MUS 4711 –
Interactive Computer Music

Course Lecture Notes

MODULE 1: MIDI and Instrument Design

PART 3: Filters and [poly~]

Objects – [biquad~] and [filtergraph~], [cascade~], [comb~], [teeth~], [allpass~], [reson~], [lores~], [onepole~], [svf~], [poly~], [midiformat], [in] and [in~], [out] and [out~], [s] and [r], [polymidiin], [mpeparse], [scale], [clip]

Biquad and Filtergraph – There are a lot of filters in MaxMSP, some of which are more useful than others. We'll start with the most useful of them, [biquad~].

[biquad~] filters a signal based on the filter coefficients you send it. These are a royal pain in the neck to calculate (see Cipriani & Giri or Loy if you really want to do this…). Fortunately, there's an object that will do this for us [filtergraph~]. [filtergraph~] is a graphical object that lets us set the filter cutoff, gain, and resonance (Q). When hooked up to an [attrui] (edit_mode), we can also set the filter type for [biquad~]: lowpass, highpass, bandpass, bandstop (notch), peaknotch (hybrid of bandpass and bandstop), lowshelf, highshelf, resonant, and allpass. There's also a "0 mode" that's display only, making [filtergraph~] unresponsive to other changes.

In addition to the graphical edits, the three right-most inlets can also control cutoff, gain, and Q of the selected filter in [filtergraph~].

Cascade and Filtergraph – [cascade~] is a cascaded set of [biquad~] peaknotch filters. This means that it essentially acts as an EQ with a number of bands equal to the number of filters (1-24) set in [attrui] with the (nfilters) setting. It can get quite messy though. I recommend resizing [filtergraph~] and/or limiting yourself to a sane number of filters…

SVF – [svf~] is a state-variable filter based on the algorithm by Hal Chamberlin. It takes a center frequency (up to 1/4 of the sample rate – this is important), and a Q value as arguments. Big deal, seems like another bandpass – except that's not all it is. The [svf~] is actually four filters in one. From left to right it simultaneously outputs a lowpass, highpass, bandpass, and notch filter, all using the same cutoff and Q values. This makes it stupidly useful for creating glitch or techno filter effects.

Other Filters – There are too many to list! But here's the major ones to know. [comb~] and [teeth~] are both comb filters. [allpass~] has a flat response but a complex phase response, delaying short transients in interesting ways. [reson~] and [lores~] are resonant bandpass and lowpass filters respectively – great for adding complexity to a signal. And finally, [onepole~] is a

one-pole lowpass – very useful for getting rid of problem signals in the high end, or in the low end after we make it into a highpass…

Making a Highpass Resonant Filter – You've probably noticed that [lores~] and [onepole~] are both lowpass filters. But what if you want a resonant highpass filter? It's actually really easy. If you take a signal and send it to a lowpass, then subtract that output from the original signal you get a highpass. Signal minus lowpass = highpass. That's it!

Poly 1 – Last week, we created basic MIDI monosynths. If only there was a way to make them polyphonic… Oh, wait. There is! Let's reopen a basic synth patch, and copy everything between [midiparse] and [gain~] (bordered in cyan) and copy that to an abstraction, a reusable bit of MaxMSP code that can be opened in another patch and reused as if it were any other object. I'm saving this as "polyExpCore."

To prep this for use in another patch, I need to make a few small changes. First, I need to add an [in] which passes data into a [poly~]. I can also use [in~] for audio signals. The output equivalents are [out] and [out~]. In either case, they all take a number as the argument – this corresponds to the inlet or outlet it will be assigned to. Pretty self-explanatory.

Next, I want to send data to the [adsr~] and to balance the sine waves and noise volume. I can't access any UI interface objects inside a poly, so I'm going to instead use [send] and [receive] to communicate with ALL instances of the balance and [adsr~]'s (I'm making an assumption that we want every note in a chord to have the same envelope…). These can also be abbreviated as [s] and [r]. Each of them takes an argument of a name, with a [s] sending to all [r]'s with the same name (or multiple [s]'s sending to the same [r], which will sum them). To make it easy, use one named pair per parameter you want to control from a different patch or patch level – I'm using noiseVol, envA, envD, envS, and envR, all of which will need corresponding [s]'s in the master level.

Finally, I've added a [thispoly~] and connected it to the first and third outlet of [adsr~]. This takes the signal and mute out of [adsr~] and uses that to turn the voice on or off within the [poly~], preventing stuck notes, drones, or accumulated feedback. It's also really efficient in terms of CPU usage.

Poly 2 – Here's how we create the polyphonic context. First, there's [poly~] with the arguments for the name of the abstraction we're using and the number of voices. Here it's polyExpCore and 8. To get data into my [r] objects, I create the corresponding [s]'s, make sure they have the same names as arguments for my [r]'s (noiseVol, envA, envD, envS, and envR), and hook them up to dials with the correct ranges (which I copied from my original – I told you to reuse your code!). To test this, I can open polyExpCore as a separate instance and test the values received.

We're nearly there, but we need to get MIDI *in* to [poly~] first. To do that I'm using the standard [midiin]-[midiparse], but now I'm connecting that to [midiformat], which formats the data as MIDI event messages, and taking the right outlet of [midiformat] and connecting that to

[poly~]. Note that while [midiparse] has a MIDI event out, it's REALLY different from what comes out of [midiformat]… we'll see why in a bit.

Coming out of [poly~] is the audio, so that goes to [gain~]-[ezdac~] and a few visualization objects. If it works, we'll look at the changes for MPE.

Poly 3 – MPE – Multidimensional Polyphonic Expression or MPE allows you to send much more data via MIDI on a per key basis. For example, you can add vibrato (through X axis pitch bend), filter cutoff (Y axis), and volume (aftertouch, Z axis, or pressure – depends on the device) on a per key basis. Compatible MPE devices include the ROLI Seaboard and Blocks, Linnstrument, Sensel Morph, Keith McMillen QuNeo QuNexus and K-Board, Haken Continuum, Eigenharp, Madrona Soundplane, Expressive-E Touché, and the Joué controllers.

Adapting our previous [poly~] patch is pretty easy. We'll start by adding the @hires 2 attribute to the [mpeparse] where [midiparse] was to enable high resolution pitch bend. Then connect [mpeparse] directly to [poly~]. I've also added a [s pB] to control the pitch bend range, and deleted the [s noiseVol] as I'll be controlling that with CC 74.

Inside the MPEpolyCore abstraction, you'll see the major changes. To deal with MPE, I need to use [polymidiin], going to [mpeparse] this is like [midiparse], but for MPE data. I send the pitch bend data to a [clip] which constrains values to within the range set as arguments, then use [scale] to map those values to a different set for the purpose of offsetting the pitch through bending. The [r pB] takes the bend range, splits it and inverts one version, then sends these to the low/high output values of scale. Unpacked velocity goes to [adsr~] as normal, while the pitch is [pak]'ed as floats with the bend data (to keep it continuous and avoid jumps), summed, and then sent to the [mtof] objects. Last but not least, I'm taking the CC out of [mpeparse], routing it to select only CC 74 data, and then scaling it to control the balance of sines and noise. Watch it in action…

Max Puzzle 4 – With your new understanding of filters and [poly~], construct a synth that pairs a filtered noise source (your choice on type) with two of the same type of antialiased oscillators, one of which is slightly detuned (I recommend adding or subtracting decimals before [mtof]). The filtered noise cutoff frequency should track with the MIDI notes input to control a thin bandpass filter that decays very quickly, leaving the oscillators to sustain. Unlike last week, everything should be controlled by a [midiin]. You may use any needed number of sliders or dials, and any other objects discussed in class as needed, but keep in mind that you will need to use [s] and [r] to communicate with the poly voices. **Remember that [poly~] requires you to use abstractions with [in] and [out~]**. After [poly~] everything should go to a controllable lowpass filter, then to a [gain~] and an [ezdac~] at the end. **HINT – Adapt your previous week's synth**. **You don't need to reinvent the wheel, and reusing code is a VERY good habit to get in to…**