Composition 001

Andy Farnell

9th June 2007

1 Synopsis

We are going to compose a very simple piece of music in Puredata. We will be introduced to a sine wave oscillator, a line segment and simple filters. Starting with a counter we will establish a timebase and use [select] operations to create part melodies. A separate note sequencer for each of four tracks will run from the same timebase. Creating abstractions, unpacking lists and modulo arithmetic are introduced.

New units used

- `[f]`: Store a float value.
- `[line~]` Audio rate line generator.
- `[mod]` Modulo operator.
- `[select]` Bang one or more outlets matching message.
- `[vline~]` More versatile audio line.
- `[unpack]` Distribute a list of values.
- `[swap]` Exchange two values.
- `[moses]` Split a stream of values at threshold.
- `[noise~]` Random audio signal, white noise.
- `[bp~]` Band pass filter with resonance.
- `[lop~]` Low pass filter.
- `[throw~]` Send audio to a multi connection bus.
- `[catch~]` Audio bus destination.

2 counting timebase

Figure 1 summary

- inlets start/stop and period
- each step is previous step plus one
- outlet bangs
- outlet time

Making a counter Let us begin with way to divide time. Place a metronome object set to 125ms (8 beat at 120bpm), add a float object, `[f]` and an addition object, `[+]`, and link them as shown so that the next value held in the float will be its last value plus one. You can verify that the counter works by attaching a number box to the float output and a toggle to the left inlet as previously. Make sure audio processing is turned on. Click the toggle to start the metronome running. Each time the float gets hit with another bang it advances one step. A trigger unit is used after the metro to split the bang messages into order, right to left - so the counter is incremented before the bang is sent.

Abstraction Delete the toggle and number box you are using to test the counter. Add two inlet boxes, one for the start/stop value and one for the period, and add two outlet boxes at the bottom, one for regular bang messages and one for float messages from the counter. We make the timebase an abstraction simply by saving it as a new file in the same directory as the main patch. We gave this the name `A-timebase` so when we use it in another patch we do so by creating a new object `[A-timebase]`. Abstractions are not the same as sub-patches which we created before by prefixing a new object with pd like `[pd mysubpatch]`, you must remember to save each abstraction you edit because it isn't a part of the main patch, the main patch just refers to it. Unlike sub-patches an abstraction also has its own memory space when loaded, so you can duplicate abstractions without worry.

A-timebase.pd

3 synth 1

Figure 2 summary

- trigger envelope
- set attack and decay
- MIDI to Hz
- line object
- amplitude envelope

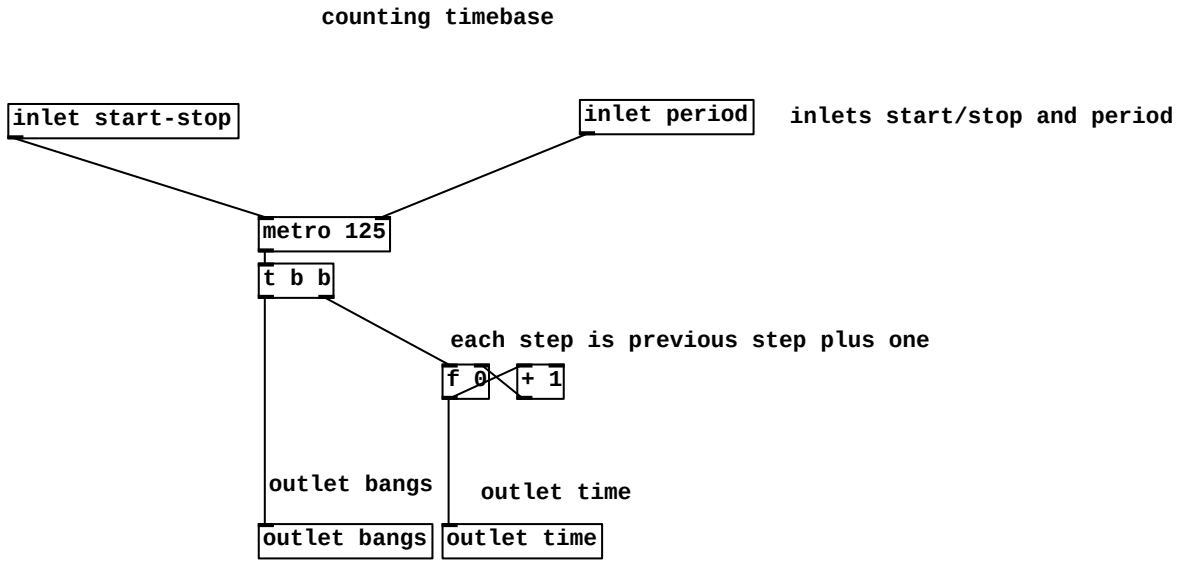

Figure 1: timebase

A simple 1 oscillator synth We need some sounds. If we take a `[osc~]` oscillator and multiply it by a falling line that will give us a basic decaying tone. The `[osc~]` oscillator has frequency as its first control inlet and one audio signal output. We want an amplitude envelope control signal that starts off at maximum and decays away to zero. The `[line~]` generator takes message arguments of the form (level to go to, time to get there), so a message `[1 0(` will send the line to a value of 1 in 0 milliseconds. Sending a further message like `[0 700(` will set it moving back to zero over 700 ms. `[1 0, 0 700(` is a pair of messages in a single message box. The comma causes both messages to be sent in left to right order instantly, so this will send the line to one in 0 milliseconds, and then back to zero in 700ms. A line envelope is a powerful component of more complex controls, we will also meet its more versatile brother `[vline~]` shortly. The result of multiplying the sinewave signal by the line is a pure tone that decays away.

MIDI control of frequency In order to control this synth with MIDI notes instead of Hz frequency let's add a `[mtof]` object. Notes appearing at the `[inlet]` will now be converted to Hz before being sent to the oscillators frequency inlet. B-synth1.pd

4 bassline sequencer

Figure 3 summary

- sequencer using select

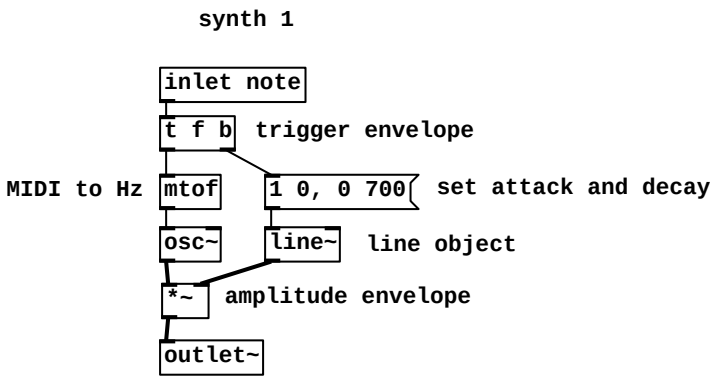

Figure 2: synth1

- note value in message boxes
- notes outlet

Sequencing the synthesiser The output of the counter in the timebase abstraction will advance forever. It is received in this patch via `[inlet time]`, which wil be connected to the timebase in the outer patch. Let's divide time into bars and measures and beats using a modulo operator. The output of `[mod n]` is bounded within 0 to n-1, n modulo n is zero, so we can get a bar counter by pushing the output from our counter through `[mod 16]`. This message stream will now count $0, 1, 2...15$ in total sixteen steps.

Picking specific notes `[select a b c]` will output a bang on one of three outputs corresponding to a, b and c if that value that matches the input message. Any ordering of values is allowed, but you will have to remember more carefully if you make a non-obvious sequence of values to match. Because the counter is cycling up over 16 steps we can use something like `[select 4 12]` to place a couple of beats in the 4 and 12 positions of a 16 beat bar. If we say `[select 0 2 4 6 8 10 12 14]` we will get eight beats on the offbeat, each activating a different bang output. That's pretty useful, so all we have to do to create a short melody is combine select with some messages. The value in the select box represents the timing pattern, the message boxes contain notes. The notes get out of the patch through `[outlet notes]` from where they will be connected to the synth. C-bassline-seq.pd

5 Playing the bass line

Figure 4 summary

- Bassline sequencer
- Bassline synth

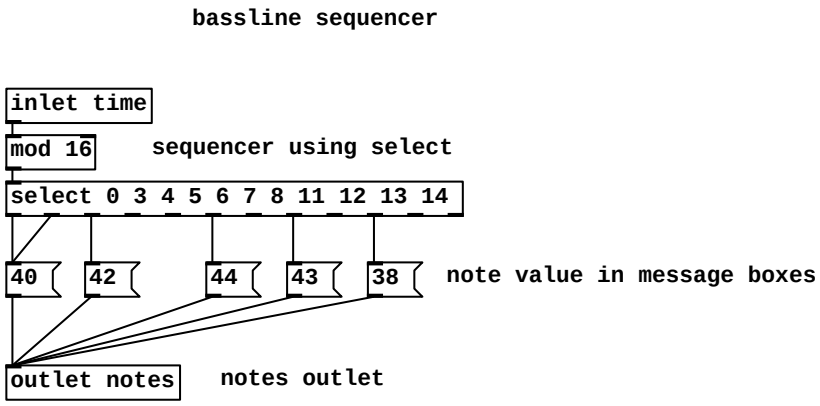

Figure 3: bassline-seq

Play the bassline The bass synth, bass line sequencer and timebase abstractions `[A-timebase]`, `[B-synth1]`, and `[C-bassline-seq]` work together in this patch to make a sound. Audio signals are sent using `[throw~]` and `[catch~]` objects, so that we can combine many synths as we go. There is link below to a soundfile of this sequence. Notice the very basic envelope produces clicks with such low frequencies.

Broadcasting variables The patch is split up to make adding more synths easier. Floats from the timebase are sent between `[s time]` and `[r time]` objects, send and receive. Think of these as bluetooth adaptors or invisible cables for puredata signals, they exist in unique pairs, several `[send]` transmitters cannot share a `[receive]`

C1-bassline-play.pd C1-bassline-play.ogg

6 synth 2

Figure 5 summary

- vibrato lfo
- set decay
- better line
- obtain square wave
- square decay amp
- rescale
- amplitude envelope

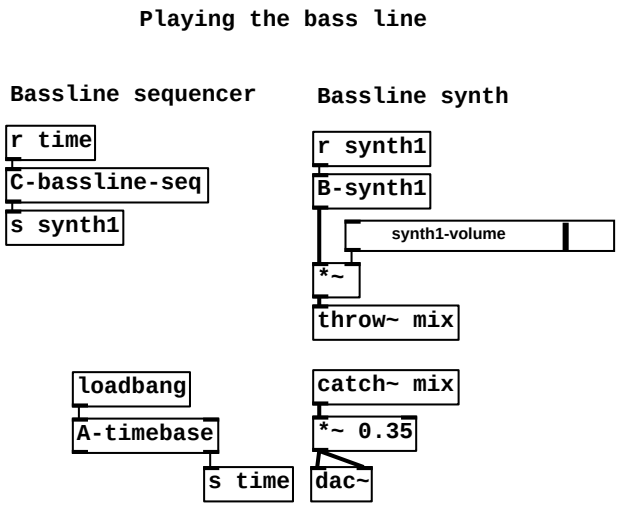

Figure 4: bassline

A better envelope Instead of `[line~]` we have used `[vline~]` which takes slightly different arguments. The `[vline~]` generator takes message arguments of the form (level to go to, time to get there, delay to wait), so a message like `[1 12 0, 0 400 12(` will send the line to one in twelve milliseconds, and then begin moving back to zero over four hundred milliseconds after a 12 millisecond delay. This allows for envelopes with a long attack phase to be specified in a single message. If we sent something like `[1 100, 0 20(` to `[line~]` it would not work. Why? Because the first message of the pair says go to 1 in 100ms but `[line~]` immediately starts moving towards 0 in 20s the instant it receives the second message, just after receiving the first. `[line~]` is an unscheduled segment generator, while `[vline~]` is scheduled and can have multiple stages all specified in the same message.

Amplitude curve To make the amplitude value decay away more quickly near the beginning we take the square of the signal. This works because the output of `[vline~]` is normalised 0 to 1. If we use `[*~]` to multiply a signal by itself we get its square as $x \times x = x^2$. One squared is one and zero squared is zero, but how does the value change over this range? It becomes curved, its rate of change becomes proportional to its value, so it dies away more quickly at the start.

Vibrato Midi note numbers are converted to Hz as before, and this time we add a constantly changing signal of amplitude 3 from another much slower oscillator running at 8Hz. Note that we used `[+~]`, the signal version of add here even though the frequency value is a message. Because the lfo sine wave is symmetrical about zero the average pitch will still be that given by `[mtof]`.

Timbre change We shape the output of our oscillator to give a different tone from the first synth. A "square wave"†6 is obtained by clipping the range of the cosine wave . This effectively chops the top and bottom of the waveform off. `[clip~ -1 1]` would have no effect on a normalised signal, `[clip~ -0.5 0.5]` will chop off half the top and bottom. `[clip~ -0.1 0.1]` removes all but the 10 percent closest to 0 creating flattened top. Multiplying it by a compensation factor brings the clipped wave back up to an audible level. As before we scale the signal path by the amplitude envelope using `[*~]`

Notes †The word `"square"` is used in two ways on this page. When we create a square-wave we are distorting a wave so that its shape appears square. When we take the square of a signal we multiply it by itself. These are two different things. D-synth2.pd

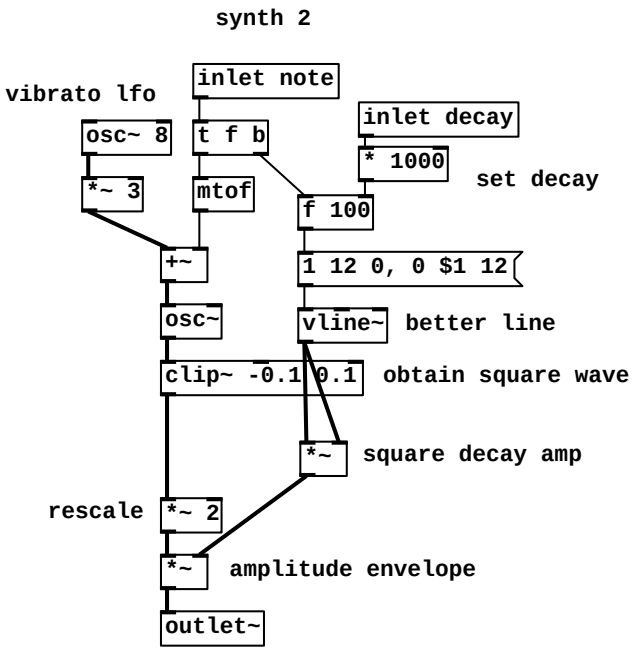

Figure 5: synth2

7 float box arpeggiator

Figure 6 summary

- cycle over 4 notes
- distribute list over 4 float boxes

- each value stored until selected
- transpose
- notes out

Sequencing the melody Wiring up `[select]` objects is fine for making fixed patterns like rhythms to store away, but not much use when you want to change a sequence rapidly without breaking connections. This time the sequence is still fixed in a particular order, just 4 steps that cycle round, but the values at each position are variable. This is sometimes called an arpeggiator, an 8 or 16 step sequencer found on many old analogue modular synths.

Unpacking lists Instead of sending each of the four notes separately into this sequencer abstraction we have just one inlet `[inlet note sequence]`, that carries a list like `[12 34 56 78(`. The `[select]` is fed by `[mod 4]` and sends a bang to each of 4 outputs in turn. These cause successive `[f]` boxes to output the float stored in them. To get the note numbers from the list we use `[unpack]` to split up the list and distribute the numbers, to the left inlet of each float box. The sequence message is always unpacked in the same left to right ordering, but remember the values appear at the outlets of `[unpack]` in right to left evaluation order just as with any Puredata object.

Transposition With a wire carrying note numbers as individual floats its easy to transpose, here we add an octave with `[+ 12]` . The third inlet, `[inlet transpose]` provides a way to control the transposition from the outside. E-melody-seq.pd

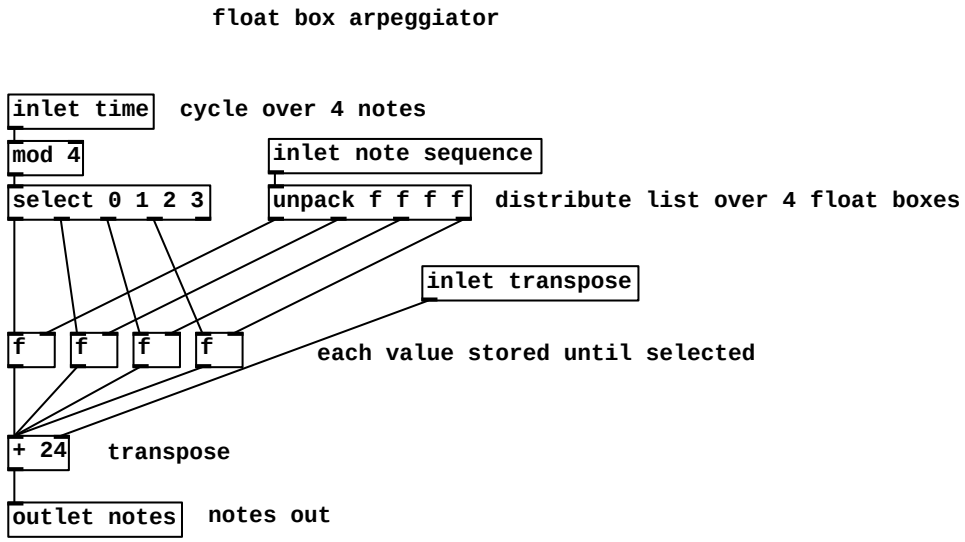

Figure 6: melody-seq

8 playing melody line

Figure 7 summary

- melody synth
- melody sequencer

Play the melody The melody sequencer and synth 2 produce this sound. Listen to the soundfile below. Hear the more "hollow" sounding square wave blip cycle forever over 4 notes. Changing the attack to 12ms reduces the click when frequency changes, and the higher pitch of this part make fast changes less obvious. E1-melody-play.pd E1-melody-play.ogg

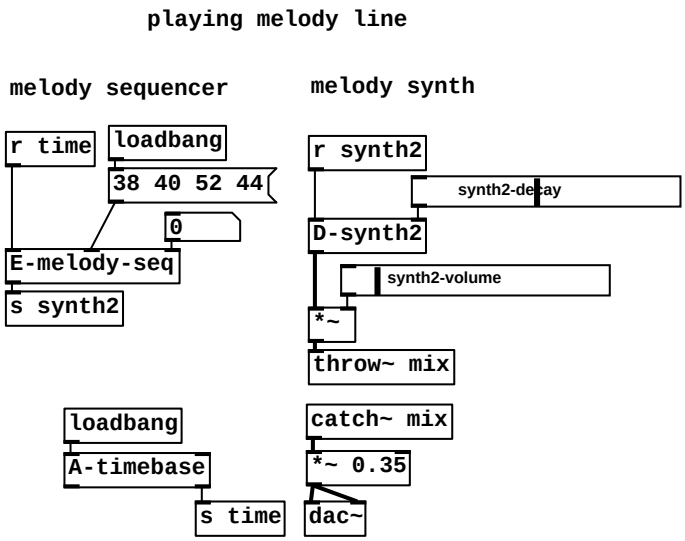

Figure 7: melody

9 synth 3

Figure 8 summary

- scale to 0 - 1
- invert amplitude range
- use note as decay time
- attack and decay
- filtered noise
- amplitude envelope

Filtered noise drums Our third synth creates splashes of noise in different frequency bands so it can be used for snare or hihat like effects. We filter some white noise using a bandbass filter, `[bp~ 100 0.5]` which has its center frequency controlled by the MIDI note frequency.

Inverting and scaling a control You can see how we adapt the MIDI note number so that we get a quieter and shorter sound when the frequency is higher. Dividing the full note range 0 to 127 with `[/127]` gives us a normalised value from zero to one. The combination of `[swap 1]` and `[-]` is an idiom that inverts a normalised signal, so that it ranges from one down to zero as the input increases. A `[swap]` object exchanges its inputs, so when used with `[-]` we get 1 minus x for an input x.

Substitution in lists For an envelope we have used an ordinary `[line~]`, activated by a message producing a variable decay envelope. The decay range is substituted in `$1`, a value ranging between zero and 400 milliseconds. Notice the second part of a message to `[line~]` can be omitted when it's a zero in the first position. Where we use $1, $2 etc†9 in a message box, each is replaced by the value of any corresponding list element that appears at the message box inlet. In this case a float in the range 0 to 400 will replace $1 in the list so it becomes something like `[1, 0 390(`, the MIDI note now controls the decay time.

Output EQ Finally we sculpt the frequency range of the filtered noise with a little EQ in the form of a `[lop~]` final stage, and then boost it back to reasonable level. `[lop~]` is a simple low pass element with one parameter for its cutoff frequency.

Notes †Do not use $0 in message box lists as this has a special meaning. F-synth3.pd

10 two bar sequencer

Figure 9 summary

- count twice as long
- extend bars with moses
- drum bar patterns

Two bars or more To extend a number of sequencers by chaining them together `[moses]` is a very useful object. The idiom shown, `[moses n][- n]`, is repeatable to create chains of events distributed along the same counter timeline. In this case two subpatches are chained together to produce two bars, 32 beats in total. While the input to `[moses 16]` is less than 16 floats will be sent to its left outlet. Note values of 16 and above will appear on the right outlet. If we subtract 16 from the second stream both now count over the 16 step range 0 to 15, but at different times thus we have created a system to chain bars together.

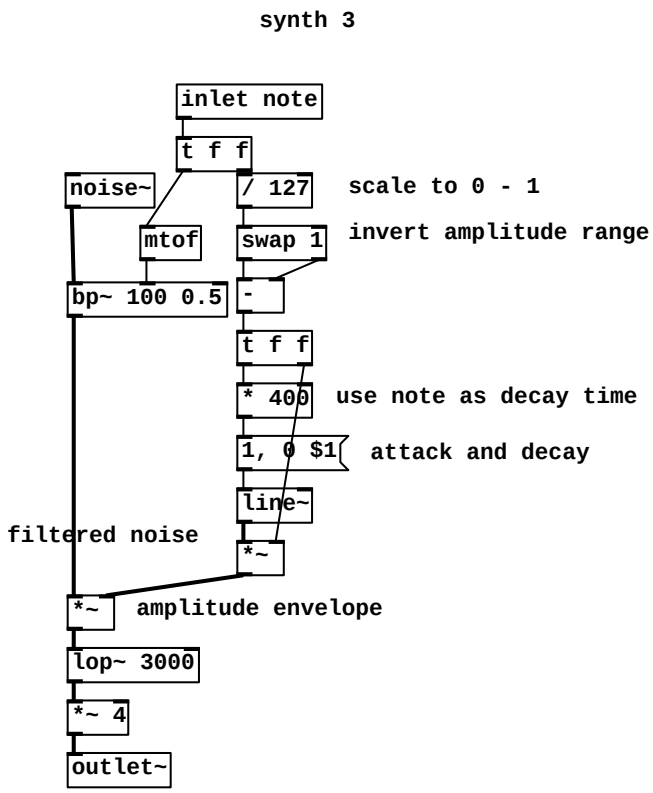

Figure 8: synth3

Two drum patterns using `[select]` are stored in separate subpatches, one for each bar. G-drum-seq.pd

11 playing the drums

Figure 10 summary

- drum sequencer
- drum synth

Play the drums The drum sequencer and synth 3 produce this sound. High bursts of noise are shorter and quieter than long, low ones. G1-drums-play.pd G1-drums-play.ogg

12 synth 4

Figure 11 summary

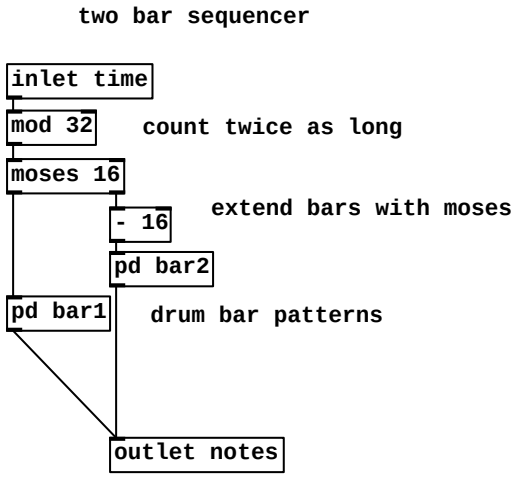

Figure 9: drum-seq

- poly voice
- 3 note poly chord
- separate trigger from list
- sustained envelope
- unpack the notes from list
- send to 3 voices
- mix and scale

Poly synth To make chords we need more than one note. Although we could treat each synth as a monophonic instrument and build up harmonies by careful sequencing, sometimes we like to be able to specify block chords that will be played in a polyphonic way. To do this we first design a single voice of the instrument. This synth is a variation on our first sinewave instrument. It has a vibrato and MIDI note input. Control of the final signal amplitude is just one multiply, so the instrument is not capable of playing separate notes as such, only chords that start and end together.

3 voice synth You can see three of the voices arranged with the frequency of each controlled by the corresponding outlet of the `[unpack]` unit. `[unpack]` splits up the list of notes sent as a chord by the sequencer. We separated the envelope trigger from the note list using [t l b] and use the bang to activate the envelope message box.

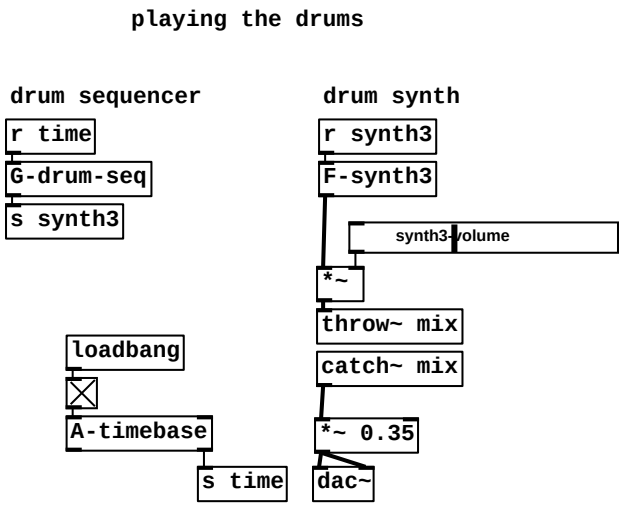

Figure 10: drums

ASR envelope Notice also that the message to `[vline~]` has a softer attack and a significant delay before the decay stage, which causes the chord to rise in 35ms and hold for a further 165ms, the remainder of the delay time, then fall to zero in 100ms each time it plays. Because 3 voices will be playing at normalised level we need to divide the output signal by 3 to get it back into a normalised range. H-synth4.pd

13 chord sequencer

Figure 12 summary

- chords stored as lists

Sequencing chords The only difference between this and the first note sequencer is that instead of a single note we send a list of three values for each chord. The pattern is 32 beats with chords being sent on the first and 15th beat. If a chord list is too long extra notes will simply have no effect. If chord lists have less than 4 members then the last note sent in the remaining positions will remain playing since no new value is unpacked.

I-chord-seq.pd

14 playing chords

Figure 13 summary

- chord sequencer
- chord synth

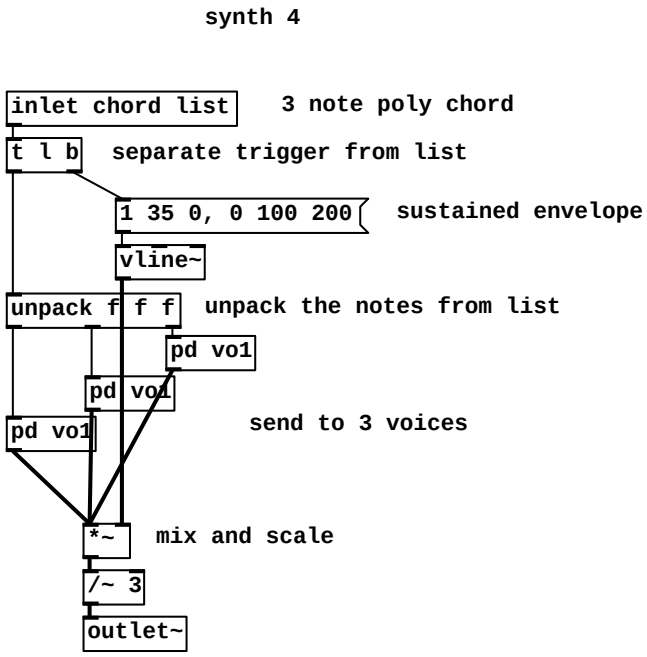

Figure 11: synth4

Play the chords The chord sequencer and synth 4 produce this sound. I1-chords-play.pd I1-chords-play.ogg

15 composition 1

Figure 14 summary

- mixer
- counter timebase
- 4 sequencers
- 4 synths with volume controls

Mixing several parts To mix two or more parts we send them to a bus with another `[throw~ mix]`. Unlike `[send~]` many different `[throw~]` units can talk to a single `[catch~]`. Before going to the bus, each channel in our mixer passes through another multiply set by a corresponding GUI fader object. The mix of audio is collected by `[catch~ mix]` and scaled to make it a little quieter.

Conclusion Here is our final composition with 4 separate synthesisers and 4 sequencers, one for each part. It plays a pattern 32 beats long with a bass

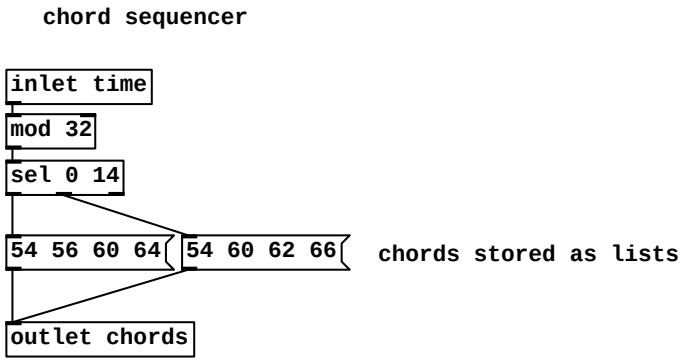

Figure 12: chord-seq

line, noise drums, arpeggio and chords. The timebase is controlled by a toggle switch so it can be started from the main patch. Z-composition1-play.pd Z-composition1-play.ogg

16 links

Composition-001.pdf Composition-001.tar.gz

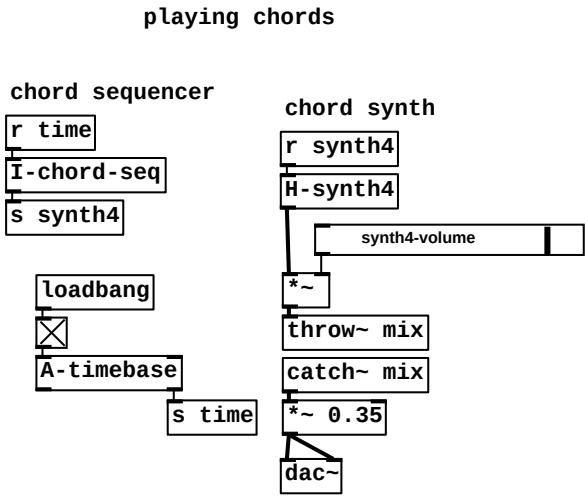

Figure 13: chords

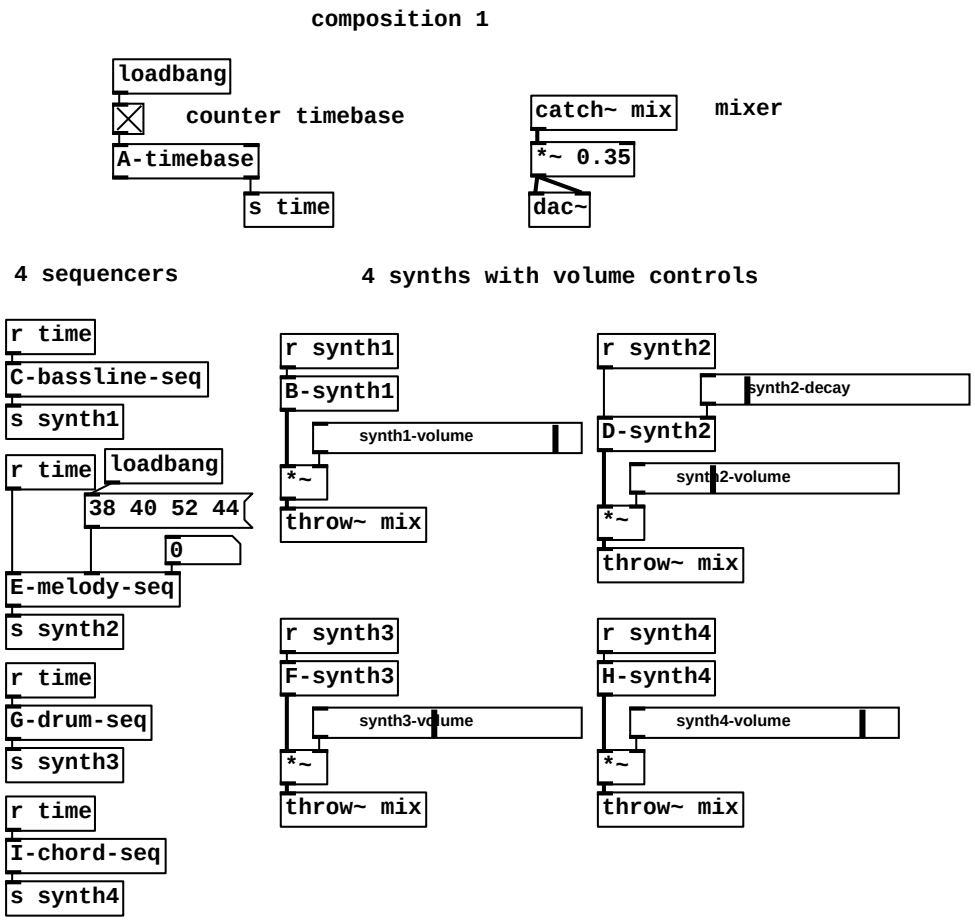

Figure 14: composition1