Circular buffer and granular synthesis

James Harkins

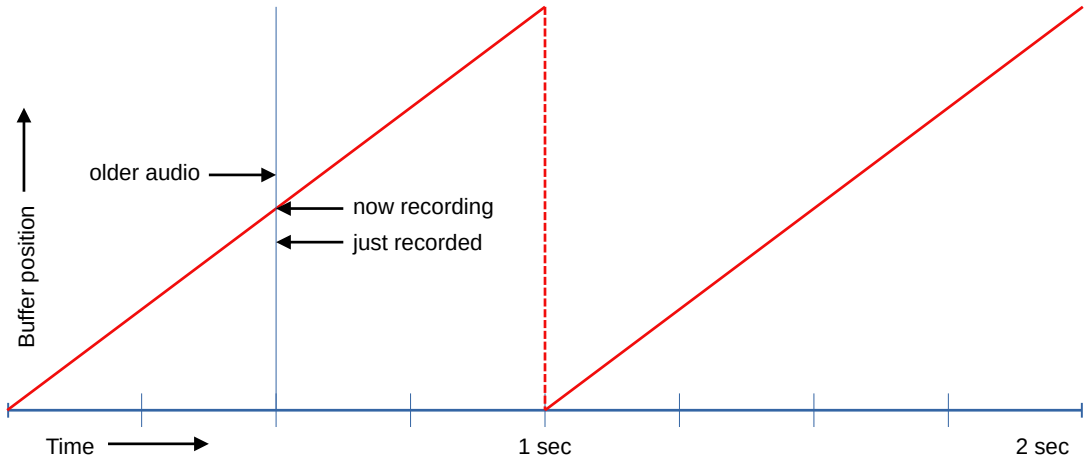
January 24, 2023

Contents

1 Recording into a circular buffer

You can imagine a "recording head" moving from the beginning of the buffer to the end. When it reaches the end, it wraps around to the beginning ("circular").

In this graph, the red line represents the position where audio has just been recorded. Slightly earlier audio is just below this line. Because the buffer is circular, you can locate older audio by continuing down below the recording point, and wrapping around at the top. For instance, at 0.5 seconds (blue vertical line), the record head is also at 0.5 seconds. A buffer position of 0.4 seconds represents audio from 0.1 seconds ago; a buffer position of 0.6 contains audio from 0.9 seconds ago.



The record head is, mathematically, a linear formula: $y_r = x \mod size$. A linear equation, in general, is written $ax + b$, where a is the *slope* and b is the *intercept*. ("Intercept" means the point where the line crosses the Y axis, when $x = 0$.) The record head's slope is 1. For convenience, we can assume that the intercept is zero (which means, for the sake of this discussion, we are counting time starting at the exact moment when recording started at the beginning of the buffer).

2 Playing grains

To play grains from a circular buffer while it's recording, there are three cases to consider:

1. Playing back at normal speed (playback rate $=$ record rate).
2. Playing back faster than normal (playback rate $>$ record rate).
3. Playing back slower than normal (playback rate $<$ record rate).

The problem is: if a line representing the grain's "playback head" crosses the record head, then playback will jump instantly from new audio to old audio, or vice versa. This will produce a click; we don't want this.

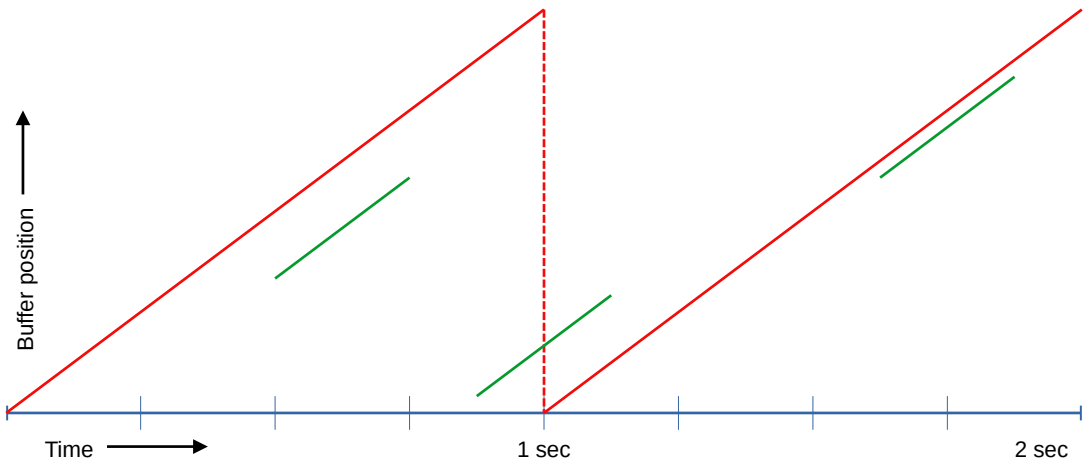
So we need to find the boundary conditions where a grain is safe.

The playback and record heads are represented by linear formulas; so, the "danger point" is the intersection of the two lines.

2.1 Normal speed

This is easy! The "playback head" for the grain has exactly the same slope as the record head. So, the lines are parallel and they will never intersect.

You can delay the grain's audio by subtracting from the record position: $y_p = (y_r - d) \mod size$.



2.2 Faster than normal

Remember that recording is at $y_r = x \mod size$.

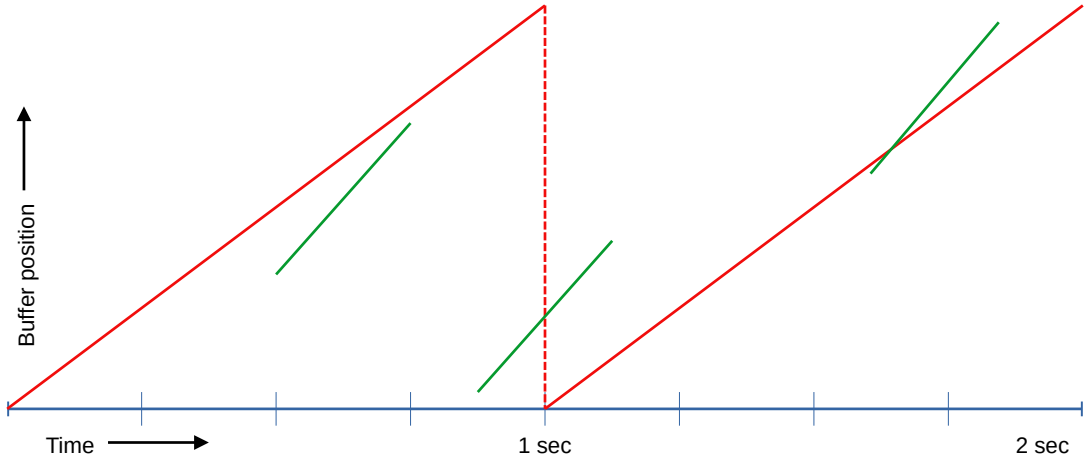
Playback at a different rate means $y_p = (ax + b) \mod size$.

If playback is faster than normal, then $a > 1$.

Here, the left two green lines are OK. They started early enough that the grain ends before it would intersect with the recording line.

The rightmost green line starts just slightly earlier than the recording position. Because it moves faster, it crosses the recording line. This is bad.

So, there is some amount of delay are relative to the recording position that is safe. If grain playback starts earlier than this position, then it's OK. If it starts later, *click*.



We know when the grain starts and when it ends. How do we know these? Because we decide when to play the grain, and how long to play it.

Since playback is faster than normal, we are only worried about the endpoint.

Known:

- x = Time at start of grain.
- t = Grain duration.
- $x + t$ = Time at end of grain.
- Record head = x (because its slope = 1). Also we can ignore modulo for now (just reapply it at playback time).
- a = Desired play rate.
- So playback is at $ax + b$ where we don't know b yet.

First step, find b so that the lines cross at $x + t$.

$$a(x + t) + b = x + t \tag{1}$$
$$b = x + t - a(x + t) \tag{2}$$
$$b = (x + t)(1 - a) \tag{3}$$

Then plug this into the playback position for x.

$$ax + b = ax + (x + t)(1 - a) \tag{4}$$
$$= ax + x - ax + t - at \tag{5}$$
$$= x + t(1 - a) \tag{6}$$

$a > 1$ in this case, so $t(1-a)$ must be negative, earlier than the record head. So this is logical.

So, if the current record position is 0.5, and you want to play a 0.1 second grain, with playback rate = 2, then the latest safe starting position is $0.5 + 0.1(1 - 2) = 0.4$. The grain's ending time is 0.6, and indeed, safe start time 0.4 + 0.1(2) = 0.6.

2.3 Slower than normal

Similar logic, except that the value $x + t(1 - a)$ is the *earliest* safe starting point, rather than the latest. If the current record position is 0.5, and you want to play a 0.1 second grain, with playback rate = 0.5, then the earliest safe starting position is $0.5 + 0.1(1 - 0.5) = 0.55$.

(Usually, in live granulation, we want to keep the delay time relatively short, that is, $y_r - y_p$ should be a relatively small positive number. So you will normally not encounter collision for slower grains.)

3 Code examples

3.1 A note about units

When implementing this technique, keep in mind that three ways of representing time will be involved:

Time unit	SuperCollider objects	Pure Data objects
Samples	Phasor, BufWr	tabread4~, poke~
Audio seconds	x + t(1-a)	x + t(1-a)
Buffer lengths	GrainBuf (start parameter)	phasor~

Each is needed at different stages. So you will need to convert time values at some points.

It's helpful to think in terms of units: samples per elapsed second, audio seconds per elapsed second, and buffers per elapsed second. Then, for instance, to convert samples/second into audio seconds/second:

$$\frac{samp}{sec} \times \frac{audiosec}{samp} = \frac{audiosec}{sec} \tag{7}$$

It might already seem a bit amusing to think of audio seconds vs. seconds—shouldn't they simply cancel out? In fact, the charts above place elapsed time on the X axis, and audio time on the Y axis. So these two concepts of time have been in the picture from the beginning.

Maintaining this distinction means that grain pitch (or playback rate) may be expressed as the number of audio seconds to play within one elapsed second: $\frac{audiosec}{sec}$.

Some general formulas:

- Buffer duration $\rightarrow$ audio second: $time_{bufdur} \times \frac{audiosec}{buffer}$. Multiply by the circular buffer's duration in seconds.
- Buffer duration $\rightarrow$ audio samples: $time_{bufdur} \times \frac{samples}{buffer}$. Multiply by the circular buffer's size in sample frames.
- Audio seconds $\rightarrow$ audio samples: $time_{audiosec} \times \frac{samples}{sec}$. Multiply by the sample rate.

Conversion in the opposite direction divides by the same quantities.

3.2 SuperCollider

The recording SynthDef should write the recording position onto a bus. Then the granulator can read from this bus and calculate valid positions based on it.

The recording synth addresses time in sample frames. For playback, the GrainBuf UGen expresses grain start time in buffer-duration units, and grain duration in seconds. So first we convert phase in samples into seconds: `var phaseSec = In.ar(posbus, 1) / sr;`. The remaining time calculations proceed in terms of seconds, until the final conversion of grain start position from seconds to buffer durations: `grainStart / bufdur`.

`grainStart` is calculated as the lesser of the recording position and the "safe start time," minus a time randomizer. The final start time is guaranteed to be earlier than this. What about the third case (slower than normal playback)? For slow grains, crossover clicks are possible only

for a very long delay time, close to the circular buffer's duration. Keeping the delay relatively short stays away from that danger zone. So it is necessary to handle only the case of faster-than-normal grains.

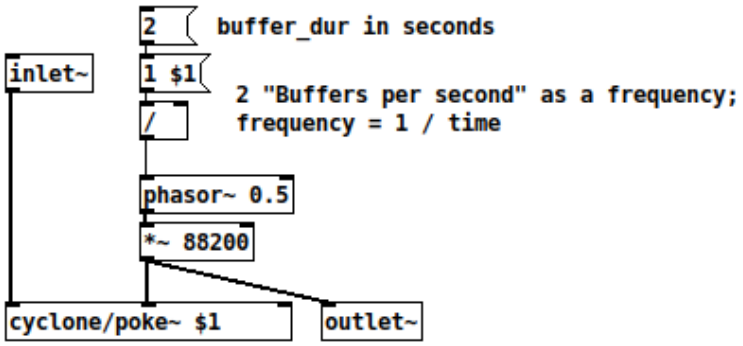
```
(
SynthDef(\circularRec, { |inbus, bufnum, posbus|
   var phase = Phasor.ar(rate: 1, start: 0, end: BufFrames.kr(bufnum));
   BufWr.ar(In.ar(inbus, 1), bufnum, phase);
   Out.ar(posbus, phase);
}).add;

SynthDef(\live_grains, { |out, gate = 1, bufnum, posbus, amp = 0.1,
   tfreq = 20, overlap = 8, rateRand = 1.05, timeRand = 0.15|
   var sr = BufSampleRate.kr(bufnum);
   var bufdur = BufDur.kr(bufnum);
   var phaseSec = In.ar(posbus, 1) / sr;
   var trig = Impulse.ar(tfreq);
   var dur = overlap / tfreq;
   var rate = TExpRand.ar(rateRand.reciprocal, rateRand, trig);
   var safeStart = phaseSec + (dur * (1 - rate));
   var grainStart = min(phaseSec, safeStart) - TRand.ar(0, timeRand, trig);
   var grains = GrainBuf.ar(2, trig, dur, bufnum, rate,
      grainStart / bufdur);
   var eg = EnvGen.kr(Env.asr(0.001, 1, 0.01), gate, doneAction: 2);
   Out.ar(out, grains * (eg * amp));
}).add;
)

s.boot;

b = Buffer.alloc(s, s.sampleRate.asInteger * 2, 1);
p = Bus.audio(s, 1);

a = Synth(\circularRec, [
   // read hardware (mic) input
   inbus: s.options.numOutputBusChannels,
   bufnum: b,
   posbus: p
]);

g = Synth(\live_grains, [
   bufnum: b,
   posbus: p,
   amp: 0.4,
   timeRand: 0.7,
   rateRand: 1.5
], target: a, addAction: \addAfter);

g.release;
a.free;
p.free;
```

```
b.free;
```

3.3 Pure Data

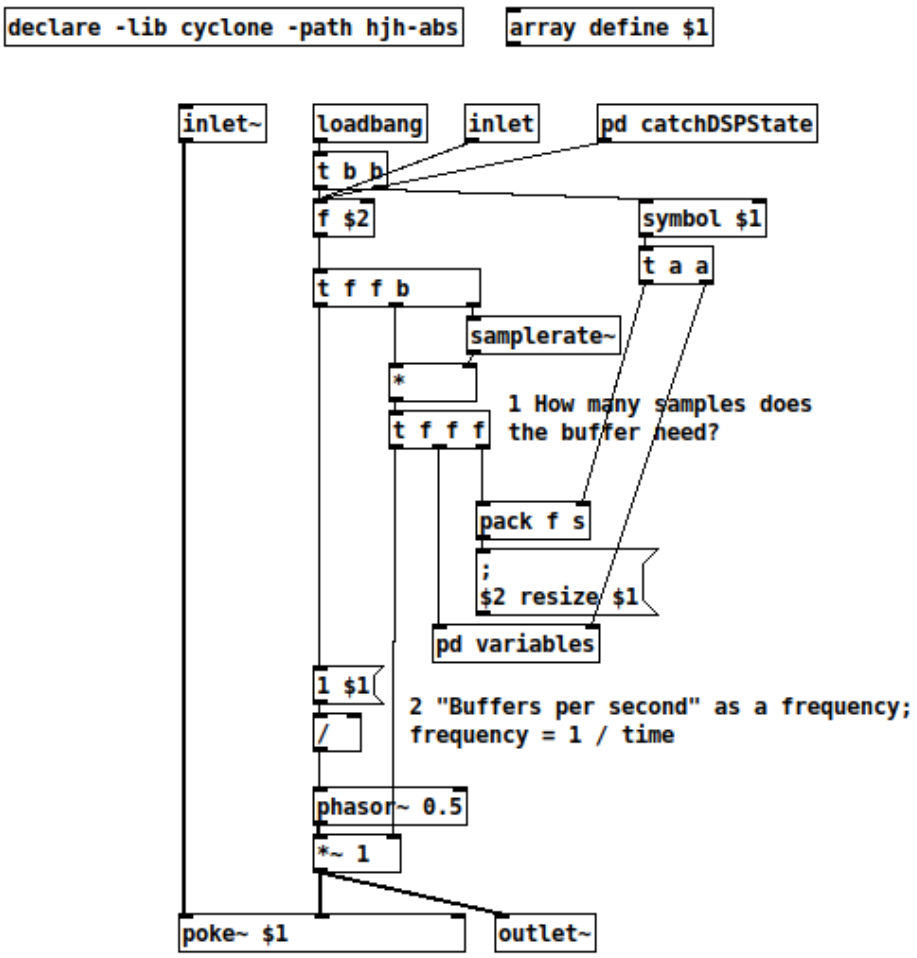
3.3.1 Circular buffer writer

The principle is relatively simple: If the buffer duration is d, then the frequency $\frac{buffers}{sec} = \frac{1}{d}$. This frequency drives a `[phasor~]`. Pd-vanilla does not provide a buffer writer that allows the user to control the record position; instead, use Cyclone's `[poke~]`. This object expects the position in sample frames, so the `[phasor~]` is multiplied by the number of frames in the buffer.[1] As in the SuperCollider example, the record position in frames is published (here, via an `[outlet~]`).

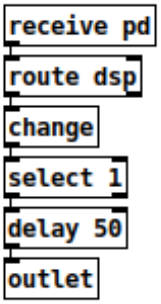


The devil, however, is in the details. Making a user-friendly abstraction out of this, with arguments for the buffer name and duration in seconds, requires some trickery, and careful attention to the order of messages.

[1] `[phasor~]` may be subject to floating-point rounding error, in which case samples may be omitted, or skipped over, during recording. It may be more reliable to use `[rpole~]` as an accumulator (though the reset mechanism could be tricky—I won't develop this idea further in this version).



The array must be sized based on the user's given number of seconds, times sample rate. What if the user created the abstraction instance, or opened the parent patch, before turning on DSP? Then the eventual sample rate is unknown. So, it is necessary to retrigger initialization logic when DSP is switched on. But, then, one will also find that `[samplerate~]` does not immediately report the correct rate! The trigger must be delayed slightly.



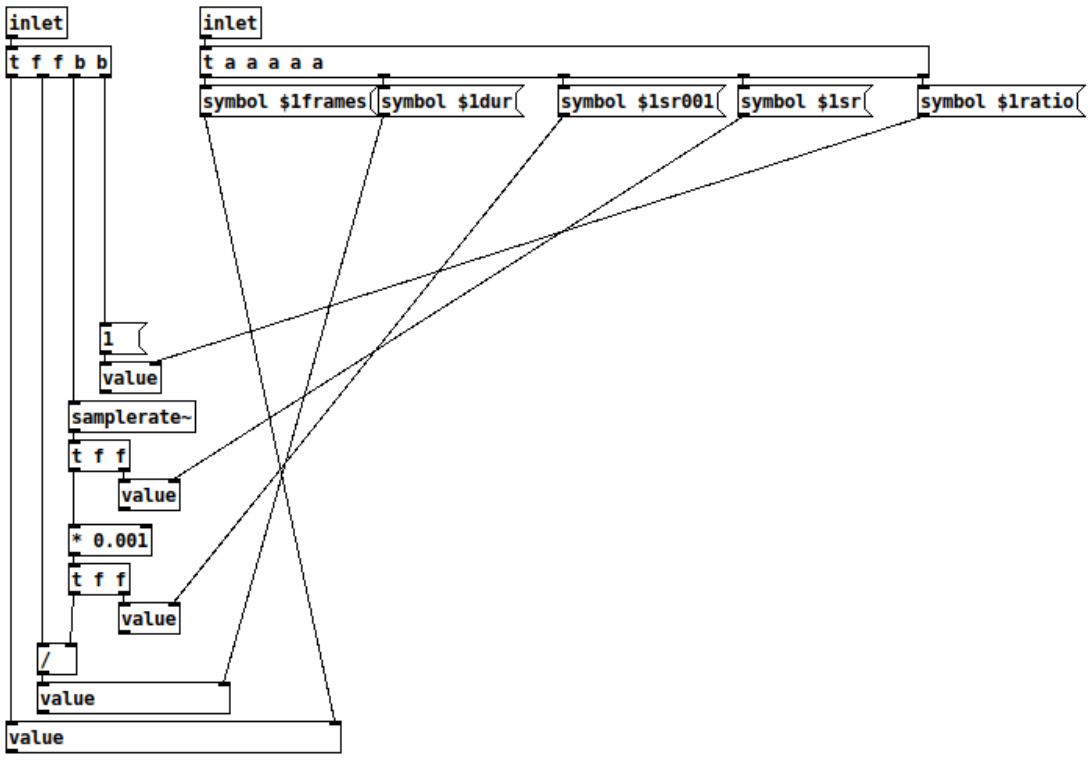
Following that, the sequence is:

1. Get the sample rate.
2. Multiply duration (seconds) by this = number of frames.
3. Pack for an array-resize message.
4. Set buffer-statistics `[value]` variables in the `[pd variables]` subpatch. (These will be used to communicate buffer size, sample rate etc. to the grain player.)

5. Feed the number of frames forward into the [phasor~] multiplier.
6. Duration (seconds) → [phasor~] frequency.

Note that the second inlet, to resize to a new number of seconds, retriggers the entire sequence.

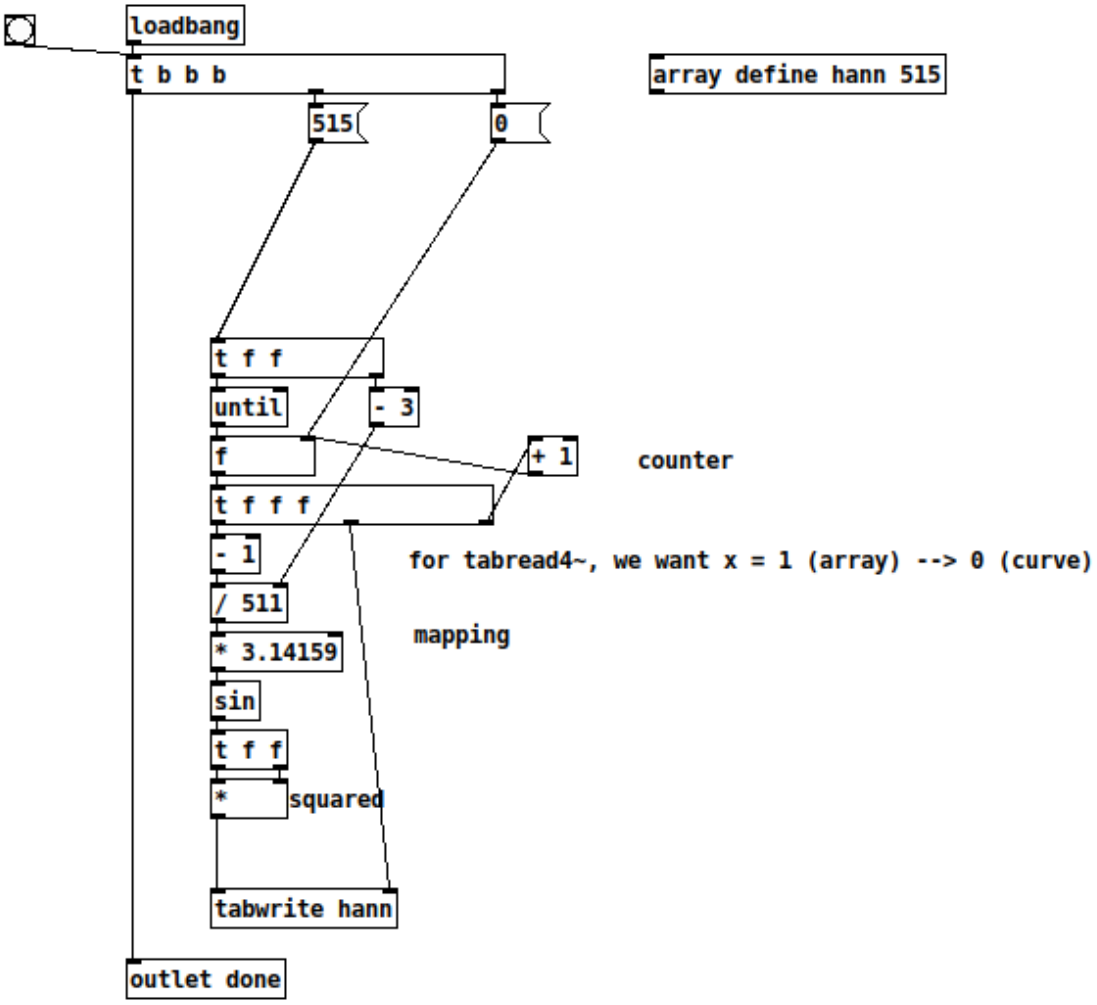
Step 4's buffer statistics are based on the [soundfiler2] abstraction in my *hjh-abs* library. [value] objects are created for number of frames, duration in ms, sample rate, samples per millisecond, and rate ratio (assumed to be 1, for a buffer not loaded from disk).



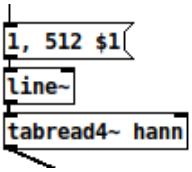
3.3.2 Grain player

Pure Data does not feature built-in granulators. So, we have to build one.

Fill a second array with a Hann window. I provide a second abstraction for this. Creating a single instance of this abstraction makes the window available globally.



Then, given a number of milliseconds for the grain duration, `[tabread4~]`-ing the array provides the envelope.

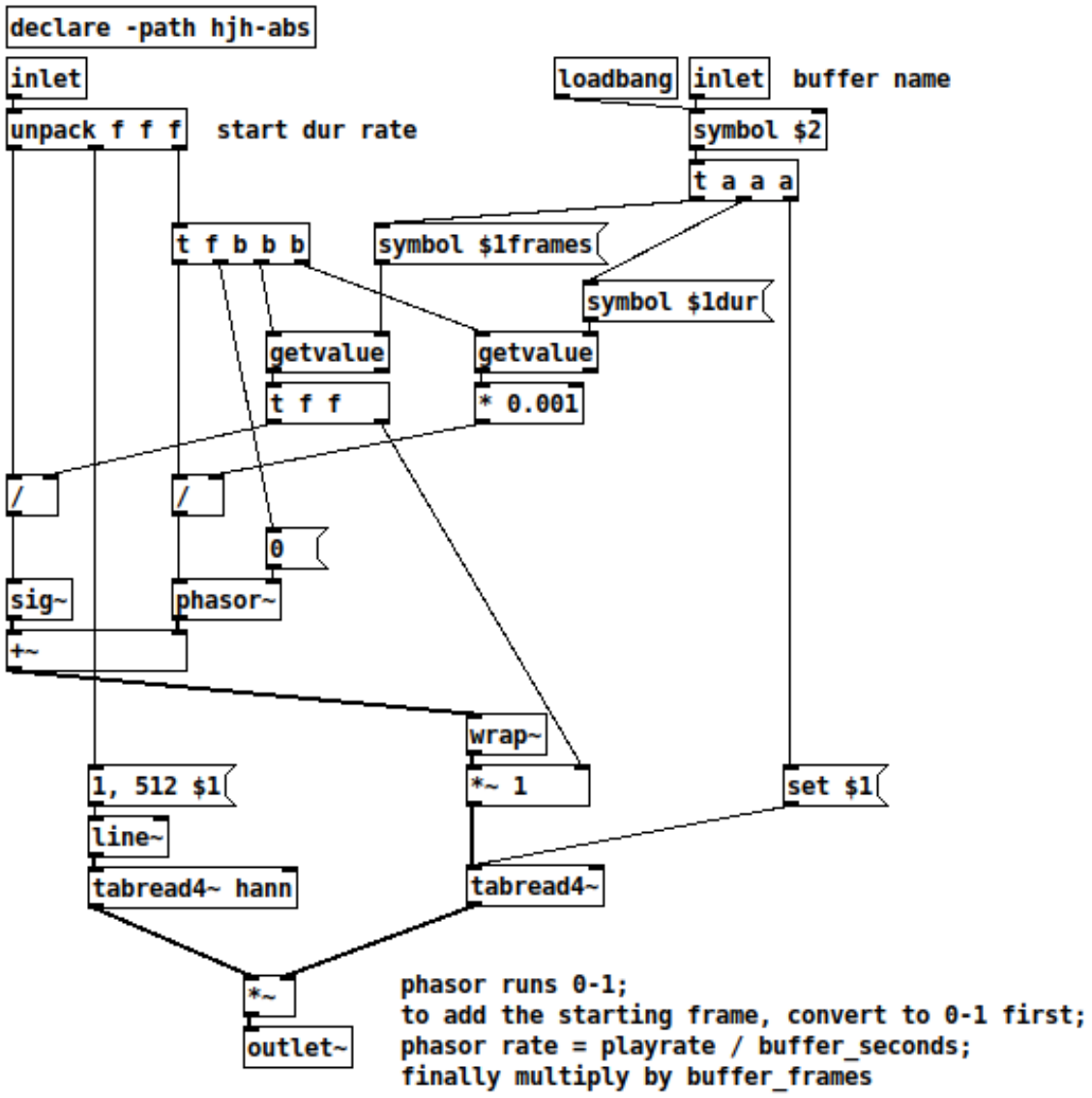


The input to trigger a grain is a list, giving the starting frame number for the grain, the duration in milliseconds, and the grain's playback rate. (This is given as a list because we will need to use this abstraction with `[clone]`, where it is easier to manage parameter lists rather than multiple inlets.)

For wraparound playback, `[phasor~]` again drives the train. Normal play speed requires a frequency of 1 ÷ buffer duration in seconds. $\frac{rate}{dur}$, then, gives the frequency for any desired playback speed. To this, the grain start position must be added. `[phasor~]` outputs time in terms of buffer duration. An incoming start time expressed in sample frames must be converted (dividing by the number of frames in the circular buffer, obtained from the buffer `[value]` variable set[2]). Because of the offset, it is necessary to `[wrap~]` to 0.0–1.0 before converting back to sample frames for reading.

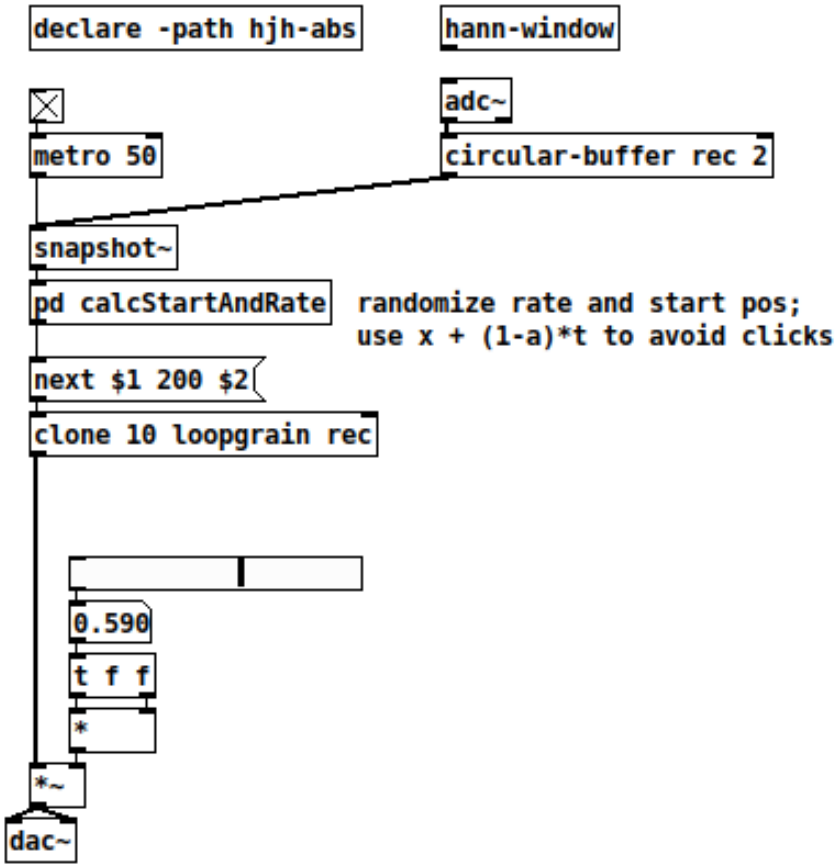
[2] `[getvalue]` is an abstraction in my *hjh-abs* library. It protects the variable from being accidentally changed, by rejecting any input other than "bang."

The remainder of this abstraction's logic prepares the other objects to refer to a specific buffer name.

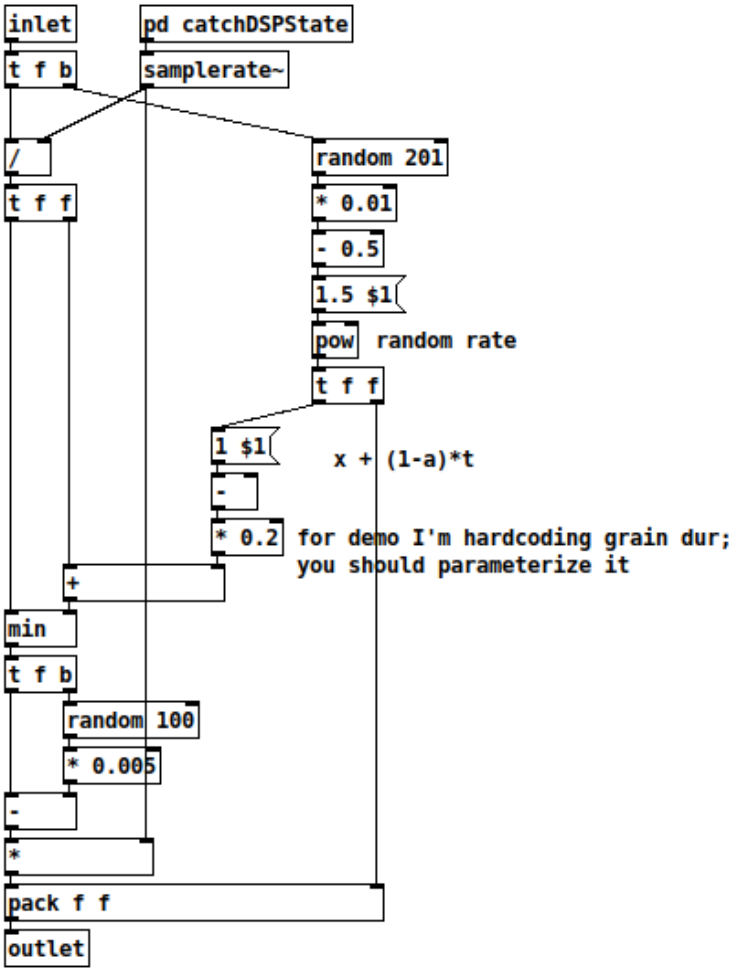


3.3.3 Granulator usage

The "loopgrain" abstraction produces one grain. For granular synthesis, we need many overlapping grains, i.e., polyphony with `[clone]`. A `[metro]` is the trigger, sampling the buffer recorder's frame position and passing it into a subpatch that randomizes the rate and start time. These parameters are packed into a list along with a "next" keyword (needed for `[clone]`).



`[pd calcStartAndRate]` implements the "safe start" formula. The right-hand branch first calculates an exponentially-distributed random rate, transposing up or down by a perfect fifth. For demonstration purposes, the grain duration is hardcoded at 200 ms = 0.2 sec, but could easily be parameterized. Then, as in the SuperCollider example, the minimum of this value or the current record position is taken, and a random offset subtracted from it.


inlet
pd catchDSPState
t f b
samplerate~
/
random 201
t f f
* 0.01
- 0.5
1.5 $1
pow
random rate
t f f
1 $1
x + (1-a)*t
-
* 0.2
for demo I'm hardcoding grain dur;
you should parameterize it
+
min
t f b
random 100
* 0.005
-
*
pack f f
outlet