Proceedings of the 17th Linux Audio Conference (LAC-19), CCRMA, Stanford University, USA, March 23–26, 2019

SEQUOIA: A LIBRARY FOR GENERATIVE MUSICAL SEQUENCERS

Chris Chronopoulos

Independent Developer
Cambridge, MA
chronopoulos.chris@gmail.com

ABSTRACT

Sequoia is a new software library for musical sequencing, with generative capabilities and sample-accurate timing. The architecture supports a variety of techniques, including polymetric sequencing, clock division, probability, and other parameters which can be manipulated in real time – or even sequenced themselves. The core library is written in C and supports JACK MIDI; Python bindings are also available.

1. MOTIVATION

In recent years, the electronic music community has shown a growing interest in the use of standalone hardware units, both for studio production and live performance [1]. Among their many appeals, these devices have the advantage of being modular - drum machines, synthesizers, samplers, sequencers, mixers, and effects units can be connected and re-connected in myriad ways to accomodate a variety of workflows. Each component serves a unique role and interfaces with other components through well-defined interfaces: line-level audio, and control signals typically in the form of MIDI or CV (control voltage).

The Linux audio ecosystem is well-poised to emulate this paradigm in software; audio routing libraries like JACK, and control signal protocols like MIDI and Open Sound Control (OSC) provide a framework for connecting standalone applications into software "rigs" suitable for composition and performance alike. Indeed, such modularity is central to the Unix philosophy: programs should "do one thing and do it well" [2]. True to form, numerous drum machines (e.g. hydrogen, drumkv1), synthesizers (zynaddsubfx, amsynth, dexed), samplers (shuriken, qsampler, petri-foo, sooperlooper), mixers (jackmixer, non-mixer), and effects (calf-plugins, guitarix) are available from popular Linux repositories. Additional utilities exist for managing audio/MIDI connections (qjackctl, catia/claudia/carla) and saving/restoring sessions (lash/ladish/nsm/aj-snapshot).

Sequencers, however, are comparatively absent from this ecosystem. Perhaps the best-established example is seq24 [3], which, albeit stable and relatively comprehensive, has not been significantly updated since 2010, and suffers from usability issues which hinder on-the-fly composition. Various sequencers exist within larger DAW applications like Ardour [4], LMMS [5], Qtractor [6], Rosegarden [7], and Muse [8], but these don't fit into the modular paradigm described here. Furthermore, the predominant interface for these software sequencers is the *piano roll*, which is well suited for editing live data captured from a MIDI controller, but less appropriate for the quick manipulation of drum patterns and arpeggios typical of dance music. For this task, a traditional *step sequencer* is desired.

But step sequencers can be quite complex. They typically feature live sequence composition, real-time manipulation, and chaining of sequences. More advanced examples include generative properties like probability, ratcheting, and meta-sequencing, in addition to step-wise parameters like microtiming and control variable modulation. With such a wide variety of features, it can be challenging to design applications which cover all the bases – but this is primarily a problem of interface design. The essentials of modern sequencing – timing, synchronization, live manipulation, etc. – can be separated from the problem of application design, and distilled into a general-purpose library, as in the "model-view-controller" paradigm [9] This is the motivation for *Sequoia*.

Figure 1: *A Sequoia session is connected to two different client applications using JACK. Here ZynAddSubFX (zyn-fusion) and drumkv1 are being used to create a simple beat. Carla is used to manage audio and MIDI connections.*

2. DESIGN

The architecture of Sequoia is based on four object classes: *session*, *sequence*, *trigger*, and *port*.

A *sequence* is a discrete series of events which steps in time with a metronome. In this sense, Sequoia is a "step sequencer", but events are not required to be evenly spaced in time (see Section 4.1). The length of a sequence is the number of steps that the sequence contains. There is no limit (aside from memory) to the length of a sequence, but once specified (via instantiation), it is fixed. This is less of a constraint than it may seem, however, as sequences can be chained together and "meta-sequenced" dynamically 5.3. Sequences have several dynamic parameters: the mute state, transpose, clock division, playhead position, playhead direction, and loop boundaries can all be modified live during playback.

Triggers (or "trigs" for short) are the event objects which may populate the steps of a sequence. They store information depending on their type; the current trigger types are:

- **Null:** (an empty trig)
- **Note:** *note value, velocity, length*

- **CC:** *number, value*

Each trigger also carries a *channel number*, a *probability* and a *microtime*. Microtime is a floating-point value in the range $[-0.5, 0.5)$, where the units are in steps. Thus a trigger can be placed half a step before or after its nomimal timing, allowing for irregular rhythms, "humanization", and swing.

Sequences run within a Sequoia *session*, which controls the tempo and transport (start/stop/pause) state applied to all contained sequences. A session can have a number of *ports* for communicating with other applications – including other Sequoia sessions. The ports can be input ("inports") or output ("outports"), have descriptive names, and can be assigned to sequences individually, or on a many-to-one basis. For example, we may have 4 sequences (kick, snare, closed hat, open hat) feeding into a single outport called "drums", while another melodic sequence feeds into an outport called "synth" – all sequencing in time within the same session.

3. API

Sequoia is implemented as a C library in the "object-oriented" style: data structures are presented as custom types with associated methods for instantiation and mutation. All library functions and data types are prefixed with `sq_*`. The full API is documented on the associated GitHub wiki; here we present a simple example which constructs and plays a 2-note sequence:

```
#include "sequoia.h"

#define STEP_RES 256

int main(void) {

  sq_session_t sesh;
  sq_session_init(&sesh, "My Session",
      STEP_RES);

  sq_sequence_t seq;
  sq_sequence_init(&seq, 16, STEP_RES);

  jack_port_t *port;
  port = sq_session_create_outport(&sesh,
            "My Port");
  sq_sequence_set_outport(&seq, port);

  sq_trigger_t trig;
  sq_trigger_init(&trig);

  sq_trigger_set_note(&trig, 60, 100, 4);
  sq_sequence_set_trig(&seq, 0, &trig);
  sq_trigger_set_note(&trig, 67, 100, 4);
  sq_sequence_set_trig(&seq, 8, &trig);

  sq_session_add_sequence(&sesh, &seq);
  sq_session_set_bpm(&sesh, 120);
  sq_session_start(&sesh);

  return 0;

}
```

Here, `STEP_RES` is the step resolution, in ticks per step. This needs to be the same for all sequences in the session – attempting to add a sequence with incompatible step resolution will result in an error. We create an outport for the session called "My Port" and set the sequence to output events through it. We then create a placeholder trigger object `trig` and use it to populate the sequence. Finally, we add the sequence to the session, set the BPM, and start sequencing.

3.1. Python Bindings

The main C library is augmented with Python bindings which obey a direct mapping between classes and methods. In Python, the example above could be written as:

```
import sequoia as sq

STEP_RES = 256

sesh = sq.session("My Session", STEP_RES)
seq = sq.sequence(16, STEP_RES)
port = sesh.create_outport("My Port")
seq.set_outport(port)

trig = sq.trig()

trig.set_note(60, 100, 4)
seq.set_trig(0, trig)
trig.set_note(67, 100, 4)
seq.set_trig(8, trig)

sesh.add_sequence(seq)
sesh.set_bpm(120)
sesh.start()
```

4. IMPLEMENTATION

A Sequoia session registers as a JACK external client whose name is the session name (specified during instantiation). Input and output ports are created as JACK MIDI ports (also named) which are served by the JACK processing callback. The API is compiled into a shared library plus header files, and can be installed e.g. in /usr/local/ for dynamic linking across multiple applications.

4.1. Timing

Timing is managed by the JACK processing thread as it executes within the context of the Sequoia session. The session keeps track of the frame count as it works to fill the JACK buffer with time-stamped MIDI events. Events are managed by the sequences which handle time as a grid of *microticks* – intervals of time much shorter than the step length which enable the microtiming functionality of the sequencer. In the code example in Section 3, the mictrotiming resolution is set to 256 ticks per step. In theory, this resolution can be set much higher, though in practice, it will be limited by CPU performance. The number of frames per tick (fpt) is:

$$\text{fpt} = 15 * \frac{\text{sr}}{\text{tps} * \text{bpm}} \tag{1}$$

where **sr** is the sample rate, **tps** is the step resolution (ticks per step), and **bpm** is the tempo in beats per minute. At 48 kHz with 256 ticks-per-step, there are 23 frames-per-tick at 120 BPM. At 4096

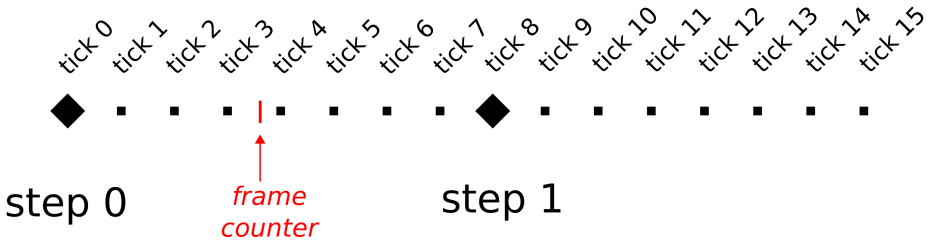

Figure 2: *Diagram visualizing the 3-tiered timing scheme used by Sequoia. At the highest level there are steps: 4 steps per beat (in the sense of "beats per minute"), and one trig per step. Going down one level, each step is composed of several "microticks" which comprise the grid for microtiming events. Here, only 8 microticks per step are shown for clarity, but a typical sequence may have 256 (or more) ticks per step. Finally, there is the frame counter, which sweeps between the microticks until it reaches a tick boundary, at which point a trigger may be fired.*

ticks-per-step, this becomes 1 frame-per-tick, which is the theoretical maximum resolution for this tempo and sample rate.

4.2. Trig-to-Microtick Translation

Although the fundamental timing grid is managed at microtick resolution, this implementation detail is hidden from the user by the trig interface. The user manages the sequence data by setting its trigs (one for each step); these trigs are then placed on the microgrid according to their microtiming. The formula is:

$$\text{tick index} = (\text{step} + \mu\text{time}) * \text{tps} \tag{2}$$

At this tick index, we place a pointer to the trig, which allows us to look up both the trig parameters (e.g. probability, length) and the sequence parameters (e.g. mute, transpose) at trig time, to ensure that we send the correct MIDI event at the correct time.

4.3. Note-Off

While note-on and control change events are recorded in the microgrid at composition time (i.e. when the user calls `sq_sequence_set_trig()`), note-off events are managed differently. To see why, consider what would happen if a C note of length 4 steps was recorded in the microgrid as a C-note-on plus a C-note-off 4 steps later. Now imagine if the sequence transpose parameter were changed in the middle of that note. The note-off would be delivered for the wrong note value, and the synthesizer downstream would be left with a hanging note. The same applies for playhead manipulation, or any number of the other sequence parameters which support live control.

The solution is to implement for each sequence a separate ring buffer, specifically for note-offs, which is always running forward. The length of this buffer is the maximum note length, which is also the length of the sequence. The buffer gets populated with a note-off (at the appropriate delay) whenever a note-on fires. When the note-off is reached by the advancing buffer pointer, it is fired, and then removed from the buffer. When a sequence (or the session) is stopped, we can optionally call a "clean" command, which sweeps through the off-buffer as quickly as possibly, delivering all remaining note-offs.

4.4. Lock-Free Parameter Control

In a running Sequoia session, the JACK thread needs immediate access to data that other threads (e.g. the UI thread) can manipulate during playback. In a non-realtime application, this would be accomplished with mutex locks [10], but in realtime audio, this is unacceptable – the audio callback must never execute code that could block for an indeterminate amount of time [11]. In lieu of mutex locks, we synchronize data between threads via lock-free message queues. For this, we use `jack_ringbuffer_t` as offered by the JACK API. We then implement a simple messaging protocol that allows for the UI thread to "set" or "get" critical data when the audio thread enters the processing callback. This allows both threads to access the data while avoiding any race conditions.

Message queuing offers a clean solution when the audio thread is running, but it can present problems when the system is in a dormant state. In this situation, for example, a queueing "getter" method would block indefinitely, waiting for the processing callback to serve the request. As another example, a user will commonly populate a sequence with trigs before adding it to a running session. If the sequence length is longer than the message queue, this would overflow the buffer and cause an error.

Ideally, the getters and setters would access data directly when operating on a dormant structure, and use message queues when the sequencer is running. In Sequoia, this branching behavior is handled automatically – the data access methods are polymorphic according to the running state of the system.

5. GENERATIVE TECHNIQUES

In addition to serving as a streamlined API for general-purpose, time-critical sequencing with real-time control, Sequoia has been designed from the ground-up with generative music techniques in mind. Here, we describe just a few of these possibilities which Sequoia enables.

5.1. Polymeter

Since there's no concept of a global step counter in Sequoia (only the per-tick frame counter managed by the session), sequences are free to run in and out of phase with each other, according to the least-common-multiple of their lengths. For example, a 16-step sequence played against a 15-step sequence will evolve through 240 steps of variation before syncing back up and repeating itself.

5.2. Probability

Trig parameters include *probability*, a floating-point value in the range $[0, 1]$ which determines what fraction of the time a trig actually fires. This applies to both note-type and CC-type triggers.

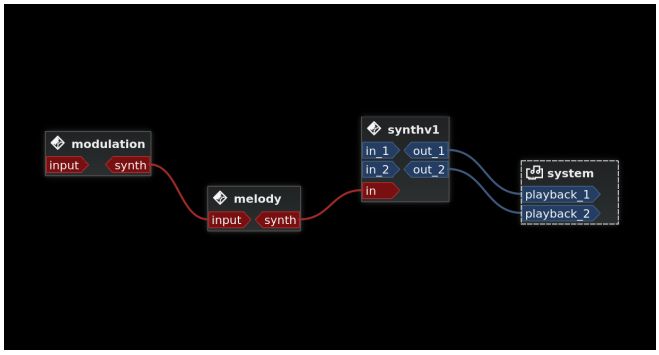

Figure 3: *Meta-sequencing. A Carla patch showing a slow modulation sequence controlling the transpose parameter of a melody sequence, which is driving the synthv1 synthesizer.*

5.3. Meta-sequencing

Meta-sequencing, simply put, is "sequences sequencing sequences". Any of the sequence parameters – playhead, loop start, loop stop, playback mode, transpose, mute state, clock divide – can be controlled live from Sequoia's MIDI-in ports. The way MIDI events map to parameter controls is determined by a mapping defined by the user upon sequence creation.

Combined with the concepts described above, this technique can be very powerful – a single, monophonic sequence can be manipulated by another (perhaps employing polymeter, probability, or clock division) to generate a much longer, stochastically evolving sequence (see Figure 3). Sequences can even be looped back into themselves to give surprising results (Figure 4) – although care must be taken in this case to avoid runaway conditions.

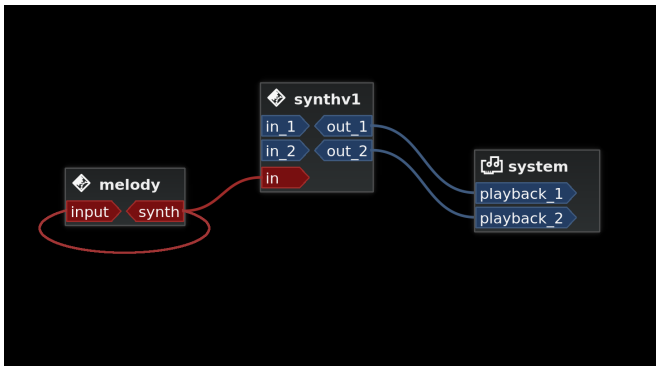

Figure 4: *Auto-sequencing. A melody sequence is fed back into itself (notice the looped-back red line from synth to input on the melody client), and the result is used to drive synthv1. Depending on the melody and the input mapping, this situation can "run away" to infinite pitch. If it doesn't, the results can be a surprising transformation of the original melody.*

5.4. Algorithmic Control

Obviously, the facility of inports and controller mappings allows for external clients (e.g. Python scripts, Pure Data patches, Geiger counters with USB connections...) to control sequence parameters in any way one might wish, thus allowing a huge variety of algorithmic methods to modulate the sequencer.

6. STATUS

Sequoia is currently in active development. The core library (libsequoia) is in a viable state, and the source code is available on GitHub under the GPL license (v3) [12]. We are also in the process of embedding the library within Ziggurat, an existing GUI sequencer application [13]. Future work will focus on developing bindings to other languages, and improving documentation.

7. REFERENCES

[1] Connor Jones, *A Live Performance Revolution is Taking Over Electronic Music*, 2016.

[2] Peter Salus, *A Quarter Century of UNIX*, Addison-Wesley, 1994.

[3] Wikipedia contributors, *Seq24*, 2019.

[4] Wikipedia contributors, *Ardour (software)*, 2019.

[5] Wikipedia contributors, *LMMS (software)*, 2019.

[6] Wikipedia contributors, *Qtractor*, 2019.

[7] Wikipedia contributors, *Rosegarden*, 2019.

[8] Wikipedia contributors, *MusE*, 2019.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, 1994.

[10] Michael Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010.

[11] Ross Bencina, *Real-time audio programming 101: time waits for nothing*, 2011.

[12] Chris Chronopoulos, *https://github.com/chronopoulos/libsequoia*, 2018.

[13] Chris Chronopoulos, *https://github.com/chronopoulos/ziggurat*, 2018.