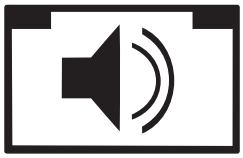
11
Pure Data Audio



SECTION 11.1

Audio Objects

We have looked at Pd in enough detail now to move on to the next level. You have a basic grasp of dataflow programming and know how to make patches that process numbers and symbols. But why has no mention been made of audio yet? Surely it is the main purpose of our study? The reason for this is that audio signal processing is a little more complex in Pd than the numbers and symbols we have so far considered, so I wanted to leave this until now.

Audio Connections

I have already mentioned that there are two kinds of objects and data for messages and signals. Corresponding to these there are two kinds of connections, audio connections and message connections. There is no need to do anything special to make the right kind of connection. When you connect two objects together, Pd will work out what type of outlet you are attempting to connect to what kind of inlet and create the appropriate connection. If you try to connect an audio signal to a message inlet, then Pd will not let you, or it will complain if there is an allowable but ambiguous connection. Audio objects always have a name ending with a tilde (∼) and the connections between them look fatter than ordinary message connections.

Blocks

The signal data travelling down audio cords is made of *samples*, single floating point values in a sequence that forms an audio signal. Samples are grouped together in *blocks*.

A block, sometimes called a *vector*, typically has 64 samples inside it, but you can change this in certain circumstances. Objects operating on signal blocks behave like ordinary message objects; they can add, subtract, delay, or store blocks of data, but they do so by processing one whole block at a time. In figure 11.1 streams of blocks are fed to the two inlets. Blocks appearing at the outlet have values which are the sum of the corresponding values in the two input blocks. Because they process signals made of blocks, audio objects do a lot more work than objects that process messages.

Audio Object CPU Use

All the message objects we looked at in the last chapters only use CPU when event-driven dataflow occurs, so most of the time they sit idle and consume no resources. Many of the boxes we put on our sound design canvases will be audio objects, so it's worth noting that they use up some CPU power just being

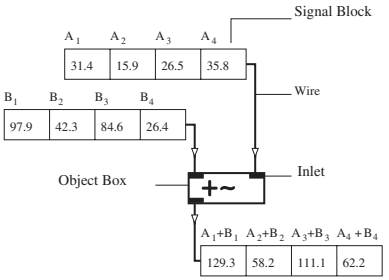


Figure 11.1
Object processing data.

idle. Whenever `compute audio` is switched on they are processing a constant stream of signal blocks, even if the blocks contain only zeros. Unlike messages, which are processed in logical time, signals are processed synchronously with the sound card sample rate. This *real-time* constraint means glitches will occur unless every signal object in the patch can be computed before the next block is sent out. Pd will not simply give up when this happens; it will struggle along trying to maintain real-time processing, so you need to listen carefully. As you hit the CPU limit of the computer you may hear crackles or pops. The DIO indicator on the Pd console shows when *over-run* errors have occurred. Click this to reset it. It is also worth knowing how audio computation relates to message computation. Message operations are executed at the beginning of each pass of audio block processing, so a patch where audio depends on message operations which don't complete in time will also fail to produce correct output.

SECTION 11.2

Audio Objects and Principles

There are a few ways that audio objects differ from message objects, so let's look at those rules now before starting to create sounds.

Fanout and Merging

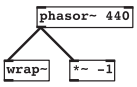


Figure 11.2
Signal fanout is okay.

You can connect the same signal outlet to as many other audio signal inlets as you like; blocks are sent in an order which corresponds to the creation of the connections, much like message connections. But unlike messages, most of the time this will have no effect whatsoever, so you can treat audio signals that fan out as if they were perfect simultaneous copies. Very seldom you may meet rare and interesting problems, especially with delays and feedback, that can be fixed by reordering audio signals (see chapter 7 of Puckette 2007 regarding time shifts and block delays).

When several signal connections all come into the same signal inlet that's also fine. In this case they are implicitly summed, so you may need to scale your signal to reduce its range again at the output of the object. You can connect as many signals to the same inlet as you like, but sometimes it makes a patch easier to understand if you explicitly sum them with a +~ unit.

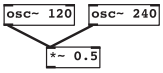


Figure 11.3
Merging signals is okay.

Time and Resolution

Time is measured in seconds, milliseconds (one thousandth of a second, written 1ms) or samples. Most Pd times are in ms. Where time is measured in samples, this depends on the sampling rate of the program or the sound card of the computer system on which it runs. The current sample rate is returned by the samplerate~ object. Typically a sample is 1/44100th of a second and is the smallest unit of time that can be measured as a signal. But the time resolution also depends on the object doing the computation. For example, metro and vline~ are able to deal in fractions of a millisecond, even less than one sample. Timing irregularities can occur where some objects are only accurate to one block boundary and some are not.

Audio Signal Block to Messages

To see the contents of a signal block we can take a snapshot or an average. The env~ object provides the RMS value of one block of audio data scaled 0 to 100 in dB, while snapshot~ gives the instantaneous value of the last sample in the previous block. To view an entire block for debugging, print~ can be used. It accepts an audio signal and a bang message on the same inlet and prints the current audio block contents when banged.

Sending and Receiving Audio Signals

Audio equivalents of send and receive are written send~ and receive~, with shortened forms s~ and r~. Unlike message sends, only one audio send can exist with a given name. If you want to create a signal bus with many-to-one connectivity, use throw~ and catch~ instead. Within subpatches and abstractions we use the signal objects inlet~ and outlet~ to create inlets and outlets.

Audio Generators

Only a few objects are signal sources. The most important and simple one is the phasor~. This outputs an asymmetrical periodic ramp wave and is used at the heart of many other digital oscillators we are going to make. Its left inlet specifies the frequency in Hz, and its right inlet sets the phase, between 0.0 and 1.0. The first and only argument is for frequency, so a typical instance of a phasor looks like phasor~ 110. For sinusoidal waveforms we can use osc~. Again, frequency and phase are set by the left and right inlets, or frequency is set by the creation parameter. A sinusoidal oscillator at concert A pitch is defined by osc~ 440. White noise is another commonly used source in sound design. The

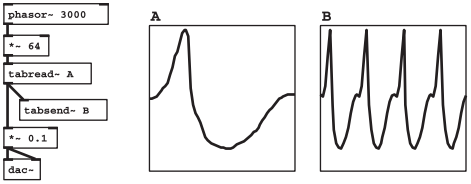


Figure 11.4
Table oscillator.

noise generator in Pd is simply `noise~` and has no creation arguments. Its output is in the range −1.0 to 1.0. Looped waveforms stored in an array can be used to implement *wavetable* synthesis using the `tabosc4~` object. This is a 4-point interpolating table ocillator and requires an array that is a power of 2, plus 3 (e.g. 0 to 258) in order to work properly. It can be instantiated like `phasor~` or `osc~` with a frequency argument. A table oscillator running at 3kHz is shown in figure 11.4. It takes the waveform stored in array `A` and loops around this at the frequency given by its argument or left inlet value. To make sound samplers we

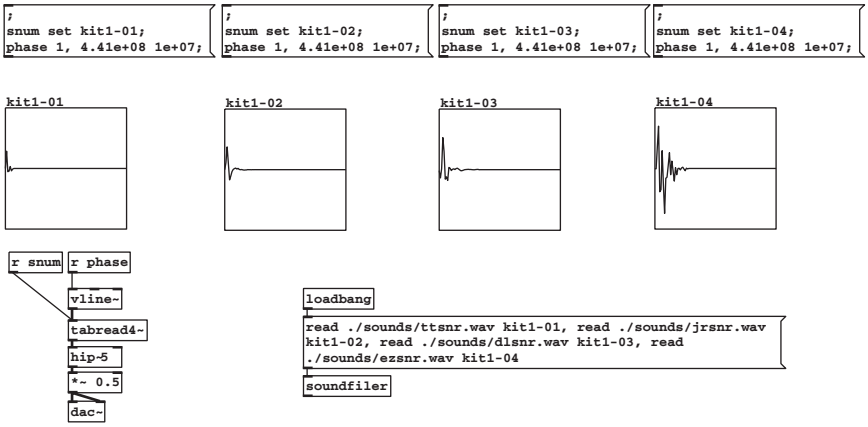


Figure 11.5
Sample replay from arrays.

need to read and write audio data from an array. The index to `tabread~` and its interpolating friend `tabread4~` is a sample number, so you need to supply a signal with the correct slope and magnitude to get the proper playback rate. You can use the special `set` message to reassign `tabread4~` to read from another array. The message boxes in figure 11.5 allow a single object to play back from more than

one sample table. First the target array is given via a message to `snum`, and then a message is sent to `phase` which sets vline~ moving up at 44,100 samples per second. The arrays are initially loaded, using a multipart message, from a `sounds` folder in the current patch directory.

Audio Line Objects

For signal rate control data the line~ object is useful. It is generally programmed with a sequence of lists. Each list consists of a pair of numbers: the first is a level to move to and the second is the time in milliseconds to take getting there. The range is usually between 1.0 and 0.0 when used as an audio control signal, but it can be any value such as when using line~ to index a table. A more versatile line object is called vline~, which we will meet in much more detail later. Amongst its advantages are very accurate sub-millisecond timing and the ability to read multisegment lists in one go and to delay stages of movement. Both these objects are essential for constructing envelope generators and other control signals.

Audio Input and Output

Audio IO is achieved with the adc~ and dac~ objects. By default these offer two inlets or outlets for stereo operation, but you can request as many additional sound channels as your sound system will handle by giving them numerical arguments.

Example: A Simple MIDI Monosynth

Using the objects we've just discussed let's create a little MIDI keyboard-controlled music synthesiser as shown in figure 11.6. Numbers appearing at the left outlet of notein control the frequency of an oscillator. MIDI numbers are converted to a Hertz frequency by mtof. The MIDI standard, or rather general adherence to it, is a bit woolly by allowing note-off to also be a note-on with a velocity of zero. Pd follows this definition, so when a key is released it produces a note with a zero velocity. For this simple example we remove it with stripnote, which only passes note-on messages when their velocity is greater than zero. The velocity value, ranging between 1 and 127, is scaled to between 0 and 1 in order to provide a rudimentary amplitude control.

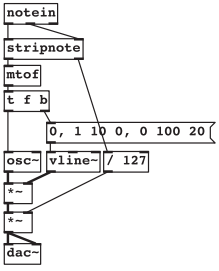


Figure 11.6
MIDI note control.

So, here's a great place to elaborate on the anatomy of the message used to control vline~ as shown in figure 11.7. The syntax makes perfect sense, but sometimes it's hard to visualise without practice. The general form has three numbers per list. It says: *"go to some value,"* given by the first number, then *"take a certain time to get there,"* which is the second number in each list. The last number in the list is a time to wait before executing the command, so it adds an extra "*wait for a time before doing it.*" What makes vline~ cool is you

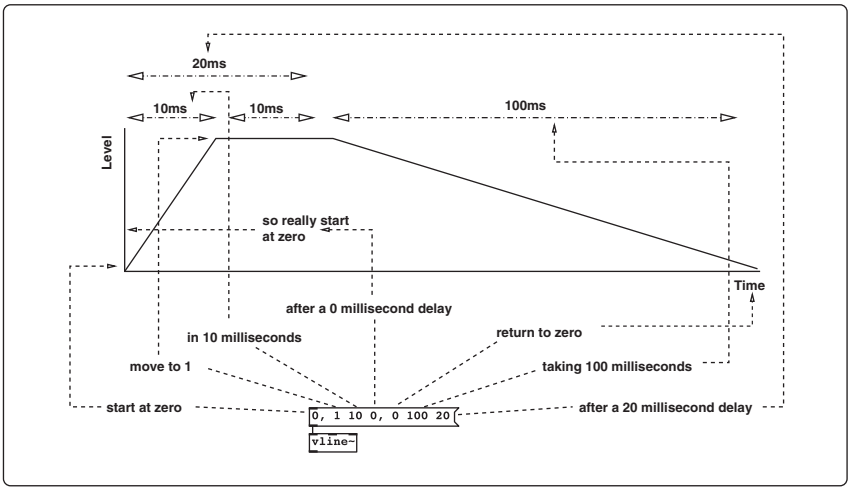


Figure 11.7
Anatomy of vline message.

can send a sequence of list messages in any order, and so long as they make temporal sense [vline~] will execute them all. This means you can make very complex control envelopes. Any missing arguments in a list are dropped in right-to-left order, so a valid exception is seen in the first element of figure 11.7 where a single 0 means *"jump immediately to zero"* (don't bother to wait or take any time getting there).

Audio Filter Objects

Six or seven filters are used in this book. We will not look at them in much detail until we need to because there is a lot to say about their usage in each case. Simple one-pole and one-zero real filters are given by [rpole~] and [rzero~]. Complex one-pole and one-zero filters are [cpole~] and [czero~]. A static biquad filter [biquad~] also comes with a selection of helper objects to calculate coefficients for common configurations, and [lop~], [hip~], and [bp~ 1] provide the standard low, high, and band pass responses. These are easy to use and allow message rate control of their cutoff frequencies and, in the case of bandpass, resonance. The first and only argument of the low and high pass filters is frequency, so typical instances may look like [lop~ 500] and [hip~ 500]. Bandpass takes a second parameter for resonance like this [bp~ 100 3]. Fast signal rate control of cutoff is possible using the versatile [vcf~] "voltage controlled filter." Its first argument is cutoff frequency and its second argument is resonance, so you might use it like [vcf~ 100 2]. With high resonances this provides a sharp filter that can give narrow bands. An even more colourful filter for use in music synthesiser designs is available as an external called [moog~], which provides a classic design that can self-oscillate.

Audio Arithmetic Objects

Audio signal objects for simple arithmetic are summarised in figure 11.8.

Object	Function
+~	Add two signals (either input will also accept a message)
-~	Subtract righthand signal from lefthand signal
/~	Divide lefthand signal by right signal
*~	Signal multiplication
wrap~	Signal wrap, constrain any signal between 0.0 and 1.0

Figure 11.8
List of arithmetic operators.

Trigonometric and Math Objects

A summary of higher maths functions is given in figure 11.9. Some signal units are abstractions defined in terms of more elementary intrinsic objects, and those marked * are only available through external libraries in some Pd versions.

Object	Function
cos~	Signal version of cosine function. Domain: −1.0 `to` + 1.0. Note the input domain is "rotation normalised."
sin~	Not intrinsic but defined in terms of signal cosine by subtracting 0.25 from the input.
atan~ *	Signal version of arctangent with normalised range.
log~	Signal version of natural log.
abs~ *	Signal version of abs.
sqrt~	A square root for signals.
q8_sqrt~	A fast square root with less accuracy.
pow~	Signal version of power function.

Figure 11.9
List of trig and higher math operators.

Audio Delay Objects

Delaying an audio signal requires us to create a memory buffer using delwrite~. Two arguments must be supplied at creation time: a unique name for the memory buffer and a maximum size in milliseconds. For example, delwrite~ mydelay 500 creates a named delay buffer "mydelay" of size 500ms. This object can now be used to write audio data to the delay buffer through its

left inlet. Getting delayed signals back from a buffer needs delread~. The only argument needed is the name of a buffer to read from, so delread~ mydelay will listen to the contents of `mydelay`. The delay time is set by a second argument, or by the left inlet. It can range from zero to the maximum buffer size. Setting a delay time larger than the buffer results in a delay of the maximum size. It is not possible to alter the maximum size of a delwrite~ buffer once created. But it is possible to change the delay time of delread~ for chorus and other effects. This often results in clicks and pops[1] so we have a vd~ variable-delay object. Instead of moving the read point, vd~ changes the rate at which it reads the buffer, so we get tape echo and Doppler-shift-type effects. Using vd~ is as easy as before: create an object that reads from a named buffer like vd~ mydelay. The left inlet (or argument following the name) sets the delay time.

References

Puckette, M.(2007). *The Theory and Technique of Electronic Music.* World Scientific.

1. Hearing clicks when moving a delay read point is normal, not a bug. There is no reason to assume that waveforms will align nicely once we jump to a new location in the buffer. An advanced solution crossfades between more than one buffer.