length 120ms. At the same time all `tabwrite~` objects are triggered, so the graphs are synchronised. All curves take the same amount of time to reach zero, but as more squaring operations are added, raising the input to higher powers, the faster the curve decays during its initial stage.

SECTION 13.2

Periodic Functions

A periodic function is bounded in range for an infinite domain. In other words, no matter how big the input value, it comes back to the place it started from and repeats that range in a loop.

Wrapping Ranges

The `wrap~` object provides just such a behaviour. It is like a signal version of `mod`. If the input a to `wrap~` exceeds 1.0 then it returns $a - 1.0$. And if the input exceeds 2.0 it gives us $a - 2.0$. Wrap is the "fractional" part of a number in relation to a division, in this case the unit 1, $a - \lfloor a \rfloor$. Let's say we have a normalised phasor which is cycling up once per second. If we pass it through `wrap~` it will be unaffected. A normalised phasor never exceeds 1.0 and so passes through unchanged. But if we double the amplitude of the phasor by multiplying by 2.0 and then wrap it, something else happens, as seen in figure 13.12.

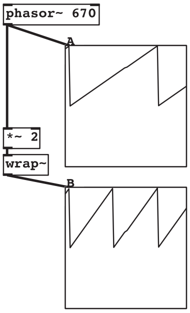

Figure 13.12
Wrapping.

Imagine the graph of a in a range of 0.0 to 2.0 is drawn on tracing paper, and then the paper is cut into two strips of height 1.0 which are placed on top of one another. Each time the phasor passes 1.0 it is wrapped back to the bottom. Consequently the frequency doubles but its peak amplitude stays at 1.0. This way we can create periodic functions from a steadily growing input, so a line that rises at a constant rate can be turned into a phasor with `wrap~`. Even more useful, we can obtain an exact number of phasor cycles in a certain time period by making the line rise at a particular rate. The `vline~` in figure 13.13 moves from 0.0 to 1.0 in 10ms. Multiplying by 3 means it moves from 0.0 to 3.0 in 10ms, and wrapping it produces three phasor cycles in a period of $10/3 = 3.333$ms, giving a frequency of $1/3.333 \times 1000 = 300$Hz.

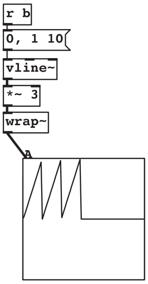

Figure 13.13
Wrapping a line.

Cosine Function

The reason for saying that the phasor is the most primitive waveform is that even a cosinusoidal oscillator can be derived from it. Notice in figure 13.14 that although the phasor is always positive in the range 0.0 to 1.0 (unipolar), the `cos~`

operation produces a *bipolar* waveform in the range −1.0 to 1.0. One complete period of the cosine corresponds to 2π, 360°, or in rotation normalised form, 1.0. When the phasor is at 0.0 the cosine is 1.0. When the phasor is at 0.25 the cosine crosses zero moving downwards. It reaches the bottom of its cycle when the phasor is 0.5. So there are two zero crossing points, one when the phasor is 0.25 and another when it is 0.75. When the phasor is 1.0 the cosine has completed a full cycle and returned to its original position.

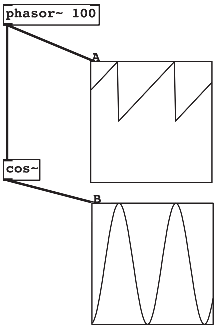

Figure 13.14
Cosine of a phasor.

SECTION 13.3

Other Functions

From time to time we will use other functions like exponentiation, raising to a variable power, or doing the opposite by taking the log of a value. In each case we will examine the use in context. A very useful technique is that arbitrary curve shapes can be formed from *polynomials*.

Polynomials

A polynomial is expressed as a sum of different power terms. The graph of $2x^2$ gives a gently increasing slope and the graph of $18x^3+23x^2-5x$ shows a simple hump weighted towards the rear which could be useful for certain kinds of sound control envelope. There are some rules for making them. The number of times the curve can change direction is determined by which powers are summed. Each of these is called a *term*. A polynomial with some factor of the a^2 term can turn around once, so we say it has one *turning point*. Adding an a^3 term gives us two turning points, and so on. The multiplier of each term is called the *coefficient* and sets the amount that term effects the shape. Polynomials are tricky to work with because it's not easy to find the coefficients to get a desired curve. The usual method is to start with a polynomial with a known shape and carefully tweak the coefficients to get the new shape you want. We will encounter some later,

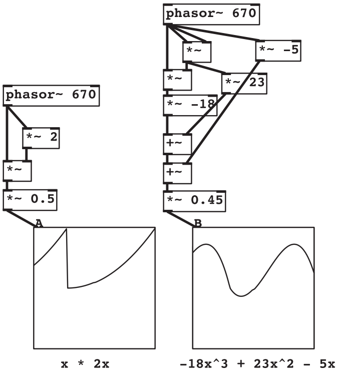

Figure 13.15
Polynomials.

like cubic polynomials, that can be used to make natural-sounding envelope curves.

Expressions

Expressions are objects with which you can write a single line of arbitrary processing code in a programmatic way. Each of many possible signal inlets x, y, z correspond to variables $\$v(x, y, z)$ in the expression, and the result is returned at the outlet. This example shows how we generate a mix of two sine waves, one 5 times the frequency of the other. The available functions are very like those found in C and follow the maths syntax of most programming languages. Although expressions are very versatile they should only be used as a last resort, when you cannot build from more primitive objects. They are less efficient than inbuilt objects and more difficult to read. The expression shown in figure 13.16 implements $Asin(2\pi\omega) + Bsin(10\pi\omega)$ for a periodic phasor ω and two mix coefficients where $B = 1 - A$. The equivalent patch made from primitives is shown at the bottom of figure 13.16.

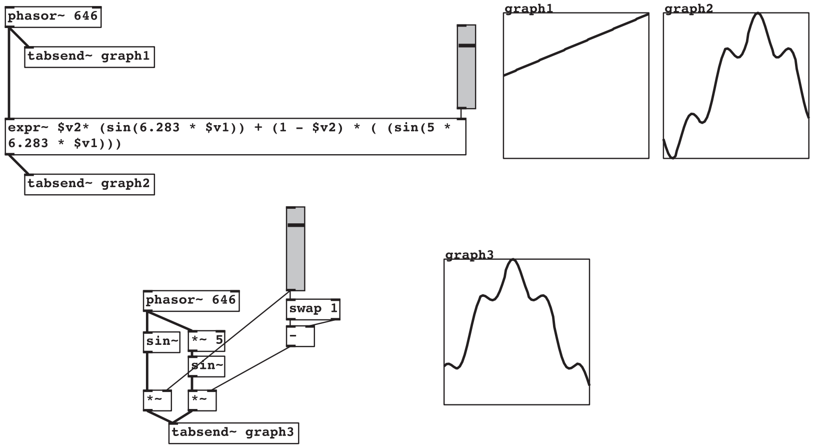

Figure 13.16
Using an expression to create an audio signal function.

SECTION 13.4

Time-Dependent Signal Shaping

So far we have considered ways to change the amplitude of a signal as a function of one or more other variables. These are all instantaneous changes which depend only on the current value of the input sample. If we want a signal to change its behaviour based on its previous features then we need to use time shaping.

Delay

To shift a signal in time we use a delay. Delays are at the heart of many important procedures like reverb, filters, and chorusing. Unlike most other Pd operations, delays are used as two separate objects. The first is a write unit that works like `send~` but sends the signal to an invisible area of memory. The second object is for reading from the same memory area after a certain time. So you always use `delwrite~` and `delread~` as pairs. The first argument to `delwrite~` is a unique name for the delay and the second is the maximum memory (as time in milliseconds) to allocate. On its own a delay just produces a perfect copy of an input signal a fixed number of milliseconds later. Here we see a 0.5ms pulse created by taking the square of a fast line from one to zero. The second graph shows the same waveform as the first but it happens 10ms later.

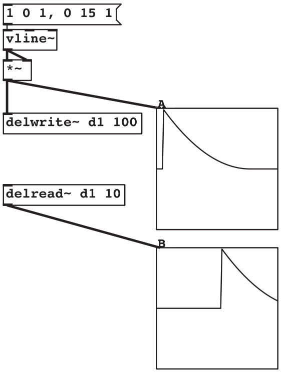

Figure 13.17
Delay.

Phase Cancellation

Assuming that two adjacent cycles of a periodic waveform are largely the same, then if we delay that periodic signal by time equal to half its period we have changed its phase by 180°. In the patch shown here the two signals are out of phase. Mixing the original signal back with a copy that is antiphase annihilates both signals, leaving nothing. In figure 13.18 a sinusoidal signal at 312Hz is sent to a delay **d1**. Since the input frequency is 312Hz its period is 3.2051ms, and half that is 1.60256ms. The delayed signal will be out of phase by half of the input signal period. What would happen if the delay were set so that the two signals were perfectly in phase? In that case instead of being zero the output would be a waveform with twice the input amplitude. For delay times between these two cases the output amplitude varies between 0.0 and 2.0. We can say for a given frequency component the output amplitude depends on the delay time. However, let's assume the delay is fixed and put it another way—for a given delay time the output amplitude depends on the input frequency. What we have created is a simple filter.

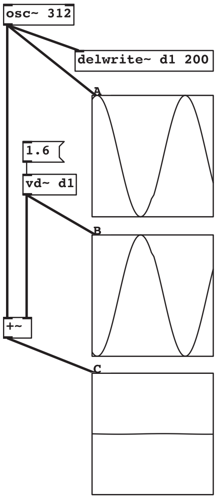

Figure 13.18
Antiphase.

Filters

When delay time and period coincide we call the loud part (twice the input amplitude) created by reinforcement a *pole*, and when the delay time equals half the period we call the quiet part where the waves cancel out a *zero*. Very basic but flexible filters are provided in Pd called `rpole~` and `rzero~`. They are tricky to set up unless you learn a little more about DSP filter theory, because the frequencies of the poles or zeros are determined by a normalised number that represents the range of 0Hz to $SR/2$Hz, where SR is the sampling rate of the patch. Simple filters can be understood by an equation governing how the output samples are computed as a function of the current or past samples. There are two kinds: those whose output depends only on past values of the *input*, which are called *finite impulse response* filters (FIR), and those whose output depends on past input values and on past *output* values. In other words, this kind has a feedback loop around the delay elements. Because the effect of a signal value could theoretically circulate forever we call this kind recursive or *infinite impulse response* filters (IIR).

User-Friendly Filters

Filters may have many poles and zeros, but instead of calculating these from delay times, sampling rates, and wave periods, we prefer to use filters designed with preset behaviours. The behaviour of a filter is determined by a built-in calculator that works out the coefficients to set poles, zeros, and feedback levels for one or more internal delays. Instead of poles and zeros we use a different terminology and talk about bands which are passed or stopped. A band has a center frequency, specified in Hz, the middle of the range where it has the most effect, and also a bandwidth which is the range of frequencies it operates over. Narrow bands affect fewer frequencies than wider bands. In many filter designs you can change the bandwidth and the frequency independently. Four commonly encountered filters are the low pass, high pass, band pass, and band cut or notch filter, shown in figure 13.19. The graphs show the spectrum of white noise after it's been passed through each of the filters. The noise would normally fill up the graph evenly, so you can see how each of the filters cuts away at a different part of the spectrum. The high pass allows more signals above its centre frequency through than ones below. It is the opposite of the low pass, which prefers low frequencies. The notch filter carves out a swathe of frequencies in the middle of the spectrum, which is the opposite of the band pass, which allows a group of frequencies in the middle through but rejects those on either side.

Integration

Another way of looking at the behaviour of filters is to consider their effect on the slope or phase of moving signals. One of the ways that recursive (IIR) filters can be used is like an accumulator. If the feedback is very high the current input is added to all previous ones. Integration is used to compute the area under a

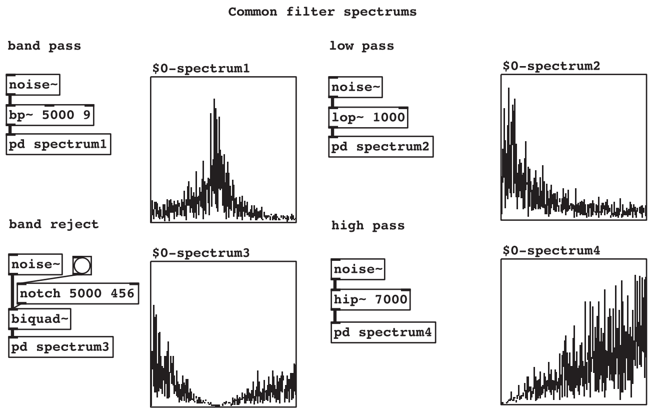

Figure 13.19
Common user-friendly filter shapes.

curve, so it can be useful for us to work out the total energy contained in a signal. It can also be used to shape waveforms; see Roberts 2009.

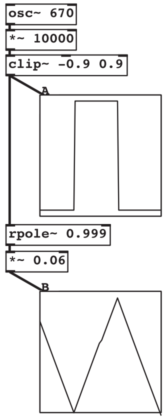

Figure 13.20
Integration.

Integrating a square wave gives us a triangle wave. If a constant signal value is given to an integrator its output will move up or down at a constant rate. In fact this is the basis of a phasor, so a filter can be seen as the most fundamental signal generator as well as a way to shape signals. Thus we have come full circle and can see the words of the great master, "It's all the same thing." A square wave is produced by the method shown in figure 13.7, first amplifying a cosinusoidal wave by a large value and then clipping it. As the square wave alternates between $+1.0$ and -1.0 the integrator output first slopes up at a constant rate, and then slopes down at a constant rate. A scaling factor is added to place the resulting triangle wave within the bounds of the graph. Experiment with integrating a cosinusoidal wave. What happens? The integral of $\cos(x)$ is $\sin(x)$, or in other words we have shifted $\cos(x)$ by 90°. If the same operation is applied again, to a sine wave, we get back to a cosine wave out of phase with the first one, a shift of 180°. In other words, the integral of $\sin(x)$ is $-\cos(x)$. This can be more properly written as a definite integral

$$\int \cos(x)\,dx = \sin(x) \tag{13.1}$$

or as

$$\int \sin(x)\, dx = -\cos(x) \tag{13.2}$$

Differentiation

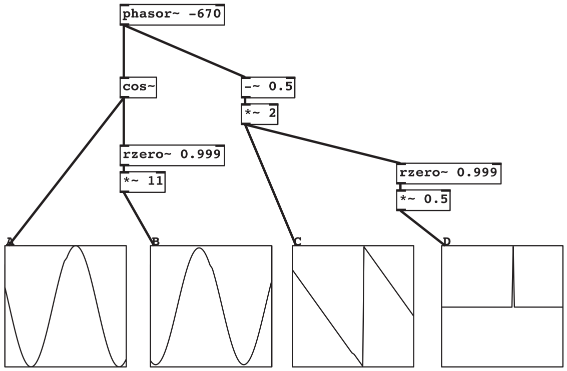

Figure 13.21
Differentiation.

The opposite of integrating a signal is differentiation. This gives us the instantaneous slope of a signal, or in other words the gradient of a line tangential to the signal. What do you suppose will be the effect of differentiating a cosine wave? The scaling factors in figure 13.21 are given for the benefit of the graphs. Perhaps you can see from the first graph that

$$\frac{d}{dx}\cos(x) = -\sin(x) \tag{13.3}$$

and

$$\frac{d}{dx}\sin(x) = \cos(x) \tag{13.4}$$

More useful, perhaps, is the result of differentiating a sawtooth wave. While the sawtooth moves slowly its gradient is a small constant, but at the moment it suddenly returns the gradient is very high. So, differentiating a sawtooth is a way for us to obtain a brief impulse spike.

References

McCartney, J. (1997). "Synthesis without Lookup Tables." *Comp. Music J.* 21(3).
Roberts, R. (2009). "A child's garden of waveforms." Unpublished ms.