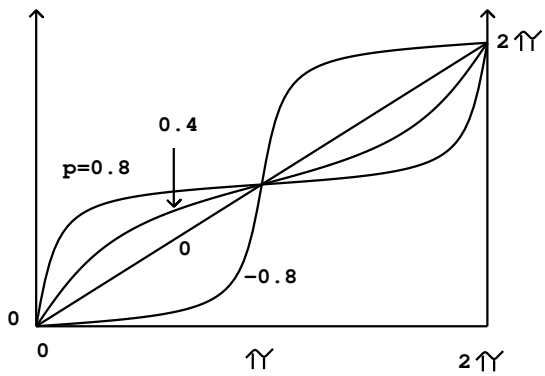


Figure 8.23: Phase response of all-pass filters with different pole locations p. When the pole is located at zero, the filter reduces to a one-sample delay.

on its frequency. We have already seen in Chapter 7 that a delay of length d introduces a phase change of $-d\omega$, at the angular frequency ω. Another class of filters, called *all-pass filters*, can make phase changes which are more widely variable functions of ω.

To design an all-pass filter, we start with two facts: first, an elementary recirculating filter and an elementary non-recirculating one cancel each other out perfectly if they have the same gain coefficient. In other words, if a signal has been put through a one-zero filter, either real or complex, the effect can be reversed by sequentially applying a one-pole filter, and vice versa.

The second fact is that the elementary non-recirculating filter of the second form has the same frequency response as that of the first form; they differ only in phase response. So if we combine an elementary recirculating filter with an elementary non-recirculating one of the second form, the frequency responses cancel out (to a flat gain independent of frequency) but the phase response is not constant.

To find the transfer function, we choose the same complex number $P < 1$ as coefficient for both elementary filters and multiply their transfer functions:

$$\frac{H(Z) = 1 - \overline{P}Z^{-1}}{P - Z^{-1}}$$

The coefficient P controls both the location of the one pole (at P itself) and the zero (at $1/P$). Figure 8.23 shows the phase response of the all-pass filter for four real-valued choices p of the coefficient. At frequencies of 0, π, and 2π, the phase response is just that of a one-sample delay; but for frequencies in between, the phase response is bent upward or downward depending on the coefficient.

Complex coefficients give similar phase response curves, but the frequencies

at which they cross the diagonal line in the figure are shifted according to the argument of the coefficient P.

8.4 Applications

Filters have a broad range of applications both in audio engineering and in electronic music. The former include, for instance, equalizers, speaker crossovers, sample rate converters, and DC removal (which we have already used in earlier chapters). Here, though, we'll be concerned with the latter.

8.4.1 Subtractive synthesis

)

Subtractive synthesis refers to using filters to shape the spectral envelope of a sound, forming another sound, usually preserving qualities of the original sound such as pitch, roughness, noisiness, or graniness. The spectral envelope of the resulting sound is the product of the spectral envelope of the original sound with the frequency response of the filter. Figure 8.24 shows a possible configuration of source, filter, and result.

The filter may be constant or time-varying. Already in wide use by the mid 1950s, subtractive synthesis reached a new height of popularity with the introduction of the voltage-controlled filter (VCF), which became widely available in the mid 1960s with the appearance of modular synthesizers. The VCF has two inputs: one for the sound to filter, and one to vary the center or cutoff frequency of the filter.

A popular use of a VCF is to control the center frequency of a resonant filter with the same ADSR generator that controls the amplitude; a possible block diagram is shown in Figure 8.25. In this configuration, the louder portion of a note (loudness roughly controlled by the multiplier at the bottom) may also be made to sound brighter, using the filter, than the quieter parts; this can mimic the spectral evolution of strings or brass instruments over the life of a note.

8.4.2 Envelope following

It is frequently desirable to use the time-varying power of an incoming signal to trigger or control a musical process. To do this, we will need a procedure for measuring the power of an audio signal. Since most audio signals pass through zero many times per second, it won't suffice to take the absolute value of the signal as a measure if its power, instead, we must make an average of the power over an interval of time long enough that its oscillations won't whow up in the power estimate, but short enough that changes in signal level are quickly reflected in the power estimate. A computation that provides a timely, time-varying power estimate is called an *envelope follower*.

The output of a low-pass filter can be viewed as a moving average of its input. For example, suppose we apply a normalized one-pole low-pass filter (as

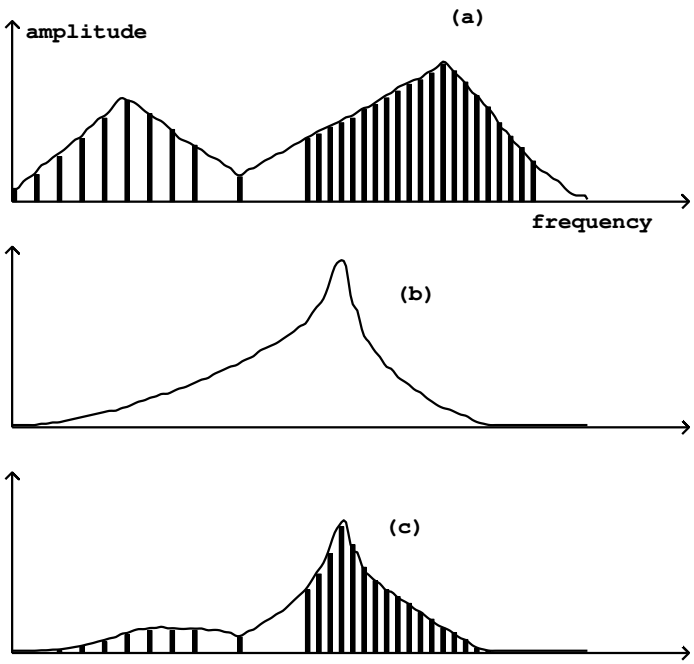


Figure 8.24: Subtractive synthesis: (a) spectrum of input sound; (b) filter frequency response; (c) spectrum of output sound.

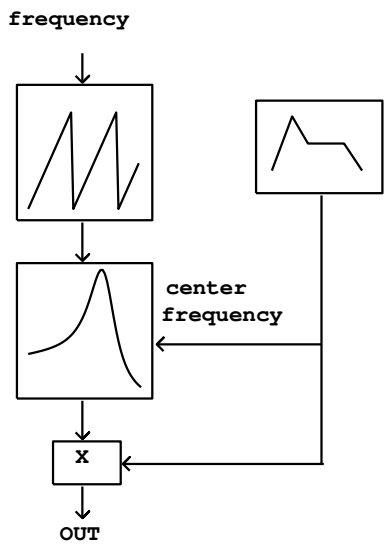


Figure 8.25: ADSR-controlled subtractive synthesis.

in Figure 8.21) to an incoming signal $x[n]$. The output (call it y[n]) is the sum of the delay output times p (real-valued for a low-pass filter), with $1-p$ times the input:

$$y[n] = p \cdot y[n-1] + (1-p) \cdot x[n]$$

so each input is averaged, with weight $1-p$, into the previous output to produce a new output. So we can make a moving average of the square of an audio signal using the diagram of Figure 8.26. The output is a time-varying average of the instantaneous power $x[n]^2$, and the design of the low-pass filter controls, among other things, the settling time of the moving average.

For more insight into the design of a suitable low-pass filter for an envelope follower, we analyze it from the point of view of signal spectra. If, for instance, we put in a real-valued sinusoid:

$$x[n] = a \cdot \cos(\alpha n)$$

the result of squaring is:

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

and so if the low-pass filter effectively stops the component of frequency 2α we will get out approximately the constant $a^2/2$, which is indeed the average power.

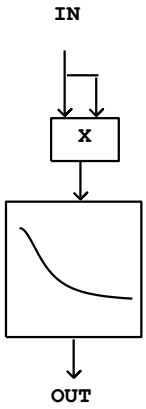


Figure 8.26: Envelope follower. The output is the average power of the input signal.

The situation for a signal with several components is similar. Suppose the input signal is now,

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

whose spectrum is plotted in Figure 8.27, part (a). (Here we have omitted the two phase terms but they will have no effect on the outcome.) Squaring the signal produces the spectrum shown in part (b). (See section 5.2.) We can get the desired fixed value of $(a^2 + b^2)/2$ simply by filtering out all the other components; ideally the result will be a constant (DC) signal. The lowest frequency we need to filter out is the minimum difference between two partials in the original signal.

Envelope followers may also be used on noisy signals, which may be thought of as signals with dense spectra. In this situation there will be difference frequencies arbitrarily close to zero frequency, and filtering them out entirely will be impossible; we will always get fluctuations in the output, but they will decrease proportionally as the filter's pass band is narrowed.

Although a narrower bass band will always give a cleaner output, whether for discrete or continuous spectra, the filter's settling time will lengthen proportionally as the bass band is narrowed. There is thus a tradeoff between getting a quick response and a smooth result.

8.4.3 Single Sideband Modulation

As we saw in Chapter 5, multiplying two real sinusoids together results in a signal with two new components at the sum and difference of the original frequencies. If we carry out the same operation with complex sinusoids, we get only one new resultant frequency; this is one result of the greater mathematical simplicity

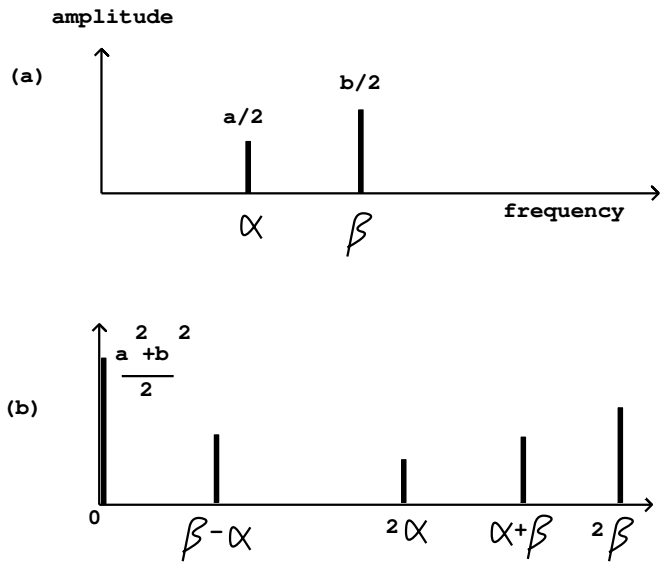


Figure 8.27: Envelope following from the spectral point of view: (a). an incoming signal with two components; (b). the result of squaring it.

of complex sinusoids as compared to real ones. If we multiply a complex sinusoid $1, Z, Z^2, \ldots$ with another one, $1, W, W^2, \ldots$ the result is $1, WZ, (WZ)^2, \ldots$, which is another complex sinusoid whose frequency, $\angle(ZW)$, is the sum of the two original frequencies.

In general, since complex sinusoids have simpler properties than real ones, is often useful to be able to convert from real sinusoids to complex ones. In other words, from a real sinusoid:

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

(with a spectral peak of amplitude a and frequency ω) we would like a way of computing a complex sinusoid:

$$X[n] = a\,(\cos(\omega n) + i\sin(\omega n))$$

so that

$$x[n] = \mathrm{Re}(X[n]).$$

We would like a linear process for doing this, so that superpositions of sinusoids get treated as if their components were dealy with separately.

Of course we could equally well have chosen the complex sinusoid with frequency $-\omega$:

$$X'[n] = a\,(\cos(\omega n) - i\sin(\omega n))$$

and in fact $x[n]$ is just half the sum of the two. In essence we need a filter that will pass through positive frequencies (actually frequencies between 0 and π,

corresponding to values of Z on the top half of the complex unit circle) from negative values (from $-\pi$ to 0, or equivalently, from π to 2π—the bottom half of the unit circle).

One can design such a filter by designing a low-pass filter with cutoff frequency $\pi/2$, and then performing a rotation by $\pi/2$ radians using the technique of section 8.3.4. However, it turns out to be easier to do it using two specially designed networks of all-pass filters with real coefficients.

Calling the transfer functions of the two filters H_1 and H_2, we design the filters so that

$$\angle(H_1(Z)) - \angle(H_2(Z)) \approx \begin{cases} \pi/2 & 0 < \angle(Z) < \pi \\ -\pi/2 & -\pi < \angle(Z) < 0 \end{cases}$$

or in other words,

$$H_1(Z) \approx iH_2(Z),\ 0 < \angle(Z) < \pi$$

$$H_1(Z) \approx -iH_2(Z),\ -\pi < \angle(Z) < 0$$

Then for any incoming real-valued signal $x[n]$ we simply form a complex number $a[n] + ib[n]$ where $a[n]$ is the output of the first filter and $b[n]$ is the output of the second. Any complex sinusoidal component of $x[n]$ (call it Z^n) will be transformed to

$$H_1 Z + iH_2(Z) \approx \begin{cases} 2H_1 Z & 0 < \angle(Z) < \pi \\ 0 & \text{otherwise} \end{cases}$$

Having started with a real-valued signal, whose energy is split equally into positive and negative frequencies, we end up with a complex-valued one with only positive frequencies.

8.5 Examples

In this section we will first introduce some easy-to-use prefabricated filters available in Pd to develop examples showing the three applications from the previous section. Then we will show some more sophisticated applications that require explicitly designed filters.

8.5.1 Prefabricated low-, high-, and band-pass filters

Patches H01.low-pass.pd, H02.high-pass.pd, and H03.band-pass.pd (Figure 8.28) show Pd's built-in filters, which implement filter designs described in sections 8.3.1, 8.3.2 and 8.1.2. Two of the patches also use a noise generator we have not introduced before; so we now introduce four new Pd objects:

`lop~`: One-pole low-pass filter. The left inlet takes a signal to be filtered, and the right inlet takes control messages to set the cutoff frequency of the filter. The filter is normalized so that the gain is one at frequency 0.

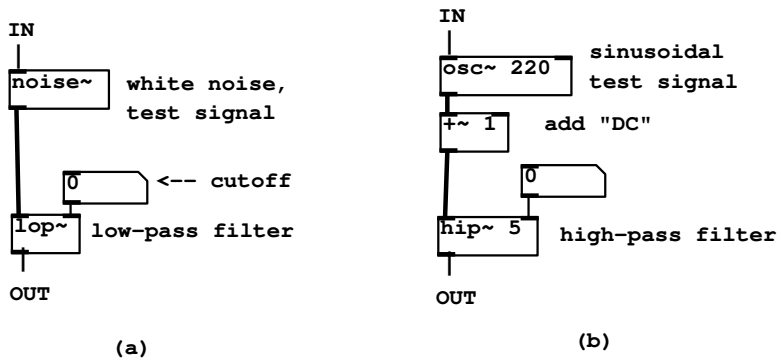


Figure 8.28: Using prefabricated filters in Pd: (a). a low-pass filter, with white noise as a test input; (b). using a high-pass filter to remove a signal component of frequency 0.

`hip~`: One-pole, one-zero high-pass filter, with the same inputs and outputs as `lop~`, normalized to have a gain of one at the Nyquist frequency.

`bp~`: Resonant filter. The middle inlet takes control messages to set the center qrequency, and the right inlet to set "q".

`noise~`: Outputs white noise. Each sample is an independent pseudo-random number, uniformly distributed from -1 to 1.

As shown in Figure 8.28, the first three example patches demonstrate these three filters. The `lop~` and `bp~` objects are demonstrated with noise as input; `hip~` as shown is used to remove the DC (zero frequency) component of a signal.

8.5.2 Prefabricated time-variable band-pass filter

Time-varying band-pass filtering, as often used in classical subtractive synthesis (section 8.4.1), can be done using the `vcf~` object, introduced here.

`vcf~` ("voltage controlled filter"): a band-pass filter, similar to `bp~`, but with a signal inlet to control center frequency. Both `bp~` and `vcf~` are one-pole resonant filters as developed in section 8.1.2; `bp~` outputs only the real part of the resulting signal, while `vcf~` outputs the real and imaginary parts separately.

Patch H04.filter.sweep.pd ()Figure 8.29) demonstrates using the `vcf~` object for a simple and characteristic subtractive synthesis task. A `phasor~` object (at top) creates a sawtooth wave to filter. (This is not especially good practice as we are not controlling the possibility of foldover; a better sawtooth generator for this purpose will be developed in Chapter 10.) The second `phasor~` object (labeled "LFO for sweep") controls the time-varying center frequency. After

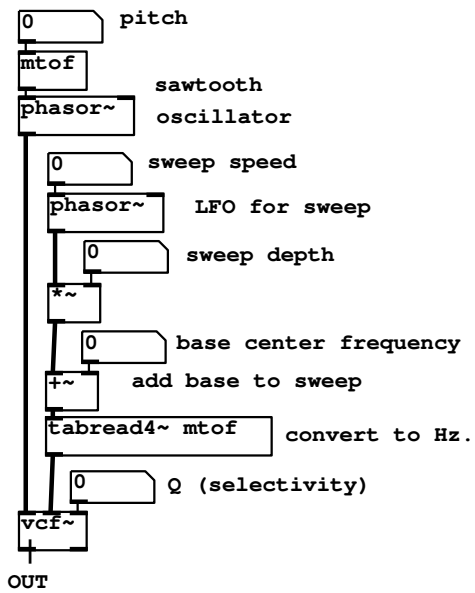


Figure 8.29: The `vcf~` band-pass filter, with its center frequency controlled by an audio signal (as compared to `bp~` which takes only control messages to set its center frequency.

adjusting to set the depth and a base center frequency (given in MIDI units), the result is converted into Hertz (using the `tabread4~` object) and passed to `vcf~` to set its center frequency. Another example of using a `vcf~` object for subtractive synthesis is demonstrated in Patch H05.filter.floyd.pd.

8.5.3 Envelope followers

Patch H06.envelope.follower.pd shows a simple realization of the envelope folower described in section 8.4.2; it is self-explanatory. An interesting application of envelope following is shown in Patch H07.measure.spectrum.pd (Figure 8.30, part (a). A famous bell sample is looped as a test sound. Rather than get the overall mean square power of the bell, we would like to estimate the frequency and power of each of its partials. To do this we sweep a band-pass filter up and down in frequency, listening to the result and/or watching the filter's output power using an envelope follower. (We use two band-pass filters in series for better isolation of the partials; this is not especially good filter design but will do in this context.) When the filter is tuned to a partial the envelope follower reports its strength.

Patch H08.heterodyning.pd (part (b) of the figure) shows an alternative way of finding partial strengths of an incoming sound; it has the advantage of reporting the phase as well as the strength. First we modulate the desired partial down to zero frequency. We use a complex-valued sinusoid as a modulator so that we get only one sideband for each component of the input. The test frequency is the only frequency that is modulated to DC; others go elsewhere. We then low-pass the resulting complex signal. (We can use a real-valued low-pass filter separately on the real and imaginary parts.) This essentially removes all the partials except for the DC one, which we then harvest. This technique will be the basis of Fourier analysis, the subject of chapter 9.

8.5.4 Single sideband modulation

As described in section 8.4.3, a pair of all-pass filters can be constructed to give roughly $\pi/2$ phase difference for positive frequencies and $-\pi/2$ for negative ones. The design of these pairs is beyond the scope of this discussion (see, for instance, [Reg93]), but Pd does provide an abstraction, `hilbert~`, to do this. Patch H09.ssb.modulation.pd, shown in Figure 8.31, demonstrates how to use the `hilbert~` abstraction to do signal sideband modulation.

The two outputs of `hilbert~`, considered as the real and imaginary parts of a complex-valued signal, are multiplied by a complex sinusoid (at right in the figure), and the real part is output. The components of the resulting signal are those of the input plus the (positive or negative) frequency of the number box.

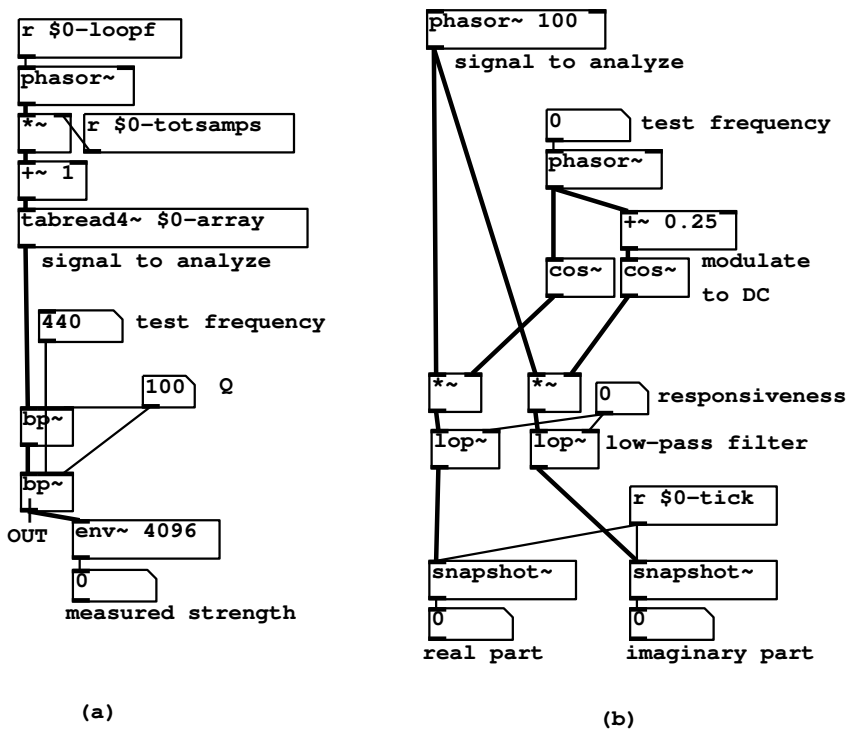


Figure 8.30: Analyzing the spectrum of a sound: (a) band-pass filtering a sampled bell sound and envelope-following the result; (b) frequency-shifting a partial to DC and reading off its real and imaginary part.

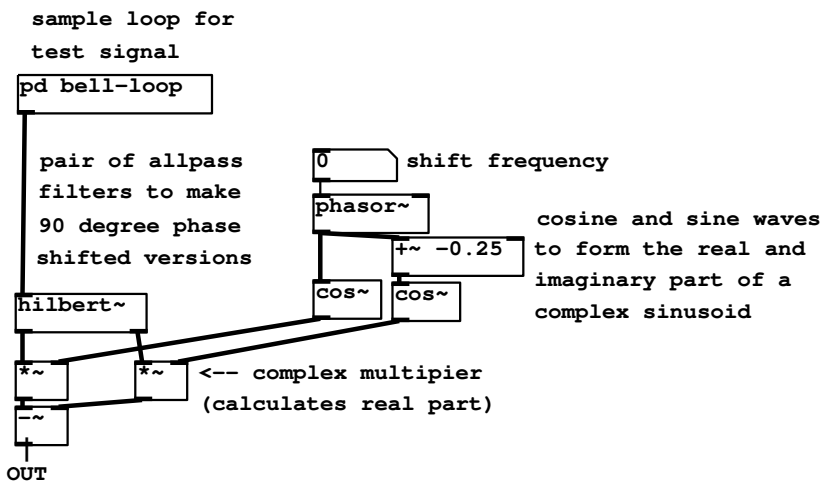


Figure 8.31: Using an all-pass filter network to make a frequency shifter.

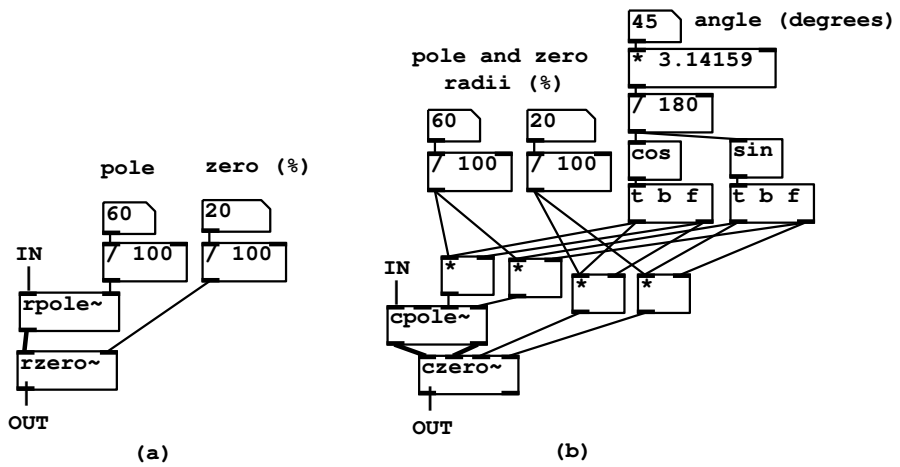


Figure 8.32: Building filters from elementary, raw ones: a. shelving; b. peaking.

8.5.5 Using elementary filters directly: shelving and peaking

No finite set of prefabricated filters could fill every possible need, and so Pd provides the elementary filters of sections 8.2.1-8.2.3 in raw form, so that the user supplies the filter coefficients explicitly. In this section we will describe patches that realize the shelving and peaking filters of sections 8.3.3 and 8.3.5 directly from elementary filters. First we introduce the six Pd objects that realize elementary filters:

`rzero~`, `rzero_rev~`, `rpole~`: elementary filters with real-valued coefficients operating on real-valued signals. The three implement non-recirculating filters of the first and secton types, and the recirculating filter. They all have one inlet, at right, to supply the coefficient. For `rzero~` and `rpole~` this coefficent gives the location of the zero or pole. The inlet for the coefficient (as well as the left inlet for the signal to filter) take audio signals. No stability check is performed.

`czero~`, `czero_rev~`, `cpole~`: elementary filters with complex-valued coefficients, operating on complex-valued signals, corresponding to the real-valued ones above. Instead of two inlets and one outlet, each of these filters has four inlets (real and imaginary part of the signal to filter, and real and imaginary part of the coefficient) and two outlets for the complex-valued output of the filter.

The example patches use a pair of abstractions to graph the frequency and phase responses of filters as explained in Patch H10.measurement.pd. Patch H11.shelving.pd (Figure 8.32, part (a)) shows how to make a shelving filter. One elementary non-recirculating filter (`rzero~`) and one elementary recirculating one (`rpole~`) are put in series. As Section 8.3.9 suggests, the `rzero~` object is placed first.

Patch H12.peaking.pd shows the peaking filter (part (b) of the figure). Here the pole and the zero are rotated by an angle ω to control the center frequency of the filter. The bandwidth and center frequency gain are equal to the shelf frequency and the DC gain of the corresponding shelving filter.

Patch H13.butterworth.pd demonstrates a three-pole, three-zero Butterwirth shelving filter. The filter itself is an abstraction, `butterworth3~`, for easy re-use.

8.5.6 Making and using all-pass filters

Patch H14.all.pass.pd(Figure 8.33 part a) shows how to make an all-pass filter out of elementary filters, in this case a non-recirculating filter, second form (`rzero_rev~`) and a recirculating filter (`rpole~`). The coefficient, ranging from -1 to 1, is controlled in hundredths.

Patch H15.phaser.pd(part b of the figure) shows how to use four all-pass filters to make a classic phaser. The phaser works by summing the input signal

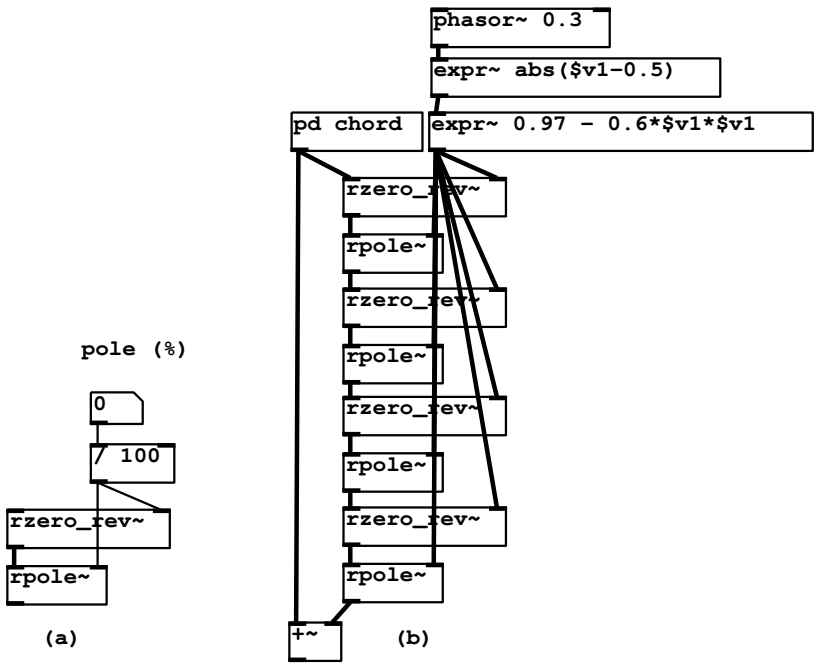


Figure 8.33: All-pass filters. (a). making an all-pass filter from elementary filters; b. using four all-pass filters to build a phaser.

with a phase-altered version of it, making interference effects. The amount of phase change is varied in time by varying the (shared) coefficient of the all-pass filters. The overall effect is somewhat similar to a flanger (time-varying comb filter) but the phaser not impose a pitch as the comb filter does.

Exercises

1. A recirculating elementary filter has a pole at $i/2$. At what angular frequency is its gain greatest, and what is the gain there? At what angular frequency is the gain least, and what is the gain there?

2. A shelving filter has a pole at 0.9 and a zero at 0.8. What are: the DC gain; the gain at Nyquist; the approximate transition frequency?

3. Suppose a complex recirculating filter has a pole at P. Suppose further that you want to combine its real and imaginary output to make a single, real-valued signal equivalent to a two-pole filter with poles at P and $\overline{P}$. How would you weight the two outputs?

Chapter 9

Fourier analysis and resynthesis

Among the applications of filters discussed in chapter 8, we saw how to use heterodyning, combined with a low-pass filter, to find the amplitude and phase of a sinusoidal component of a signal (section 8.5.3). In this chapter we will refine this technique into what is called *Fourier analysis*. In its simplest form, Fourier analysis takes as input any periodic signal (of period N) and outputs the complex-valued amplitudes of its N possible sinusoidal components. These N complex amplitudes can theoretically be used to reconstruct the original signal exactly. This reconstruction is called *Fourier resynthesis*.

In this chapter we will start by developing the theory of Fourier analysis and resynthesis of periodic sampled signals. Then we will go on to show how to apply the same techniques to arbitrary signals, whether periodic or not. Finally, we will develop some standard applications such as the phase vocoder.

9.1 Fourier analysis of periodic signals

Suppose $X[n]$ is a complex-valued signal with period N, a positive integer. (We are using complex-valued signals rather than real-valued ones because the mathematics will turn out simpler.) The values of $X[n]$ for $n = 0, \ldots, N-1$ (one period) determine $X[n]$ for all integer values of n.

Suppose further that $X[n]$ can be written as a sum of sinusoids of frequency 0, $2\pi/N$, $4\pi/N$, ..., $2(N-1)\pi/N$. These are the partials, starting with the zeroth, for a signal of period N. We stop after the Nth partial because the next one would have frequency 2π, equivalent to frequency 0, which is already on the list.

Given the values of X, we wish to find the complex amplitudes of the partials. Suppose we want the kth partial, where $0 \leq k < N$. The frequency of the partial is $\omega = 2\pi k/N$. We can get its amplitude by modulating X downward ω radians per sample in frequency, so that the kth partial is modulated to frequency

zero. Then we pass the signal through a low-pass filter with such a low cutoff frequency that nothing but the zero-frequency partial remains. Such a filter will essentially average the N samples of its periodic input. In summary, to measure a sinusoidal component of a periodic signal, modulate it down to DC and then average over one period.

Let U be the unit-magnitude complex number with argument $2\pi/N$ (the fundamental frequency of a signal with period N):

$$U = \cos(\omega) + i\sin(\omega)$$

The kth partial of the signal $X[n]$ is of the form:

$$P_k[n] = A_k \left[U^k\right]^n$$

where A_K is the complex amplitude of the partial, and the frequency of the partial is:

$$\angle(U^k) = k\angle(U) = k\omega$$

We're assuming for the moment that the signal $X[n]$ can actually be written as a sum of the n partials, or in other words:

$$X[n] = P_0[n] + P_1[n] + \cdots + P_{N-1}[n]$$

$$= A_0\left[U^0\right]^n + A_1\left[U^1\right]^n + \cdots + A_{N-1}\left[U^{N-1}\right]^n$$

By the heterodyne-filtering argument above, we expect to be able to measure each A_k by multiplying by the sinusoid of frequency $-k\omega$ and averaging over a period:

$$A_k = \frac{1}{N}\left(\left[U^{-k}\right]^0 X[0] + \left[U^{-k}\right]^1 X[1] + \cdots + \left[U^{-k}\right]^{N-1} X[N-1]\right)$$

This is such a useful formula that it gets its own notation. The *Fourier transform* of a signal $X[n]$, over N samples, is defined as:

$$\mathcal{FT}\left\{X[n]\right\}(k) = V^0 X[0] + V^1 X[1] + \cdots + V^{N-1} X[N-1]$$

where $V = U^{-k}$. The Fourier transform is a function of the variable k. So far k has taken integer values but the formula makes sense for any value of k if we define V more carefully as:

$$V = \cos(-k\omega) + i\sin(-k\omega)$$

where, as defined above, $\omega = 2\pi/N$ is the angular frequency associated with the period N.

9.1.1 Fourier transform as additive synthesis

Now consider an arbitrary signal $X[n]$ that repeats every N samples. (Previously we had assumed that $X[n]$ could be obtained as a sum of sinusoids, but we haven't yet found out whether any periodic $X[n]$ can be obtained that way.) Let

$$\begin{aligned} Y[k] &= \mathcal{FT}\left\{X[n]\right\}(k) \\ &= \left[U^{-k}\right]^0 X[0] + \left[U^{-k}\right]^1 X[1] + \cdots + \left[U^{-k}\right]^{N-1} X[N-1] \end{aligned}$$

be the Fourier transform for $k = 0, ..., N-1$. Looking hard at $Y[k]$ we see that it is a sum of complex sinusoids, with complex amplitudes $X[n]$ and frequencies $-n\omega$ for $n = 0, \ldots, N-1$. In other words, $Y[k]$ can be considered as a waveform in its own right, whose mth component has strength $X[-m]$. We can also find the amplitude of the partials of $Y[k]$ using the Fourier transform on $Y[k]$. Equating the two expressions for the partial amplitudes gives:

$$\frac{1}{N}\mathcal{FT}\left\{Y[k]\right\}(m) = X[-m]$$

(The expression $X[-m]$ makes sense because X is a periodic signal). This means in turn that $X[-m]$ can be obtained by summing sinusoids with amplitudes $Y[-k]/N$. The same analysis starting with $X[-m]$ shows that $X[m]$ is obtained by summing sinusoids using $Y[k]/N$ as their amplitudes. So now we know that any periodic $X[n]$ can indeed be obtained as a sum of sinusoids. Furthermore, we know how to reconstruct a signal from its Fourier transform, if we know its value for the integers $k = 0, \ldots, N-1$.

9.1.2 Periodicity of the Fourier transform

If X[n] is, as above, a signal that repeats every N samples, the Fourier transform of X[n] also repeats itself every N units of frequency, that is,

$$\mathcal{FT}\left\{X[n]\right\}(k+N) = \mathcal{FT}\left\{X[n]\right\}(k)$$

for all real values of k. This follows immediately from the definition of the Fourier transform, since the factor

$$V = \cos(-k\omega) + i\sin(-k\omega)$$

is unchanged when we add N (or any multiple of N) to k.

9.2 Properties of Fourier transforms

In this section we will investigate what happens when we take the Fourier transform of a (complex) sinusoid. The simplest situation is a sinusoid of frequency zero (DC). After we derive the Fourier transform of that, we will develop some properties of Fourier transforms that allow us to apply that result to more general sinusoids.

9.2.1 Fourier transform of DC

Now let $X[n] = 1$ for all n (this repeats with any desired integer period $N > 1$). From the preceding discussion, we expect to find that

$$\mathcal{FT}\left\{X[n]\right\}(k) = \begin{cases} N & k = 0 \\ 0 & k = 1, \ldots, N-1 \end{cases}$$

We will often need to know the answer for non-integer values of k however, and for this there is nothing better to do than to calculate the value directly:

$$\mathcal{FT}\left\{X[n]\right\}(k) = V^0 X[0] + V^1 X[1] + \cdots + V^{N-1} X[N-1]$$

where V (which depends on k) is as before. This is a geometric sum, and as long as $V \neq 1$ we get:

$$\mathcal{FT}\left\{X[n]\right\}(k) = \frac{V^N - 1}{V - 1}$$

We now symmetrize the top and bottom in the same way as we earlier did in Section 7.3. To do this let:

$$\xi = \cos(\pi k/N) + i \sin(\pi k/N)$$

so that $\xi^2 = V$. Then taking appropriate powers of ξ out of the numerator and denominator gives:

$$\mathcal{FT}\left\{X[n]\right\}(k) = \xi^{N-1} \frac{\xi^N - \xi^{-N}}{\xi - \xi^{-1}}$$

It's easy now to simplify the numerator:

$$\xi^N - \xi^{-N} = (\cos(\pi k) + i \sin(\pi k)) - (\cos(\pi k) - i \sin(\pi k)) = 2i \sin(\pi k)$$

and similarly for the denominator, giving:

$$\mathcal{FT}\left\{X[n]\right\}(k) = (\cos(\pi k(N-1)/N) + i \sin(\pi k(N-1)/N)) \frac{\sin(\pi k)}{\sin(\pi k/N)}$$

Putting it all together, whether $V = 1$ or not, we have

$$\mathcal{FT}\left\{X[n]\right\}(k) = (\cos(\pi k(N-1)/N) + i \sin(\pi k(N-1)/N))\, D_N(k)$$

where $D_N(k)$, known as the *Dirichlet kernel*, is defined as

$$D_N(k) = \begin{cases} N & k = 0 \\ \frac{\sin(\pi k)}{\sin(\pi k/N)} & k \neq 0,\ -N < k < N \end{cases}$$

Figure 9.1 shows the Fourier transform of $X[n]$, with $N = 100$. The transform repeats every 100 samples, with a peak at $k = 0$, another at $k = 100$, and

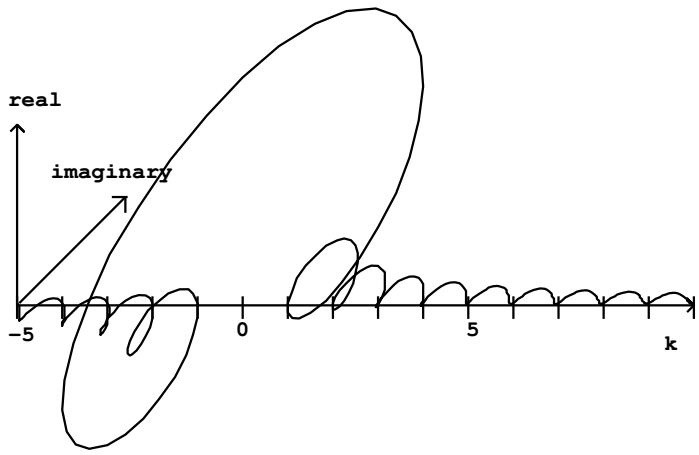


Figure 9.1: The Fourier transform of a signal consisting of all ones. Here N=100, and values are shown for k ranging from -5 to 10. The result is complex-valued and shown as a projection, with the real axis pointing up the page and the imaginary axis pointing away from it.

so on. The figure endeavors to show both the magnitude and phase behavior using a 3-dimensional graph projected onto the page. The phase term

$$\cos(\pi k(N-1)/N) + i\sin(\pi k(N-1)/N)$$

acts to twist the values of $\mathcal{FT}\{X[n]\}(k)$ around the h axis with a period of approximately two. The Dirichlet kernel $D_N(k)$, shown in Figure 9.2, controls the magnitude of $\mathcal{FT}\{X[n]\}(k)$. It has a peak, two bins wide, around $k = 0$. This is surrounded by one-bin-wide *sidelobes*, alternating in sign and gradually decreasing in magnitude as k increases or decreases away from zero. The phase term rotates by almost π radians each time the Dirichlet kernel changes sign, so that the product of the two stays roughly in the same complex half-plane for $k > 1$ (and in the opposite half-plane for $k < -1$). The phase rotates by about 2π radians over the peak from $k = -1$ to $k = 1$.

9.2.2 Shifts and phase changes

Section 7.2 showed how time-shifting a signal changes the phases of its sinusoidal components, and section 8.4.3 showed how multiplying a signal by a complex sinusoid shifts its component frequencies. These two effects each correspond to an identity involving the Fourier transform.

First we consider a time shift. If $X[n]$, as usual, is a complex-valued signal that repeats every N samples, let $Y[n]$ be $X[n]$ delayed d samples:

$$Y[n] = X[n-d]$$

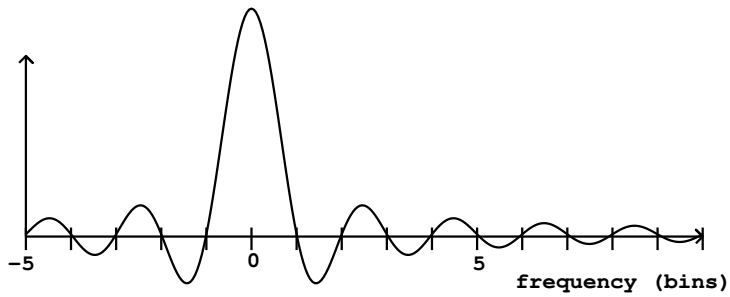


Figure 9.2: The Dirichlet kernel, for $N = 100$.

which also repeats every N samples since X does. We can reduce the Fourier transform of $Y[n]$ this way:

$$\begin{aligned}
\mathcal{FT}\left\{Y[n]\right\}(k) &= V^0 Y[0] + V^1 Y[1] + \cdots + V^{N-1} Y[N-1] \\
&= V^0 X[-d] + V^1 X[-d+1] + \cdots + V^{N-1} X[-d+N-1] \\
&= V^d X[0] + V^{d+1} X[1] + \cdots + V^{d+N-1} X[N-1] \\
&= V^d \left(V^d X[0] + V^{d+1} X[1] + \cdots + V^{d+N-1} X[N-1]\right) \\
&= V^d \mathcal{FT}\left\{X[n]\right\}(k)
\end{aligned}$$

We therefore get the TIME SHIFT FORMULA FOR FOURIER TRANSFORMS:

$$\mathcal{FT}\left\{Y[n]\right\}(k) = (\cos(-dk\omega) + i\sin(-dk\omega))\,\mathcal{FT}\left\{X[n]\right\}(k)$$

So the Fourier transform of $Y[n]$ is a phase term times the Fourier transform of $X[n]$. The phase is changed by $-dk\omega$, a multiple of the frequency k.

Now suppose instead that we change our starting signal $X[n]$ by multiplying it by a complex exponential Z^n with angular frequency α:

$$Y[n] = Z^n X[n]$$

$$Z = \cos(\alpha) + i\sin(\alpha)$$

The Fourier transform is:

$$\begin{aligned}
\mathcal{FT}\left\{Y[n]\right\}(k) &= V^0 Y[0] + V^1 Y[1] + \cdots + V^{N-1} Y[N-1] \\
&= V^0 X[0] + V^1 Z X[1] + \cdots + V^{N-1} Z^{N-1} X[N-1] \\
&= (VZ)^0 X[0] + (VZ)^1 X[1] + \cdots + (VZ)^{N-1} X[N-1] \\
&= \mathcal{FT}\left\{X[n]\right\}(k + \frac{\alpha}{\omega})
\end{aligned}$$

We therefore get the PHASE SHIFT FORMULA FOR FOURIER TRANSFORMS:

$$\mathcal{FT}\left\{(\cos(\alpha) + i\sin(\alpha))X[n]\right\}(k) = \mathcal{FT}\left\{X[n]\right\}(k + \frac{\alpha N}{2\pi})$$

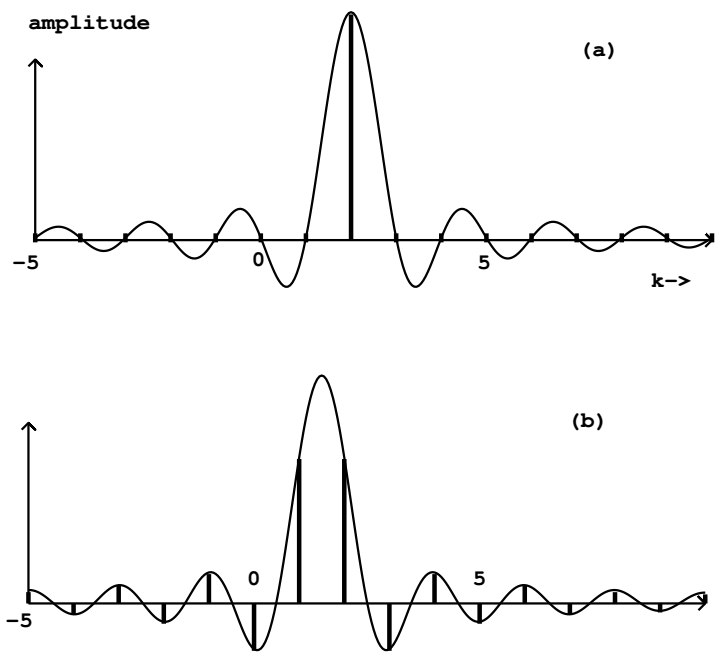


Figure 9.3: Fourier transforms of complex sinusoids, with $N = 100$; the frequencies are (a) 2ω ; (b) 1.5ω.

9.2.3 Fourier transform of a sinusoid

We can use the phase shift formula above to find the Fourier transform of any complex sinusoid Z^n with frequency α, simply by setting $X[n] = 1$ in the formula and using the Fourier transform for DC:

$$\mathcal{FT}\left\{Z^n\right\}(k) = \mathcal{FT}\left\{1\right\}(k + \frac{\alpha}{\omega})$$

$$= \left[\cos(\Phi(k) + i\sin(\Phi(k))\right] D_N(k + \frac{\alpha}{\omega})$$

where D_N is the Dirichlet kernel and Φ is an ugly phase term:

$$\Phi(k) = \pi(k + \frac{\alpha}{\omega})(N-1)/N)$$

If the sinusoid's frequency α is an integer multiple of the fundamental frequency ω, the Dirichlet kernel is shifted by an integral value. In this case the zero crossings of the Dirichlet kernel line up with integer values of k, so that only one partial is nonzero. This is pictured in Figure 9.3 part (a).

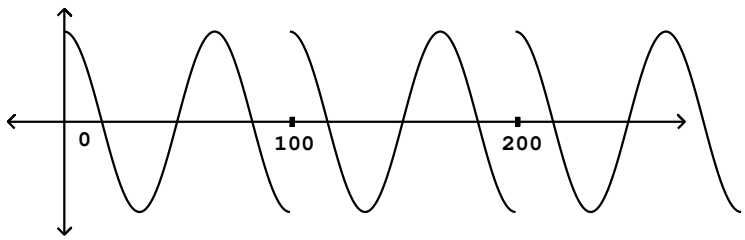


Figure 9.4: A complex sinusoid with frequency $\alpha = 1.5\omega = 3\pi/N$, forced to repeat every N samples. (N was arbitrarily set to 100; only the real part is shown.)

Part (b) shows the result when the frequency α falls halfway between two integers. The partials have amplitudes falling off roughly as $1/k$ in both directions, measured from the actual frequency α. That the energy should be spread over many partials, when after all we started with a single sinusoid, might seem surprising at first. However, as shown in Figure 9.4, the signal repeats at a period N which disagrees with the frequency of the sinusoid. As a result there is a discontinuity at the beginning of each period.

9.3 Fourier analysis of non-periodic signals

Most signals aren't periodic, and even a periodic one might have an unknown period. So we should be prepared to do Fourier analysis on signals without the comforting assumption that the signal to analyze repeats at a fixed period N. Of course, we can simply take N samples of the signal and *make* it periodic; this is essentially what we did in the previous section, in which a pure sinusoid gave us the complicated Fourier transform of Figure 9.3, part (b).

However, it would be better to get a result in which the response to a pure sinusoid were better localized around the corresponding value of k. We can accomplish this using the enveloping technique shown in Figure 2.7 (page 40). Applying this technique to Fourier analysis will not only improve our analyses, but will also shed new light on the enveloping looping sampler of Chapter 2.

Given a signal x[n], periodic or not, defined on the points from 0 to $N-1$, the technique is to envelope the signal before doing the Fourier analysis. The envelope shape is known as a *window function*. Given a window function $w[n]$, the *windowed Fourier transform* is:

$$\mathcal{FT}\left\{w[n]x[n]\right\}(k)$$

Much ink has been spilled over the design of suitable window functions for particular situations, but here we will consider the simplest one, named the

Hann window function (the name is usually corrupted to "Hanning" in DSP circles). The Hann window is:

$$w[n] = \frac{1}{2} - \frac{1}{2}\cos(2\pi n/N)$$

It is easy to analyze the effect of multiplying a signal by the Hann window before taking the Fourier transform, because the Hann window can be written as a sum of three complex exponentials:

$$w[n] = \frac{1}{2} - \frac{1}{4}U^n - \frac{1}{4}U^{-n}$$

where as before, U is a unit-magnitude complex number with argument $2\pi/N$. We can now calculate the windowed Fourier transform of a sinusoid Z^n with angular frequency α as before:

$$\begin{aligned}
&\mathcal{FT}\left\{w[n]Z^n\right\}(k) \\
&= \mathcal{FT}\left\{\frac{1}{2}Z^n - \frac{1}{4}(UZ)^n - \frac{1}{4}(U^{-1}Z)^n\right\}(k) \\
&\approx \left[\cos(\Phi(k) + i\sin(\Phi(k))\right] M(k + \frac{\alpha}{\omega})
\end{aligned}$$

where the phase term has been simplified to:

$$\Phi(k) \approx \pi(k + \frac{\alpha}{\omega}))$$

and the magnitude function is:

$$M(k) = \left[\frac{1}{2}D_N(k) + \frac{1}{4}D_N(k+1) + \frac{1}{4}D_N(k-1)\right]$$

The magnitude function $M(k)$ is graphed in Figure 9.5. The three Dirichlet kernel components are also shown separately.

The main lobe of $M(k)$ is four harmonics wide, twice the width of the main lobe of the Dirichlet kernel. The sidelobes, on the other hand, have much smaller magnitude. Each sidelobe of $M(k)$ is a sum of three sidelobes of $D_n(k)$, one attenuated by 1/2 and the others, opposite in sign, attenuated by 1/4. They do not cancel out perfectly but they do cancel out fairly well.

The sidelobes reach their maximum amplitudes near their midpoints, and we can estimate their amplitudes there, using the approximation:

$$D_N(k) \approx \frac{N sin(\pi k)}{\pi k}$$

and setting $k = 3/2, 5/2, \ldots$ gives sidelobe amplitudes, relative to the peak height N, of:

$$\frac{2}{3\pi} \approx -13\text{dB}, \frac{2}{5\pi} \approx -18\text{dB}, \frac{2}{7\pi} \approx -21\text{dB}, \frac{2}{9\pi} \approx -23\text{dB}, \ldots$$

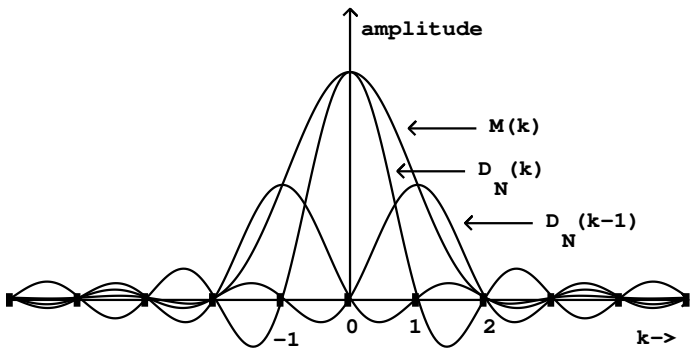


Figure 9.5: The magnitude M(k) of the Fourier transform of the Hann window function. It is the sum of three (shifted and magnified) copies of the Dirichlet kernel D_N, with $N = 100$.)

The sidelobes drop off progressively more slowly so that the tenth one is only attenuated about 30 dB and the 32nd one about -40 dB. On the other hand, the Hann window sidelobes are attenuated by:

$$\frac{2}{5\pi} - \frac{1}{2}[\frac{2}{3\pi} + \frac{2}{7\pi}] \approx -32.30\text{dB}$$

and -42, -49, -54, and -59 dB for the next four sidelobes.

This implies that applying a Hann window will allow us to isolate sinusoidal components from each other better in a Fourier transform (than if no Hann window is applied.) If a signal has many sinusoidal components, the sidelobes engendered by each one will interfere with the main lobe of all the others. Reducing the amplitude of the sidelobes reduces this interference.

Figure 9.6 shows a Hann-windowed Fourier analysis of a signal with two sinusoidal components. The two are separated by about 5 times the fundamental frequency ω, and for each we see clearly the shape of the Hann window's Fourier transform. Four points of the Fourier analysis lie within the main lobe of $M(k)$ corresponding to each sinusoid. The amplitude and phase of the individual sinusoids are reflected in those of the (four-point-wide) peaks. The four points within a peak which happen to fall at integer values k are successively one half cycle out of phase.

To fully resolve the partials of a signal, we should choose an analysis size N large enough so that $\omega = 2\pi/N$ is no more than a quarter of the frequency separation between neighboring partials. For a periodic signal, for example, the partials are separated by the fundamental frequency. For the analysis to fully resolve the partials, the analysis period N must be at least four periods of the signal.

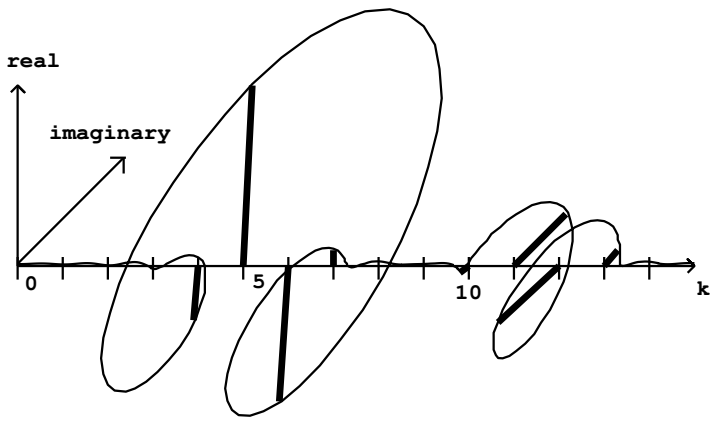


Figure 9.6: The Hann-windowed Fourier transform of a signal with two sinusoidal components, at frequencies 5.3 and 10.6 times the fundamental, and with different complex amplitudes.

In some applications it works to allow the peaks to overlap as long as the center of each peak is isolated from all the other peaks; in this case the four-period rule may be relaxed to three or even slightly less.

9.4 Fourier analysis and reconstruction of audio signals

Fourier analysis can sometimes be used to resolve the component sinusoids in an audio signal. Even when it can't go that far, it can separate a signal into frequency regions, in the sense that for each k, the kth point of the Fourier transform would be affected only by components close to the nominal frequency $k\omega$. This suggests many interesting operations we could perform on a signal by taking its Fourier transform, transforming the results, and then reconstructing a new, transformed, signal from the modified transform.

Figure 9.7 shows how to carry out a Fourier analysis, modification, and reconstruction of an audio signal. The first step is to divide the signal into *windows*, which are segments of the signal, of N samples each, usually with some overlap. Each window is then shaped by multiplying it by a windowing function (Hann, for example). Then the Fourier transform is calculated for the N points $k = 0, 1, \ldots, N-1$. (Sometimes it is desirable to calculate the Fourier transform for more points than this, but these N points will suffice here.)

The Fourier analysis gives us a two-dimensional array of complex numbers. Let H denote the number of samples each window is advanced past the previous

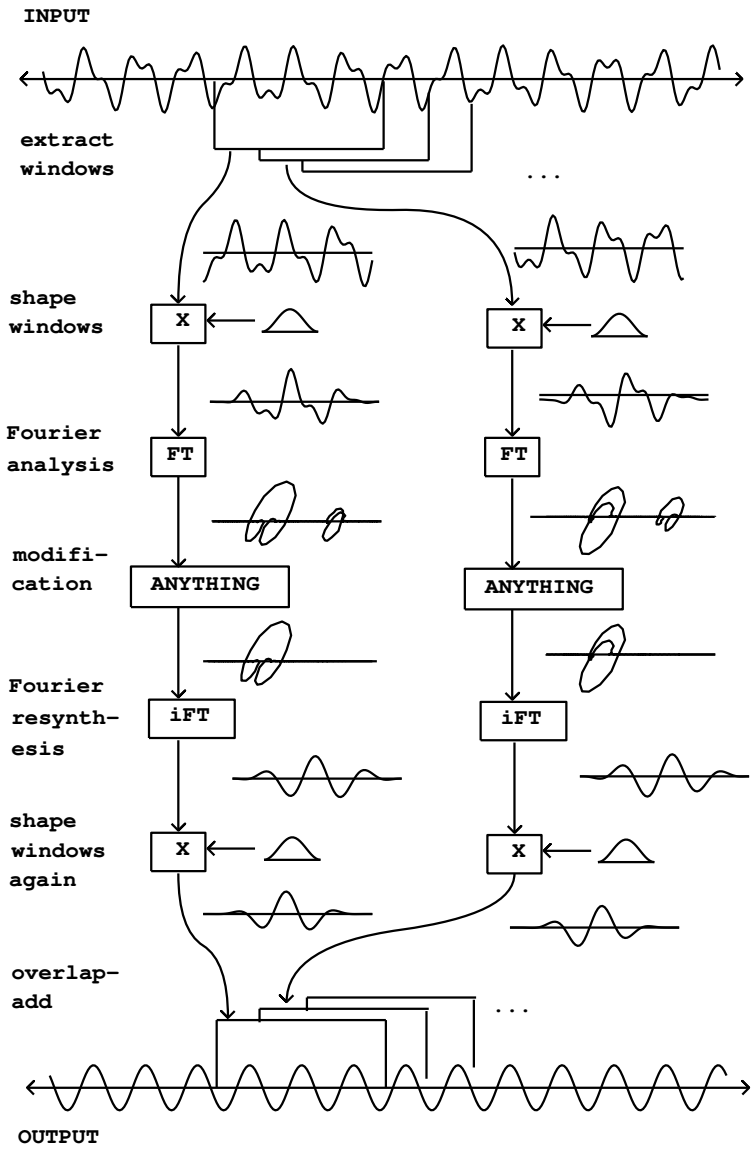


Figure 9.7: Sliding-window analysis and resynthesis of an audio signal using Fourier transforms. In this example the signal is filtered by multiplying the Fourier transform with a desired frequency response.

window. Then for each $m = \ldots, 0, 1, \ldots$, the mth window consists of the N points starting at the point mH. The nth point of the mth window is $mH + n$. The windowed Fourier transform is thus equal to:

$$S[m, k] = \mathcal{FT}(W(n)X[n - mH])(k)$$

This is both a function of time (m, in units of H samples) and of frequency (k, as a multiple of the fundamental frequency ω). Fixing the frame number m and looking at the windowed Fourier transform as a function of k:

$$S[k] = S[m, k]$$

gives us a measure of momentary spectrum of the signal $X[n]$. On the other hand, fixing a frequency k we can look at it as the kth channel of an N-channel signal:

$$C[m] = S[m, k]$$

From this point of view, the windowed Fourier transform separates the original signal $X[n]$ into N narrow frequency regions, or bands.

Having computed the windowed Fourier transform, we next apply any desired modification. In the figure, the modification is simply to replace the upper half of the spectrum by zero, which gives a highly selective low-pass filter. Two other possible modifications, narrow-band companding and vocoding, are described in the following sections.

Finally we reconstruct an output signal. To do this we apply the inverse of the Fourier transform (labeled "iFT" in the figure). As shown in Section 9.1.1 this can be done by taking another Fourier transform, normalizing, and flipping the result backwards. In case the reconstructed window does not go smoothly to zero at its two ends, we apply the Hann windowing function a second time. Doing this to each successive window of the input, we then add the outputs, using the same overlap as for the analysis.

If we use the Hann window and an overlap of four (that is, choose N a multiple of four and space each window $N/4$ samples past the previous one), we can reconstruct the original signal faithfully by omitting the "modification" step. This is because the iFT undoes the work of the FT, and so we are multiplying each window by the Hann function squared. The output is thus the input, times the Hann window function squared, overlap-added by four. An easy check shows that this comes to the constant $3/2$, so the output equals the input adjusted by a gain factor.

The ability to reconstruct the input signal exactly is useful because some types of modification may be done by degrees, and so the output can be made to vary smoothly between the input and some transformed version of it.

9.4.1 Narrow-band companding

A *compander* is a tool that amplifies a signal with a variable gain, depending on the signal's measured amplitude. The term is a contraction of "compressor" and

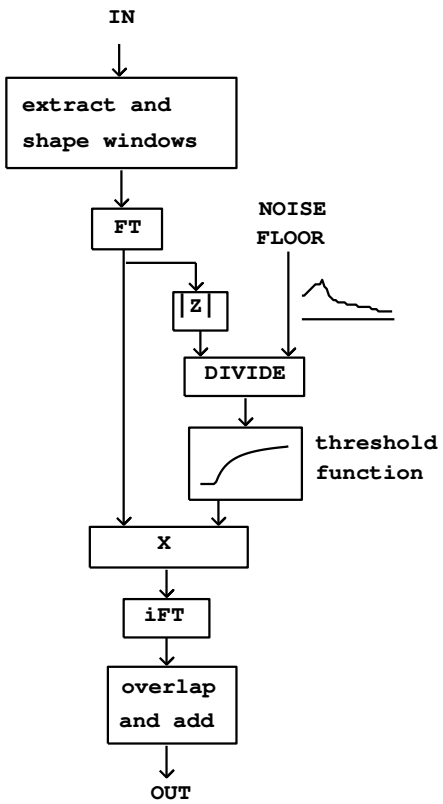


Figure 9.8: Block diagram for narrow-band noise suppression by companding.

"expander". A compressor's gain increases as the input level increases, so that its *dynamic range*, that is, the overall variation in signal level, is reduced. An expander does the reverse, increasing the dynamic range. Frequently the gain depends not only on the immediate signal level but on its history; for instance the rate of change might be limited or there might be a time delay.

By using Fourier analysis and resynthesis, we can do companding individually on narrow-band channels. If $C[m]$ is one such band, we apply a gain $g[m]$ to it, to give $g[m]C[m]$. Although $C[m]$ is a complex number, the gain is a non-negative real number. In general the gain could be a function not only of $C[m]$ but also of any or all the previous samples in the channel: $C[m-1]$, $C[m-2]$, and so on. Here we'll consider the simplest situation where the gain is simply a function of the magnitude of the current sample: $|C[m]|$.

The patch diagrammed in Figure 9.8 shows one very useful application of companding, called a *noise gate*. Here the gain $g[m]$ depends on the channel amplitude $C[m]$ and a noise floor which is a function f of the channel number k. For clarity we will introduce the frequency subscript k to the gain, now written as $g[m,k]$, and to the windowed Fourier transform $S[m,k] = C[m]$. The gain is given by:

$$g[m,k] = \begin{cases} 1 - f[k]/|S[m,k]| & |S[m,k]| > f[k] \\ 0 & \text{otherwise} \end{cases}$$

Whenever the magnitude $S[m,k]$ is less than the threshold $f[k]$ the gain is zero and so the amplitude $S[m,k]$ is replaced by zero. When greater, multiplying the amplitude by $g[m,k$ reduces the the magnitude downward to $|S[m,k]| - f[k]$. Since the gain is a non-negative real number, the phase is preserved.

In the figure, the gain is computed as a thresholding function of the ratio $x = |S[m,k]|/f[k]$ of the signal amplitude above the noise floor; the threshold is $g(x) = 1 - 1/x$ when $x < 1$ and zero otherwise, although other thresholding functions could easily be substituted.

This technique is useful for removing noise from a recorded sound. We either measure or guess values of $f[k]$ according to a noise floor. Because of the design of the gain function $g[m,k]$, only amplitudes which are above the noise floor reach the output. Since this is done on narrow frequency bands, most of the noise can be removed even while the signal itself, in the frequency ranges where it is louder than the noise floor, is mostly preserved.

This operation is also useful as a pre-processor before applying a non-linear operation, such as distortion, to a sound. It is often best to distort only the most salient frequencies of the sound. Subtracting the noise-gated sound from the original gives a residual signal which can be passed through undistorted.

9.4.2 Timbre stamping (classical vocoder)

A second application of Fourier analysis and resynthesis is a time-varying filter capable of making one sound take on the evolving spectral envelope of another. This is widely known in electronic music circles as a *vocoder*, named, not quite accurately, after the original Bell Laboratories vocal analysis/synthesis device of

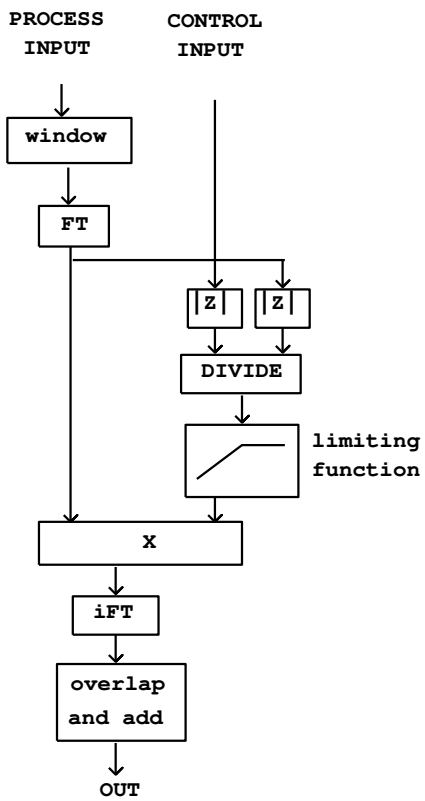


Figure 9.9: Block diagram for timbre stamping (AKA "vocoding").

that name. The technique described here is more accurately called *timbre stamping*. Two input signals are used, one to be filtered, and the other to control the filter via its time-varying spectral envelope. The windowed Fourier transform is used both on the control signal input to estimate its spectral envelope, and on the filter input in order to apply the filter.

A block diagram for timbre stamping is shown in Figure 9.9. As in the previous example, the timbre stamp acts by multiplying the complex-valued windowed Fourier transform of the filtering input by non-negative real numbers, hence scaling their amplitudes but leaving their phases intact. Here the twist is that we want simply to replace the magnitudes of the original, $|S[m,k]|$, with magnitudes obtained from the control input (call them $|T[m,k]|$, say). The necessary gain would thus be,

$$g[m,k] = \frac{|T[m,k]|}{|S[m,k]|}$$

In practice it is best to limit the gain to some maximum value (which might depend on frequency) since otherwise channels containing nothing but noise, sidelobes, or even truncation error might be raised to audibility. So a suitable limiting function is applied to the gain before using it.

9.5 Phase

So far we have operated on signals by altering the magnitudes of their windowed Fourier transforms, but leaving phases intact. The magnitudes encode the spectral envelope of the sound. The phases, on the other hand, encode frequency and time, in the sense that phase change from one window to a different one accumulates, over time, according to frequency. To make a transformation that allows independent control over frequency and time requires analyzing and reconstructing the phase.

Figure 9.10 shows how the phase of the Fourier transform changes from window to window, using a complex sinusoid as input. The sinusoid's frequency is 3ω, so that the peak in the Fourier transform is centered at $k = 3$. If the initial phase is ϕ, then the neighboring phases can be filled in as:

$$\begin{array}{lll}
\angle S[0,2] = \phi + \pi & \angle S[0,3] = \phi & \angle S[0,4] = \phi + \pi \\
\angle S[1,2] = \phi + H\alpha + \pi & \angle S[1,3] = \phi + H\alpha & \angle S[1,4] = \phi + H\alpha + \pi \\
\angle S[2,2] = \phi + 2H\alpha + \pi & \angle S[2,3] = \phi + 2H\alpha & \angle S[2,4] = \phi + 2H\alpha + \pi
\end{array}$$

This gives an excellent way of estimating the frequency α: pick any channel whose amplitude is dominated by the sinusoid and subtract two successive phase to get $H\alpha$:

$$H\alpha = \angle S[1,3] - \angle S[0,3]$$

$$\alpha = \frac{\angle S[1,3] - \angle S[0,3] + 2p\pi}{H}$$

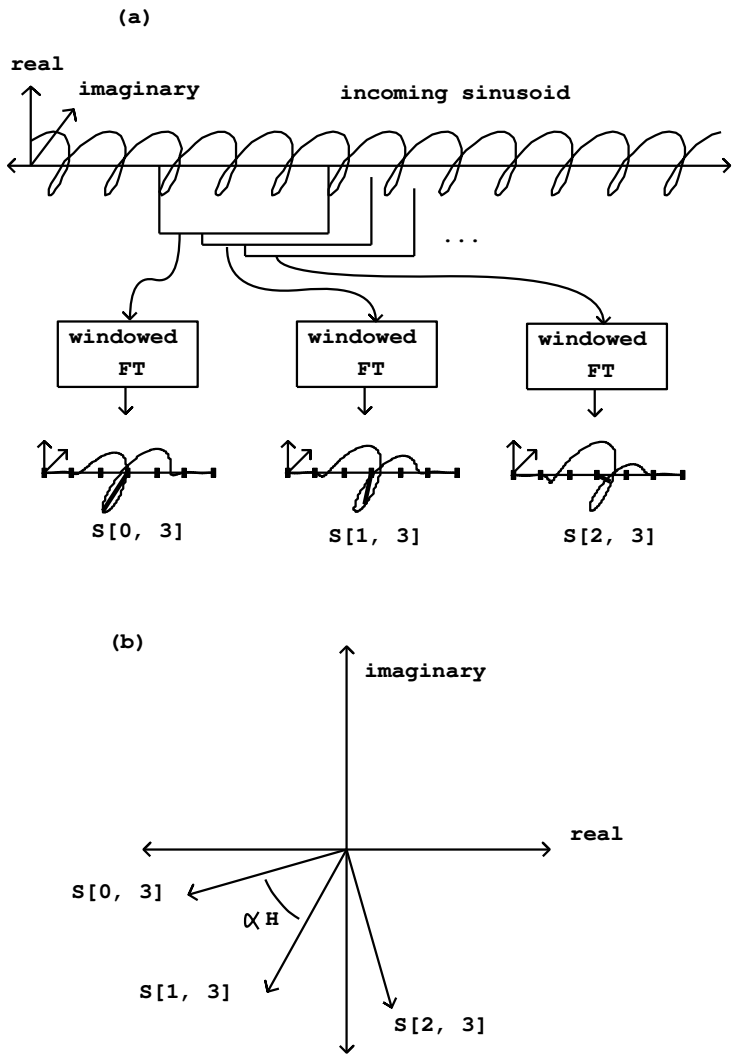


Figure 9.10: Phase in windowed Fourier analysis: (a) a complex sinusoid analyzed on three successive windows; (b) the result for a single channel (k=3), for the three windows

where p is an integer. There are H possible frequencies, spaced by $2\pi/H$. If we are using an overlap of 4, that is, $H = N/4$, the frequencies are spaced by $8\pi/N = 4\omega$. Happily, this is the width of the main lobe for the Hann window, so no more than one possible value of α can explain any measured phase difference within the main lobe of a peak. The correct value of p to choose is that which gives a frequency closest to the nominal frequency of the channel, $k\omega$.

In the analysis/synthesis examples of the previous section, the phases of the output are derived directly from the phases of an input. This is appropriate when the output signal corresponds in time with the input signal. Sometimes time modifications are desired, for instance to do time stretching or contraction. Alternatively the output phase might depend on more than one input, for instance for attempting to morph between one sound and another.

In these situations, the important thing is to try to maintain the appropriate phase relationships between successive resynthesis windows, and also between adjacent channels. These two sets of relationships are not always compatible, however. We will make it our primary obligation to honor the relations between successive resynthesis windows, and worry about phase relationships between channels afterward.

Suppose we want to construct the mth spectrum $S[m,k]$ for resynthesis (having already constructed the previous one, number $m-1$). Suppose we wish the phase relationships between windows $m-1$ and m to be those of a signal $x[n]$, but that the phases of window number $m-1$ might have come from somewhere else and can't be assumed to be in line with our wishes.

To find out how much the phase of each channel should differ from the previous one, we do two analysis of the signal $x[n]$, separated by the same hop size H that we're using for resynthesis:

$$T[k] = \mathcal{FT}(W(n)X[n])(k)$$

$$T'[k] = \mathcal{FT}(W(n)X[n+H])(k)$$

Figure 9.11 shows the process of phase accumulation, in which the output phases each depend on the previous output phase and the phase difference for two windowed analyses of the input. Figure 9.12 illustrates the phase relationship in the complex plane. The phase of the new output $S[m,k]$ should be that of the previous one plus the difference between the phases of the two analyses:

$$\begin{aligned}\angle S[m,k] &= \angle S[m-1,k] + (\angle T'[k] - \angle T[k]) \\ &= \angle\left(\frac{S[m-1,k]T'[k]}{T[k]}\right)\end{aligned}$$

Here we used the fact that multiplying or dividing two complex numbers gives the sum or difference of their arguments.

If the desired magnitude is a real number a, then we should set $S[m,k]$ to:

$$S[m,k] \;=\; a \;\cdot\; \left|\frac{S[m-1,k]T'[k]}{T[k]}\right|^{-1} \;\cdot\; \frac{S[m-1,k]T'[k]}{T[k]}$$

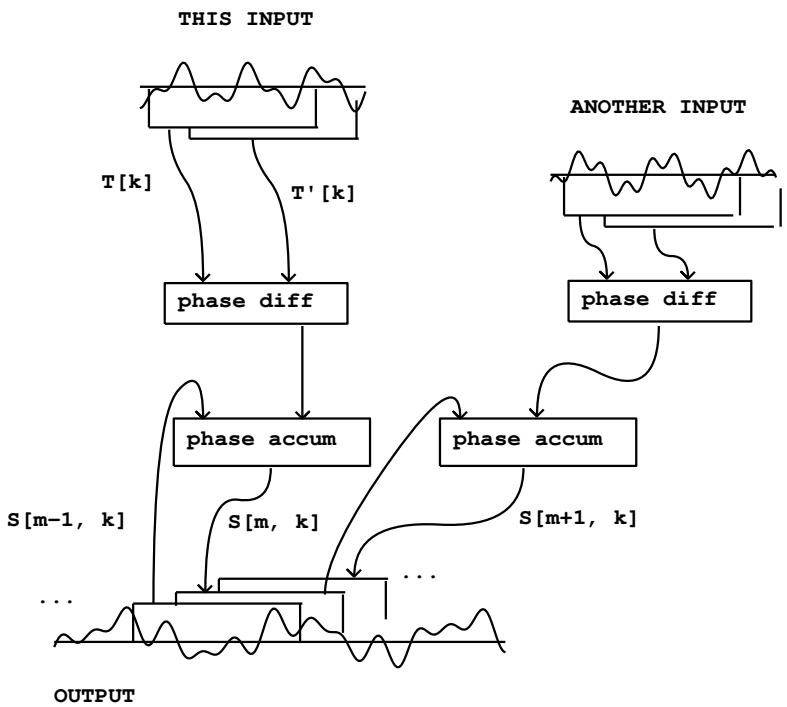


Figure 9.11: Propagating phases in resynthesis. Each phase, such as that of $S[6, k]$ here, depends on the previous output phase and the difference of the input phases.

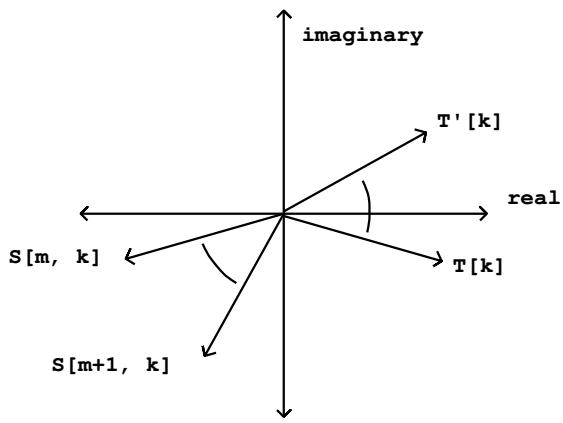


Figure 9.12: Phases of one channel of the analysis windows and two successive resynthesis windows.

The magnitudes of the second and third terms cancel out, so that the magnitude of $S[m,k]$ reduces to a; the first two terms are real numbers so the argument is controlled by the last term.

If we want to take magnitude from the spectrum T as well, we can set $a = |T'[k]|$, giving a simpler formula:

$$S[m,k] \;=\; \left|\frac{S[m-1,k]}{T[k]}\right|^{-1} \cdot \frac{S[m-1,k]T'[k]}{T[k]}$$

9.5.1 Phase relationships between channels

In the scheme above, the phase of each $S[m,k]$ depends only on the previous value for the same channel. The phase relationships between neighboring channels are left to chance. This is adequate in some situations, but sometimes the incoherence of neighboring channels gives rise to an unintended chorus effect. We would ideally like for $S[m,k]$ and $S[m,k+1]$ to have the same phase relationship as for $T'[k]$ and $T'[k+1]$, but also for the phase relationship between $S[m,k]$ and $S[m-1,k]$ to be the same as between $T'[k]$ and $T[k]$.

These $2N$ equations for N phases in general will have no solution, but we can alter the equation for $S[m,k]$ above so that whenever there happens to be a solution to the over-constrained system of equations, the reconstruction algorithm homes in on the solution. This approach is called *phase locking* [Puc95b], and has the virtue of simplicity although more sophisticated techniques are available [DL97]).

The condition for an exact solution to exist at stage m is that:

$$\angle T[k+1] - \angle T[k] = \angle S[m-1, k+1] - \angle S[m-1, k]$$

or, rearranging:

$$\angle \left\{ \frac{S[m-1, k+1]}{T[k+1]} \right\} = \angle \left\{ \frac{S[m-1, k]}{T[k]} \right\}$$

With this in mind, we can rewrite the recursion formula for $S[m, k]$:

$$S[m, k] \;=\; |R[k]|^{-1} \;\cdot\; R[k]T'[k]$$

with

$$R[k] \;=\; \frac{S[m-1, k]}{T[k] \;\cdot\; |S[m-1, k]|}$$

where the $R[k]$ should all be in phase. The trick is now to replace $R[k]$ for each k with the sum of three neighboring ones. The computation is then:

$$S[m, k] \;=\; |R'[k]|^{-1} \;\cdot\; R'[k]T'[k]$$

with

$$R'[k] = R[k+1] + R[k] + R[k-1]$$

If the channels are already in the correct phase relationship, this has no effect (the resulting phase will be the same as if only $R[k]$ were used.) But in general the sum will share two terms in common with its neighbor:

$$R'[k+1] = R[k+2] + R[k+1] + R[k]$$

so that the R' will tend to point more in the same direction than the R do. Applying this iteratively will eventually line all the R' up to the same phase, as long as the phase relationships between the measured spectra T and T' allow it.

9.6 Phase bashing

In section 2.3 on enveloped sampling we saw how to make a periodic waveform from a recorded sound, thereby borrowing the timbre of the original sound but playing it at a specified pitch. If the window into the recorded sound is made to precess in time, the resulting timbre varies in imitation of the recorded sound.

One important problem arises, which is that if we take waveforms from different windows of a sample (or from different samples), there is no guarantee that the phases of the two match up. If they don't, the result is heard as ugly-sounding frequency deviations (since frequency modulation can be thought of as phase slippage). This can be dealt with using Fourier analysis and resynthesis [Puc05].

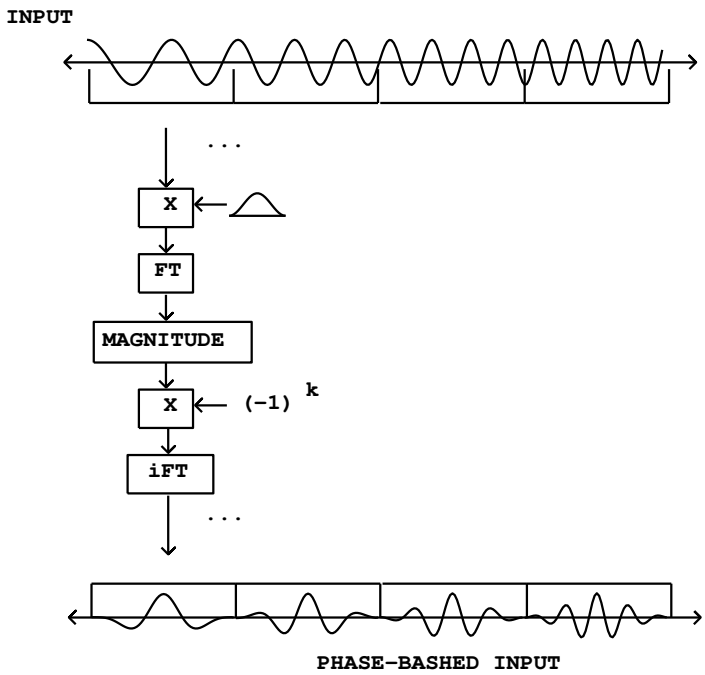


Figure 9.13: Phase-bashing a recorded sound (here, a sinusoid with rising frequency) to give a series of oscillator wavetables.

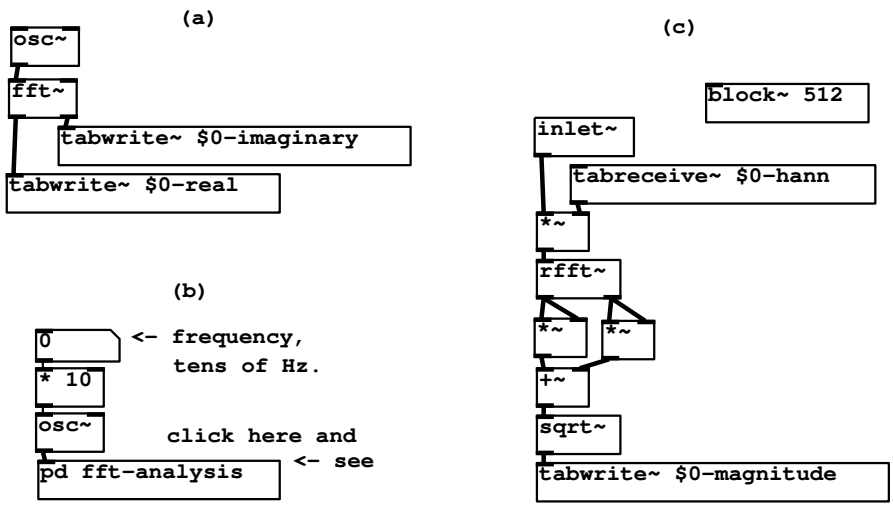


Figure 9.14: Fourier analysis in Pd: (a) the fft ~ object; (b) using a subwindow to control block size of the Fourier transform; (c) the subwindow, using a real Fourier transform (the `fft~`object) and the Hann windowing function.

Figure 9.13 shows the simplest way to use Fourier analysis to align phases in a series of windows in a recording. We simply take the FFT of the window and then align the phase so that it is zero for even values of k and π for odd ones. The phase at the center of the window is thus zero for both even and off values of k. To set the phases (the arguments of the complex amplitudes in the spectrum) in the desired way, first we find the magnitude, which can be considered a complex number with argument zero. Then multiplying by $(-1)^k$ adjusts the amplitude so that it is positive and negative in alternation. Then we take the inverse Fourier transform, without even bothering to window again on the way back; we will probably want to apply a windowing envelope later anyway as was shown in Figure 2.7.

9.7 Examples

Fourier analysis and resynthesis in Pd

Patch H01.low-pass.pd (Figure 9.14 part a) demonstrates computing the Fourier transform of an audio signal using the `fft~` object:

`fft~`: Fast Fourier transform. The two inlets take audio signals representing the real and imaginary parts of a complex-valued signal. The window size N is given by the block size of the window. One Fourier transform is done on each block.

The Fast Fourier transform [III03] reduces the computational cost if Fourier analysis in Pd to only that of between 5 and 15 `osc~` objects in typical configurations. The FFT algorithm in its simplest form takes N to be a power of two, which is also a constraint on block sizes in Pd.

Patch H02.high-pass.pd (Figure 9.14 parts b and c) shows how to control the block size using a `block~` object, how to apply a Hann window, and a different version of the Fourier transform. Part (b) shows the invocation of a subwindow which in turn is shown in part (c). New objects are:

`rfft~`: real Fast Fourier transform. The imaginary part of the input is assumed to be zero. Only the first $N/2 + 1$ channels of output are filled in (the others are determined by symmetry). This takes half the computation time of the (more general) `fft~`object.

`tabreceive~`: repeatedly outputs the contents of a table (a floating-point array). Each block of computation outputs the same first N samples of the table.

In this example, the table "$0-hann" holds a Hann window function of length 512, in agreement with the specified block size. The signal to be analyzed appears (from the parent patch) via the `inlet~` object. The channel amplitudes (the output of the `rfft~` object) are reduced to real-valued magnitudes: the real and imaginary parts are squared separately, the two squares are added, and the result passed to the `sqrt~` object). Finally the magnitude is written (controlled by a connection not shown in the figure) via `tabwrite~` to another table, "$0-magnitude", for graphing.

Patch H03.band-pass.pd (Figure 9.15) shows how to analyze and resynthesize an audio signal following the strategy of Figure 9.4. As before there is a subwindow to do the work at a block size appropriate to the task; the figure shows only the sub-window. We need one new object for the inverse Fourier transform:

`rifft~`: real inverse Fast Fourier transform. Using the first $N/2 + 1$ points of its inputs (taken to be a real/imaginary pair), and assuming the appropriate values for the other channels by symmetry, reconstructs a real-valued output. No normalization is done, so that a `rfft~`/`rifft~` pair together result in a gain of N. The `ifft~` object is also available which computes an unnormalized inverse for the `fft~` object, reconstructing a complex-valued output.

The `block~` object, in the subwindow, is invoked with a second argument which specifies an overlap factor of 4. This dictates that the sub-window will run four times every $N = 512$ samples, at even intervals of 128 samples. The `inlet~` object does the necessary buffering and rearranging of samples so that its output always gives the 512 latest samples of input in order. In the other direction, the `outlet~` object adds segments of its previous four inputs to carry out the overlap-add scheme shown in Figure 9.4.

The 512-sample blocks are multiplied by the Hann window both at the input and the output. If the `rfft~` and `rifft~` objects were connected without any modifications in between, the output would perfectly reconstruct the input.

A modification is applied, however: we simply multiply each channel by a

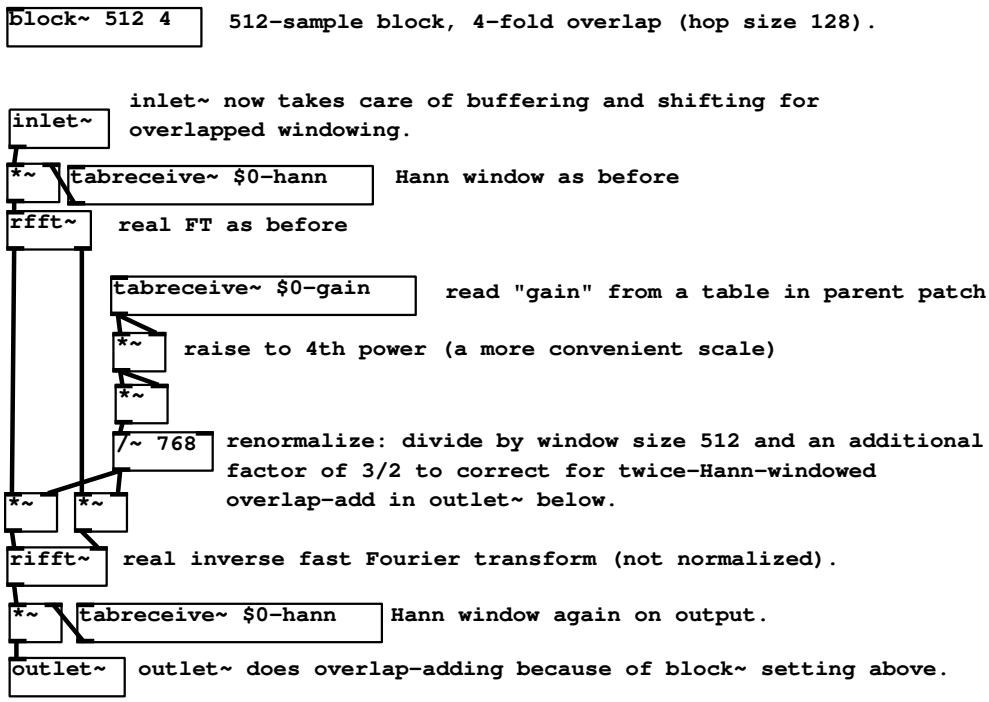


Figure 9.15: Fourier analysis and resynthesis, using `block~` to specify an overlap of 4, and `rifft~` to reconstruct the signal after modification.

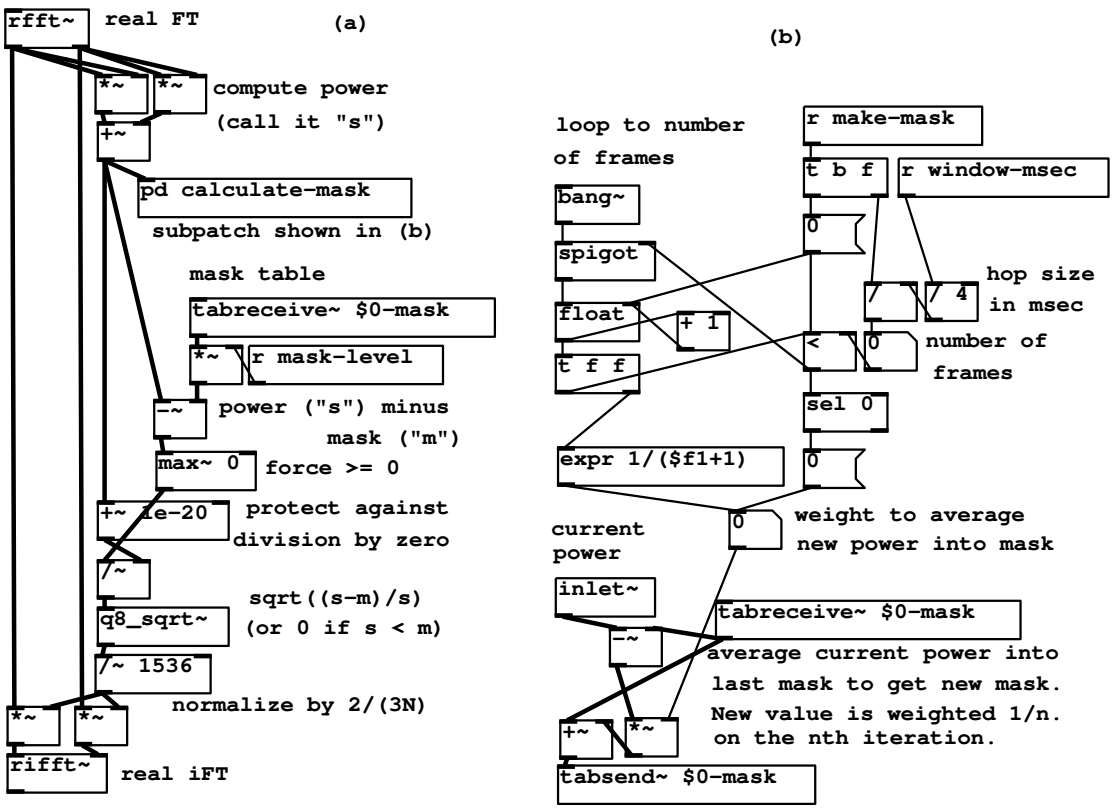


Figure 9.16: Noise suppression as an example of narrow-band companding.

(positive real-valued) gain. The complex-valued amplitude for each channel is scaled by separately multiplying the real and imaginary parts by the gain. The gain (which is different for each channel) comes from another table, named "$0-gain". The result is a graphical equalization filter; by mousing in the graphical window for this table, you can design gain-frequency curves.

There is an inherent delay introduced by using `block~` to increase the block size (but none if it is used, as shown in Chapter 7, to reduce block size relative to the parent window.) The delay can be measured from the inlet to the outlet of the sub-patch, and is equal to the difference of the two block sizes. In this example the buffering delay is 512-64=448 samples. Blocking delay does not depend on overlap, only on block sizes.

Narrow-band companding: noise suppression

Patch H04.filter.sweep.pd (Figure 9.16) shows an example of narrow-band companding using Fourier analysis/resynthesis. (This is a realization of the block

diagram of Figure 9.8.) Part (a) of the figure shows a filter configuration similar to the previous example, except that the gain for each channel is now a function of the channel magnitude.

For each k, if we let $s[k]$ denote the power in channel k, and let $m[k]$ be a mask level (a level presumably somewhat higher than the noise power for channel k), then the gain in channel k is given by

$$\begin{cases} \sqrt{\frac{s[k]-m[k]}{s[k]}} & s[k] > m[k] \\ 0 & \text{otherwise} \end{cases}$$

The power in the kth channel is thus reduced by $m[k]$, or by zero, whichever is greater.

The mask itself is the product of the measured average noise in each channel, which is contained in the table `$0-mask`, multiplied by a value named "mask-level". The average noise is measured in a subpatch (`pd calculate-mask`), whose contents are shown in part (b) of the figure. To carry out control computations for each block of computation, we are using a new object:

`bang~`: send a bang in advance of each block of computation. The bang appears at the logical time of the first sample in each block (the earliest logical time whose control computation affects that block and not the previous one), as shown in Figure 3.2.

The power averaging process is begun by sending a time duration in milliseconds to `make-mask`. The patch computes the equivalent number of blocks b and generates a sequence of weights: $1, 1/2, 1/3, \ldots, 1/b$, by which each of the b following blocks' power is averaged into whatever the mask table held at the previous block. At the end of b blocks the table holds the equally-weighted average of all b power measurements. Thereafter, the weight for averaging new power measurements is zero, so the measured average stops evolving.

To use this patch for classical noise suppression requires at least a few seconds of recorded noise without the "signal" present. This is played into the patch, and its duration sent to `make-mask`, so that the `$0-mask` table holds the average measured noise power for each channel. Then, making the assumption that the noisy part of the signal rarely exceeds 10 times its average power (for example), `mask-level` is set to 10, and the signal to be noise-suppressed is sent through part (a) of the patch. The noise will be almost all gone, but those channels in which the signal exceeds 20 times the noise power will only be attenuated by 3dB, and higher-power channels progressively less. (Of course, actual noise suppression might not be the most interesting application of the patch; one could try masking any signal from any other one.)

Timbre stamp ("vocoder")

Patch I05.compressor.pd is another channel compander which is presented in preparation for patch I06.timbre.stamp.pd, which we will describe next. This is a realization of the timbre stamp of Figure 9.9, slightly modified.

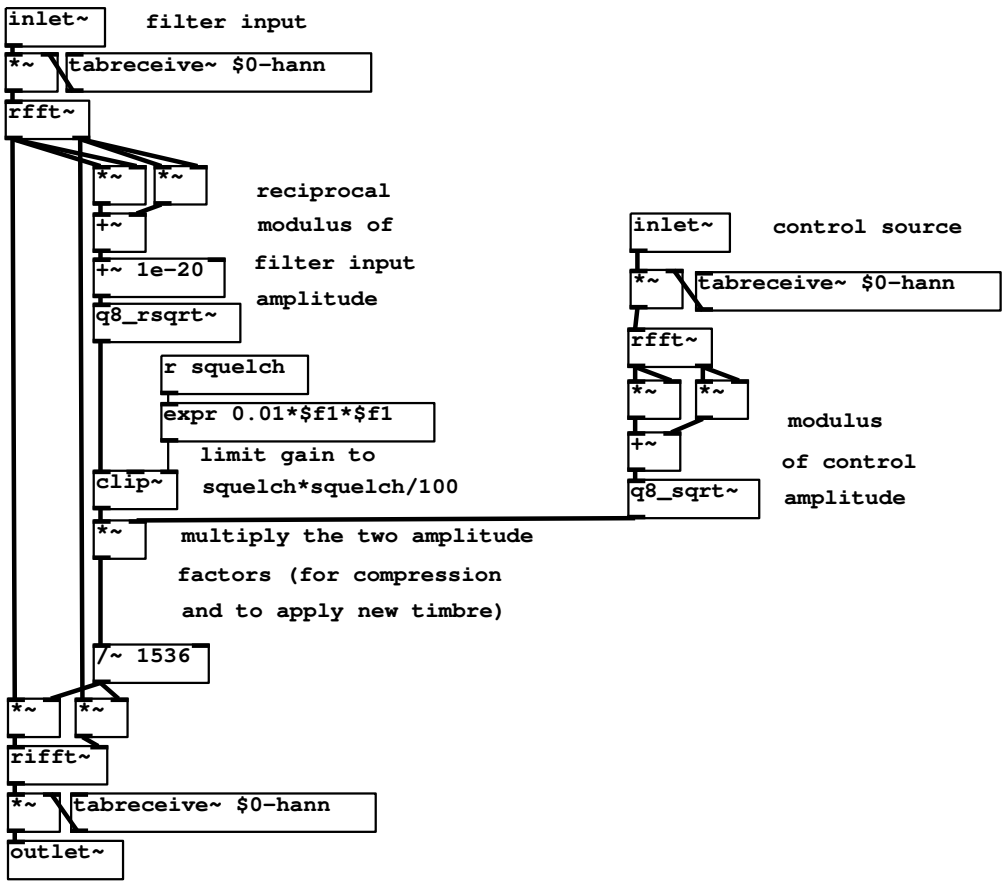


Figure 9.17: Timbre stamp.

There are two inputs, one at left to be filtered (and whose Fourier transform is used for resynthesis after modifying the magnitudes), and one at right which acts as a control source. In the simplest case, if the two magnitudes are $f[k]$ for the filter input and $c[k]$ for the control source, we just "whiten" the filter input, multiplying by $1/f[k]$, and then stamp the control magnitudes onto the result by further multiplying by $c[k]$. In practice, we must limit the gain to some reasonable maximum value. In this patch this is cone by limiting the whitening factor $1/f[k]$ to a specified maximum value. This is done by the `clip~` object. The limit is controlled by the `squelch` parameter, which is squared and divided by 100 to make the units reasonable.

Another possible scheme is to limit the gain after forming the quotient $c[k]/f[k]$. The gain limitation may in either case be frequency dependent. Finally, it is sometimes useful to raise the gain to a power p between 0 and 1; if 1, this is a timbre stamp and if 0, it passes the filter input through unchanged, and values in between give a smooth interpolation between the two.

Phase vocoder time bender

The phase vocoder usually refers to the general technique of passing from (complex-valued) channel amplitudes to pairs consisting of (real-valued) magnitudes and phase precession rates ("frequencies"), and back, as described in Figure 9.11 (section 9.5). In patch I07.phase.vocoder.pd (figure 9.18), we use this technique for the specific goal of time-stretching and/or time-contracting a recorded sound under real-time control. That is, we control, at any moment in real time, the location in the recorded sound we hear.

Two new objects are needed:

`tabsemd~`: the companion object for `tabreceive~`, repeatedly copies its input to the contents of a table. Affects up to the first N samples of the table,

`lrshift`: Shift a block left or right according to its argument. If the argument is positive, each block of the output is the input shifted that number of spaces to the right, filling zeros in as needed on the left. A negative argument shifts to the left, filling zeros in at the right.

The process starts with a sub-patch, `pd read-windows`, which outputs two Hann-windowed blocks of the recorded sound, a "back" one and a "front" one 1/4 window further forward in the recording. The window shown uses the two outputs of the sub-patch to guide the amplitude and phase change of each channel of its own output.

The top two `tabreceive~` objects recall the previous block of complex amplitudes sent to the `rifft~` object at bottom, corresponding to $S[m-1,k]$ in the discussion of section 9.5. The patch as a whole computes $S[m,k]$ and then its Hann windowed inverse FT for output.

After normalizing $S[m-1,k]$, its complex conjugate (equal to its inverse) is multiplied by the windowed Fourier transform of the "back" window $T[k]$,

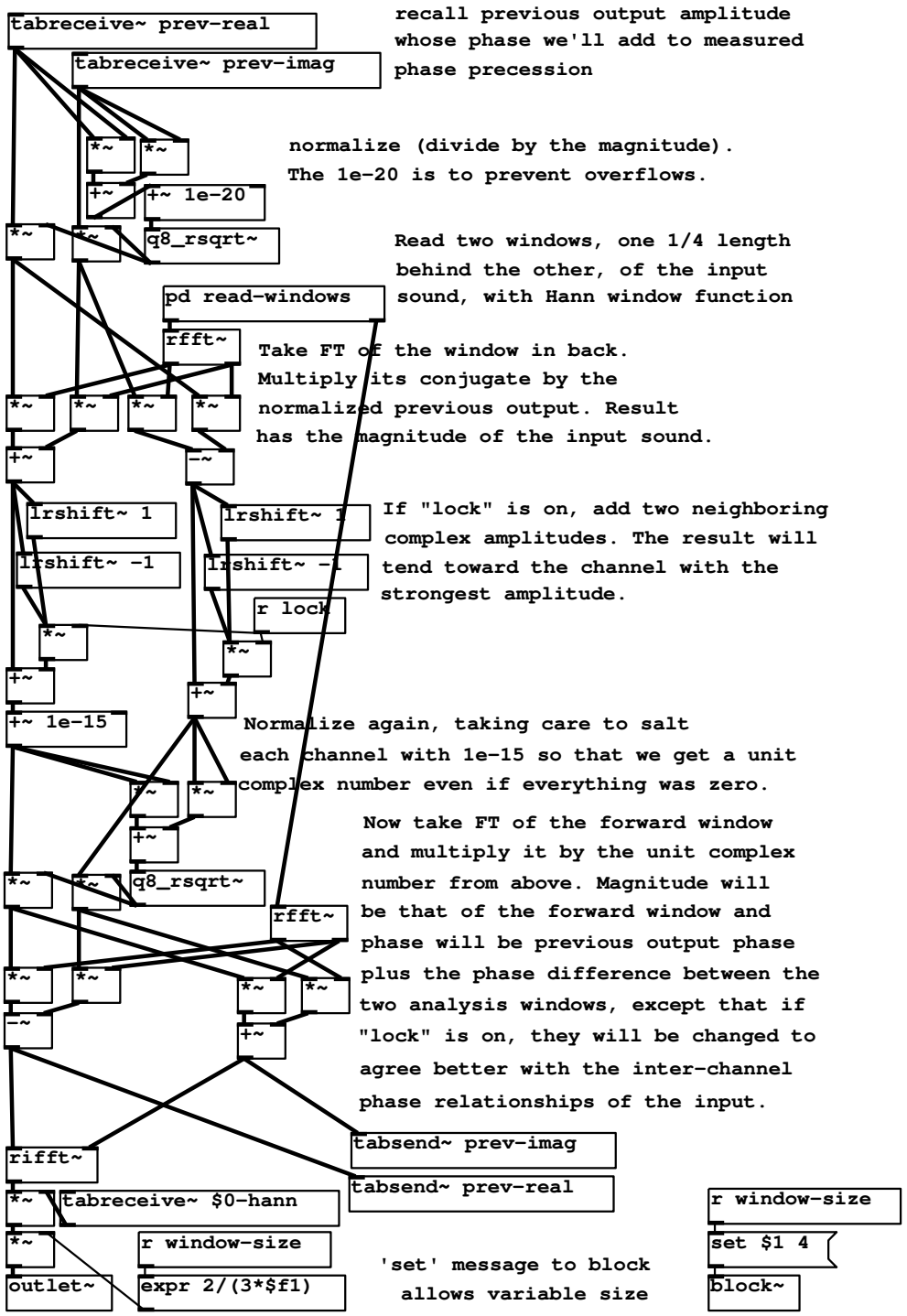


Figure 9.18: Phase vocoder for time stretching and contraction.

giving the product

$$R[k] \;=\; \frac{S[m-1,k]}{T[k] \;\cdot\; |S[m-1,k]|}$$

Next, depending on the value of the parameter "lock", the computed value of $R[k]$ is conditionally replaced with the phase-locking version $R'[k]$. This is done using `lrshift` objects, whose outputs are added into $R[k]$ if `lock` is set to one, or otherwise not if it is zero. The result is then normalized and multiplied by the Hann-windowed Fourier transform of the "front" window ($T'[k]$) to give $S[m,k]$.

Three other applications of Fourier analysis/resynthesis, not pictured here, are provided in the Pd examples. First, patch I08.pvoc.reverb.pd shows how to make a phase vocoder whose output recirculates as in a reverberator, except that individual channels are replaced by the input when it is more powerful than what is already recirculating. The result is a more coherent-sounding reverberation effect than can be made in the classical way using delay lines.

Patch I09.sheep.from.goats.pd demonstrates the (imperfect) technique of separating pitched signals from noisy ones, channel by channel, based on the phase coherence we should expect from a Hann-windowed sinusoid. If three adjacent channels are approximately π radians out of phase from each other, they are judged to belong to a sinusoidal peak. Channels belonging to sinusoidal peaks are replaced with zero to extract the noisy portion of the signal, or all others are replaced with zero to give the sinusoidal portion.

Patch I10.phase.bash.pd returns to the wavetable looping sampler of Figure 2.7, and shows how to align the phases of the sample so that all components of the signal have zero phase at points 0, N, $2N$, and so on. In this way, two copies of a looping sampler placed N samples apart can be coherently cross-faded. A realistic, pitched version of the original soundfile can be made using daisy-chained cross-fades.

Exercises

1. A signal $x[n]$ is 1 for $n = 0$ and 0 otherwise (an impulse). What is its Fourier transform as a function of k?

2. For what integer values of k is the Fourier transform of the N-point Hann window function nonzero?

3. Suppose an N-point Fourier transform is done on a complex sinusoid of frequency 2.5ω where $\omega = 2\pi/N$ is the fundamental frequency. What percentage of the signal energy lands in the main lobe, channels $k = 2$ and $k = 3$? If the signal is Hann windowed, what percentage of the energy is now in the main lobe (which is then channels 1 through 4)?

Chapter 10

Classical waveforms

Up until now we have primarily taken two approaches synthesizing repetitive waveforms: wavetable synthesis (Chapter 2) and waveshaping (Chapters 5 and 6). This chapter introduces a third approach, in which waveforms are built up explicitly from line segments with controllable endpoints. This approach is historically at least as important as either of the other two, and was dominant during the analog synthesizer period, approximately 1965-1985. For lack of a better name, we'll use the term *classical waveforms* to denote waveforms composed of line segments.

The waveforms include the *sawtooth*, *triangle*, and *rectangle* waves pictured in figure reffig10.01, among many other possibilities. As we will see, the salient features of these waveforms are their discontinuities, either jumps (changes in value) or corners (changes in slope). In the figure, the sawtooth and rectangle waves have changes in value (once per cycle for the sawtooth, and twice for the rectangle), but the slope is constant (negative for the sawtooth wave, zero for the rectangle wave). The triangle wave has no discontinuous jumps, but the slope changes discontinuously.

To use classical waveforms effectively, it is usefile to understand how the shape of the waveform is reflected in its Fourier series. (To compute them we need materials from Chapter 9, which is why this chapter appears here and not earlier.) We will also need strategies for digitally synthesizing classical waveforms. These waveforms will prove to be much more susceptible to foldover problems than any we have treated before, so we will have to pay especially close attention to its control.

In general, our strategy for dealing with sampling effects will be to consider first those sampled waveforms whose period is an integer N. Then if we want to obtain a waveform of a non-integral period (call it τ, say) we approximate τ as a quotient N/R of two integers. Conceptually at least, we can then synthesize the desired waveform with period N, and then take only one of each R samples of output. This last, down-sampling step is where the foldover is produced, and a careful analysis of it will help us control it.

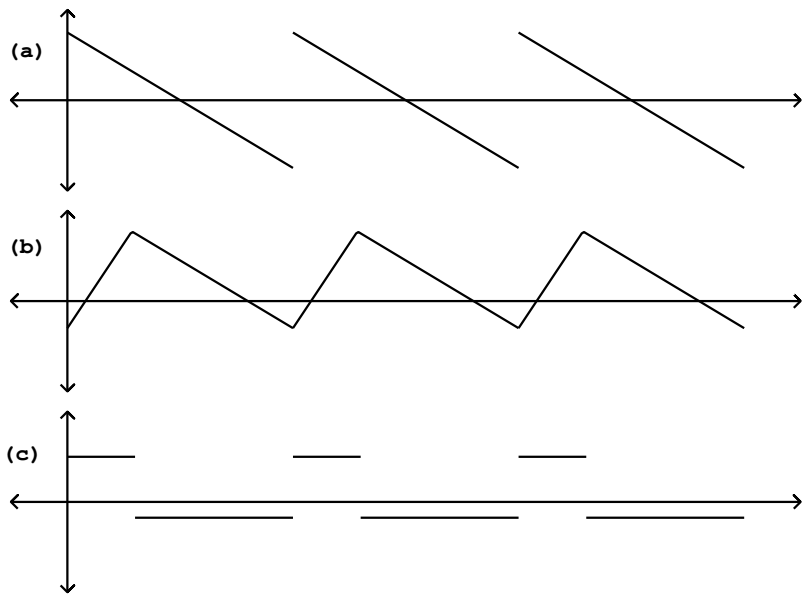


Figure 10.1: Classical waveforms: (a) the sawtooth, (b) the triangle, and (c) the rectangle waves, shown as functions of a real parameter (not sampled).

10.1 Symmetries and Fourier series

Before making a quantitative analysis of the Fourier series of the classical waveforms, we pause to make two useful observations about symmetries in waveforms and the corresponding symmetries in the Fourier series. First, a Fourier series may consist only of even or odd-numbered harmonics; this will be reflected in symmetries comparing a waveform to its displacement by half a cycle. Second, the Fourier series may contain only real- or pure imaginary-valued coefficients (corresponding to the cosine or sine functions). This is reflected in symmetries comparing the waveform to its reversal in time.

In this section we will assume that our waveform has an integer period N, and furthermore, for simplicity, that N is even (if it isn't we can just up-sample by a factor of two). We know from chapter 9 that any (real or complex valued) waveform $X[n]$ is equal to its Fourier series (whose coefficients we'll denote by $Y[k]$):

$$X[n] = Y[0] + Y[1]U^n + \cdots + Y[N-1]U^{(N-1)n}$$

or, equivalently,

$$X[n] = Y[0] + Y[1](\cos(\omega n) + i\sin(\omega n)) + \cdots$$

$$+Y[N-1](\cos(\omega(N-1)n) + i\sin(\omega(N-1)n))$$

where $\omega = 2\pi/N$ is the fundamental frequency of the waveform, and $U = \cos(\omega) + i\sin(\omega)$ is the unit-magnitude complex number whose argument is ω.

To analyze the first symmetry we delay the signal $X[n]$ by a half-cycle. Since $U^{N/2} = -1$ we get:

$$X[n+N/2] = Y[0] - Y[1]U^n + Y[2]U^{2n} \pm \cdots$$

$$+Y[N-2]U^{(N-2)n} - Y[N-1]U^{(N-1)n}$$

We combine this with the original series in two different ways. Letting X' denote half the sum of the two:

$$X'[n] = \frac{X[n] + x[n+N/2]}{2} = Y[0] + Y[2]U^{2n} + \cdots + Y[N-2]U^{(N-2)n}$$

and X'' the difference:

$$X''[n] = \frac{X[n] - x[n+N/2]}{2} = Y[1]U^n + Y[3]U^{3n} + \cdots + Y[N-1]U^{(N-1)n}$$

we see that X' consists only of even-numbered harmonics (including DC) and X'' only of odd ones.

Furthermore, if X happens to equal itself shifted a half cycle, that is, if $X[n] = X[n+N/2]$, then looking at the definition of X' we get $X'[n] = X[n]$ and $X''[n] = 0$. This implies that, in this case, $X[n]$ has only even numbered harmonics. Indeed, this should be no surprise, since in this case $X[n]$ would have to repeat every $N/2$ samples, so its fundamental frequency is twice as high than in general for period N.

In the same way, if $X[n] = -X[n+N/2$, then X can have only odd-numbered harmonics. This allows us easily to split any desired waveform into its even- and odd-numbered harmonics. (This is equivalent to using a comb filter to extract even or odd harmonics; see Chapter 7.)

For the second symmetry we compare $X[n]$ with its time reversal, $X[-n]$ (or. equivalently, since X repeats every N samples, with $X[N-n]$). The Fourier series becomes:

$$X[-n] = Y[0] + Y[1](\cos(\omega n) - i\sin(\omega n)) + \cdots$$

$$+Y[N-1](\cos(\omega(N-1)n) - i\sin(\omega(N-1)n))$$

(since the cosine function is even and the sine function is odd). In the same way as before we can extract the cosines by forming $X'[n]$ as half the sum:

$$X'[n] = \frac{X[n] + X[-n]}{2} = Y[0] + Y[1]\cos(\omega n) + \cdots + Y[N-1]\cos(\omega(N-1)n)$$

and $X''[n]$ as half the difference over i:

$$X'[n] = N\frac{X[n] + X[-n]}{2i} = Y[1]\sin(\omega n) + \cdots + Y[N-1]\sin(\omega(N-1)n)$$

So if $X[n]$ satisfies $X[-n] = X[n]$ the sinusoidal components are all cosines; and if $X[-n] = -X[n]$ they are all sines; and as before we can decompose any $X[n]$ (that repeats every N samples) into the two.

10.1.1 Sawtooth waves and symmetry

As an example, we apply the shift symmetry (even and odd harmonics) to a sawtooth wave. Figure 10.2 (part a) shows the original sawtooth wave and part b shows the result of shifting by a half cycle. The sum of the two (part c) drops discontinuously whenever either one of the two copies does so, and traces a line segment whenever both component sawtooth waves do; so it in turn becomes a sawtooth wave, of half the original period (twice the fundamental frequency). The difference of the two sawtooth waves has slope zero except for the discontinuities. The discontinuities coming from the original sawtooth wave jump in the same direction (negative to positive), but those coming from the shifted one are negated and jump from positive to negative. The result is a *square wave*, a special case of the rectangle wave where the two component segments have the same duration.

This symmetry was used to great effect in the design of Buchla analog synthesizers in the 1970s; instead of offering a single sawtooth generator, Buchla designed an oscillator which outputs the even and odd harmonic portions separately, so that cross-fading between the two allows a continuous control over the relative strengths of the even and odd harmonics in the analog waveform.

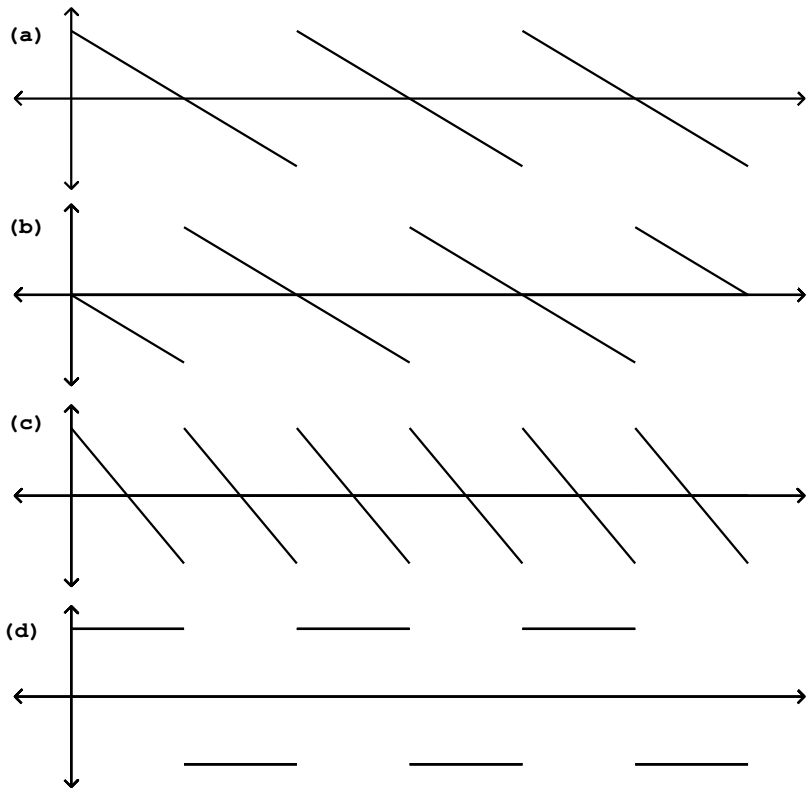


Figure 10.2: Using a symmetry relation to extract even and odd harmonics from a sawtooth wave: (a) the original sawtooth wave; (b) shifted by 1/2 cycle; (c) their sum (another sawtooth wave at twice the frequency); (d) their difference (a square wave).

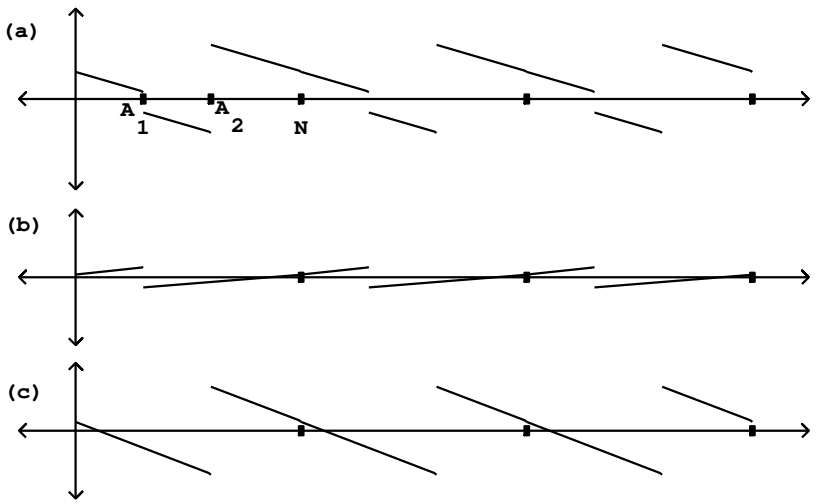


Figure 10.3: Decomposing a waveform with two discontinuities (a) as a sum of two sawtooth waves two sawtooth waves (b and c).

10.2 Decomposing the classical waveforms into sawtooth and parabolic waves

Among the several conclusions we can draw from the even/odd harmonic decomposition of the sawtooth wave (Figure 10.2), one is that a square wave can be decomposed into a linear combination of two sawtooth waves. We can carry this idea further, and show how to compose any classical waveform having only jumps (discontinuities in value) but no corners (discontinuities in slope) as a sum of sawtooth waves of various phases and amplitudes. We then develop the idea further, showing how to generate waveforms with corners (either in addition to, or instead of, jumps) using another fundamental waveform we'll call the *parabolic wave*.

Suppose first that a waveform of period N has discontinuities at j different points, $A_1, \ldots, A_j$, all lying on the cycle between 0 and N, at which the waveform jumps by values $d_1, \ldots, d_j$. A negative value of d_1, for instance, would mean that the waveform jumps from a higher to a lower value at the point A_1, and a positive value of d_1 would mean a jump from a lower to a higher value.

For instance, Figure 10.3 shows (part a) a classical waveform with two jumps: $(A_1, d_1) = (0.3, -0.3)$ and $(A_2, d_2) = (0.6, 1.3)$. Parts (b) and (c) show sawtooth waves which each has one of the two jumps. The sum of the two sawtooth waves reconstructs the waveform of part (a), except for a possible constant (DC) offset.

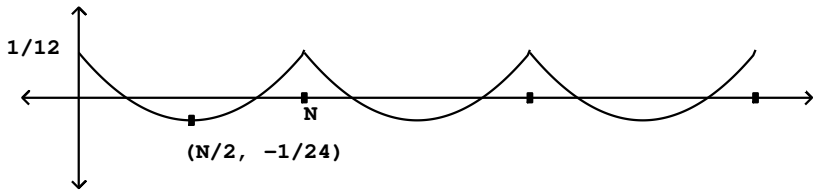


Figure 10.4: The parabolic wave.

The sawtooth wave with a jump of one unit at the point zero is given by

$$s[n] = n/N - 1/2$$

over the period $0 \leq n \leq N-1$, and repeats for other values of n. A sawtooth wave with a jump at A, d is given by $s'[n] = ds[n-A]$. The sum of all the sawtooth waves necessary would be:

$$x[n] = d_1 s[n - A_1] + \cdots + d_j s[n - A_j]$$

The slopes of the segments of the classical waveform of part (a) are all equal, and equal the sum of the slopes of the component sawtooth waves, equal in general to

$$-\frac{d_1 + \cdots + d_j}{N}$$

The square and rectangle waves have horizontal line segments (slope zero); for this to happen in general the jumps must add to zero: $d_1 + \cdots + d_j = 0$.

To decompose classical waveforms with corners we use the parabolic wave, which, over a single period from 0 to N, is equal to

$$p[n] = (\frac{n}{N} - \frac{1}{2})^2/2 - 1/24$$

as shown in Figure 10.4. It is a second-order (quadratic) polynomial in the variable n, arranged so that it reaches a maximum halfway through the cycle at $n = N/2$, the DC component is zero (or in other words, the average value over one cycle of the waveform is zero), and so that the slope changes discontinuously by $-1/N$ at the beginning of the cycle.

To construct a waveform with any desired number of corners (suppose they are at the points $B_i, \ldots, B_l$, at which we want the slope changes to equal $c_1, \ldots, c_l$), we sum up the necessary parabolic waves:

$$x[n] = -Nc_1 p[n - B_1] - \cdots - Nc_l p[n - B_l]$$

An example is shown graphically in Figure 10.5.

If the sum $x[n]$ is to contain line segments (not segments of curves), the n^2 terms in the sum must sum to zero. From the expansion of $x[n]$ above, this

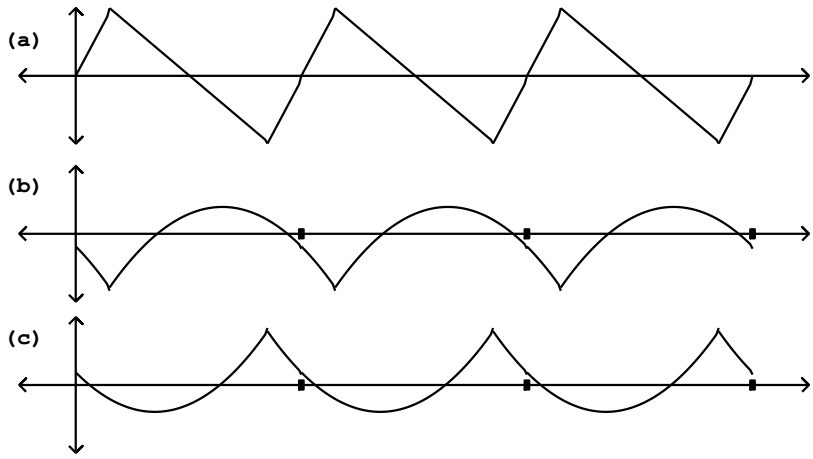


Figure 10.5: Decomposing a triangle wave (part a) into two parabolic waves (b and c).

implies that $c_1 + \cdots + c_l = 0$. Analyses of complex waveforms (as in the figure) will always satisfy this condition because the changes in slope, over a cycle, must all add to zero for the waveform to connect with itself.

10.3 Fourier series of the elementary waveforms

In general, given a repeating waveform $X[n]$, we can evaluate its Fourier series coefficients $Y[k]$ by directly evaluating the Fourier transform:

$$Y[k] = \frac{1}{N}\mathcal{FT}\{X[n]\}(k)$$

$$= \frac{1}{N}\left[X[0] + U^{-k}X[1] + \cdots + U^{-(n-1)k}X[n-1]\right]$$

but doing this directly for sawtooth and parabolic waves will require massive algebra (or somewhat less if we resort to differential calculus, which would require more mathematics background). Instead, we rely on properties of the Fourier transform to relate the transform of a signal $x[n]$ with its *first difference*, defined as $x[n] - x[n-1]$. The first difference of the parabolic wave will turn out to be a sawtooth, and that of a sawtooth will be simple enough to evaluate directly, and thus we'll get the desired Fourier series.

In general, to evaluate the strength of the kth harmonic, we'll make the assumption that N is much larger than k, or equivalently, that k/N is negligible.

We start from the Time Shift Formula for Fourier Transforms (Chapter 9.2.2) setting the time shift to one sample:

$$\mathcal{FT}\{x[n-1]\} = [\cos(k\omega) - i\sin(k\omega)]\,\mathcal{FT}\{x[n]\}$$

$$\approx (1 + i\omega k)\mathcal{FT}\{x[n]\}$$

Here we're using the fact that $\omega k = 2\pi k/N$ is much smaller than unity to make the approximations,

$$\cos(\omega k) \approx 1\ ,\ \sin(\omega k) \approx \omega k$$

which are good to within a negligible error on the order of $(k/N)^2$. Now we plug this result in to evaluate:

$$\mathcal{FT}\{x[n] - x[n-1]\} \approx i\omega k\mathcal{FT}\{x[n]\}$$

Sawtooth wave

First we apply this to the sawtooth wave $s[n]$. For $0 \le n < N$ we have:

$$s[n] - s[n-1] = -\frac{1}{N} + \begin{cases} 1 & n = 0 \\ 0 & \text{otherwise} \end{cases}$$

Ignoring the DC offset of $-\frac{1}{N}$, this gives an *impulse*, zero everywhere except one sample per cycle. The summation in the Fourier transform only has one term, and we get:

$$\mathcal{FT}\{s[n] - s[n-1]\}(k) = 1,\ k \neq 0,\ -N < k < N$$

We then apply the difference formula backward to get:

$$\mathcal{FT}\{s[n]\}(k) \approx \frac{1}{i\omega k} = \frac{-iN}{2\pi k}$$

valid for integer values of k with $k \neq 0$ and $k << N$. (To get the second form of the expression we plugged in $\omega = 2\pi/N$ and $1/i = -i$.)

This analysis doesn't give us the DC component $\mathcal{FT}\{s[n]\}(0)$, because we would have had to divide by $k = 0$. Instead, we can evaluate the DC term directly as the sum of all the points of the waveform: it's approximately zero by symmetry.

To get a Fourier series in terms of familiar real-valued sine and cosine functions, we combine corresponding terms for negative and positive values of k. The first harmonic ($k = \pm 1$) is:

$$\frac{1}{N}\left[\mathcal{FT}\{s[n]\}(1) \cdot U^n + \mathcal{FT}\{s[n]\}(-1) \cdot U^{-n}\right]$$

$$\approx \frac{-i}{2\pi}\left[U^n - U^{-n}\right]$$

$$= \frac{\sin(\omega n)}{\pi}$$

and similarly for other values of k, the kth harmonic is

$$\frac{\sin(2\omega n)}{k\pi}$$

and the entire Fourier series is:

$$s[n] \approx \frac{1}{\pi}\left[\sin(\omega n) + \frac{\sin(2\omega n)}{2} + \frac{\sin(3\omega n)}{3} + \cdots\right]$$

Parabolic wave

The same analysis, with some differences in sign and normalization, works for parabolic waves. First we compute the difference:

$$\begin{aligned} p[n] - p[n-1] &= \frac{\left(\frac{n}{N} - \frac{1}{2}\right)^2 - \left(\frac{n-1}{N} - \frac{1}{2}\right)^2}{2} \\ &= \frac{\left(\frac{n}{N} - \frac{N}{2N}\right)^2 - \left(\frac{n}{N} - \frac{N-2}{2N}\right)^2}{2} \\ &= \frac{\frac{2n}{N^2} - \frac{1}{N} + \frac{1}{N^2}}{2} \\ &\approx -s[n]/N. \end{aligned}$$

So (again for $k \neq 0$, small compared to N) we get:

$$\begin{aligned} \mathcal{FT}\{p[n]\}(k) &\approx \frac{-1}{N} \cdot \frac{-iN}{2\pi k} \cdot \mathcal{FT}\{s[n]\}(k) \\ &\approx \frac{-1}{N} \cdot \frac{-iN}{2\pi k} \cdot \frac{-iN}{2\pi k} \\ &= \frac{N}{4\pi^2 k^2} \end{aligned}$$

and as before we get the Fourier series:

$$p[n] \approx \frac{1}{2\pi^2}\left[\sin(\omega n) + \frac{\sin(2\omega n)}{4} + \frac{\sin(3\omega n)}{9} + \cdots\right]$$

Square and symmetric triangle waves

To demonstrate how to obtain Fourier series for classical waveforms in general, consider first the square wave,

$$x[n] = s[n] - s[n - \frac{N}{2}]$$

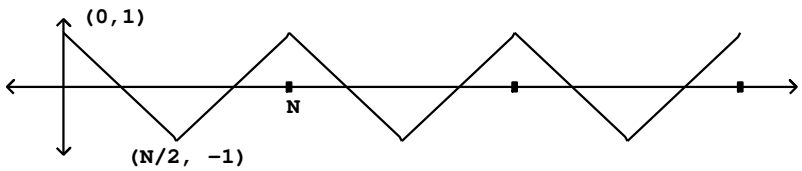


Figure 10.6: Symmetric triangle wave, obtained by superposing parabolic waves with (B, c) pairs equal to $(0, 4/N)$ and $(N/2, -4/N)$.

equal to $1/2$ for the first half cycle ($0 <= n < N/2$) and $-1/2$ for the rest. We get the Fourier series by plugging in the Fourier series for $s[n]$ twice:

$$x[n] \approx \frac{1}{\pi}\left[\sin(\omega n) + \frac{\sin(2\omega n)}{2} + \frac{\sin(3\omega n)}{3} + \cdots\right.$$

$$\left.-\sin(\omega n) + \frac{\sin(2\omega n)}{2} - \frac{\sin(3\omega n)}{3} \pm \cdots\right]$$

$$= \frac{2}{\pi}\left[\sin(\omega n) + \frac{\sin(3\omega n)}{3} + \frac{\sin(5\omega n)}{5} + \cdots\right]$$

The symmetric triangle wave (Figure 10.6) given by

$$x[n] = \frac{4}{N}p[n] - \frac{4}{N}p[n - \frac{N}{2}]$$

similarly comes to

$$x[n] \approx \frac{4}{\pi^2}\left[\cos(\omega n) + \frac{\cos(3\omega n)}{9} + \frac{\cos(5\omega n)}{25} + \cdots\right]$$

General (non-symmetric) triangle wave

A general, non-symmetric triangle wave appears in Figure 10.7. Here we have arranged the cycle so that, first, the DC component is zero (so that the two corners have equal and opposite heights), and second, so that the midpoint of the shorter segment goes through the point $(0, 0)$.

The two line segments have slopes equal to $1/A$ and $-2/(N - 2A)$, so the decomposition into component parabolic waves is given by:

$$x[n] = \frac{N^2}{AN - 2A^2}(p[n + A] - p[n - A])$$

(here we're using the periodicity of $p[n]$ to replace $p[n - (N - A)]$ by $p[n + A]$).)

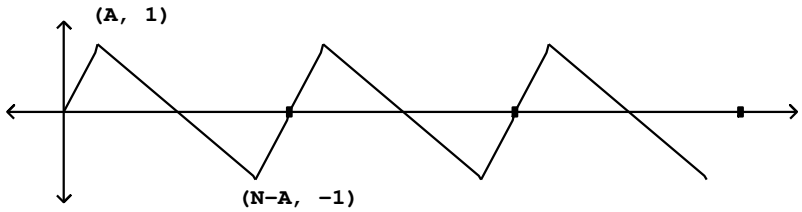


Figure 10.7: Non-symmetric triangle wave, with vertices at $(A, 1)$ and $(N - A, -1)$.

The most general way of dealing with linear combinations of elementary (parabolic and/or sawtooth) waves is to go back to the Fourier components, as we did in finding the series for the elementary waves themselves. But in this particular case we can use a trigonometric identity to avoid the extra work of converting back and forth. Just plug in the trigonometric series:

$$x[n] = \frac{N^2}{2\pi^2(AN - 2A^2)}\left[\cos(\omega(n-A)) - \cos(\omega(n+A)) + \frac{\cos(2\omega(n-A)) - \cos(2\omega(n+A))}{4} + \cdots\right]$$

Now we use the identity,

$$\cos(a) - \cos(b) = 2\sin(\frac{b-a}{2})\sin(\frac{a+b}{2})$$

so that, for example,

$$\cos(\omega(n-A)) - \cos(\omega(n+A)) = 2\sin(2\pi A/N)\sin(\omega n)$$

(Here again we used the definition of $\omega = 2\pi/N$.) This is a simplification since the first sine term does not depend on n, and hence is just an amplitude term. Applying the identity to all the terms of the expansion for $x[n]$ gives:

$$x[n] = a[1]\sin(\omega n) + a[2]\sin(2\omega n) + \cdots$$

where the amplitudes of the components are given by:

$$a[k] = \frac{1}{\pi^2(A/N - 2(A/N)^2)}\frac{\sin(2\pi kA/N)}{k^2}$$

Notice that the result does not depend separately on the values of A and N, but only on their ratio, A/N (this is not surprising because the shape of the waveform depends on this ratio). If we look at small values of k:

$$k < \frac{1}{4A/N}$$

the argument of the sine function is less than $\pi/2$ and using the approximation $\sin(\theta) \approx \theta$ we find that $a[k]$ drops off as $1/k$, just as the partials of a sawtooth wave. But for larger values of k the sine term oscillates between 1 and -1, so that the amplitudes drop off irregularly as $1/k^2$.

Figure 10.8 shows the partial strengths with A/N set to 0.03; here, our prediction is that the $1/k$ dependence should extend to $k \approx 1/(4 \cdot 0.03) \approx 8.5$, in rough agreement with the figure.

Another way to see why the partials should behave as $1/k$ for low values of k and $1/k^2$ thereafter, is to compare the period of a given partial with the length of the short segment, $2A$. For partials numbering less than $N/4A$, the period is at least twice the length of the short segment, and at that scale the waveform is nearly indistinguishable from a sawtooth wave. For partials numbering in excess of $N/2A$, the two corners of the triangle wave are at least one period apart, and at these higher frequencies the two corners (each with $1/k^2$ frequency dependence) are resolved from each other. In the figure, the notch at partial 17 occurs at the wavelength $N/2A \approx 1/17$, at which wavelength the two corners are one cycle apart; since the corners are opposite in sign they cancel each other.

10.4 Predicting and controlling foldover

Now we descend to the real situation, in which the period of the waveform cannot be assumed to be arbitrarily long and integer-valued. Suppose (for definiteness) we want to synthesize tones at 440 Hz. (A above middle C), and that we are using a sample rate of 44100 Hz, so that the period is about 100.25 samples. Theoretically, given a very high sample rate, we would expect the fiftieth partial to have magnitude 1/50 compared to the fundamental and a frequency about 20 kHz. If we sample this waveform at the (lower) sample rate of 44100, then partials in excess of this frequency will be aliased, as described in Section 3.1. The relative strength of the folded-over partials will be on the order of -32 decibels—highly audible. If the fundamental frequency is raised further, more and louder partials reach the Nyquist frequency (half the sample rate) and begin to fold over.

Foldover problems are much less pronounced for waveforms with only corners (instead of jumps) because of the faster dropoff of higher partial frequencies; for instance, a triangle wave at 440 Hz. would get twice the dropoff, or -64 decibels. In general, though, waveforms with discontinuities are a better starting point for subtractive synthesis (the most popular classical technique). And furthermore, subtractive filtering will not remove foldover once it is present in an audio signal.

10.4.1 Oversampling

As a first line of defense against foldover, we can synthesize the waveform at a much higher sample rate, apply a low-pass filter whose cutoff frequency is set to the Nyquist frequency (for the original sample rate), then downsampling. For example, in the above scenario (44100 sample rate, 440 Hz. tone) we could

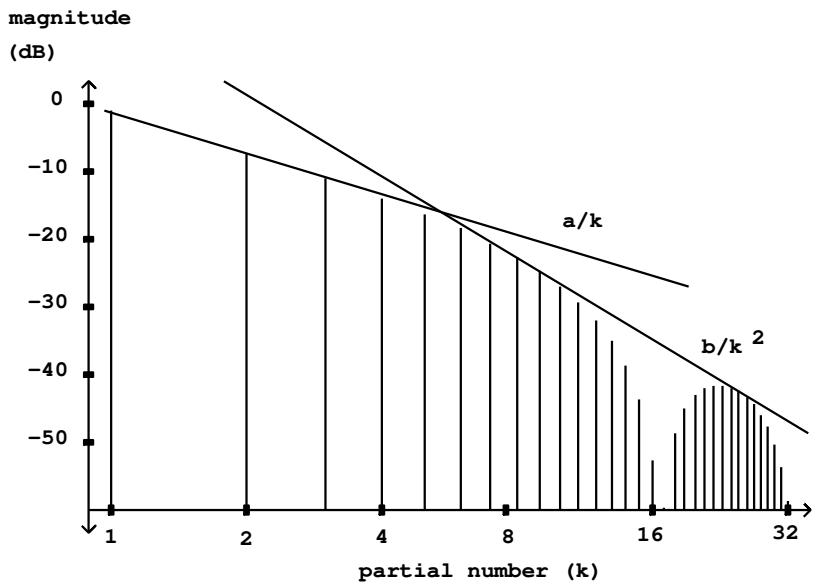


Figure 10.8: Magnitude spectrum of a triangle wave with $A/N = 0.03$. The two line segments show $1/k$ and $1/k^2$ behavior at low and high frequencies.

generate the sawtooth at a sample rate of $16 \cdot 44100 = 705600$ Hz. We would then only care about frequencies in excess of $705600 - 20000 = 685600$ Hz. (so that they fold over into audible frequencies; foldover to ultrasonic frequencies normally won't concern us) so the first problematic partial is $685600/440 = 1558$, whose amplitude is -64dB relative to that of the fundamental.

As before, this attenuation degrades by 6 dB for every octave the fundamental is raised, so that a 10 kHz. sawtooth only enjoys a 37 dB drop from the fundamental to the loudest foldover partial. On the other hand, raising the sample rate by an additional factor of two reduces foldover by the same amount. If we really wish to get 60 decibels of foldover rejection—all the way up to a 10 kHz. fundamental—we will have to oversample by a factor of 256, to a sample rate of about 11 million Hertz.

10.4.2 Sneaky triangle waves

For low fundamental frequencies, oversampling is an easy way to get adequate foldover protection. If we wish to allow higher frequencies, we will need more sophisticated approaches to the problem. One of these is to replace discontinuities by ramps, or in other words, to reduce component sawtooth waves by triangle waves, as treated in the previous section, with values of A/N small

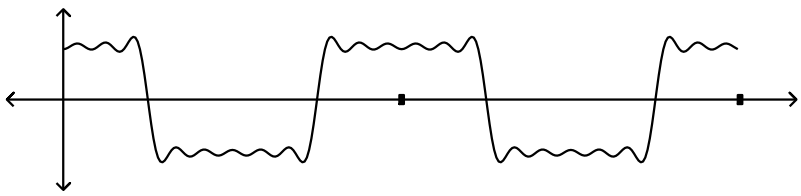


Figure 10.9: A square wave, band-limited to partials 1, 3, 5, 7, 9, and 11. This can be regarded approximately as a series of band-limited step functions arranged end to end.

enough so that the result sounds like a sawtooth wave, but large enough to control foldover.

Returning to Figure 10.8, suppose for example we imitate a sawtooth wave with a triangle wave with A equal to two samples, so that the first notch falls on the Nyquist frequency. Partials above the first notch (the 17th partial in the figure) will fold over; the worst of them is about 40 dB below the fundamental. On the other hand, the partial strengths start dropping faster than those of a true sawtooth wave at about half the Nyquist frequency. This is acceptable in some, but not all, situations.

The triangle wave strategy can be combined with oversampling to improve the situation further. Again in the context of Figure 10.8, suppose we oversample by a factor of 4, and set the first notch at the original sample rate. The partials up to the Nyquist frequency (partial 8, at the fundamental frequency shown in the figure) follow those of the true sawtooth wave fairly well. Foldover sets in only at partial number 48, and is 52 dB below the fundamental. This overall behavior holds for any fundamental frequency up to about one quarter the sample rate (after which A exceeds $N/2$). Setting the notch frequency to the original sample rate is equivalent to setting the segment of length $2A$ to two samples.

10.4.3 Transition splicing

In the point of view developed in this chapter, the energy of the spectral components of classical waves can be attributed entirely to their jumps and corners. This is artificial, of course: the energy really emanates from the entire waveform. Our derivation of the spectrum of the classical waveforms uses the jumps and corners as a bookkeeping device, and this is possible because the entire waveform is determined by their positions and magnitudes.

In this light, the problem of making band-limited versions of classical waveforms can be attacked by making band-limited versions of the jumps and corners. Since the jumps are the more serious foldover threat, we will focus on them here, although the approach described here works perfectly well for corners as well.

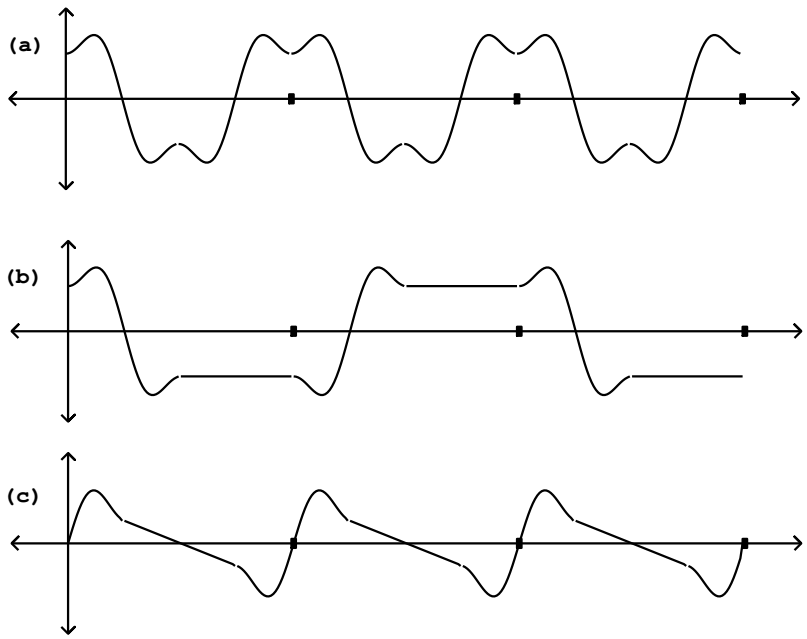


Figure 10.10: Stretching a band-limited square wave (a) to an arbitrary period: (b) by splicing in horizontal segments; (c) using the same step transition for a sawtooth wave.

To construct a band-limited step function, all we have to do is add the Fourier components of a square wave, as many as we like, and then harvest the step function at any one of the jumps. Figure 10.9 shows the partial Fourier sum corresponding to a square wave, using partials 1, 3, 5, 7, 9, and 11. The cutoff frequency can be taken as 12ω (if ω is the fundamental frequency).

If we double the period of the square wave, to arrive at the same cutoff frequency, we would add twice as many Fourier partials, up to number 23, for instance. Extending this process forever, we would eventually see the ideal band-limited step function, twice per (arbitrarily long) period.

In practice we can do quite well using only the first two partials (one and three times the fundamental). Figure 10.10, part (a), shows a two-partial approximation of a square wave. The cutoff frequency is four times the fundamental; so if the period of the waveform is eight samples, the cutoff is at the Nyquist frequency. Part (b) of the figure shows how we could use this step function to synthesize, approximately, a square wave of twice the period. If the cutoff frequency is the Nyquist frequency, the period of the waveform of part (b) is 16 samples. Each transition lasts 4 samples, because the band-limited square wave has a period of eight samples.

We can make a band-limited sawtooth wave by adding the four-sample-long

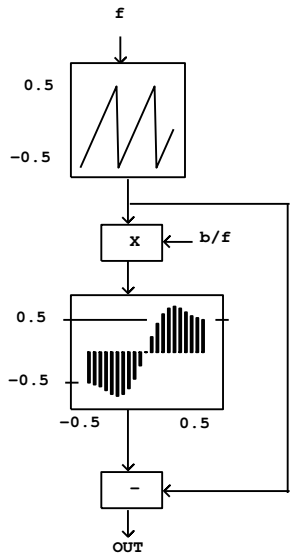


Figure 10.11: Block diagram for making a sawtooth wave with a spliced transition.

transition to a ramp function so that the end of the transition function meets smoothly with the beginning of the next one, as shown in part (c) of the figure.

There is one transition per period, so the period must be at least four samples: the highest fundamental frequency we can synthesize this way is half the Nyquist frequency. For this or lower fundamental frequency, the foldover products all turn out to be 60 dB or lower than the strength of the fundamental.

Figure 10.11 shows how to generate a sawtooth wave with a spliced transition. The two parameters are f, the fundamental frequency, and b, the band limit, assumed to be at least as large as f. We start with a digital sawtooth wave (a phasor) ranging from -0.5 to 0.5 in value. The transition will take place at the middle of the cycle, when the phasor crosses 0. The traversal takes a constant amount of time, $1/b$, regardless of f. The table lookup is taken to be non-wraparound, so that inputs out of range output either -0.5 or 0.5.

At the end of the cycle the phasor discontinuously jumps from -0.5 to 0.5, but the output of the transition table jumps an equal and opposite amount, so the result is continuous. During the portion of the waveform in which the transition table is read at one or the other end-point, the output describes a straight line segment.

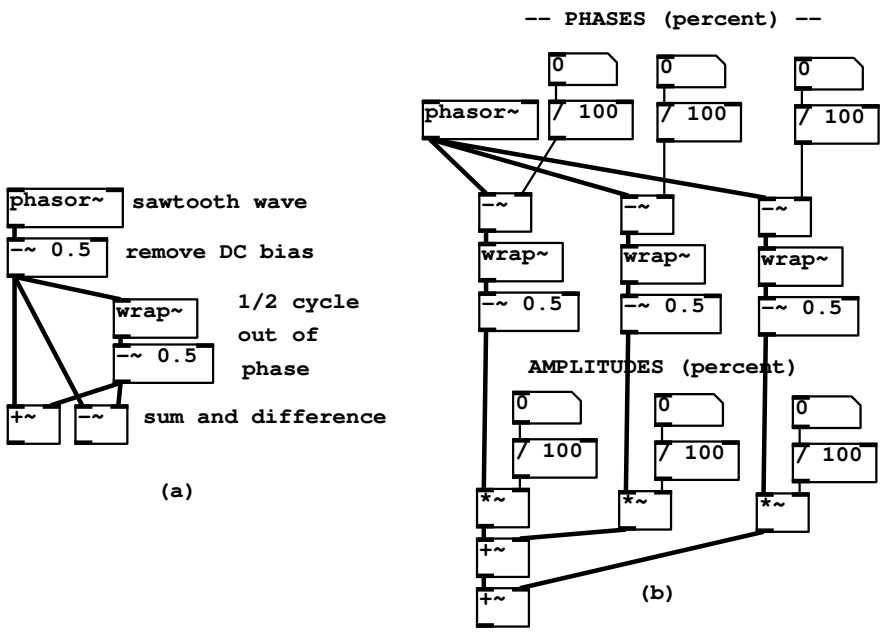


Figure 10.12: Combining sawtooth waves: (a). adding and subtracting sawtooth waves 1/2 cycle out of phase, to extract even and odd harmonics; (b). combining three sawtooth waves with arbitrary amplitudes and phases.

10.5 Examples

Combining sawtooth waves

Patch J01.even.odd.pd (Figure 10.12 part a) shows how to combine sawtooth waves in pairs to extract the even and odd harmonics. The resulting waveforms are as shown in Figure 10.3. Patch J02.trapezoids.pd (part b) demonstrates combining three sawtooth waves at arbitrary phases and amplitudes; the resulting classic waveform has up to three jumps and no corners. The three line segments are horizontal as long as the three jumps add to zero; otherwise the segments are sloped to make up for the the unbalanced jumps so that the result repeats from one period to the next.

Patch J03.pulse.width.mod.pd (not shown) combines two sawtooth waves, of opposite sign, with slightly different frequencies so that the relative phase changes continuously. Their sum is a rectangle wave whose width varies in time. This is known as pulse width modulation ("PWM").

Patch J04.corners.pd (Figure 10.13 shows how to add parabolic waves to make a combined waveform with three corners. Each parabolic wave is com-

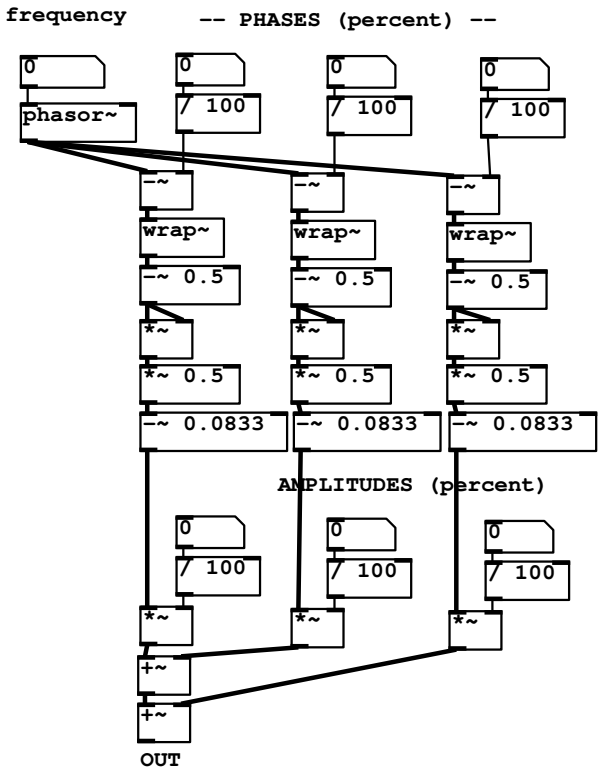


Figure 10.13: Combining parabolic waves to make a waveform with three corners.

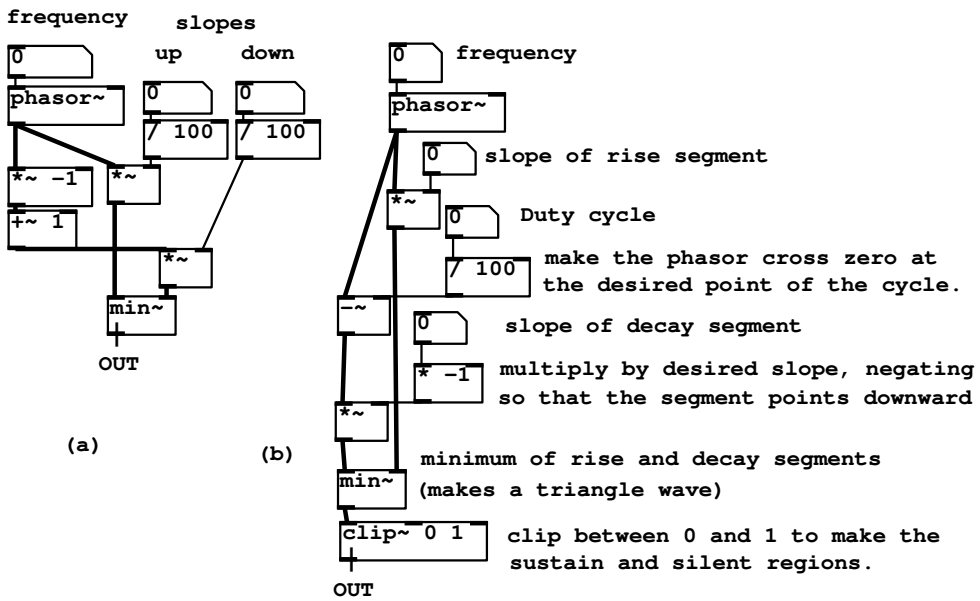


Figure 10.14: Alternative techniques for making waveforms with corners: (a), a triangle wave as the minimum of two line segments; (b). clipping a triangle wave to make an "envelope".

puted from a sawtooth wave (ranging from -0.5 to 0.5) by squaring it, multiplying by 0.5, and subtracting the DC component of -1/12, or -0.08333. The patch combines three such parabolic waves with controllable amplitudes and phases. As long as the amplitudes sum to zero, the resulting waveform consists of line segments, whose corners are located according to the three phases and have slope changes according to the three amplitudes.

Strategies for bandlimiting sawtooth waves

Patch J05.triangle.pd (Figure 10.14 part a) shows a simple way to make a triangle wave, in which only the slope of the rising and falling segment are specified. A phasor supplies the rising shape (its amplitude being the slope), and the same phasor, subtracted from one, gives the decaying shape. The minimum of the two linear functions follows the rising phasor up to the intersection of the two, and then follows the falling phasor back down to zero at the end of the cycle.

A triangle wave can be clipped above and below to make a trapezoidal wave, which can be used either as an audio-frequency pulse or, at a lower fundamental frequency, as a repeating ASR (attack/sustain/release) envelope. Patch

J06.enveloping.pd (Figure 10.14 part b) demonstrates this. The same rising shape is used as in the previous example, and the falling shape differs only in that its phase is set so that it falls to zero at a controllable point (not necessarily at the end of the cycle as before). The `clip~` object prevents it from rising above 1 (so that, if the intersection of the two segments is higher than one, we get a horizontal "sustain" segment), and also from falling below zero, so that once the falling shape reaches zero, the output is zero for the rest of the cycle.

Patch J07.oversampling.pd shows how to use upsampling to reduce foldover when using a `phasor~` object as an audio sawtooth wave. A subpatch, running at 16 times the base sample rate, contains the `phasor~` object and a three-pole, three-zero Butterworth filter to reduce the amplitudes of partials above the Nyquist frequency of the parent patch (running at the original sample rate) so that the output won't fold over when it is downsampled at the `outlet~` object. Patch J08.classicsynth.pd demonstrates using upsampled phasors as signal generators to make an imitation of a classic synthesizer doing subtractive synthesis.

Patch J09.bandlimited.pd shows how to use transition splicing as an alternative way to generate a sawtooth wave with controllable foldover. This has the advantage of being more direct (and usually less compute-intensive) than the upsampling method. On the other hand, this technique depends on using the reciprocal of the fundamental frequency as an audio signal in its own right (to control the amplitude of the sweeping signal that reads the transition table) and, in the same way as for the PAF technique of Chapter 6, care must be taken to avoid clicks if the fundamental frequency changes discontinuously.

Exercises

1. A `phasor~` object has a frequency of 441 Hz. (at a sample rate of 44100 Hz.) What is the amplitude of the DC component? The fundamental? The partial at 22050 Hz. (above which the partials fold over)?

2. A square wave oscillates between 1 and -1. What is its RMS amplitude?

3. In section 10.3 a square wave was presented as an odd waveform whose Fourier series consisted of sine (and not cosine) functions. If the square wave is advanced 1/8 cycle in phase, so that it appears as an even function, what does its Fourier series become?

4. A rectangle wave is 1 for 1/4 cycle, zero for 3/4 cycles. What are the strengths of its harmonics at 0, 1, 2, 3, and 4 times the fundamental?

Index

* ~, 19
bang ~, 292
block ~, 211
bp ~, 256
catch ~, 107
clip ~, 138
cos ~, 54
cpole ~, 261
czero$_r$ev ~, 261
czero ~, 261
dac ~, 21
delay, del, 77
delread ~, 206
delwrite ~, 206
div, 113
env ~, 84
expr, 56
fft ~, 288
fiddle ~, 137
ftom, 24
hip ~, 52, 256
inlet, 102
inlet ~, 102
line, 81
line ~, 23
loadbang, 56
lop ~, 255
lrshift ~, 294
makenote, 113
mod, 113
moses, 79
mtof, 24
noise ~, 256
notein, 86
osc ~, 19
outlet, 102
outlet ~, 102
pack, 50
pipe, 77
poly, 113
receive, 25
receive ~, 56
rfft ~, 289
rifft ~, 289
rpole ~, 261
rzero$_r$ev ~, 261
rzero ~, 261
r, 25
r ~, 56
samphold ~, 56
select, sel, 79
send, 25
send ~, 56
snapshot ~, 84
stripnote, 86
switch ~, 211
s, 25
s ~, 56
tabosc4 ~, 49

tabread4 ~, 50
tabreceive ~, 289
tabsend ~, 294
tabwrite ~, 50
throw ~, 110
trigger, t, 86
unpack, 107
until, 170
vcf ~, 256
vd ~, 208
vline ~, 81
wrap ~, 56

abstraction (Pd), 102
additive synthesis, 14
    examples, 107, 110
ADSR envelope generator, 89
aliasing, 62
amplitude, 3
    complex, 177
amplitude, measures of, 5
amplitude, peak, 5
amplitude, RMS, 5
angle of rotation, 189
argument (of a complex number), 175
arguments
    creation, 18
audio signals, digital, 3

band-pass filter, 224
bandwidth, 147, 224
beating, 25
box, 17
    GUI, 18
    message, 17
    number, 18
    object, 17

carrier frequency, 132, 149
carrier signal, 122
center frequency, 147, 224
cents, 8
Chebychev polynomials, 140
class, 17
classical waveforms, 297
clipping, 29
clipping function, 126
coloration, 192
comb filter, 182
    recirculating, 183
compander, 277
complex conjugate, 228
complex numbers, 174
compound filter, 231
continuous spectrum, 121
control, 63
control stream, 65
    numeric, 66
creation arguments, 18

decibels, 6
delay
    compound, 76
    in Pd, 77
    on control streams, 76
    simple, 76
delay network
    linear, 178
delay, audio, 178
detection
    of events, 71
digital audio signals, 3
Dirichlet kernel, 268
discrete spectrum, 119
distortion, 127
Doppler effect, 198
duty cycle, 42
dynamic, 7
dynamic range, 279

echo density, 192
envelope follower, 250
envelope generator, 10, 89
    ADSR, 89
    resetting, 96
equalization, 224
event, 65

feedback, 160

filter, 181, 221
all-pass, 249
band-pass, 224
Butterworth, 239
compound, 231
high-pass, 222
low-pass, 222
non-recirculating elementary, 227
peaking, 224
shelving, 224
first difference, 304
foldover, 62
formant, 147
Fourier analysis, 265
Fourier transform, 266
fast (FFT), 288
phase shift formula, 270
time shift formula, 270
windowed, 272
frequency domain, 179
frequency modulation, 132
frequency response, 181
frequency, angular, 3
fundamental, 14

gain, 181
granular synthesis, 37
GUI box, 18

half step, 8
Hann window function, 273
Hanning window function, 152
harmonic signal, 119
harmonics, 14
high-pass filter, 222

imaginary part of a complex number, 174
impulse, 179, 305
impulse response, 183
index (waveshaping), 126, 140
index of modulation, 132
inharmonic signal, 121
interference, 173
intermodulation, 130

Karplus-Strong synthesis, 211

logical time, 63
low-pass filter, 222

magnitide (of a complex number), 175
merging control streams, 76
in Pd, 77
message box, 17
messages, 18, 77
MIDI, 8
modulating signal, 122
modulation
frequency, 132
ring, 122
modulation frequency, 132
muting, 95

noise gate, 279
noisy spectrum, 121
non-recirculating elementary filter, 227
number box, 18
numeric control stream
in Pd, 77
Nyquist theorem, 61

object box, 17
octave, 8
oscillator, 9

parabolic wave, 302
partials, 121
passband, 222
patch, 9, 17
peaking filter, 224
period, 13
phase locking, 285
phase-aligned formant (PAF), 158
polar form (of a complex number), 175
pole-zero plot, 234
polyphony, 98
power, 6
pruning control streams, 76
in Pd, 77

quality, 247

real part of a complex number, 174
real time, 63
rectangle wave, 297
rectangular form (of a complex number), 175
reflection, 190
resynchronizing control streams, 77
    in Pd, 79
ring modulation, 122
ripple, 222

sample number, 3
sample rate, 5
sampling
    examples, 113
sawtooth wave, 30, 297
settling time, 247
shelving filter, 224
sidebands, 122
sidelobe, 269
sidelobes, 154
signals, digital audio, 3
spectral envelope, 36, 121
spectrum, 119
square wave, 300
stable delay network, 184
stopband, 224
stopband attenuation, 224
subpatches, 102
switch-and-ramp technique, 96
synthesis
    subtractive, 250

tags, 101
tasks, 98
timbre stamping, 281
timbre stretching, 41, 149
time domain, 179
time invariance, 178
time sequence, 65
toggle switch, 25
transfer function, 94, 126, 221
transient generator, 89
transition band, 224
triangle wave, 297

unit generators, 9
unitary delay network, 188

vocoder, 279
voice bank, 98

wave packet, 152
waveshaping, 126
wavetable lookup, 29
    noninterpolating, 30
wavetables
    transposition formula for looping, 35
    transposition formula, momentary, 36
window, 5, 275
window function, 272
window size, 204

Bibliography

[Bal03] Mark Ballora. *Essentials of Music Technology*. Prentice Hall, Upper Saddle River, New Jersey, 2003.

[Ble01] Barry Blesser. An interdisciplinary synthesis of reverberation viewpoints. *Journal of the Audio Engineering Society*, 49(10):867–903, 2001.

[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.

[DL97] Mark Dolson and Jean Laroche. About this phasiness business. In *Proceedings of the International Computer Music Conference*, pages 55–58, Ann Arbor, 1997. International Computer Music Association.

[GM77] John M. Grey and James A. Moorer. Perceptual evaluations of synthesized musical instrument tones. *Journal of the Acoustical Society of America*, 62:454–462, 1977.

[Har87] William M. Hartmann. Digital waveform generation by fractional addressing. *Journal of the Acoustical Society of America*, 82:1883–1891, 1987.

[III03] Julius Orion Smith III. *Mathematics of the Discrete Fourier Transform (DFT), with Music and Audio Applications*. W3K Publishing, Menlo Park, California, 2003.

[KS83] Kevin Karplus and Alex Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55, 1983.

[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.

[PB87] T. W. Parks and C.S. Burrus. *Digital filter design*. Wiley, New York, 1987.

[Puc95a] Miller S. Puckette. Formant-based audio synthesis using nonlinear distortion. *Journal of the Audio Engineering Society*, 43(1):224–227, 1995.

[Puc95b] Miller S. Puckette. Phase-locked vocoder. In *IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics*, 1995.

[Puc01] Miller S. Puckette. Synthesizing sounds with specified, time-varying spectra. In *Proceedings of the International Computer Music Conference*, pages 361–364, Ann Arbor, 2001. International Computer Music Association.

[Puc05] Miller S. Puckette. Phase bashing for sample-based formant synthesis. In *Proceedings of the International Computer Music Conference*, pages ??–??, Ann Arbor, 2005. International Computer Music Association.

[Reg93] Phillip A. Regalia. Special filter design. In Sanjit K. Mitra and James F. Kaiser, editors, *Handbook for digital signal processing*, pages 907–978. Wiley, New York, 1993.

[RM69] Jean-Claude Risset and Max V. Mathews. Analysis of musical instrument tones. *Physics Today*, 22:23–40, 1969.

[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.

[Ste96] Kenneth Steiglitz. *A Digital Signal Processing Primer*. Addison-Wesley, Menlo Park, California, 1996.

[Str85] John Strawn, editor. *Digital Audio Signal Processing*. William Kaufmann, Los Altos, California, 1985.