Casting a Line: A Flexible Approach to Soundfile Playback Using libPd in Ninja Jamm

Dr Edward Kelly
Synchroma Audio Engineering / Ninja Tune Records
11 Spenser Road
London, UK, SE24 0NS
synchroma@gmail.com

Abstract

This paper describes a strategy for implementation of both smooth playback and glitch-free scrambling of audio on multiple, independent tracks, using a single instance of an augmented version of the phasor~ object: phasorbars~. Each track has control of the playback point, so that functions to re-order slices of each looped sample are independent for each of the four tracks in the Ninja Jamm mobile app. Formulas used in calculating how to offset the single phase signal to achieve rhythmic flexibility within the bar are discussed, and a method for achieving on-the-fly envelopes on each re-ordered slice of audio are detailed.

Keywords

libPd, soundfile, synchronization, Ableton link.

1 Introduction

Ninja Jamm[1] is an app for iOS and Android that gives the user a set of tools with which to remix tracks by a large and growing number of artists on an iPhone, iPad or suitable Android device[2]. At its heart lies a highly complex Pd patch, with a suite of external objects created by the author. This runs all the audio and sequencing functions of the app. The GUI for the app is also a Pd patch using Pd Open Frameworks (POF) by Antoine Rousseau[3], who is also responsible for the GUI Pd Patch, and an Objective C wrapper and extra tweaks to the GUI infrastructure by Christopher Rice of Holderness Media[4], and of course Matt Black of Ninja Tune, who initiated the project and continues to manage it. This creates the environment in which Ninja Jamm operates on an iOS or Android device. In its original inception it was designed to run on an iPhone 4, and given the lack of processing power and memory of this device, a means had to be found whereby all four tracks of audio had to be run from a single phasor~ object. This is complicated by the jamming functions that beat-slice the audio, where each of the four tracks may be programmed to play from any point or slice within the bar on any rhythmic pulse.

The very first version of the app had poor sound quality, due to the fact a de-click mechanism was used on each slice, regardless of whether re-ordering of the material was taking place. In order for tracks to be played back cleanly when no re-ordering is taking place, and because due to low CPU power a simple tabread~ mechanism had to be implemented. Subsequently, memory usage was decreased by the creation of hextab~ and hexread4~ objects, using 16 bit memory rather than 32 bit table~ and tabread4~ objects, but preserving the 4-point interpolation algorithm devised by Miller Puckette and sacrificing load and save functions for a simple copy of data with translation from 32 bit to 16 bit arrays. From 152% CPU load on the iPhone 4, it was reduced to 71% before the 16 bit objects were introduced, and 56% after.

A multiple-bar spanning object was created called phasorbars~. Whereas in the original version the phasor~ output was used to read a single bar's worth of audio and metro objects provided the clock controls, phasorbars~ provides a single ramp signal that loops through multiple bars, and generates 1/8th, 1/16th and 1/24th note clocks from the ramp. These clocks are generated from the ramp/phasor~ signal, so that re-ordered slices will always trigger in time with tempo changes no matter how quickly these changes occur.

Beyond this point, it was discovered that there were discrepancies between control-rate and

1 http://ninjajamm.com

2 Although there is a significant latency with the Android operating system related to the round-trip audio path, but this is improving with newer iterations of Android.

3 https://github.com/Ant1r/ofxPof

4 http://www.holdernessmedia.com

audio rate block-boundaries that had to be reconciled at control-rate, while audio ramps kept-up with the error, so the wrap_overshoot~ object was developed in order to account for this difference in timing due to objects receiving control messages further down the audio chain and implementing them at the end of an audio block. This external deliberately overshoots a signal value of 1 when it is reached until the end of the current block of audio, so that the wrapped signal only returns to 0 when control-rate offsets take effect. Further developments in the app that have led to a more traditional phasor~ object in phasorbars~, but retaining the bar counters that drive the sequencing object in the app, most notably the use of the Ableton Link technique, whereby many instances of the same mobile app may be synchronized together, developed by Peter Brinkmann for Pd[1].

2 Slicing the Line

2.1 The Bar as a Single Phase-Cycle

The basic premise is that 1 cycle of phasor~ output = 1 bar. Within the app there is presently only the capacity for 4/4 time signatures, but the techniques detailed here can be adjusted to accommodate other rhythmic structures with relative ease. The frequency of the phasor~ is set to:

$$f = (60/BPM)/4$$

This assumes a bar length of 4 and a rhythm value of a quarter-note for the BPM (beats-per-minute), hence a 4/4 time signature. This formula that is familiar to many practitioners of electronic music is easily adjusted to account for different pulse and bar structures.

Figure 1 shows the connection structure between control elements of the patch (voice_control.pd) the phasorbars~ signal from the inlet~ and the audio generation objects (voice.pd).

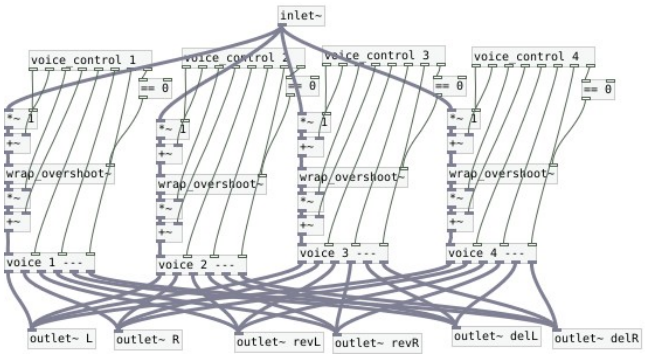

Figure 1. Control and Signal Paths for the Four Tracks or Voices.

The way in which this is structured is such that the length of each clip in bars is reciprocated to multiply the signal, and a float in fractional terms may be used to offset the incoming phasor~ if the loop is offset, that is if the first bar of the loop to play is not the first bar of the sample. The multiplier is unique to each track as is the offset, so that the output of wrap~ is always a vector from 0 to 1. After the wrap~ object, a separate *~ object multiplies the signal by the length of the clip in samples, and an offset in samples may be added for the beat-slicing to occur. This also requires a de-click mechanism when jumps are made from one portion of an audio array to another, since the nature of the app is that it is a platform for many existing works by many different artists – essentially a publishing platform for a record label to release music in an interactive state. The material is unpredictable at the sample level from within the Pd patch.

2.2 De-Click the Jumps but Swim in the River

It is a requirement of the app that the user is able to switch between clips (i.e. audio arrays, with 8 per track) or invoke beat-slicing or arbitrary loops at any instant. In order that this process could happen as efficiently as possible, the first version of the app considered audio in 1/8th chunks, so one quaver pulse is equal to 0.125 of the output of the (originally used) phasor~ object. An event-based sequencer is used to change clips and set loop lengths within the song, with loops initiated either by the user or automatically when a jamming function or clip change is enacted by the user, with a de-click function used at every 1/8th note boundary, or when user interaction occurs.

There are clear disadvantages to this, in that there are very short dropouts in the audio that distort the material every 1/8th note even if it is playing normally, from the start of the audio array to the end. The phasorbars~ object was created to overcome this, but first, the de-click mechanism is discussed.

2.2.1 Block-Based Click Math

The default DSP block size in Pd is 64 samples. One of the great consistencies of Pd is its 32 bit architecture, so that if I calculate the length of the block in milliseconds it has the same precision as the audio architecture. A

single block of DSP measured in milliseconds is:

$$t = 64 / (SR / 1000)$$

At 44100Hz sample rate (SR), the time (t) is 1.45125ms, a fundamental unit of time within libPd, unless block~ or switch~ objects are used to change this value.

If a change is initiated in the playback point, or the clip is changed, or a mute etc. it happens at the next block-boundary as these events are outside of the audio (signal-driven) tileline, or more precisely a quantized timeline to 64 sample blocks. Given that we are dealing with recorded audio here, there may well be a discontinuity (a click) in the audio stream. Since we cannot go back in time to start the fade before the event occurred, it is necessary to delay the output of the sample reader by a few milliseconds. Then it is possible to fade out the play back for one audio block (1.45125ms). Experimental tests show that delaying the audio stream by 2 blocks (2.9025ms at 44100Hz SR) and fading up after 3 blocks (4.35375ms at 44100Hz SR) is enough to eliminate clicks.

It is a very slight trade-off in turns of sluggishness of the interactions that is barely noticeable. User-selected clips start 4.35375ms after they are triggered. So do sequenced beat-slicing chunks, but since the beat-slicing algorithms are all quantized and automatic any delay is more-or-less irrelevant to the user experience of those functions. Figure 2 shows the delay and vline~ objects in the Voice.pd patch, with the message sent to vline~ when a change occurs shown in figure 3.

Figure 2. The vline~ object is used to attenuate delayed audio while changes happen to the source array or read point.

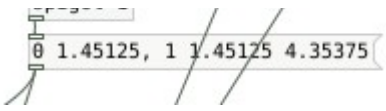

Figure 3. The de-click message sent to vline~, ensuring that discontinuities happen when the track is silenced.

2.2.2 The No-Click Playback Mode

In contrast to all this manipulation of the playback point or source array, it is also possible to play clips from start to finish, in an arrangement specified by a file within the tune pack. If de-click is being used when it is not needed (e.g. when playing an 8-bar loop from start-to-finish) then audio distortion is the result.

The answer is to have two different signal paths within each voice. One where manipulation takes place and an other where a smooth ramp played the audio back with no intervention. In order for this to be completely smooth however, the was a need for an object where the phasor~ signal spans 0 to 1 in 1 bar, but can transition to bar 2, 3, 4 etc. according to the maximum length, i.e. the length of the longest loop. So a transition from 1-bar-at-a-time plus offsets and e.g. 8 seamless bars of playback is made possible by the *~ and +~ objects on either side of the wrap~ object in the Voice.pd patch. Another object has been created to handle the phasor~ signal spanning multiple bars, and sending commands to the sequencing object as well as generating clocks for events. This is the phasorbars~ object discussed below.

3 Phasorbars~, A Single Synchronization Vector for Simple Rhythmic Values

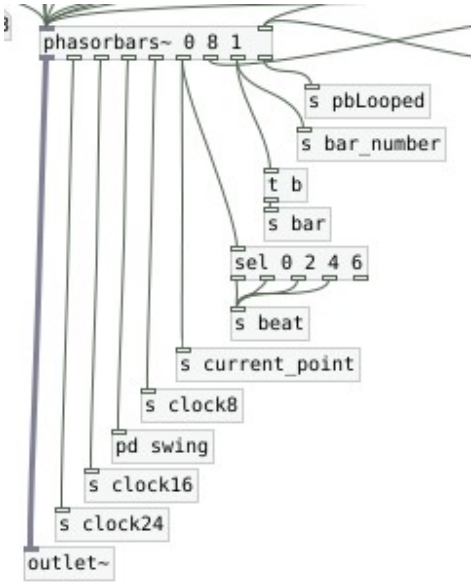

Figure 4. The phasorbars~ object.

Parameters within phasorbars~ are the phase (0-1 just as with the phasor~ object), loop length (how long is the sequence loop) whether the object is looping or not, and the cycle point (the current bar number within the loop). The bar number is incremented on each bar when

loop mode is not initialized, and this feeds into the sequencing object for playback mode. If the app is not looping, the bar number is incremented at the end of the bar offset + loop length, and phasorbars~ is slaved to the sequence which is organized with a resolution of 1 bar. If the object is in loop mode however, the sequence is slaved to the repetition of the loop, itself stored in the sequence object.

The point is that it goes from 0 to 1, 1 to 2, 2 to 3 etc on the signal outlet, while remaining aware of where it is in the sequence (the baroffset and length parameters).

Given that an individual object is used both to set the loop length in loop mode, and also send commands to change the mute, clip and clip offset in sequence mode - the sequence storage object also knows which bar each sample in each of the four tracks is on, fed from the bar number outlet of phasorbars~, so that beat-slicing is accurate when the signal is selected where a 0 to 1 ramp equals one bar. The sequencer sets the parameters for the sample-looping mechanism in phasorbars~, and the sample looping mechanism tells the sequencer when to move on, or to loop.

Its incremental bar counter interacts with the sequencer object to enable both smooth playback modes and the original (slice-based) mode. Its signal output is similar to phasor~ except that instead of a floating-point audio signal that changes between 0 and 1, it then continues the audio ramp through 2, 3 etc. until it reaches the loop length, when it starts again at zero. The only drawback to this approach is that there is a limit to how long an uninterrupted loop can continue, a limit of 4000000 discrete values, which matches with the default maximum length of a sample array. This evaluates to just over 52 bars at 140BPM, 4/4 time signature (or 2^22 rounded down to the nearest million). The point of this is that the chopped audio (with declick at every slice) is only used when some re-organization of the material is happening, and the rest of the time (i.e. when samples are playing from start-to-finish) the phasorbars ramp is playing bar 0, then 1, then 2, then 3 etc on a continuous sample-accurate ramp.

3.1 Voices.pd

Another feature of this object is that it has 1/8th, 1/16th and 1/24th rhythmic outputs that are tied to the progress of the internal phasor~ algorithm. Rapid changes in tempo have no impact on the accuracy of the clocks, since changes in tempo are directly translated into the speed of the phasor~ from which the clock thresholds are taken, and hence quantized changes may occur.

This leads back to the network of pre- and post- wrap~ *~ and +~ objects, and what they represent (see fig. 1). Given that the signal into wrap~ always emerges as a 0 to 1 vector, it can either be a bar with optional beat-slicing, or for example (if multiplied by 0.25) 4 bars with clean playback, depending on what happens on the other end of wrap~. Manipulating the numbers on either side of the wrap~ element allows for complete flexibility with regards to the playback strategy, from a single signal that is modified to read any of the audio arrays from any point.

Having stated this, it was recently discovered that some of the mechanisms in the pre- and post-wrap~ elements are not functioning as they originally were since the Pd patch has been modularized. A click was heard at the end of each bar, where bar lengths in samples were being incremented after the wrap~ object in the signal path, at control-rate.

3.2 wrap_overshoot~

The principle of the wrap_overshoot~ object is founded on the fact that control-rate calculations using +~ or *~ right inlets *after* a wrapped line~ or phasor~ signal do not take effect until the end of an audio block (64 samples). The wrap~ object is mathematically pure in that it allows a positive signal vector to inhabit strict boundaries between 0 and 1. What wrap_overshoot~ does is that when the signal hits the threshold of 1, it will wrap it only at the end of the current signal block. This means that control-rate messages sent into signal arithmetic objects further down the signal path are synchronized with the wrapping of the signal driving the sample readers, and so a slight overshoot (see figure 5) allows the playback of multiple-bar arrays to be click-free.

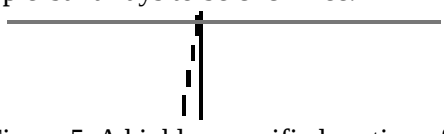

Figure 5. A highly magnified portion of the output from wrap_overshoot~ at a bar boundary.

Some may say this is a kludge, and there are grounds for this supposition. It is often the case with commercial projects that deadlines and external pressures lead to unusual solutions to

problems. There is an argument for a complete re-write of the voice_control.pd mechanism. A window of time is needed for this to happen.

3.3 The linkline patch

The development of Ableton Link[2] has meant that there is a way for multiple devices to be synchronized. Coldcut (Matt Black who manages this project, and Jonathan More) have been using this as a live performance tool for their music in recent months, with linked iPads as performance tools. The phasorbars~ external also has a slave mode, whereby the link phase determines the signal output. Peter Brinkmann's externals for link[5] have enabled the use of link as a synchronization device, and this is quite simple to apply to a sequencing patch or app. However, when playing recorded material there is a problem, since the phase information is sent at control rate.

A patch was devised in a similar way to wrap_overshoot~. Given the block length of 1.45125ms, each block is a ramp from the previous value to the next using the line~ object. When the phase messages from the abl_link external go from high to low values, the low value has 1 added to it as the destination of the line~, and the next block starts at the lower value and goes to the next. The mechanism is shown in figure 6.

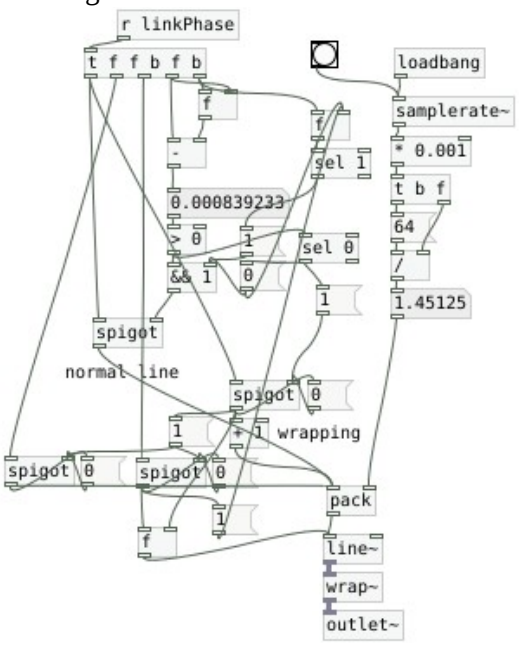

Figure 6. The linkLine.pd abstraction

Returning to the de-clicked beat slicing mechanism, it is possible to determine offsets and multipliers to any point in the bar from the single phasor~ signal if you know where we are.

How we work out the way to that point is another matter.

4 From Where to Here

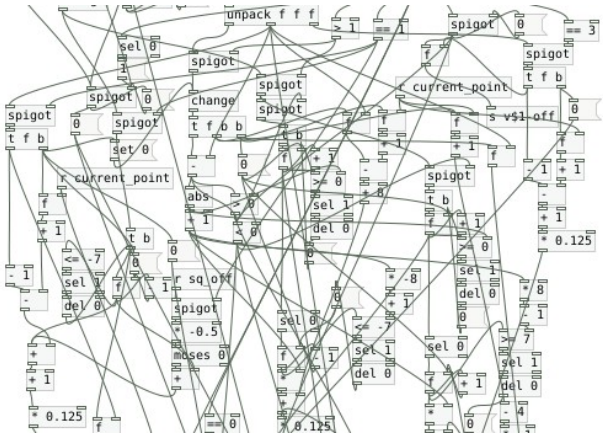

Figure 7. The drill and arbitrary loop mechanisms alone are fiendishly complicated, considering what they do.

There is a way of calculating at any instant the value of an offset to apply to a ramp, provided we know what value the ramp is at in a particular moment. The phasorbars~ object assists us with this task, since its event outlets observe the strict right-to-left ordering of Pd. In fact, with more efficient programming of the patch, perhaps one or more outlets could be eliminated. But the philosophy is that longer structural units happen before shorter ones ,with the sample-playback ramp being the finest resolution.

At any quantized moment, for a given resolution of 1/8th, to offset the phasor~ ramp W (where-we-are) to a point D (destination), an offset A is applied so that:

$$A = 1/8(D - W)$$

Where D and W are the number of 1/8th notes. Of course this value should be wrapped for the offsets to work properly, so that the complete version is:

$$A = 1/8(((D - W) + 8) \% 8)$$

Of course the modulus only applies to integers, and then we take 1/8 of the value, but this is easily ironed-out in patching.

This is deceptively simple, but the realization that this sort of momentary voodoo was necessary caused some angst-ridden moments, to get the CPU load down from 152% to 71% on an iPhone 4!

Its reverse equivalent:

5 https://github.com/nettoyeurny/pd_link_bridge

$$A = 1/8((D - 8) * -1 + 8 - W)$$

can be used when the input signal is running backwards, being multiplied by -1.

5 Missing Artefacts

There is a large amount of detail missing from this article, such as what the sequencer is (a linear database of possible states) or how the sequence object swaps roles with the phasorbars~ object (looping is triggered by user interactions and interventions, line modes are allocated depending on whether to slice). But the point of this explanation and the release of phasorbars~ is not to allow unlimited versions of Ninja Jamm. It is so that these methods can be utilised in ways unpredictable from here.

The future of Ninja Jamm is Jamm Pro, where this becomes a compositional tool for mixing and remixing user-generated material. There are many forthcoming additions to Ninja Jamm that will see the light of day with the version in development. Many aspects not discussed here, such as the inclusion of audio copy and paste functions, load-your-own-sample, elastic audio, snapshots and other customization options that are too far-ranging for a single paper. But the libPd platform has shown its resilience in this app in that the app has existed in a public release for over four years, and its development continues.

6 Conclusion

The techniques discussed here are in some ways obsolete. The rapid growth of mobile devices' speed, memory and overall power means that we can be less frugal with resources than we would otherwise be. However, as this author who grew up in the era of 8-bit computing and 16 colors knows, a little can go a long way.

By understanding the architecture of block sizes in Pd, and clever use of delays and vline~ it is possible to make interactions with recorded audio sound (almost) as smooth and clean as a hard-disk edit, but on-the-fly.

There is great simplicity in the structure of this app, in that it assumes a rigid hierarchy of electronic dance music as it's mode of transmission. However, each of the algorithms and formulas in this paper can be tweaked to suit different situations, such as bar lengths and time signatures, with suitable adaptation of the algorithms published here.

The phasorbars~ and wrap_overshoot~ externals, and the linkLine abstraction are available on my personal website[3].

7 Acknowledgements

Thanks go to Seeper[6] who developed the original interface, and of course to Matt Black and Jonathan More of Coldcut, and the Ninja Tune record label. Special thanks go to the current development team of Christopher Rice and Antoine Rousseau, Peter Brinkmann for the link externals and our pack assembler Aneek Thapar. Also great thanks to Dan Wilcox who develops ofxPd, the branch of libPd used in Ninja Jamm.

References

[1] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, H. C. Steiner: *Embedding Pure Data with libpd*, Weimar, 2011.

[2] https://www.ableton.com/en/link/

[3] http://sharktracks.co.uk/html/ninjaTools.html

6 http://seeper.com