

Chapter 1

Sinusoids, Amplitude and Frequency

Electronic music is usually made using a computer, by synthesizing or processing *digital audio signals*. These are sequences of numbers,

$$\dots, x[n - 1], x[n], x[n + 1], \dots$$

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*. An example of a digital audio signal is the *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 $\omega n + \phi$. The initial phase is the phase at the zeroth sample ($n = 0$).

Figure 1.1 (part a) shows a sinusoid graphically. The horizontal axis shows successive values of n and the vertical axis shows the corresponding 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 an idealization of it.

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

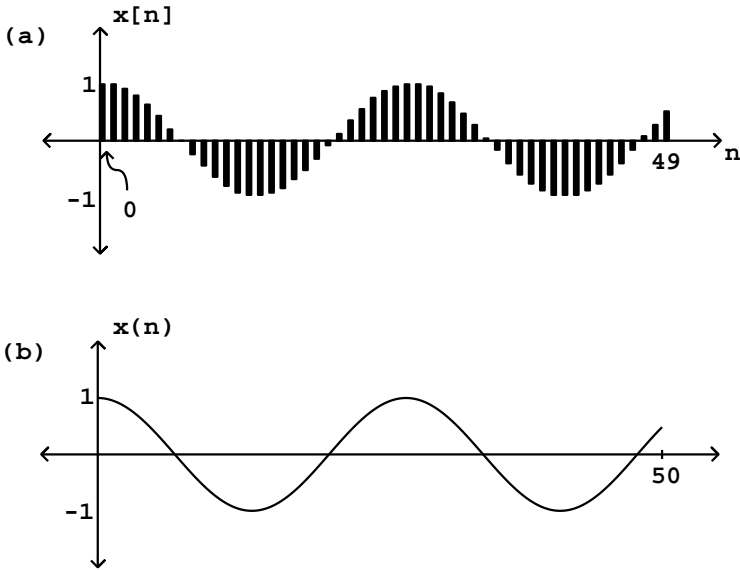


Figure 1.1: A digital audio signal, showing its discrete-time nature (part a), and idealized as a continuous function (part b). This signal is a (real-valued) sinusoid, fifty points long, with amplitude 1, angular frequency 0.24, and initial phase zero.

sounds, which is why sinusoids, and combinations of sinusoids, can be used to achieve many 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. The time t is related to the sample number n 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 Hertz (i.e., 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 [Mat69, pp. 4-10].

1.1 Measures of Amplitude

The most fundamental property of a digital audio signal is its amplitude. Unfortunately, a signal's amplitude has no one canonical definition. Strictly speaking, all the samples in a digital audio signal are themselves amplitudes, and we also spoke of the amplitude a of the sinusoid as a whole. It is useful to have measures of amplitude for digital audio signals in general. Amplitude is 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,

$$x[M], x[M + 1], \dots, x[M + N - 1]$$

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

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

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:

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

(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 of Page 1 is approximately a and its RMS

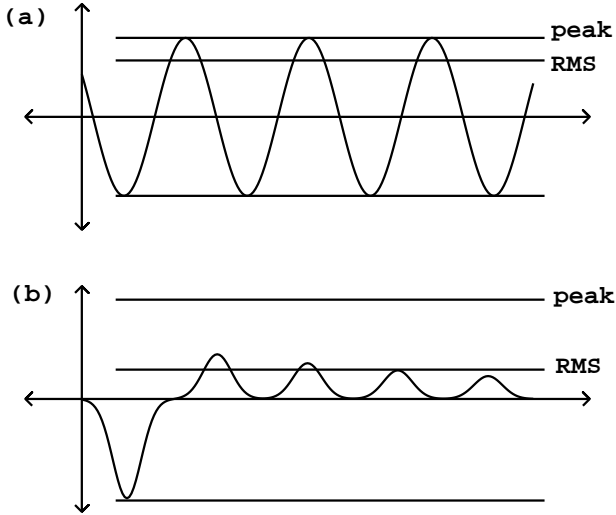


Figure 1.2: Root mean square (RMS) and peak amplitudes of signals compared. For a sinusoid (part a), the peak amplitude is higher than RMS by a factor of $\sqrt{2}$.

amplitude about $a/\sqrt{2}$. Figure 1.2 shows the peak and RMS amplitudes of two digital audio signals.

1.2 Units of Amplitude

Two amplitudes are often better compared using their ratio than their difference. Saying that one signal's amplitude is greater than another's by a factor of two might be 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 comparisons, we often express amplitudes in logarithmic units called *decibels*. If a is the amplitude of a signal (either peak or RMS), 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

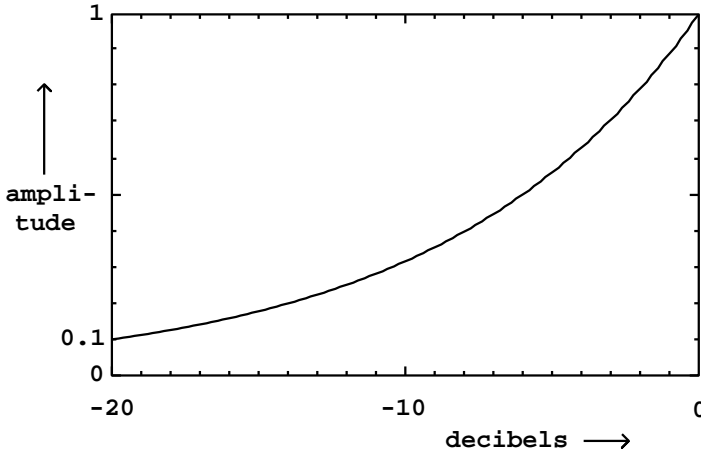


Figure 1.3: The relationship between decibel and linear scales of amplitude. The linear amplitude 1 is assigned to 0 dB.

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. The relationship between linear amplitude and amplitude in decibels is graphed in Figure 1.3.

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 level of $-\infty$.

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

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

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

Amplitude is related in an inexact way to 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 1, 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 that of $x[n]$, multiplied 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 measured 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 a MIDI pitch m and a frequency in cycles per second f , apply the Pitch/Frequency Conversion formulas:

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

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

Middle C, corresponding to MIDI pitch $m = 60$, comes to $f = 261.626$ 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]).

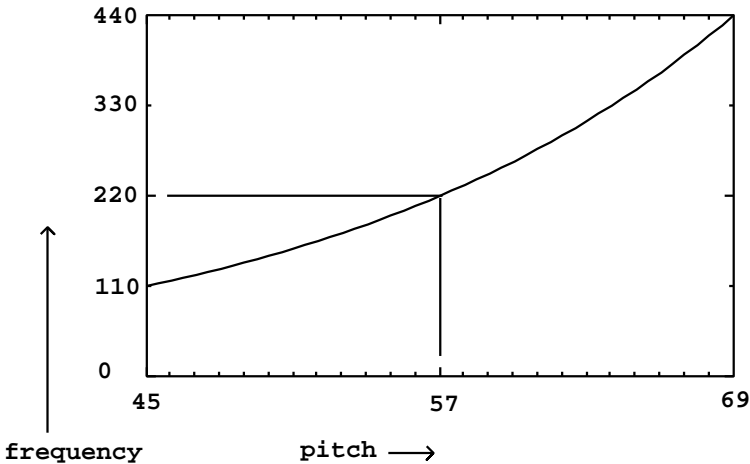


Figure 1.4: The relationship between "MIDI" pitch and frequency in cycles per second (Hertz). The span of 24 MIDI values on the horizontal axis represents two octaves, over which the frequency increases by a factor of four.

A half-step comes to a ratio of about 1.059 to 1, or about a six percent increase in frequency. Half-steps are further divided into *cents*, each cent being one hundredth of a half-step. As a rule of thumb, it might take about three cents to make a discernible 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 1.4.

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 18), we'll 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 1.5 (part a) shows a pictorial representation of a sinusoidal oscillator as an icon. The input is a frequency (in cycles per second), and the output is a sinusoid of peak amplitude one.

Figure 1.5 (part b) shows how to multiply the output of a sinusoidal oscillator by an appropriate scale factor $y[n]$ to control its amplitude. Since the oscillator's peak amplitude is 1, the peak amplitude of the product is about $y[n]$, assuming $y[n]$ changes slowly enough and doesn't become negative in value.

Figure 1.6 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 an audible "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.

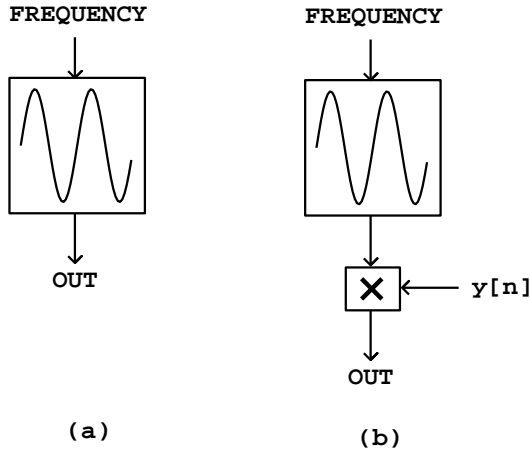


Figure 1.5: Block diagrams for (a) a sinusoidal oscillator; (b) controlling the amplitude using a multiplier and an amplitude signal $y[n]$.

For now we can state two general rules without justifying them. First, pure sinusoids are the 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 made using an *envelope generator*. Figure 1.7 shows a network in which an envelope generator is used to control the amplitude of an oscillator. Envelope generators vary widely in design, but we will focus on the simplest kind, which generates line segments as shown in Figure 1.6 (part c). 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:

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

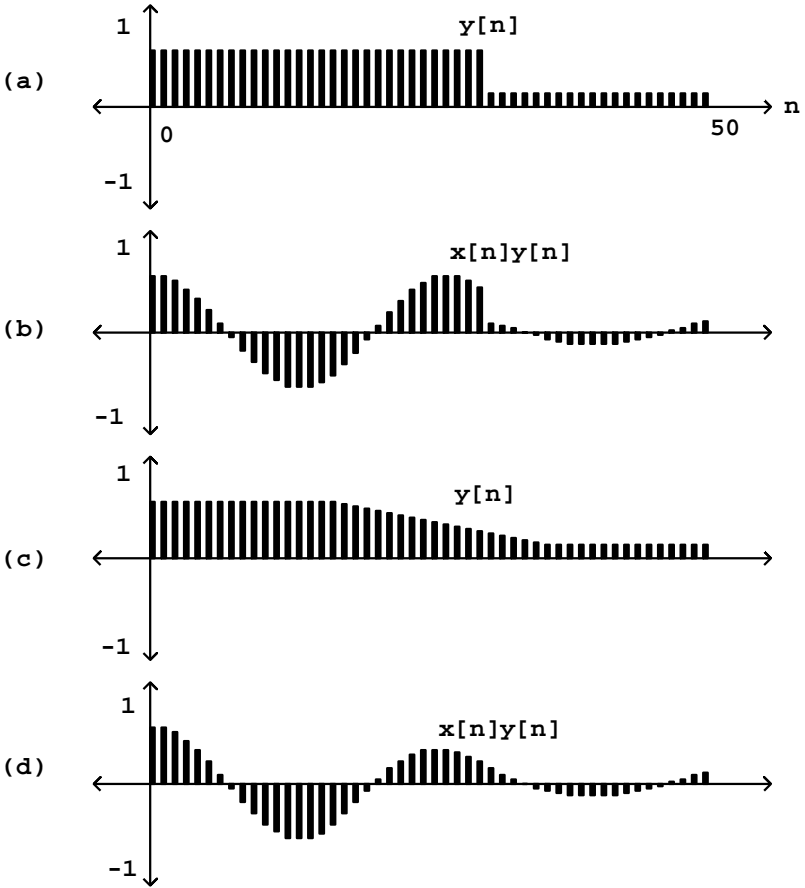


Figure 1.6: Two amplitude functions (parts a, c), and (parts b, d), the result of multiplying them by the pure sinusoid of Figure 1.1.

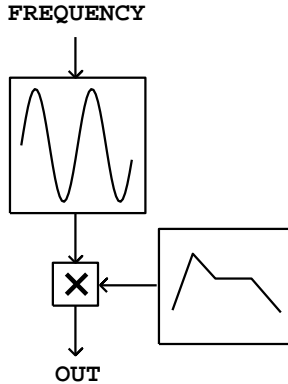


Figure 1.7: Using an envelope generator to control amplitude.

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, especially 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.

The envelope generator dates from the analog era [Str95, p.64] [Cha80, p.90], as does the rest of Figure 1.7; oscillators with controllable frequency were called voltage-controlled oscillators or VCOs, and the multiplication step was done using a voltage-controlled amplifier or VCA [Str95, pp.34-35] [Cha80, pp.84-89]. 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 $k \cdot x[n]$ (where $k \geq 0$) has amplitude kA . The mean power of the scaled signal changes by a factor of k^2 . The situation gets more complicated when two different signals are added together; just knowing the amplitudes of the two does not suffice to know the amplitude of the sum. The two amplitude measures do at least obey triangle inequalities; for any

two signals $x[n]$ and $y[n]$,

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

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

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

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

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

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

The covariance may be positive, zero, or negative. Over a sufficiently large window, the covariance of two sinusoids with different frequencies is negligible compared to the mean power. Two signals which have no covariance are called *uncorrelated* (the correlation is the covariance normalized to lie between -1 and 1). In general, for two uncorrelated signals, the power of the sum is the sum of the powers:

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

Put in terms of amplitude, this becomes:

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

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. For now 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.

A 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 repeat after $2\pi/\omega$ samples. Such a sum is called a *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.

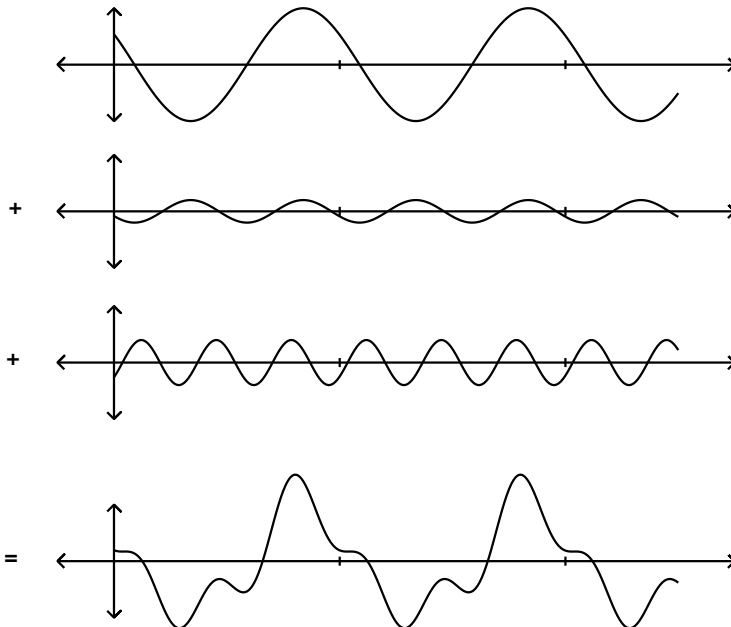


Figure 1.8: A Fourier series, showing three sinusoids and their sum. The three component sinusoids have frequencies in the ratio 1:2:3.

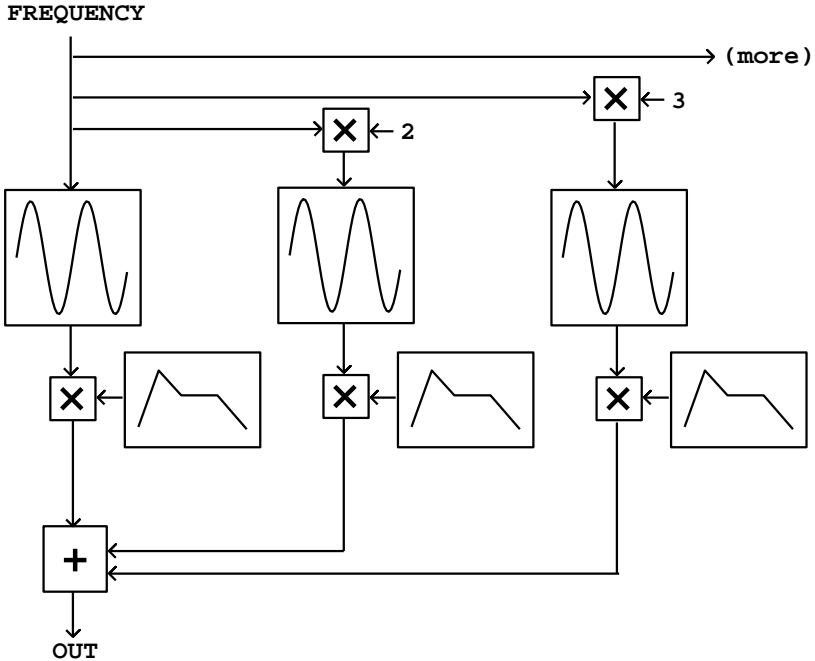


Figure 1.9: Using many oscillators to synthesize a waveform with desired harmonic amplitudes.

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 $\omega, 2\omega, \dots$ are at intervals of 0, 1200, 1902, 2400, 2786, 3102, 3369, 3600, \dots cents above the fundamental; this sequence of pitches is sometimes called the *harmonic series*. The first six of these are all quite 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 exactly 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 (a perfect fifth in Western terminology) is almost exactly seven half-steps, 4:3 (a perfect fourth) is just as near to five half-steps, and the ratios 5:4 and 6:5 (perfect major and minor thirds) are fairly close to intervals of four and three half-steps, respectively.

A Fourier series (with only three terms) is shown in Figure 1.8. The first three graphs are of sinusoids, whose frequencies are in a 1:2:3 ratio.

The common period is marked on the horizontal 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 1.9 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 rang with the sound of additive synthesis.

1.8 About the Software Examples

The examples for this book use Pure Data (Pd), and to understand them you will have to learn at least something about Pd itself. Pd is an environment for quickly realizing computer music applications, primarily intended for live music performances. Pd can be used for other media as well, but we won't go into that here.

Several other patchable audio DSP environments exist besides Pd. The most widely used one is certainly Barry Vercoe's Csound [Bou00], which differs from Pd in being text-based (not GUI-based). This 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. Genealogically, Csound belongs to the so-called Music N languages [Mat69, pp.115-172].

Another open-source 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 a more suitable tool than Csound for tasks like writing loops or maintaining complex data structures.

Finally, Pd has a widely-used sibling, 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 apply it in Max/MSP and vice versa, and even to port patches from one to the other, but the two 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, because Pd is an implementation environment, not a specification language.

A Pd patch, such as the ones shown in Figure 1.10, consists of a collection of *boxes* connected in a network. 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 a pointing device). The message in this case consists simply of the number “21”.
- an *object box*. Object boxes have a rectangular border; they interpret the text to create objects when you load a patch. Object boxes may hold 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 holds an adder. In most Pd patches, the majority of boxes are of type “object”. The first word typed into an object box specifies its *class*, which in this case is just “+”. Any additional (blank-space-separated) words appearing in the box are called *creation arguments*, which specify the initial state of the object when it is created.
- a *number box*. Number boxes are a particular type of *GUI box*. Others include push buttons and toggle switches; 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 fixed 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 Figure 1.10 (part 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 1.10 (part 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~`

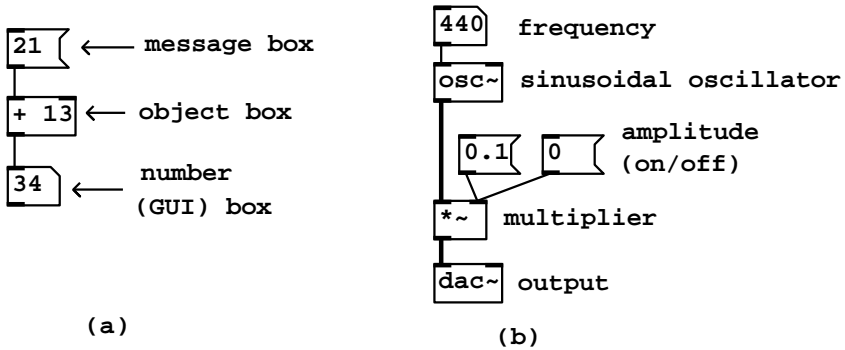


Figure 1.10: (a) three types of boxes in Pd (message, object, and GUI); (b) a simple patch to output a sinusoid.

object) 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 depends on the box the connection comes from; so, for instance, the + object outputs messages, but the *~ object outputs a signal. The inputs of a given object may or may not accept signals (but they always accept messages, even if only to convert them to signals). As a convention, object boxes with signal inputs or outputs 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 (or perhaps just the latest one) and download the archive containing the associated revision of the examples (you may also download an archive of the HTML version of this book 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 load them correctly if you have moved 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 find a different version of the examples which might not correspond to the text you're reading.

1.9 Examples

Constant amplitude scaler

Example A01.sinewave.pd, shown in Figure 1.11, 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:

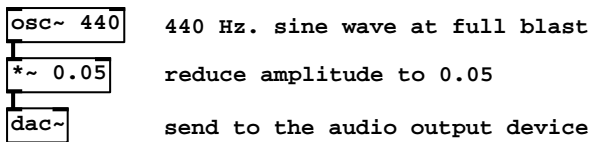
osc ~: sinusoidal oscillator. The left hand side input and the output are digital audio signals. The input is taken to be a (possibly time-varying) frequency in Hertz. The output is a sinusoid at the specified frequency. If nothing is connected to the frequency inlet, the creation argument (440 in this 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 also be sent messages which are automatically converted to the desired audio signal.

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

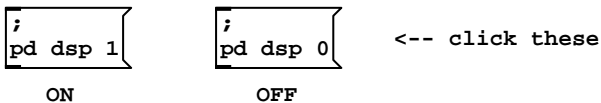
dac ~: audio output device. Depending on your hardware, this might not actually be a Digital/Analog Converter as the name suggests; but in general, it allows you to send any audio signal to your computer's audio output(s). If there are no creation arguments, the default behavior is to output to channels 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 use two or more output channels, or may not

MAKING A SINE WAVE

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



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



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

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

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

Figure 1.11: The contents of the first Pd example patch: A01.sinewave.pd.

have the audio output device open at all; consult the Pd documentation for details.

The two message boxes show a peculiarity in the way messages are parsed in message boxes. Earlier in Figure 1.10 (part a), 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 this 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”) in the Pd release, but they will only be touched on here as necessary to demonstrate the audio signal processing techniques that are the subject of this book.

Amplitude control in decibels

Example A02.amplitude.pd shows how to make a crude amplitude control; the active elements are shown in Figure 1.12 (part a). There is one new object class:

`dbtorms` : Decibels to linear 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` 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;

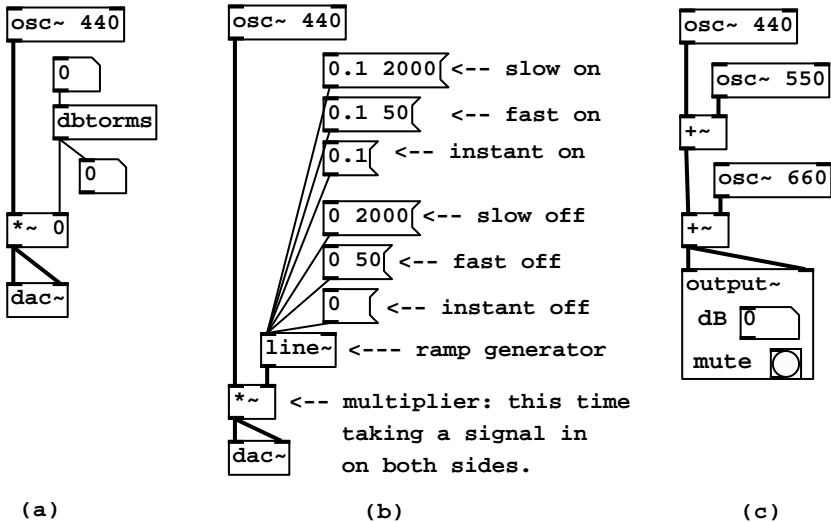


Figure 1.12: The active ingredients to three patches: (a) A02.amplitude.pd; (b) A03.line.pd; (c) A05.output.subpatch.pd.

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 may be some extra work to do.

Smoothed amplitude control with an envelope generator

As Figure 1.6 shows, one way to make smooth amplitude changes in a signal without clicks is to multiply it by the output of an envelope generator as shown in block diagram form in Figure 1.7. This may be implemented in Pd using the `line~` object:

`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 in other words, its output range should conform to whatever input it may be connected to.

Example A03.line.pd demonstrates the use of a `line~` object to control the amplitude of a sinusoid. The active part is shown in Figure 1.12

(part 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; a message with a single number such as "0.1" is always equivalent to the pair, "0.1 0". Almost any other object will retain the previous value for the right inlet, instead of resetting it to zero.

Example A04.line2.pd shows the `line~` object's output graphically. Using the various message boxes, you can recreate the effects shown in Figure 1.6.

Major triad

Example A05.output.subpatch.pd, whose active ingredients are shown in Figure 1.12 (part 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 opening the properties menu for 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 best 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

Example A06.frequency.pd (Figure 1.13) 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`: convert MIDI pitch to frequency units according to the Pitch/Frequency Conversion Formulas (Page 7). Inputs and outputs are messages ("tilde" equivalents of the two also exist, although like `dbtorms~` they're expensive in CPU time). The `ftom` object's output is -1500 if 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 (described 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. Earlier 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 the number to its output. However, a number box

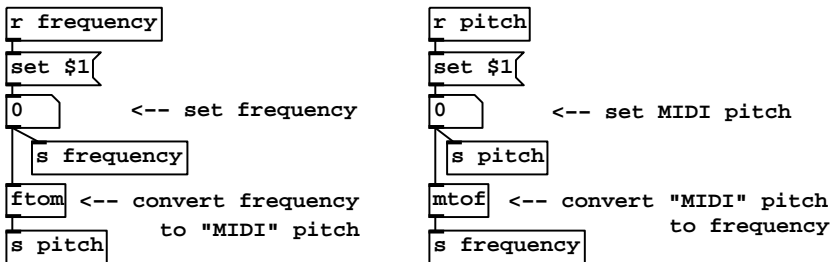


Figure 1.13: Conversion between pitch and frequency in A06.frequency.pd.

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 `receive` objects 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 broke.

More additive synthesis

The major triad (Example A06.frequency.pd, Page 22) shows one way to combine several sinusoids together by summing. There are many other possible ways to organize collections of sinusoids, of which we’ll show two. Example A07.fusion.pd (Figure 1.14) 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).

The second, third, and fourth oscillators 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 1 and 0 alternately when clicked

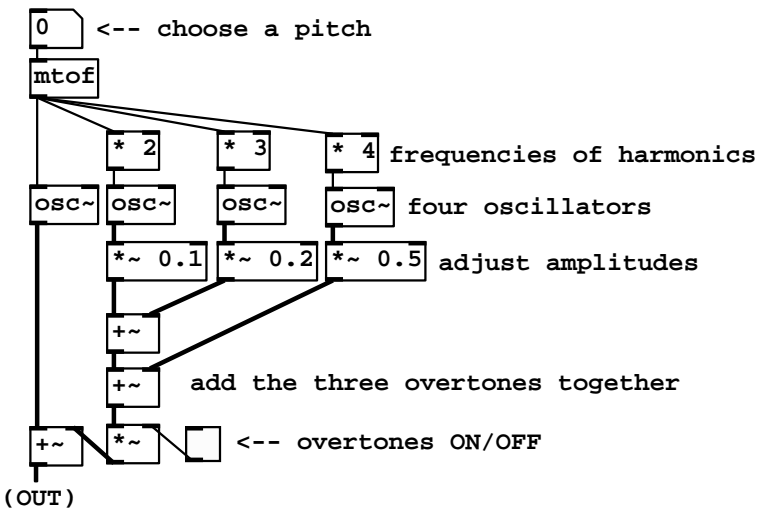


Figure 1.14: Additive synthesis using harmonically tuned oscillators.

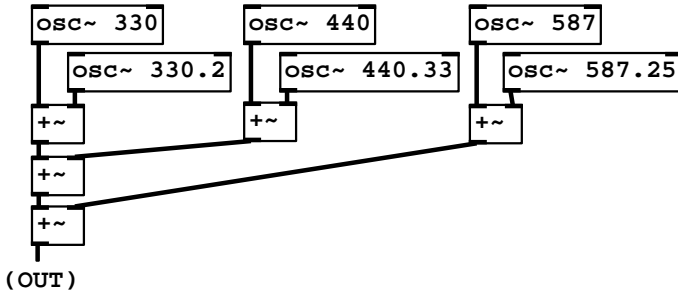


Figure 1.15: Additive synthesis: six oscillators arranged into three beating pairs.

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.

Example A08.beating.pd (Figure 1.15) shows another possibility, in which six oscillators are tuned into three pairs of neighbors, for instance 330 and 330.2 Hertz. These pairs slip into and out of phase with each other, so that the amplitude of the sum changes over time. Called *beating*, this phenomenon is frequently used for musical effects.

Oscillators may be combined in other ways besides simply summing their output, and a wide range of resulting sounds is available. Example 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. A sinusoid (Page 1) has initial phase $\phi = 0$ and angular frequency $\omega = \pi/10$. What is its period in samples? What is the phase at sample number $n = 10$?
2. Two sinusoids have periods of 20 and 30 samples, respectively. What is the period of the sum of the two?

3. If 0 dB corresponds to an amplitude of 1, how many dB corresponds to amplitudes of 1.5, 2, 3, and 5?
4. Two uncorrelated signals of RMS amplitude 3 and 4 are added; what's the RMS amplitude of the sum?
5. How many uncorrelated signals, all of equal amplitude, would you have to add to get a signal that is 9 dB greater in amplitude?
6. What is the angular frequency of middle C at 44100 samples per second?
7. Two sinusoids play at middle C (MIDI 60) and the neighboring C sharp (MIDI 61). What is the difference, in Hertz, between their frequencies?
8. How many cents is the interval between the seventh and the eighth harmonic of a periodic signal?
9. If an audio signal $x[n], n = 0, \dots, N - 1$ has peak amplitude 1, what is the minimum possible RMS amplitude? What is the maximum possible?