MUS 4711 –
Interactive Computer Music

Course Lecture Notes

MODULE 2: Audio Processing

PART 7: FX Dissection and Pedal Board 1

Objects – [send~], [receive~], [dspstate~] (discussed, not included), [overdrive~], [delay~], [nw.gverb~]

Prep - [send~] and [receive~] function exactly the same as [s] and [r]. They require a name as an argument and use that to route between the pairs without any wiring. The only difference is that these are for audio, and their names cannot be abbreviated.

In every case, I've used a [live.slider] to control Dry/Wet balance (attached to an [expr 1 - $f1] to invert). The reason for [live.slider] as opposed to a regular [slider] is that it has the ability to exponentially scale the output, which is how we perceive things like pitch/volume. It's also useful for dealing with scaling times for delay, feedback, etc. They also have another MAJOR important feature. They can have an initial value loaded with the patch – it's like having a bonus [loadmess] built in!

More on the Dry/Wet balance: you'll notice that I'm **not** scaling the output of either signal. This is because the balance slider is doing that for me. While [receive~] *is* summing the signals, we don't need to do any multiplication or division by the number of signals going in to it. We're all good!

Delay – The simplest of all effects, and one of the most effective. A signal going into [delay~] is delayed by a number of samples (**not** MS). The fact that it operates on samples makes it considerably faster and higher resolution than the [tapin~] [tapout~] method, but does mean that we need to multiply it by the sample rate. If you're not sure of the sample rate (which you should be…), use [dspstate~] to get that information.

Reverb – Classically, reverb can be created using a number of delays and feedback. Unfortunately, [delay~] doesn't like feedback, and [tapin~] and [tapout~] aren't sample level accurate. Fortunately, Nathan Wolek (professor at Stetson University) has created a library of REALLY user-friendly externals for MaxMSP that includes the classic Gverb algorithm. From the Package Manager install the lowkeyNW library, then create the [nw.gverb~] object. Two inlets, signal is hot, time in MS is cold. Again, super easy.

Now, let's compare that to the reverbs modeled by Tom Erbe (UCSD professor and reverb aficionado). Things will get a little complicated as we look at this…

Overdrive – Overdrive works by using waveshaping to deliberately exceed the maximum audio signal range of a system (usually +/- 1), resulting in "soft clipping," a removal of values above/below the range that's pretty close to vacuum tube overdrive… Just digitally managed. Watch the [scope~] to see it in action. Note that the sliders for [overdrive~] start at 1 as "off." Also note that the sliders have two different value ranges allowing for different aggressiveness of overdrives. Finally, be aware of the Amp scaler. While a signal that's being processed by overdrive distortion is not *nominally* or *practically* louder than any other digital signal, it is *perceptually* louder as more of its values are hitting the upper ceiling of the DSP's range. This lack of contrast can make it sound much louder than it actually is, which is why the Amp control is included.

Chorus – In order to simulate the sound of multiple people performing the same thing as an ensemble (including slight tuning deviations), a delayed signal is mixed with its source. A periodic random signal is used to control the delay time, creating deviations in tuning around the input signal. Feedback is used to create a decay and slight deviation in future tuning of the processed signal. Note that the maximum rate is 8Hz, max time is 20MS, and max feedback is 50%. Beyond that, you start to approach flanging, which is closely related, but pushes farther into larger values.

Flanger – Note how insanely close this is to chorus… other than a few changes to value ranges and the obvious. Here, the [rand~] is replaced by [cycle~] to generate a periodic signal that **constantly** varies the delay from 0-20MS *in a periodic fashion*. This means that two identical signals are being mixed together, with a small but changing period, creating the effect of multiple peak/notch filters filtering the signal, *but in a harmonic relationship that emphasizes the partials similar to manually cranking the knobs on a comb filter***.** Classic effect and a personal favorite.

LFO – Tremolo, plain and simple. Or at least it would be except I'm allowing you to select the type of waveform used to create the effect! All three oscillators ([cycle~], [phasor~], and [tri~]) receive the same frequency information, and are all sent to multipliers that receive the same depth value. After that, they are multiplied by a toggle (0 or 1) that is controlled by a [umenu] and [sel]. When a given oscillator type is selected, [sel] sends a bang to the corresponding output, which sends a 1 to the oscillator's [toggle], and 0 to the other two [toggle]'s! this is a great control trick, and is slightly more CPU efficient than some graphical methods.

For a bonus, the frequency slider moves into the lower audio range (33Hz). Take that further into the audio range for ring modulation…

Synth (Audiopocalypse) – This is a personal creation. It's essentially fixed rate additive synthesis where the five sine waves modulators are not in a harmonic relationship to the fundamental (which would be the signal going in). These are summed with the original via a [*~] instead of a [+~] to keep them from constantly outputting a signal even when there is no input.

The result goes through two [onepole~] filters. The first is a standard lowpass, then the result goes through the second, which is subtracted, making it function as a highpass. The final output is amplified similar to the Overdrive effect.

Max Puzzle 6 (basic version) – Pedal Board Part 1: Now that you've examined some of the classic audio effects, let's combine them into a pedal board. Arrange them in sequence so that the audio flows from an [ezadc~] through four of the effects we've modeled in sequence to create a truly awesome guitar sound. Make sure that all of the "pedals" are labeled and arranged in individual panels in Presentation mode with all the controls labeled. They should be laid out in the same sequence you are using them. Save your settings with a [preset], you should include at least four cool/effect transformed ones, and one pass through version (I usually put this in the last slot). **Remember that you can easily test the sounds with one of the included audio files in MaxMSP, and then replace it with an [ezadc~]**.

PART 8: FX Dissection and Pedal Board 2 (Advanced/Optional)

Objects – [p], [inlet], [outlet], [pfft~] and [gizmo], [retune~] (discussed in contrast to [gizmo]), [transratio] (built-in MaxMSP subpatch to calculate [gizmo~] transpositions), [t b f], [f], [bpatcher], [pattr], [pattrstorage]

Review Exponential mode in [live.slider] and [live.dial]!

FM Synth – Similar to the FM synth, but as in Audiopocalypse, the input signal is the modulator, and the sine wave is the carrier. The [kslider] or frequency slider sets the carrier frequency, depth controls depth of modulation. An envelope follower subpatch (which we'll look at in depth in week 12) tracks the amplitude levels so that it only outputs a signal when something is actually happening (e.g. no drones).

Two things to note on subpatches: first, they are always created with the object [p] then a name for the subpatch. Second, they communicate with the patch a level above using [inlet] and [outlet] objects. They are **not** the same as [in]/[out], as you can see from their shape. They have no type (they can be audio or data) or arguments. They are labelled based on horizontal position left to right from 1 – X. Commenting these with regular comments and with the Comment Behavior in Inspector (creates a mouseover hint) is ***HIGHLY*** recommended.

PitchShift – Here's where it gets complicated. Like on a traditional sampler (EXS24 in Logic…), it assumes that your root note is C3. Reset that as needed if it's functionally important. Then adjust the note on the [kslider] in relation to C3 – up or down transposes by that number of semi-tones. The root key goes into a [t b f] object, which when triggered executes its actions sequentially from Right to Left – first it outputs the float that it received, then it bangs the [f] stored float, sending those to [- 60.] This goes into the [transratio] abstraction built in with

MaxMSP – all it does is transpose MIDI to pitch ratios for [gizmo~], a pitch shifting abstraction that exists inside a [pfft~] (Polyphonic Fast Fourier Transform). Essentially, an FFT breaks a signal into a set number of samples (must be a power of 2), with an overlap factor. It handles window size automatically. Then, it splits the signal into real and imaginary (spectral) data, as well as a sync signal. The long and short is that it lets you edit data in the frequency domain independent of the time domain.

You don't have to know *how* this works, just that it *does*.

Once that's done, [gizmo~] works by analyzing the FFT bins, detecting peaks in the spectrum and shifting them along the frequency axis. Thus, pitch shifting the input signal. There will always be a slight delay introduced into the equation via this process as well – the larger the transformation of pitch, the longer the delay.

LinearPan – This is how we've been controlling panning and Dry/Wet balance up until now. Pretty easy, but it has a slight problem when the two signals reach 50%...

CosinePan – This version applies the Cosine Panning Law. What it does is applies a cosine wave scaled and offset as a multiplier to the incoming signal, varying the rate between left and right, but it scales the signal as it approaches the center point so that there is no accumulation of the signal above the maximum value! Super useful!

[bpatcher] – No PRESET CONTROLS by default. You'll need to use [pattr] with #1_ Variables. This lets you generate a unique context for each variable in the bpatcher which is passed on to the top-level patcher and it's [pattrstorage] object. You can then store/recall using the messages, or interpolate between them using a float. The @savemode argument is set to attempt to autosave whenever the patcher is saved. If it can't, it will prompt the user to save. The saved JSON or XML files should have the same name as you gave to the [pattrstorage] – if it does, it will attempt to reload that on start. Kind of a pain, but VERY useful for communicating with [bpatcher] objects you want to save/recall.

Another, potentially more useful application is the fact that you can hook up a float (any type) to [pattrstorage], and it will interpolate smoothly between sequential saved states. Think about the applications for *that*…

Max Puzzle 6 (advanced version) – Pedal Board Part 2, now with [bpatcher]: Similar to last week, create a pedal board, but with ALL the effects we've used in [bpatcher] versions – Chorus, Delay, Flanger, FM synth, LFO, Overdrive, Pitch Shifter, Synth, Reverb, and a Panner of your choice. Arrange them in the order you are using them. All [bpatcher]'s must have [pattr] objects associated with all UI elements (sliders, dials, etc.), and the top-level patch will need [pattrstorage] with at least four cool versions saved and one "default" (use save slot 1 for the default this time). All [bpatcher]'s should conform to the same design aesthetic in terms of panel and UI colors and height. Audio will need to flow from an [ezadc~] through the pedal

board, to two [gain]'s (due to the panning), and out to an [ezdac~]. **Remember that you can easily test the sounds with one of the included audio files in MaxMSP, and then replace it with an [ezadc~]**.