Synthesis in MSP

Synthesis in MSP is similar to the use of the old modular synthesizers (or their on screen emulators, like Tassman.) We have an assortment of primitive functions that we must assemble to generate the sounds we want. MSP is not as advanced as the hard core computer music programs like Csound and ChucK, but the list of available functions is being augmented all the time.

Any modular synthesis scheme divides objects into three basic operation: generation, processing and control, and these are available in standard MSP objects. (Advanced techniques like granular synthesis and modeling are possible with third party external objects.)

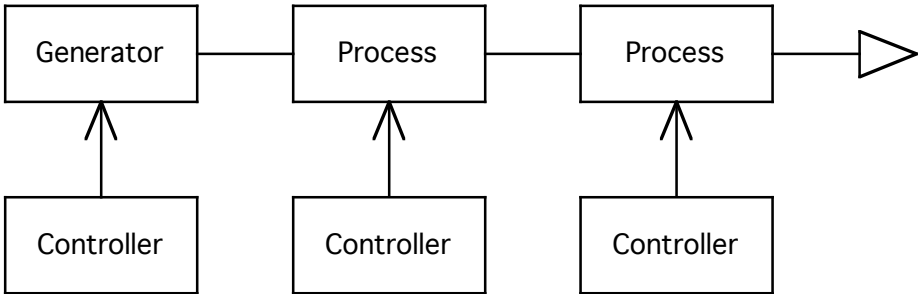


Figure 1.

Generators produce raw signal, with control of aspects like frequency and waveform. Processors adjust gain, and alter the waveform with filtering and other processes. All of this is mediated by various control functions.

Sound Generators

Oscillators

The cycle~ object is the basic oscillator. When audio is on it puts out a sine wave at he indicated frequency.

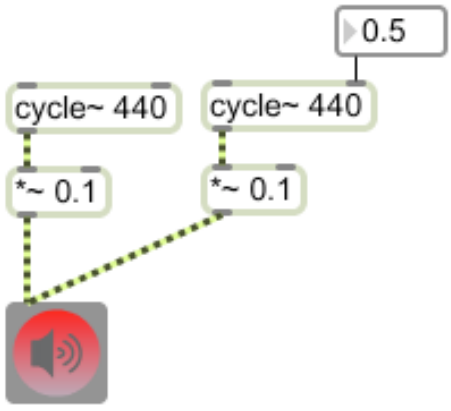


Figure 2.

There are two inlets to cycle~ the left one controls the frequency, the right one controls phase. Frequency is in hertz.

Phase is a fraction of the wave table, so 0.25 is 90 degrees, 1.0 is a full 360. The patch at the left doesn't make any sound, because 0.5 is 180 degrees out of phase, and you know what that does!

Phasor~

Phasor~ is the most primitive oscillator. Every digital oscillator is based on looking up values in a wave table , which is simply a chunk of memory containing a recording of one cycle of the waveform. To get different pitches, some math is required to step across entries of the wave table.[1] The phasor~ is the object that does that math. What goes into phasor~ is a float (or signal) with the frequency. What comes out is a signal going from 0.0 to 1.0 that some other objects can convert into a wave.

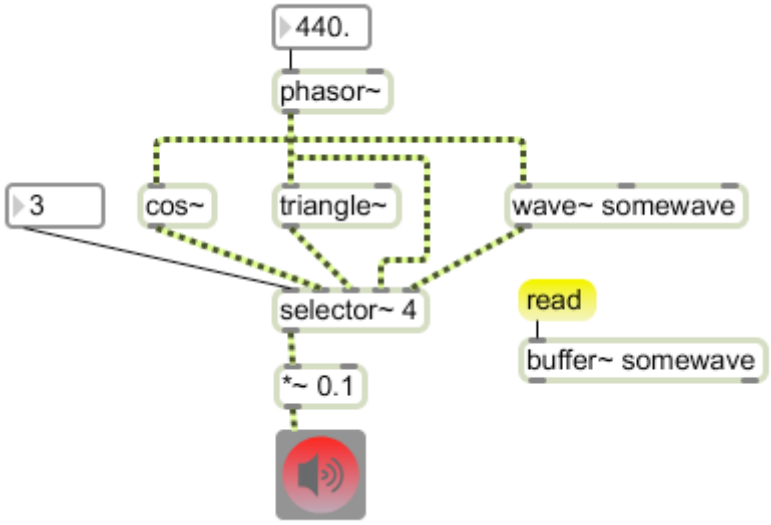


Figure 3.

Cos~ produces a cosine wave, triangle~ produces a triangle wave, and wave~ can have anything you want. Notice that listening to the phasor~ itself gives a sawtooth. The downside of listening to phasor~ (and to some extent triangle~) is that the sharp corners of the waveform imply harmonic content above the sampling rate. This produces aliasing[2].

Band Limited oscillators

To get away from the aliased sound, the objects saw~, rect~ and tri~ have waveforms that do not have Nyquist limit problems. They sound just like they ought to.

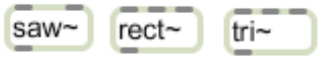
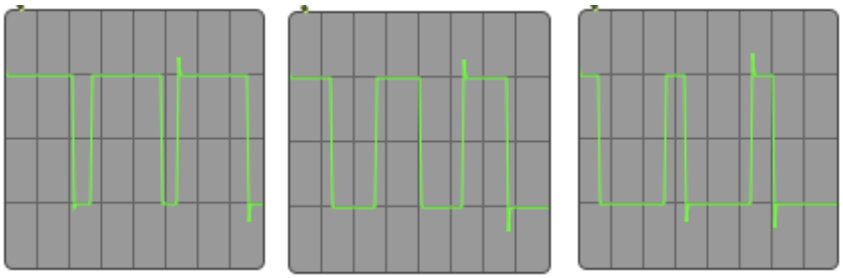


Figure 4.

[1] It's simple really. Assume a table of 512 places that contains one complete cycle. If the sampling rate is 44100 Hz, reading each value in turn would give a pitch of 86.132 Hz. To get 172 Hz, read every other sample. For frequencies in between, you could skip one place every third sample and so forth, or make up numbers in between what you've got. The latter is called interpolation. To get lower pitches, play some samples twice, or interpolate as needed.

[2] Remember the Nyquist limit. You can't go above half the sample rate.

If you use these with phasor~, connect to the right inlet. The left inlet controls frequency like cycle~. Rect~ and tri~ have a center inlet for duty cycle.



0.2 0.5 0.75

Figure 5.

Figure 5 shows the output of rect~ with varying duty cycle. Notice the analog style ringing at the corners of the wave.

Noise

The noise~ and pink~ objects are a constant source of white and pink noise.

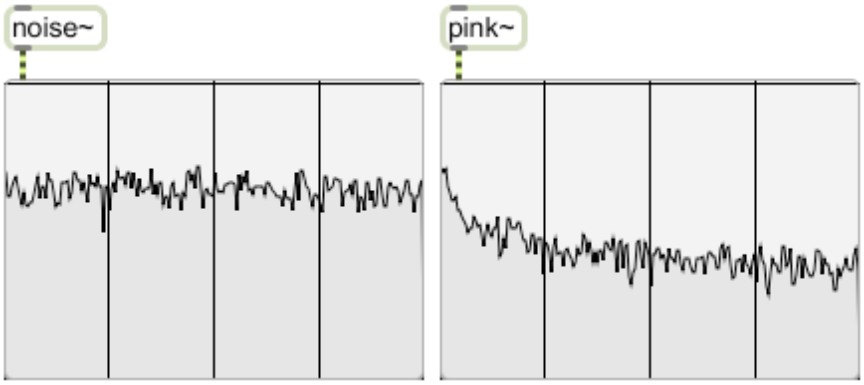


Figure 6.

White noise has equal power per Hz (on average) so, given that we hear in octaves and the top octave of our hearing has as many Hz as all the rest put together, we hear white noise as a high, hissy sound. Pink noise has equal power per octave, so we should hear something better balanced. Because math and our ears don't quite match, pink noise is usually heard as bass heavy.

Signal Processors

Amplitude

All of the MSP generators produce signals that swing from –1.0 to 1.0. This amplitude is easily adjusted by a *~ object. This multiplies every sample value by the argument.

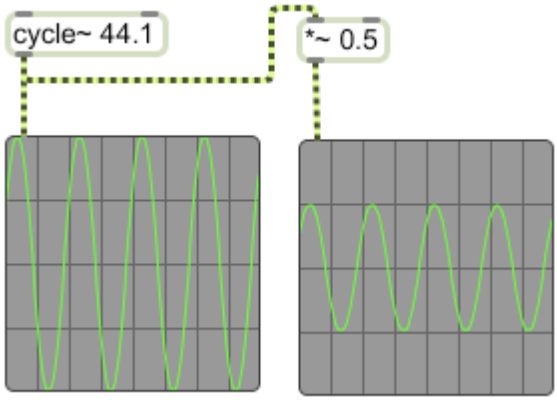


Figure 7.
An argument of 0 shuts the signal off. A float or signal applied to the right inlet will adjust the signal level. You can use a dbtoa object to calculate the appropriate multiplier for a desired dB change.

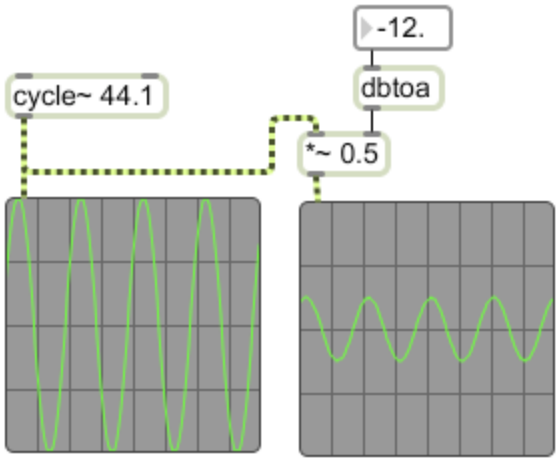


Figure 8.
You don't need to remember 20logA/B anymore, but you do need to remember that a negative dB value specifies a reduction in gain.

A Taste of Filters

MSP 2 features several nice filters. The most useful is probably lores~, which is a lowpass filter similar to those found on our modular machines. You will also find svf~ very familiar. This is a state variable filter, the circuit that gives simultaneous high, low bandpass and notch functions.

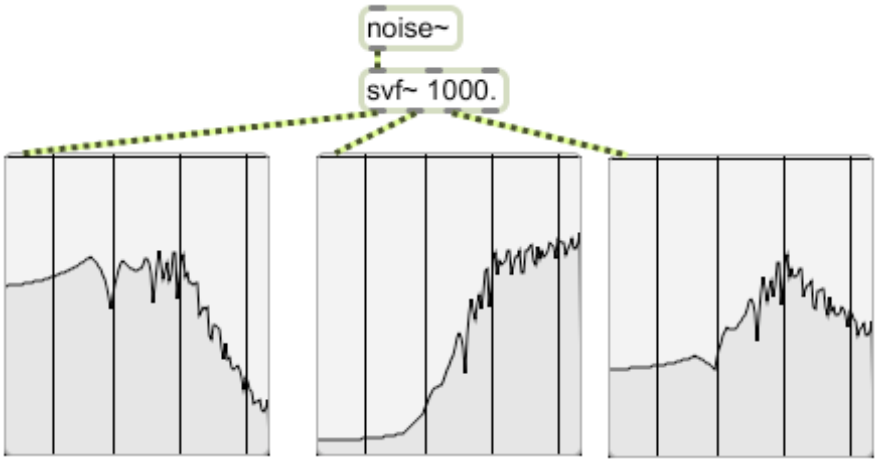


Figure 9. Low pass, high pass and band pass outputs of svf~

My favorite filter is the fast fixed filter bank, the fffb~. This gives the sound of the old 1/3rd octave filter. Here's a patch that shows how to control the band amplitudes with multislider.

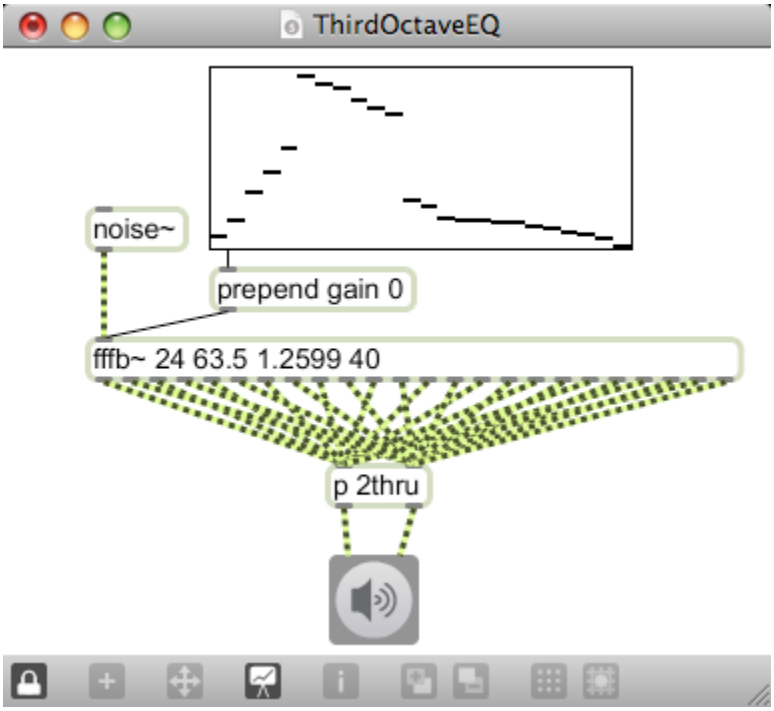


Figure 10.

The arguments set the number of filters, the frequency of the first filter, the frequency ratio (here it's the cube root of 2), and the Q of the filters. Each filter has its own output, which you can sum as shown or use for independent processing. (The 2thru subpatch merely holds the outputs together so I don't have to draw 24 connections if I change the filter destination.)

There's a thorough discussion of the MSP filters in the essay MSP Filters.

Wave shaping

The phasor~ can use wave~ to play a buffer~ containing an arbitrary waveform. You can draw a simple shape into a buffer~ with this mechanism:

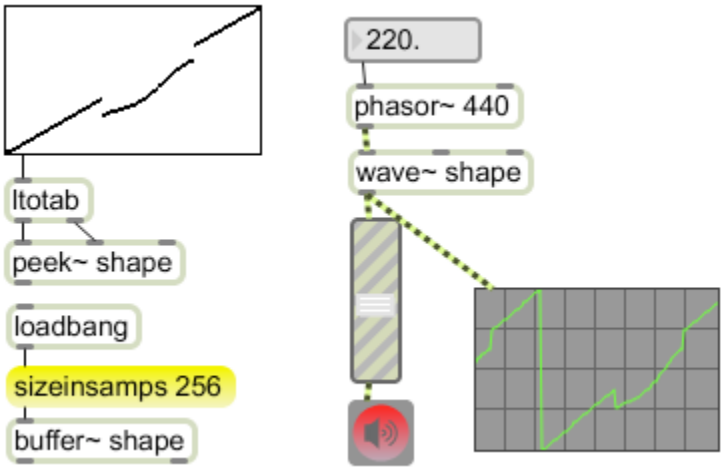


Figure 11.

The drawing area is a multislider with 256 sliders. The peek~ object addresses a buffer~ as if it were a table, so ltotab will format the output of the multislider to fill the buffer.

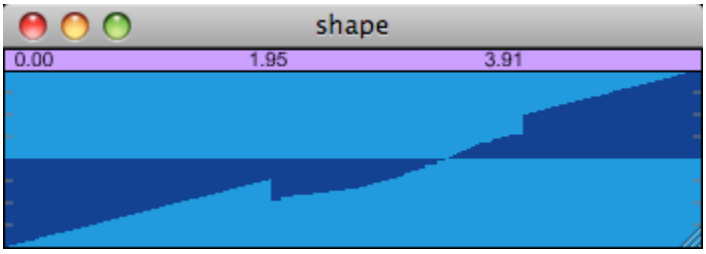


Figure 12.

With only 256 points in the waveform the sound will be a bit edgy. There are techniques available in jitter to support bigger tables, but that is seldom necessary. Drawing waveforms is a good source of interesting sounds, but wave~ is also a powerful processing tool. If we modify the output of cycle~ so that its output range is within 0 to1 we can use the shape drawn in the buffer to distort the output wave.

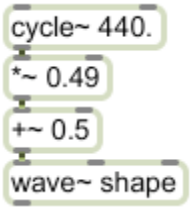
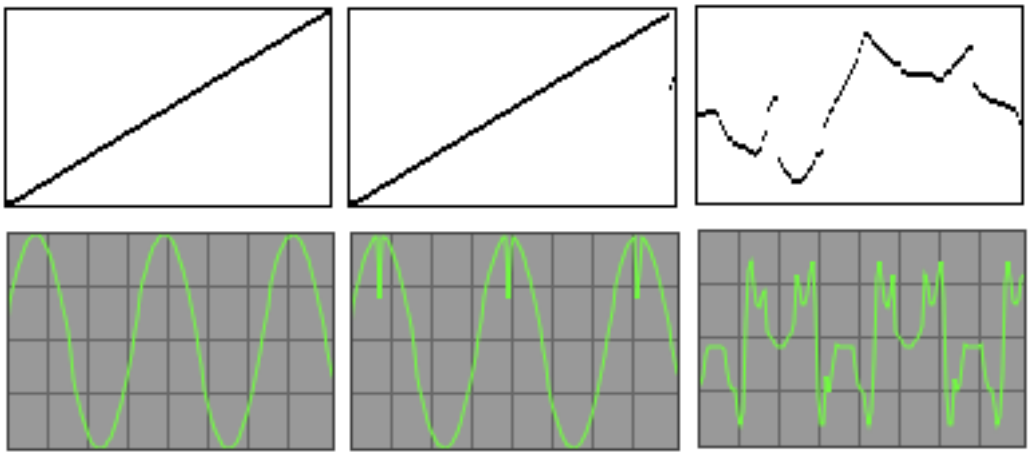


Figure 13.

I've found it's best not to use the end values of the buffer, so multiplying by 0.49 and adding 0.5 works well. Figure 14 illustrates the effect. If the shape is a straight line nothing happens, except a slight change in amplitude.



No change "Moog glitch" Messy

Figure 14.

If the shape is not straight, the wave is bent. In the middle drawing, all but the highest values of the cycle~ are unaffected, but at the right of the graph surprising low values are returned, giving a glitch similar to the old Moog oscillators. Messier shapes produce wild distortion. You can apply this process to any signal, not just sine waves.

Building Synthesizers

Control function are derived from basic Max objects. These are combined with signal generation and processing techniques to create instruments. Here are some examples.

Basic Beep

The line~ object will give us the equivalent of the envelope generator. It can be used with cycle~ and a multiplier to give us this:

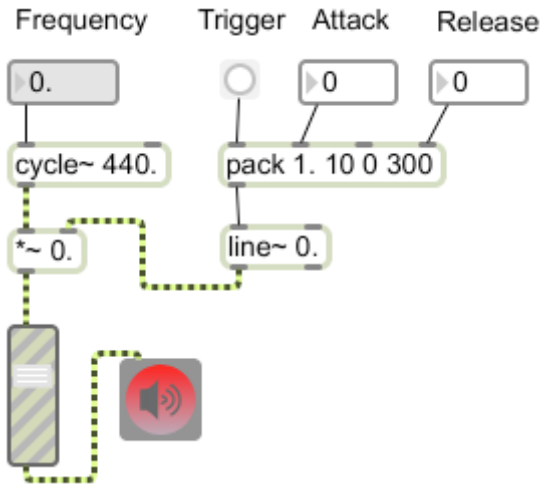


Figure 15.

Line~ is controlled by pairs of numbers in a message. The numbers are target, time; target, time; and so on, up to 46 pairs. When line~ receives the message it starts putting out numbers in a ramp from where it is now to the target. This is a signal that continues putting out the final target value until a new message comes in. You can also control line~ by ints; time in the right and target in the left. The right outlet of line~ bangs when the process is complete. Line does not operate unless audio is turned on.

The pack object in figure 15 contains a default envelope- notice the target for the attack is 1 and the final value is 0. Most envelopes will be similar, possibly with more line segments. The attack time is adjusted with the second inlet and the release time is adjusted via the last inlet. A bang to the pack object sends the envelope. If you set a long release and click the button quickly, you will hear only slight attacks, because when a new message is received, line starts at the current value. A message like this will force line~ back to 0 on each trigger.

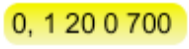


Figure 16.

ADSR~

The adsr~ object provides a more elaborate envelope.

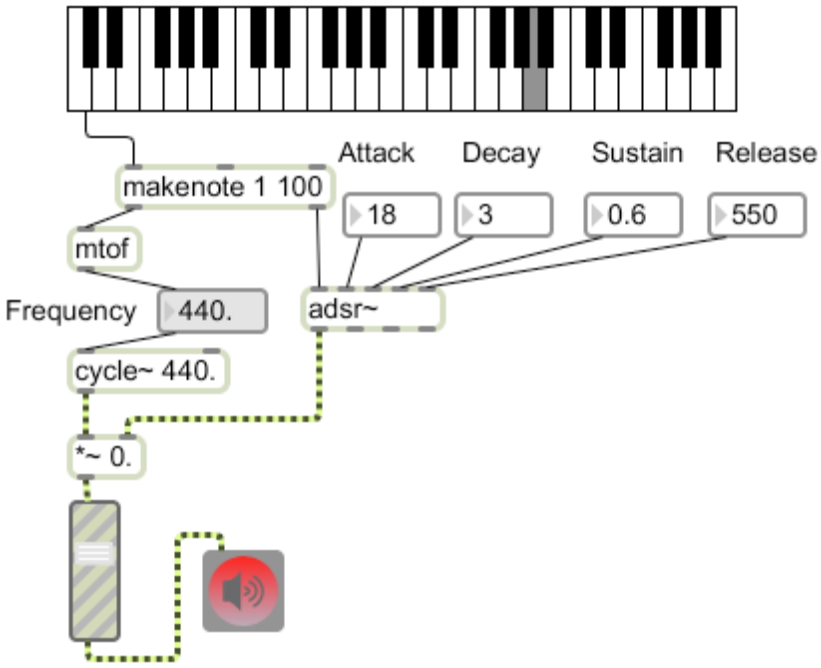


Figure 17.

This is the classic attack, decay, sustain and release module, which produces a signal like this:

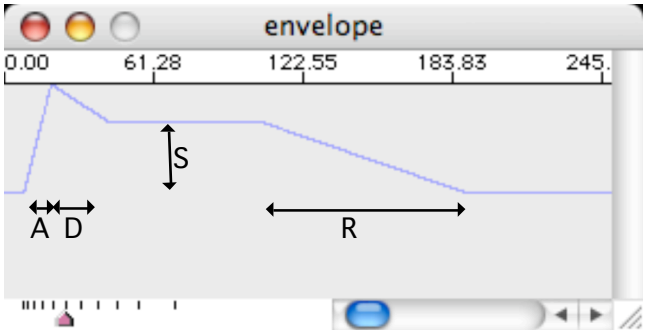
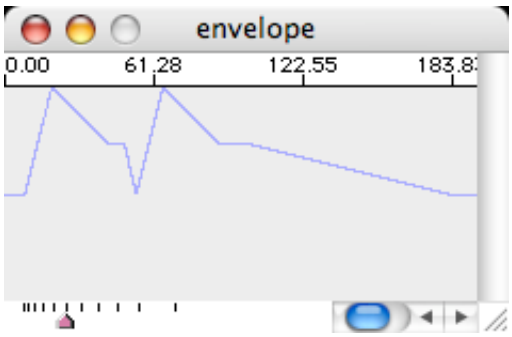


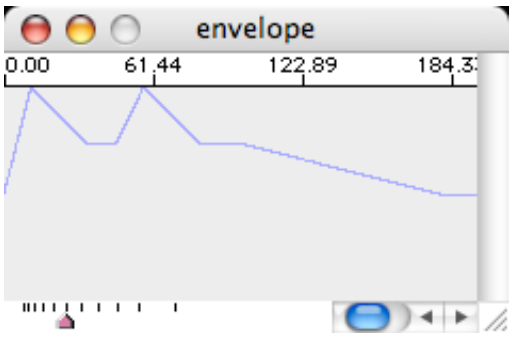
Figure 18.

The values vary from 0. to 1, perfect for amplitude control with a *~ object. The adsr~ is gated with a 1 and 0, and the parameters are set with floats.

The adsr~ has a legato feature, which affects what happens if the envelope is retriggered before it has completed its cycle. Normally, it shuts down briefly (with a ramp set by a retrigger $1 message), but if legato 1 has been received, the top of the attack and decay will be repeated with no actual break in sound.



Retriggering with legato 0



Retriggering with legato 1

Figure 19.

There are three more outlets:

- The second outlet sends a signal, which reflects the trigger input 1 or 0.
- The third outlet sends mute 0 when the envelope starts and mute 1 when it is done. This is useful for controlling poly~ functions as you will see.
- The fourth outlet is for parameter dumps a la jitter.

MIDI control

The next step is to trigger the sound from a MIDI keyboard. This uses objects you are already familiar with:

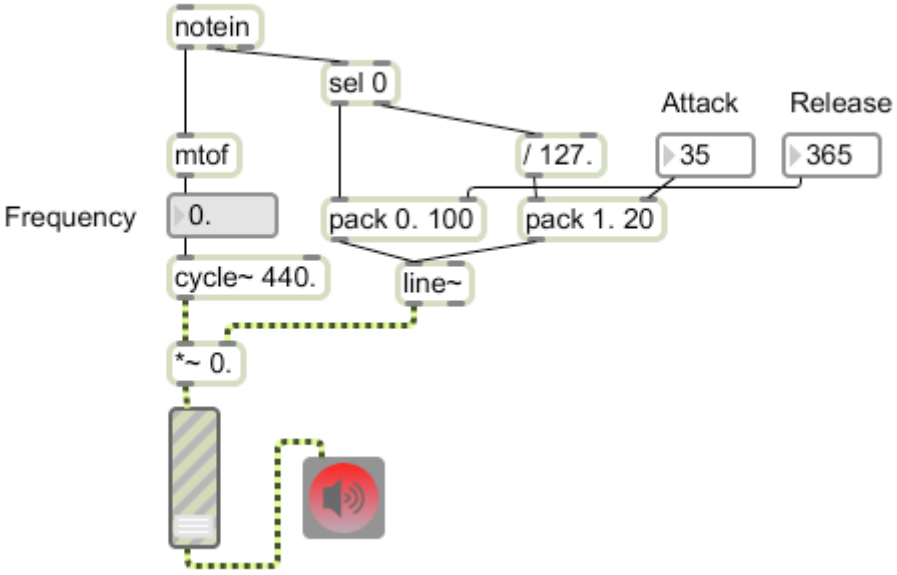
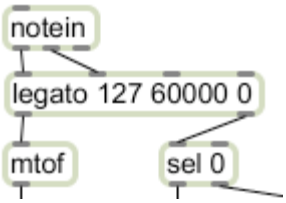


Figure 20.

The trick to MIDI control is to divide the envelope control into two parts-- a pack with the attack time and another with the release time. A select (sel) object either triggers the release directly, or send the velocity value to be divided by 127. This creates a number between 0 and 1 as the target value for line.

When you try this, you will find that it works, but that if you try to play too fast, you'll get chopped off notes. That's because when notes overlap, the note off for the first note will happen after the note on for the second note. There are a variety of ways to fix this; the easiest being to stick a Legato (that's an Lobject) after the notein:



Legato is like makenote, but only allows one note to be active at a time. The arguments are velocity, duration and overlap. (Overlap can stretch the noteoff past the next note.) If we set the duration longer than we are likely to play, legato can be used as a kind of filter for notein, turning off notes even if our keyboard technique is a bit sloppy.

Figure 21.

Polyphony

Polyphony is a pretty tough problem. First, you must have a separate signal generator (called a "voice") for each note you want to play at a time. In addition, there must be a system for connecting the notes off to the voice that is playing that particular pitch. In MSP this is handled by the **poly~** object. The poly~ object is a "wrapper", which can include multiple copies of a subpatcher and control them in a polyphonic way. It works like this:

First, build a voice patch.

This is almost exactly like the subpatchers we used before. Here's one based on figure 20.

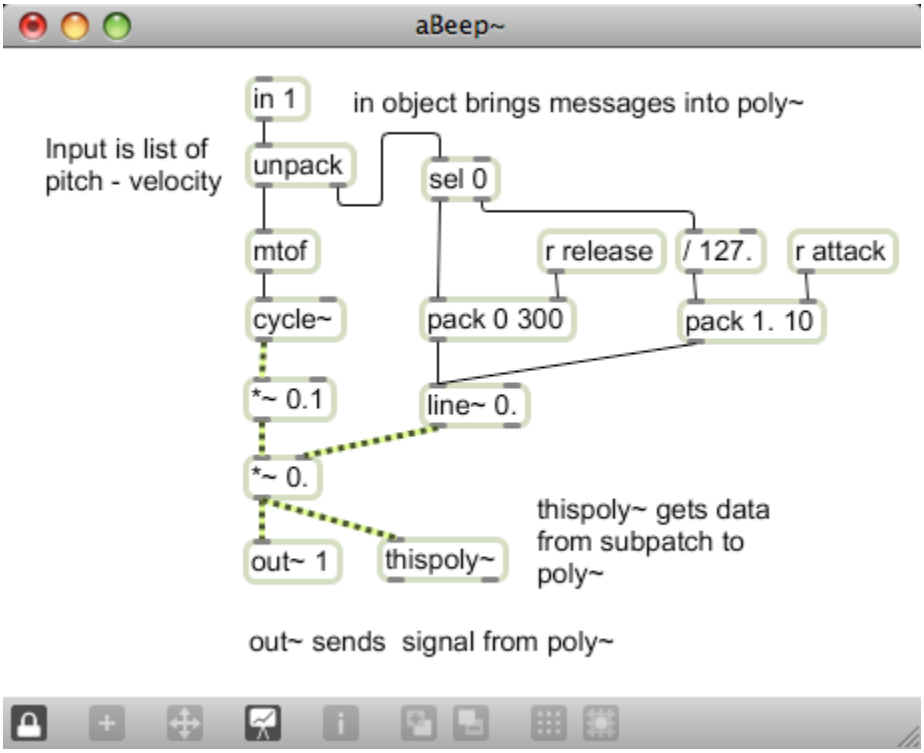


Figure 22.

The main difference is that **in** and **out~** objects have replaced the input and output sections and the envelope times are set remotely via send and receive. The argument 1 for the in object means it will be connected to inlet 1 of Poly~. The in object accepts messages-- use **in~** to bring in a signal. **Thispoly~** is the means the instrument sends information back to the poly~ wrapper. When thispoly~ receives a signal of 0, the poly~ will stop audio processing for the voice and it will be available for a new note.

Second, load the voice into poly~

Poly~ takes two arguments: the name of your file, and the number of voices you want available. A new instance of your patch will be created for each voice. Poly~ will acquire enough inlets to match the ins and outs of the voice.

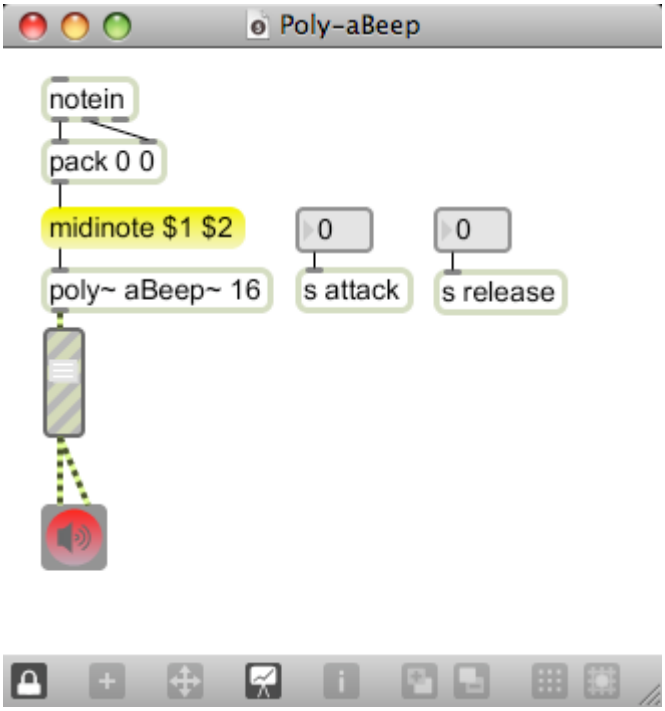


Figure 23.

To make a note sound, send the message **midinote** with the pitch and velocity. Poly~ will activate an unused voice and send it the data. When the note off occurs, poly~ will send the pitch and a 0 velocity to the same voice.

You could use the message **note** to start a note. This will send the arguments after note to the first unused voice. There is no tracking of sounding pitches with the note message, so this only works when instruments have predefined durations.

You can use send and receive objects to broadcast shared data to all of the poly instances or you can define inlets for the job. You can use the **target** message to route data to specific voices. For instance, once the message **target 4** is received, poly will route all data messages to voice 4. **Target 0** selects all voices.

You can send signals to voices with in~. Signals are always sent to all voices. If a voice patcher has both an in 2 and in~ 2 defined, the poly~ will route data sent to it second inlet to in 2 and signals at that inlet to in~ 2. If you have an out 2 and out~ 2 defined, poly will have 4 outlets, two for signals and 2 for data messages.

For more details on using Poly~, see the essay titled Working With Poly~.

Getting Beyond Beeps

The various waveform objects provide some variety of sound, but for really interesting synthesis, we need to explore some other strategies.

Buffer Based Sample Player

We can play short recorded sounds with a bit of preparation. The preparation is to use an audio editor to make a wav or aiff file with each sound we want to play. The patch will be simplest if all of the samples are exactly the same length.

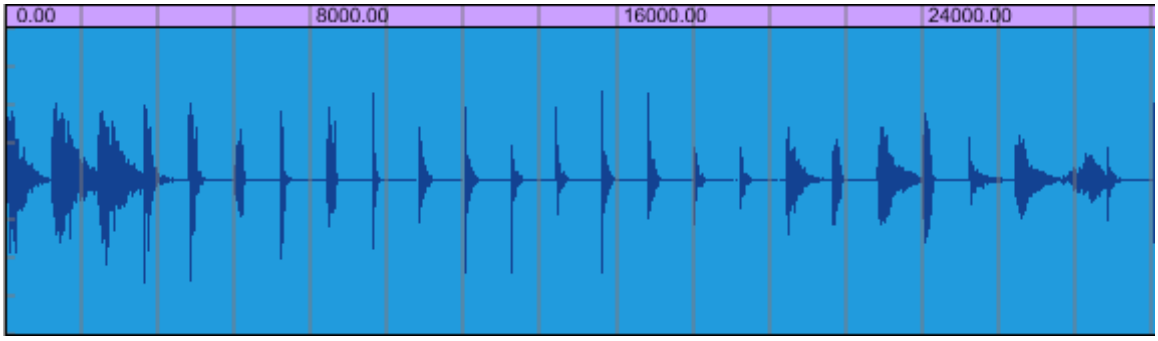


Figure 24.
Start each sample at some multiple of the same duration. Here each is 1200 ms in length.

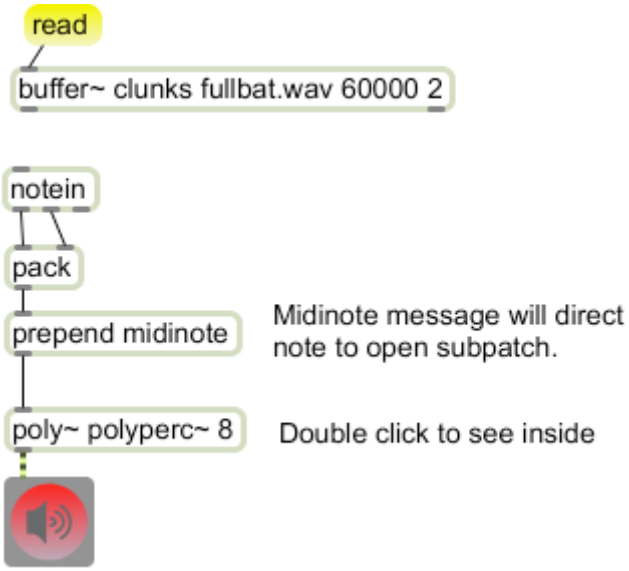


Figure 25.

The top level patcher looks like figure 25. There is a buffer~ to hold the sample file, and a poly~ object to contain the player.
The voice patch contains a groove~ object that points to the buffer~

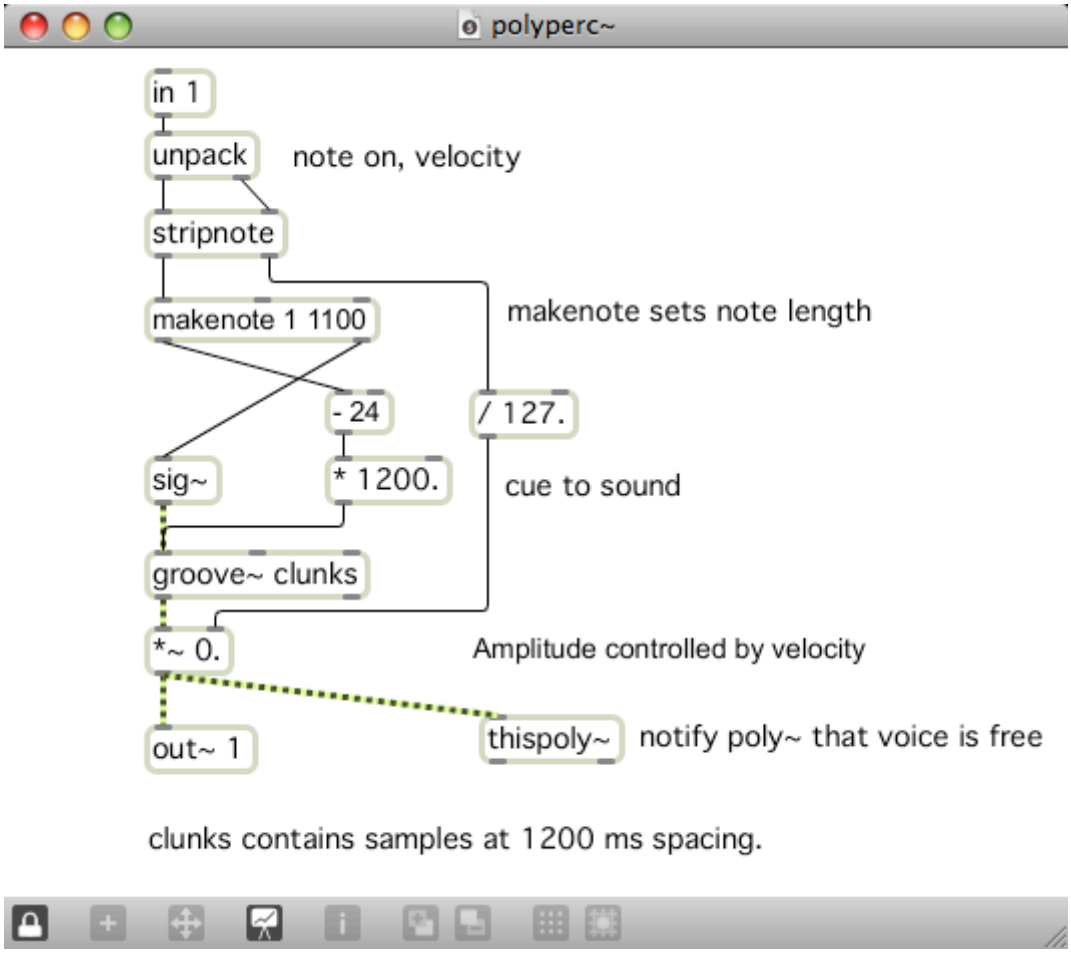


Figure 26.

Since the sounds are percussive, we use a makenote to specify a duration a bit shorter than the standard sample length. The note number from makenote is used to cue the groove~ object to the sound desired. First the number assigned to the first sample is subtracted, then the result is multiplied by the sample duration. If the samples were not the same length, you could use a named coll to point to the start position of each. I'd only do that if I had to mix a few long samples in a with a lot of short ones.

This isn't a true sampler, but it's useful when you want to play simple sounds on cue.

Doubling

A fair amount of richness can be added to a beep type voice by simply running two oscillators in parallel. Here is a patch to experiment with:

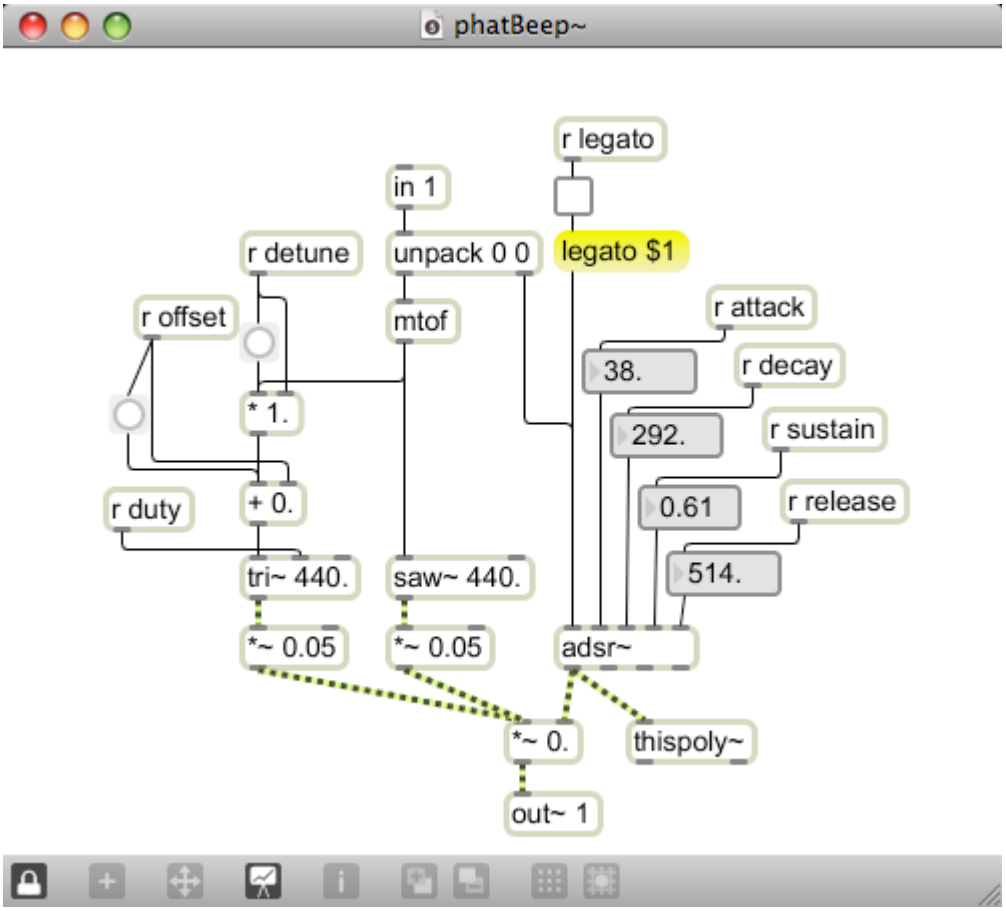


Figure 27.
In this patch, the basic saw~ sound is augmented by a triangle at nearly the same frequency. How much the frequency differs is determined by the offset and detune parameters.

- Offset adds a constant number of Hz to the frequency.
- Detune multiplies the frequency but a value, which should be close to 1.0

The effect of both of these is quite similar. When the two generators are running at slightly different frequencies, the fundamentals will change relative phase, a process known as beating. If this is done with two sine waves, the sound disappears at a rate equal to the difference in frequencies. (The beat frequency) Since the tri~ does not have the same partials as saw~ only the lower partials are affected in this patch. You can get subtly different effects with other combinations, such as rect~ with variable duty cycle.

The difference between the using offset and detuning is apparent. With offset, the beat frequency is the same for all notes, with detuning, the beat frequency increases with pitch. Detuning is best expressed in cents, which can be calculated with this function:

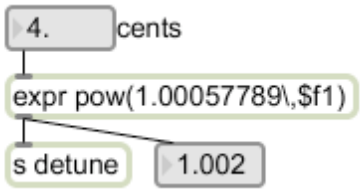


Figure 28.

The value 1.000577789 is the 1200th root of two. Figure 29 shows the wrapper patch.

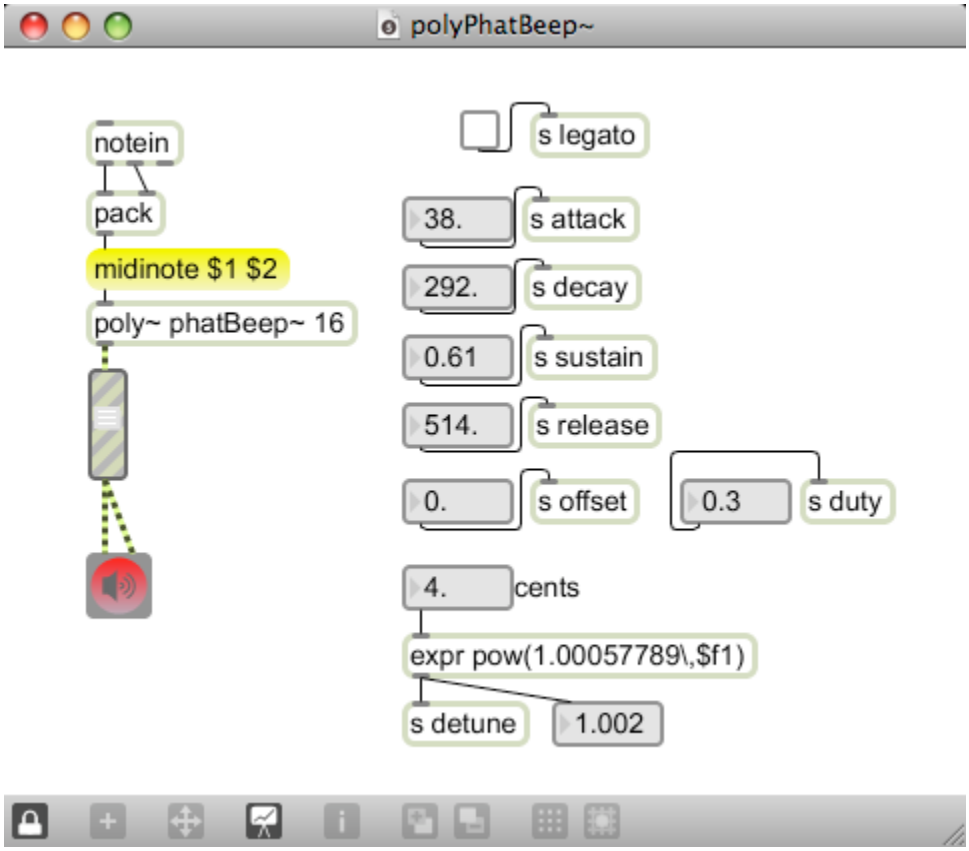


Figure 29.

Dynamic Filters

Multiplication is not the only way to apply an envelope to a signal. If the envelope controls the cutoff frequency of a lowpass filter, the sound will be controlled in a interesting way. With a harmonically rich input, the fundamental will appear first, followed by additional partials as the filter frequency sweeps upward. On the release, the process is reversed, with slow releases tailing into a pure pitch that fades away. The result is often quite attractive. A lot depends on the type of filter and the precise settings. Here is another patch to experiment with:

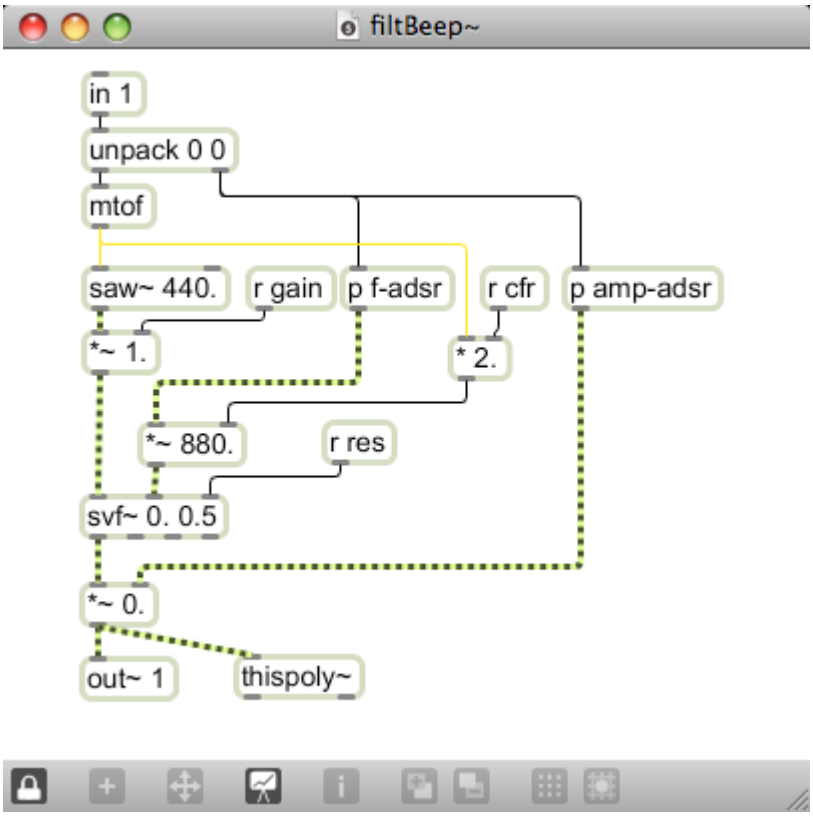


Figure 30.

This is similar to what has come before, but a state variable filter, svf~ has been added before the final *~. The arguments to svf~ set the frequency to 0 and the resonance to 0.5. The subpatchers f-adsr and amp-adsr each contain an adsr~ and assorted receive objects to set the envelope parameters. These are shown in figure 31. Note that the velocity input is scaled to a range of 0.-1.0.

If you follow the cords in the filtBeep~ patch, you can see that the note number is converted to a frequency by mtoa. This is multiplied by 2 or anything received as cfr (center frequency ratio) to set the base frequency of the filter. The filter envelope is multiplied by this.

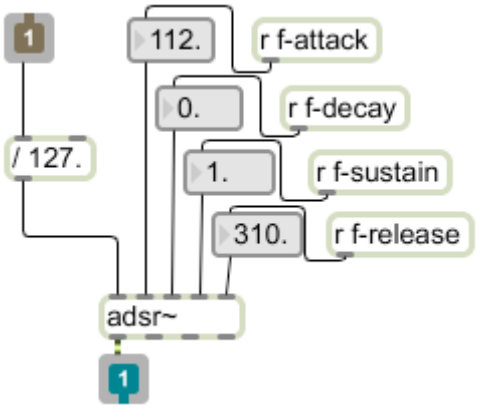


Figure 31.

The value received as res also has a profound effect. This is controlling the resonance of the filter, which is a tendency to exaggerate frequencies near the cutoff. With res near 1.0 the partials of the signal will be heard individually as the filter sweeps. Lower settings give a less amusing sound. The gain parameter reduces the output of the saw~ object. This may be come necessary when the resonance of the filter is increased.

Figure 32 is a controller patch for filtBeep~.

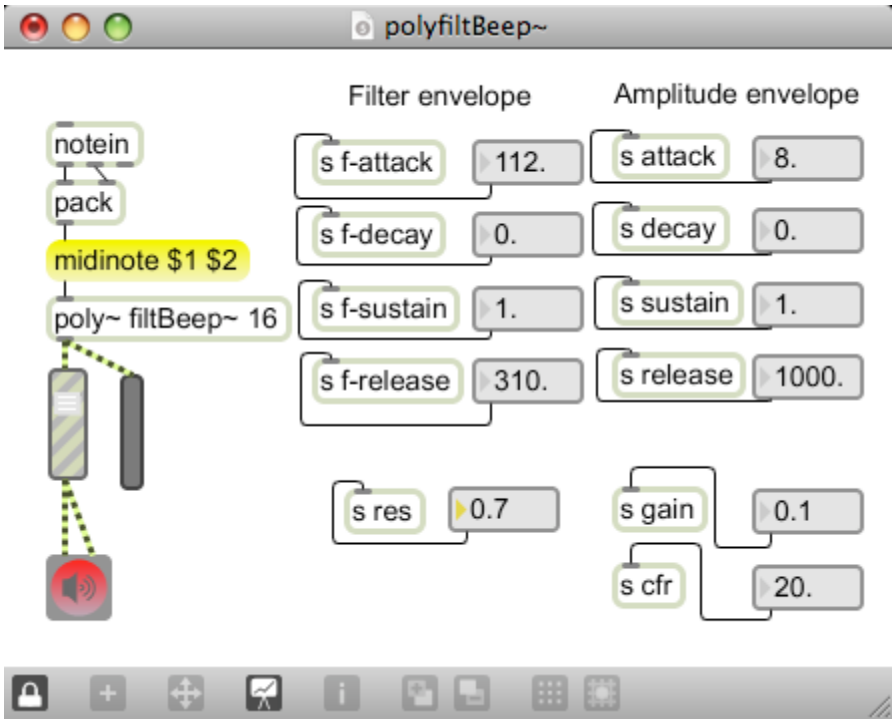


Figure 32.

FM

Tutorial 11 gives a pretty good demonstration of FM. Figure 33 shows my version, which behaves much like the operators in FM8.

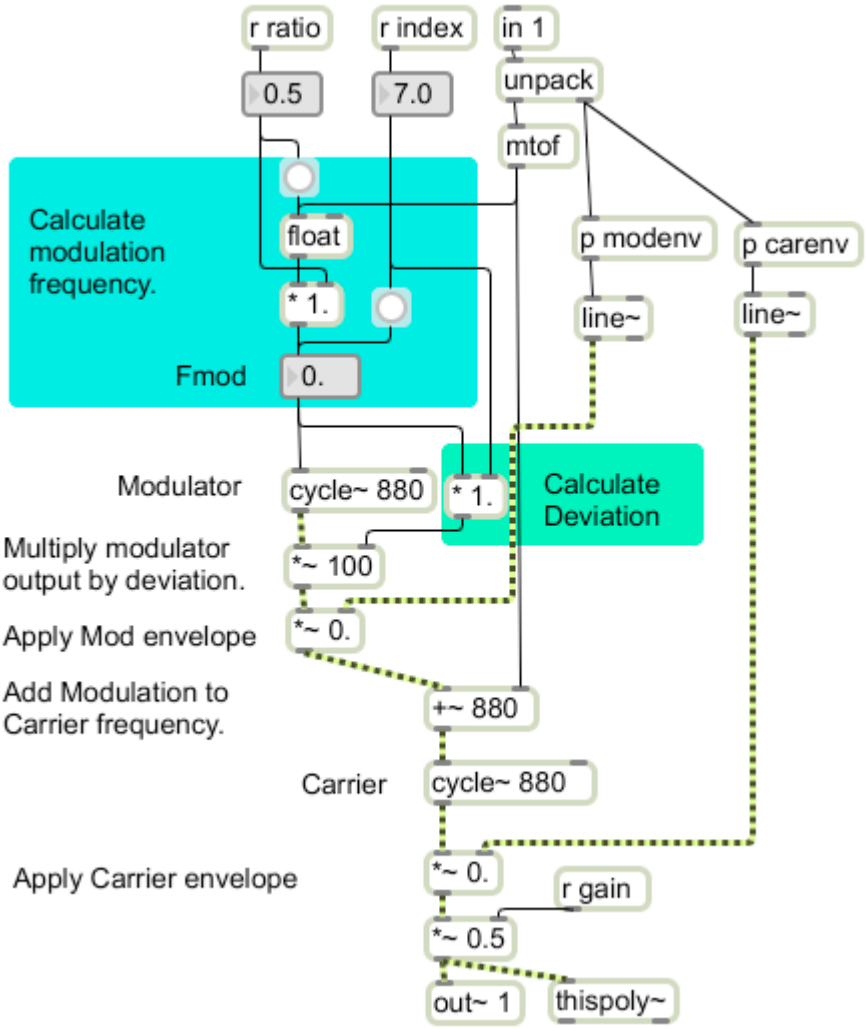


Figure 33.

Simple FM requires two oscillators, patched so one (modulator) modifies the frequency of the other (carrier). The amount of modulation is set by the index and frequency of the modulator. In classic FM, the modulation frequency is related to the carrier frequency by a specified ratio. Thus, on each note, the modulation frequency is derived from the pitch and the deviation is calculated from the modulation frequency and the index. The output of the modulator is added to the pitch frequency and applied to the frequency control of the carrier. The modenv and carenv subpatchers provide individual attack and decay for carrier and modulator.

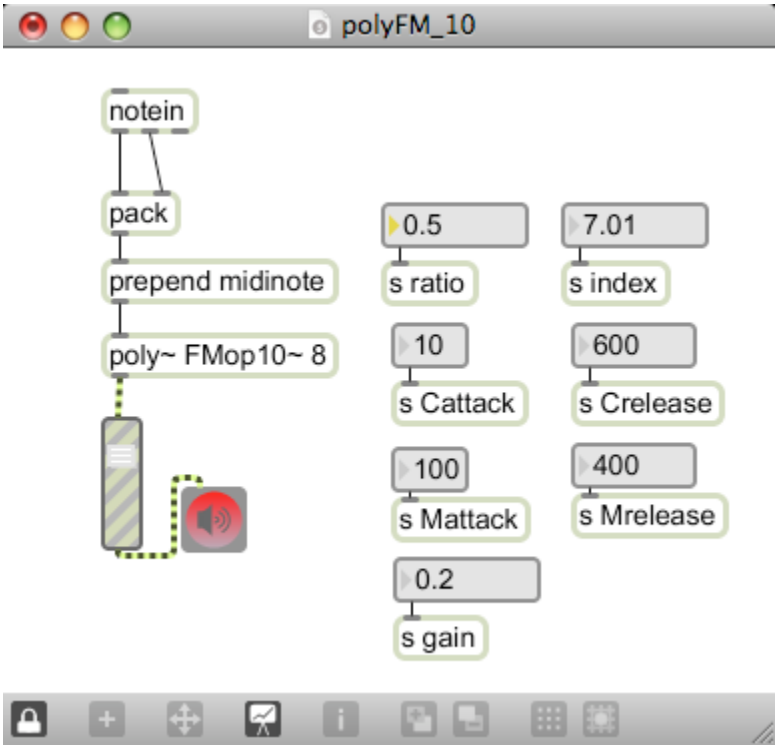


Figure 44.

There is a more detailed exploration of this in the tutorial Max & FM.

Polyphonic Additive Synthesis

The advent of powerful processors has given live performance a tool previously reserved for off-line file rendering: additive synthesis. With the oscbank~ object, it is possible to draw a spectrum and play the values directly. Figure 45 is the wrapper patch. The desired spectrum is a list produced by multislider.

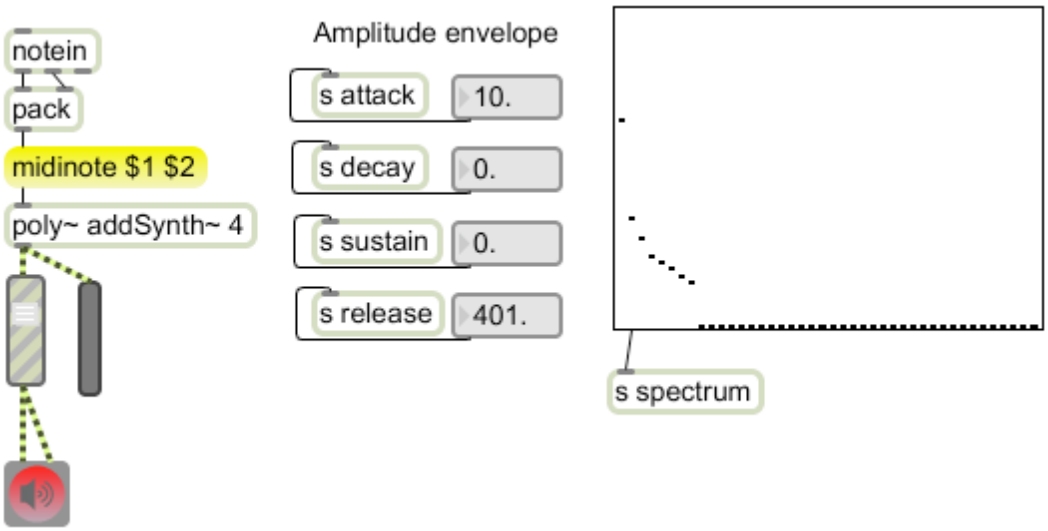


Figure 45.
The addSynth~ voice patch is relatively simple.

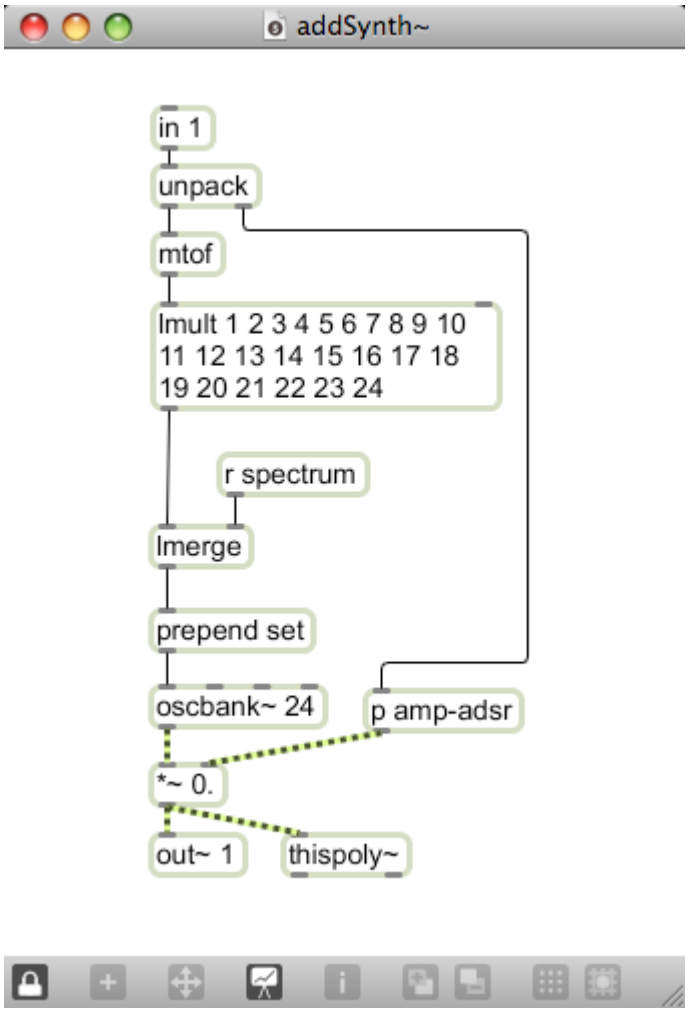


Figure 45.

The Lmult object (an lobject) generates a harmonic series from the input pitch. This is then merged (frequencies alternating with amplitudes) and applied to an oscbank~. The envelope is taken right out of figure 30.