13
Shaping Sound

The signal generators we've seen so far are the phasor, cosinusoidal oscillator, and noise source. While these alone seem limited they may be combined using shaping operations to produce a great many new signals. We are going to make transformations on waveforms, pushing them a little this way or that, moulding them into new things. This subject is dealt with in two sections: amplitude-dependent shaping where the output depends only on current input values, and time-dependent signal shaping where the output is a function of current and past signal values.

SECTION 13.1
Amplitude-Dependent Signal Shaping

Simple Signal Arithmetic

Arithmetic is at the foundation of signal processing. Examine many patches and you will find, on average, the most common object is the humble multiply, followed closely by addition. Just as all mathematics builds on a few simple arithmetic axioms, complex DSP operations can be reduced to adds and multiplies. Though it's rarely of practical use, it's worth noting that multiplication can be seen as repeated addition, so to multiply a signal by two we can connect it to both inlets of [+~] and it will be added to itself. The opposite of addition is subtraction. If you are subtracting a constant value from a signal it's okay to use [+~], but express the subtracted amount as a negative number, as with [+~ -0.5], though of course there is a [-~] unit too. Addition and multiplication are commutative (symmetrical) operators, so it doesn't matter which way round you connect two signals to a unit. On the other hand, subtraction and division have ordered arguments: the right value is subtracted from, or is the divisor of, the left one. It is common to divide by a constant, so [*~] is generally used with an argument that's the reciprocal of the required divisor. For example, instead of dividing by two, multiply by half. There are two reasons for this. First, divides were traditionally more expensive so many programmers are entrenched in the habit of avoiding divides where a multiply will do. Second, an accidental divide by zero traditionally causes problems, even crashing the program. Neither of these things are actually true of Pd running on a modern processor, but because of such legacies you'll find many algorithms written accordingly. Reserve divides for when you need to divide by a variable signal, and multiply by decimal fractions everywhere else unless you need rational numbers with good accuracy.

This habit highlights the importance of the function and makes your patches easier to understand. Arithmetic operations are used to scale, shift, and invert signals, as the following examples illustrate.

A signal is scaled simply by multiplying it by a fixed amount, which changes the difference between the lowest and highest values and thus the peak to peak amplitude. This is seen in figure 13.1 where the signal from the oscillator is halved in amplitude.

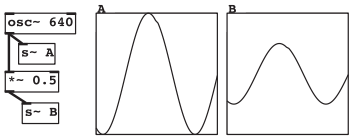

Figure 13.1
Scaling a signal.

Shifting involves moving a signal up or down in level by a constant. This affects the absolute amplitude in one direction only, so it is possible to distort a signal by pushing it outside the limits of the system, but it does not affect its peak to peak amplitude or apparent loudness since we cannot hear a constant (DC) offset. Shifting is normally used to place signals into the correct range for a subsequent operation, or, if the result of an operation yields a signal that isn't centered properly to correct it, shifting swings it about zero again. In figure 13.2 the cosine signal is shifted upwards by adding 0.5.

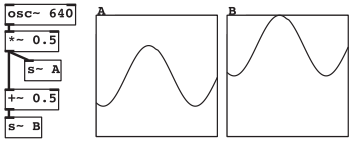

Figure 13.2
Shifting a signal.

In figure 13.3 a signal is inverted, reflecting it around the zero line, by multiplying by -1.0. It still crosses zero at the same places, but its direction and magnitude is the opposite everywhere. Inverting a signal changes its phase by π, 180° or 0.5 in rotation normalised form, but that has no effect on how it sounds since we cannot hear absolute phase.

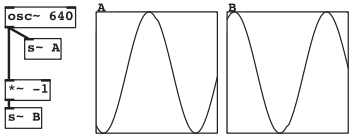

Figure 13.3
Inverting a signal.

The complement of a signal a in the range 0.0 to 1.0 is defined as $1 - a$. As the phasor in figure 13.4 moves upwards the complement moves downwards, mirroring its movement. This is different from the inverse; it has the same direction as the inverse but retains the sign and is only defined for the positive range between 0.0 and 1.0. It is used frequently to obtain a control signal for amplitude or filter cutoff that moves in the opposite direction to another control signal.

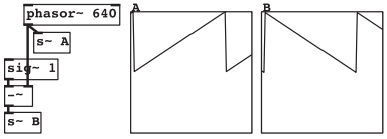

Figure 13.4
Signal complement.

For a signal a in the range 0.0 to x the reciprocal is defined as $1/a$. When a is very large then $1/a$ is close to zero, and when a is close to zero then $1/a$ is very large. Usually, since we are dealing with normalised signals, the largest

input is $a = 1.0$, so because $1/1.0 = 1.0$ the reciprocal is also 1.0. The graph of $1/a$ for a between 0.0 and 1.0 is a curve, so a typical use of the reciprocal is shown in figure 13.5. A curve is produced according to $1/(1+a)$. Since the maximum amplitude of the divisor is 2.0 the minimum of the output signal is 0.5.

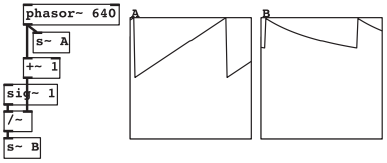

Figure 13.5
Signal reciprocal.

Limits

Sometimes we want to constrain a signal within a certain range. The `min~` unit outputs the minimum of its two inlets or arguments. Thus `min~ 1` is the minimum of one and whatever signal is on the left inlet; in other words, it clamps the signal to a maximum value of one if it exceeds it. Conversely, `max~ 0` returns the maximum of zero and its signal, which means that signals going below zero are clamped there forming a lower bound. You can see the effect of this on a cosine signal in figure 13.6.

Think about this carefully; the terminology seems to be reversed but it is correct. You use `max~` to create a minimum possible value and `min~` to create a maximum possible value. There is a slightly less confusing alternative `clip~` for situations where you don't want to adjust the limit using another signal. The left inlet of `clip~` is a signal and the remaining two inlets or arguments are the values of upper and lower limits; so, for example, `clip~ -0.5 0.5` will limit any signal to a range of one centered about zero.

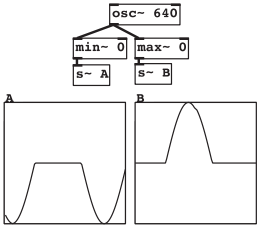

Figure 13.6
Min and max of a signal.

Wave Shaping

Using these principles we can start with one waveform and apply operations to create others like square, triangle, pulse, or any other shape. The choice of starting waveform is usually a phasor, since anything can be derived from it. Sometimes it's best to minimise the number of operations, so a cosine wave is the best starting point.

One method of making a square wave is shown in figure 13.7. An ordinary cosine oscillator is multiplied by a large number and then clipped. If you picture a graph of a greatly magnified cosine waveform, its slope will have become extremely steep, crossing through the area between -1.0 and 1.0 almost vertically. Once clipped to a normalised range what remains is a square wave, limited to between 1.0 and -1.0 and crossing suddenly halfway through. This method produces a waveform that isn't band-limited, so when used in synthesis you should keep it to a fairly low-frequency range to avoid aliasing.

A triangle wave moves up in a linear fashion just like a phasor, but when it reaches the peak it changes direction and returns to its lowest value at the

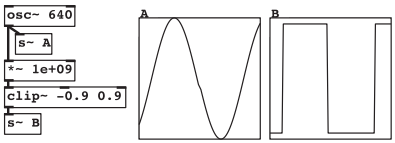

Figure 13.7
Square wave.

same rate instead of jumping instantly back to zero. It is a little more complicated to understand than the square wave. We can make a signal travel more or less in a given time interval by multiplying it by a constant amount. If a signal is multiplied by 2.0 it will travel twice as far in the same time as it did before, so multiplication affects the slope of signals. Also, as we have just seen, multiplying a signal by −1.0 inverts it. That's another way of saying it reverses the slope, so the waveform now moves in the opposite direction. One way of making a triangle wave employs these two principles.

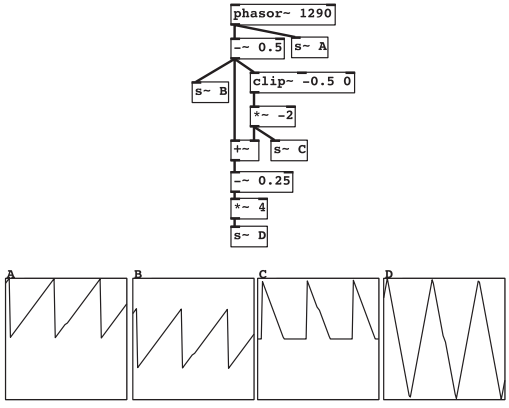

Figure 13.8
Triangle.

Starting with a phasor (graph *A*) at the top of figure 13.8, and shifting it down by 0.5 (graph *B*), the first half of it, from 0.0 to 0.5, is doing what we want. If we take half and isolate it with clip~ we can then multiply by −1.0 to change the slope, and by 2.0 to double the amplitude, which is the same as multiplying by −2.0. During the first half of the source phasor, between 0.5 and 1.0, the right branch produces a falling waveform (graph *C*). When we add

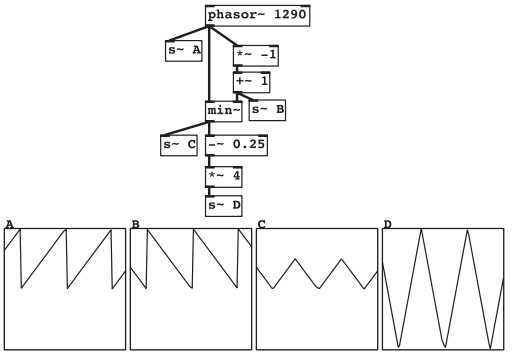

Figure 13.9
Another way to make a triangle wave.

that back to the other half, which is shifted down by 0.5 the sum is a triangle wave once normalised (graph D).

An alternative formula for a triangle wave, which may be slightly easier to understand, uses min~ and is shown in figure 13.9. Starting with a phasor again, (graph A) and adding one to the inverse produces a negative moving phasor with the same sign but opposite phase (graph B). Taking the minima of these two signals gives us a triangle wave, positive with amplitude 0.5 (graph C). This is recentered and normalised (graph D).

Squaring and Roots

One common function of a signal a is a^2, another way of writing $a \times a$. A multiplier is the easiest way to perform squaring. If you connect a signal to both inlets of a multiplier it is multiplied by itself. The effect of squaring a signal is twofold. Its amplitude is scaled as a function of its own amplitude. Amplitude values that are already high are increased more, while values closer to zero are increased less. Another result is that the output signal is only positive. Since a minus times a minus gives a plus, there are no squares that are negative. The reverse of this procedure is to determine a value r which if multiplied by itself gives the input a. We say r is the square root of a. Because finding square roots is a common DSP operation that requires a number of steps, there's a built-in sqrt~ object in Pd. Without creating complex (imaginary) numbers there are no square roots to negative numbers, so the output of sqrt~ is zero for these values. The effect of making the straight phasor line between 0.0 and 1.0 into a curve is clear in figure 13.10, graph A; likewise the curve bends the other way for the square root in graph B. Remembering that a minus times a minus gives a plus you can see that whatever the sign of a signal appearing at both inlets of the multiplier, a positive signal is output in graph C. Making either sign of the cosine wave positive like this doubles the frequency. In graph D an absence of negative square roots produces a broken sequence of positive pulses, and the

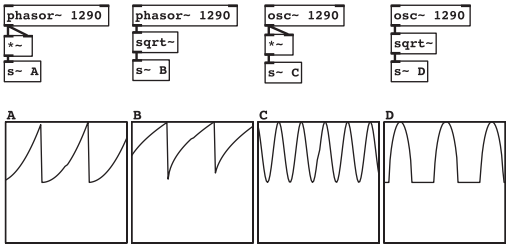

Figure 13.10
Square roots.

effect of the square root operation is to change the cosine curve to a parabolic (circular) curve (notice it is more rounded).

Curved Envelopes

We frequently wish to create a curve from a rising or falling control signal in the range 0.0 to 1.0. Taking the square, third, fourth, or higher powers produces increasingly steep curves, the class of *parabolic* curves. The quartic envelope is frequently used as a cheap approximation to natural decay curves. Similarly, taking successive square roots of a normalised signal will bend the curve the other way.[1] In figure 13.11 three identical line segments are generated each of

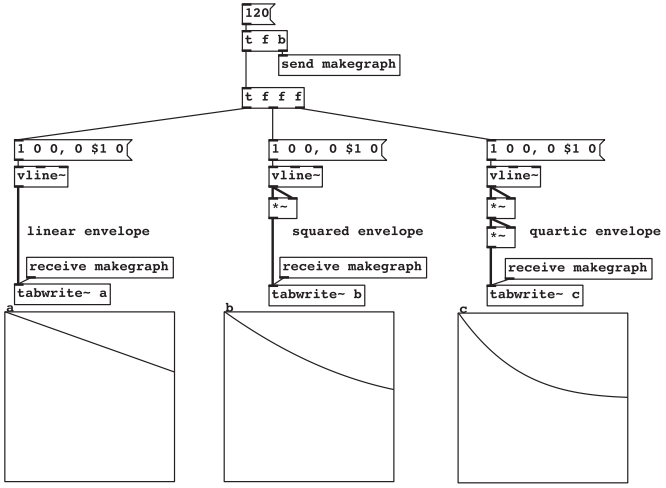

Figure 13.11
Linear, squared, and quartic decays.

1. See McCartney 1997 for other identities useful in making efficient natural envelopes.

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.

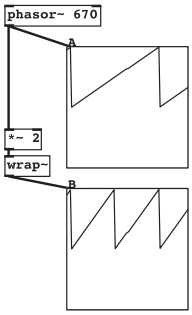

Figure 13.12
Wrapping.

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.

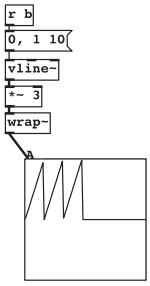

Figure 13.13
Wrapping a line.

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~