AN APPLICATION OF THE ACTOR MODEL OF CONCURRENCY IN PYTHON: A EUCLIDEAN RHYTHM MUSIC SEQUENCER

Daniel P. Prince and Saverio Perugini
Department of Computer Science
University of Dayton
300 College Park
Dayton, Ohio 45469–2160
(937) 229–4079
saverio@udayton.edu

ABSTRACT

We present a real-time sequencer, implementing the *Euclidean rhythm algorithm*, for creative generation of drum sequences by musicians or producers. We use the Actor model of concurrency to simplify the communication required for interactivity and musical timing, and generator comprehensions and higher-order functions to simplify the implementation of the Euclidean rhythm algorithm. The resulting application sends Musical Instrument Digital Interface (MIDI) data interactively to another application for sound generation.

INTRODUCTION

A vast range of music production software has become available recently due to the increasing power of personal computers and increasing popularity of audio production among hobbyists and professionals. While an expanse of robust and effective music production software is available, musicians and producers desire more flexible and creative software applications. The Euclidean sequencer described here addresses this issue by providing musicians with an array of simple sequencers that can be controlled in real-time to create appealing rhythms.

The *Euclidean rhythm algorithm* is a compelling approach to creating interesting musical sequences with a minimal user interface, as the possible sequences model many common rhythms in popular and world music using easily adjustable parameters [14]. The use of the Euclidean rhythm algorithm to create interactive musical instruments is under-explored; more intuitive interfaces for real-time performance are necessary to support its widespread use for this purpose. Comparable sequencing approaches such as the common Roland TR-808 style step sequencers require the user to manually enter the states of every step in a sequence, which allow the user to precisely define the sequence that they are configuring. However, these step sequencing strategies require comparatively more data entry by the user, and can feel tedious at worst. In this work, the Euclidean rhythm algorithm is explored for its immediacy and ability to quickly define complex sequences.

Audio production software necessitates a unique set of technical demands to facilitate an experience consistent with modern standards. Software used for music production must not only provide precise timing for the generation and playback of musical passages that often involves demanding digital signal processing algorithms, but it must also provide an interface to a user that allows for real-time configuration of the model behind the music. This combination of synchronous and asynchronous work is a demanding task for synchronization models. Classical approaches to synchronization, such as semaphores do not scale well to applications with many concurrent tasks. Due to the complex synchronization issues related to creating an audio production application, the Actor model can be used to simplify the communication scheme. The fault-tolerant quality of actors are also helpful for live music performance, where reliability is important. For instance, it is possible for an application using actors to continue playing music even if there is an error in the actor that manages the GUI thread.

This paper is organized as follows. The Musical Terminology subsection introduces the reader to music vocabulary necessary to

discuss the work. The Modeling Euclidean Rhythms using Generator Comprehensions subsection describes the Euclidean rhythm algorithm and describes how generators are well suited to its implementation. The Performance Controls subsection describes the features available for a user to interact with the sequencer in real time. The Synchronized Actors through Message Passing section describes how the Actor model is used to facilitate communication between concurrent demands in the application. The Evaluation subsection briefly summarizes the results of the work. Finally, the CONCLUSION section summarizes the work and provides a few remarks on proposed future work.

RELATED WORK

The Actor model is a scaleable and fault-tolerant approach to concurrency [3, 7, 11]. This model of concurrency is especially applicable in functional programming languages such as Elixir and Scala due to the absence of mutable state [9]. However, libraries supporting the use of actors exist for a wide range of languages. We use the *Pykka* actor library for Python to support our use of the Actor model of concurrency within Python [2]. The *Pykka* library is based on the design of the *Akka* library for actors in Java and Scala [1].

The ability to use computers to map any physical sensor to a computational model and generate interesting musical output has resulted in a wide range of applications. Several recent examples include the use of tangible wooden pucks placed on a table read with rotating optical sensors for real-time sequencing [5], a sequencer that uses only a user's specified degree of excitement as input to a hidden Markov model for selecting techno loops [10], and even the use of biological organisms for step sequencing [6]. We built a traditional GUI for the user to input parameters and real-time commands through a a mouse and keyboard. The use of other input devices for more intuitive or creative control is suggested as future work.

Other recent work that uses Euclidean rhythms for music gen-

eration include *XronoMorph*—a sequencer that uses geometrical input that uses models related to Euclidean rhythms for composition as well as education and performance [13]. Another application is a multi-robot system that cooperates to generate real-time algorithmic music based on Euclidean rhythms [4]. Despite this work, the Euclidean sequencer represents an interesting computational model that is still under-explored for musical performance and composition.

TECHNICAL DETAILS

Musical Terminology

We define the following terms for both textual consistency and readers without a working knowledge of music theory and technology.

- **Sequencer**: a device used to record and transmit musical notes for the purpose of representing a performance electronically.
- **Sequence**: a series of notes stored in a sequencer that represents a musical phrase.
- **Part**: the sequence that corresponds to a single instrument. In this application, there are six parts.
- **Step**: the smallest discrete musical unit represented by the sequencer. In a given sequence, a step either indicates the presence of a note onset or the lack of a note onset. Here, one step is assumed to be the duration of a sixteenth note, although it could be reconfigured for any other note division.
- **Beat**: a musical unit of time perceived by the listener as the regular occurring pulse of a piece of music (usually represented by a quarter note).
- **Beats Per Minute** (BPM): a common rate used to describe the tempo, or speed, of a piece of music.

Modeling Euclidean Rhythms using Generator Comprehensions

The two most interesting parameters in the Euclidean rhythm algorithm are k and n, where k indicates the number of note onsets and n indicates the length of the sequence in steps. The Euclidean rhythm algorithm spaces the k note onsets as evenly as possible into the n available steps, but some uneven spacing occurs when an even spacing is not possible over the n discrete divisions.

After the n^{th} step in a Euclidean rhythm sequence, the next n steps always have the same step states as first n steps, repeating infinitely as long as the sequence is playing. This particular property of Euclidean rhythms means that the resulting sequences formed by Euclidean rhythms are well suited to programming constructs such as *generator comprehensions*—an application of *lazy evaluation* to simulate data structures of infinite size [8]. In the developed Python application, a list of the native generator class is used to represent the current sequences for each part. To get the next step from each part at a given time, Python's `next` function for getting a value from a generator is mapped across the generator sequence corresponding to each part.

Figure 1 shows a simple example of a pair of Euclidean rhythms that are common in many forms of music. Each vertical grid line indicates the musical duration of one sixteenth note, the same amount of time represented by one step. The top sequence uses the parameters $k = 1, n = 3$, while the bottom sequence uses the parameters $k = 1, n = 4$. This results in one note being placed in every three and four steps, respectively. Interestingly, for any given combination of part sequences, the resulting rhythm periodically repeats after a number of steps equal to the least common multiple of the n values for each part has been reached. In this example, the combined sequence repeats after 12 steps, or at the "1.4" and "2.3" marks shown in Figure 1. Figure 2 shows a more musically interesting result, which uses the common eighth note pattern on the closed hi-hat part using the parameters $k = 1, n = 2$ and the equally ubiq-

Figure 1: A screen capture from *Ableton Live* illustrating the relationship between two simple Euclidean sequences.

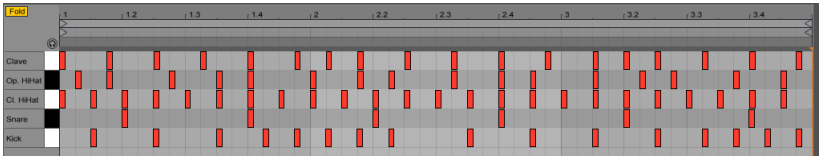

Figure 2: A screen capture from *Ableton Live* illustrating an example of a complex drum machine part created by the Euclidean sequencers.

uitous backbeat on beats 2 and 4 on the snare using the parameters $k = 1, n = 8$.

Performance Controls

The graphical user interface to this application provides a variety of performance options for the musician controlling it. Due to the simple parameters that determine the result of a single Euclidean rhythm, each sequence can be adjusted in real-time to control the dynamics and groove of the performance. Figure 3 shows the application's GUI.

A user can spend as much time as necessary selecting the k and n parameters for an individual sequence, and then apply the changes to the next note using one sequence's start button. Additionally, a single sequence can be restarted by clicking the start button again. This allows the performer to alter a sequence's relationship to the period of the other sequences by restarting it before its end, and also to repeat the beginning of a sequence multiple times in quick succession, allowing an option for a more manual control of what is otherwise a mostly automatic rhythm generation.

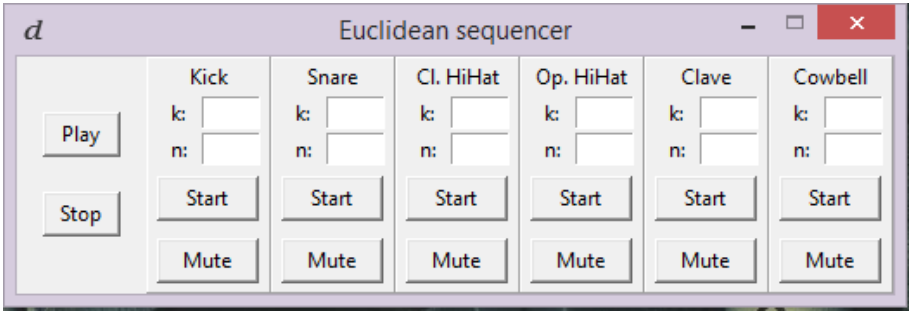

Figure 3: The GUI of the application in Windows 8.1.

The mute button per part particularly allows the performer a simple toggle for control of dynamics. This option enables for an easy use of a technique commonly found in pop and electronic music, where a single percussion part drops out for a number of measures to provide a breakdown of a lower dynamic level, or to provide anticipation of an upcoming change in dynamics in a future measure.

Synchronized Actors through Message Passing

Figure 4 depicts the overall system architecture of the Euclidean sequencer application, and its relationship to the audio processing chain in which it is situated. In particular, the left side of Figure 4 presents the model of concurrent actors (nodes) and the messages passed between them (directed edges) in the application, which are summarized in Table 1. The `NoteActor` manages the logical state of the application, including the current sequence parameters and their mute states. It waits for the periodic `tick` message from the `TimingActor` to determine when it should send MIDI messages on its output. It also waits for asynchronous configuration messages from the `GuiActor` that indicate that the user has interacted with the GUI and that the state of the application should be updated.

The `TimingActor` consists mainly of a loop which waits for a set period of seconds that define the steps of the rhythms in the appli-

Table 1: Description of actors in the application.

Actor Name	Action
`TimingActor`	Count musical divisions of time.
`NoteActor`	Generate rhythms and send notes.
`GuiActor`	Display status and enable interaction.

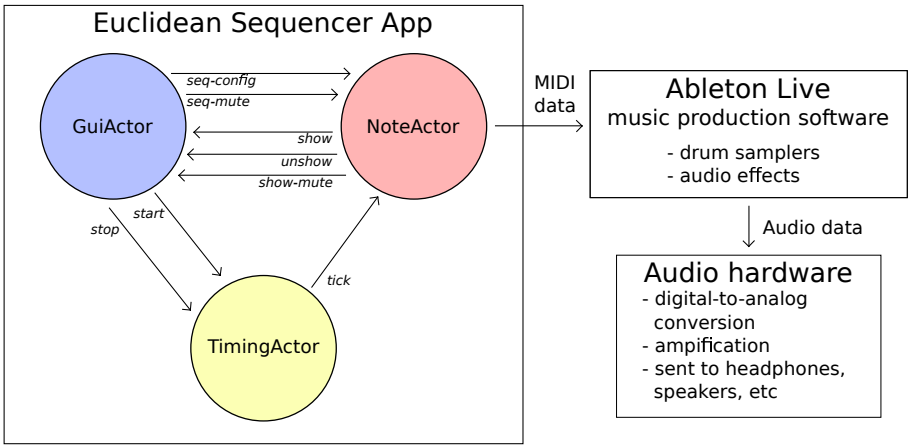

Figure 4: System architecture of the Euclidean sequencer application, and its relationship to the audio processing chain in which it is situated.

cation. Using the default tempo of 120 BPM, the `TimingActor` waits 0.125 seconds between each note. The calculation used to reach the period between ticks is described by the following equation:

$$\left(120\frac{beats}{min}\right)^{-1} \times 60\frac{sec}{min} \times \left(4\frac{beats}{step}\right)^{-1} = 0.125\frac{sec}{step}$$

The `GuiActor` handles the asynchronous interaction with the user through the GUI. The `GuiActor` sends the message `seq-config` to the `NoteActor` when the user enters a new combination of sequence parameters, and it sends `seq-mute` when the user presses the mute button.

Evaluation

It is imperative that any interactive, real-time music software application react to input from the user in a consistent and responsive way. In general, these applications should be responsive on the order of several milliseconds and with low variation of latency to be considered effective for live music performance [15].

The sequencer application is only responsible for generating Musical Instrument Digital Interface (MIDI) [12], which is a relatively simple computational operation. After the MIDI events are generated, they are sent to *Ableton Live* for generating the corresponding audio signal, which is a more demanding computational task. The combination of the sequencer application and *Ableton Live* result in a musical system that experiences no perceptible latency when running on an *Apple Macbook Pro*, Early 2015 model. The system likely has latency on the order of several milliseconds and seems to be usable for live performance.

CONCLUSION

We developed a Euclidean rhythm sequencer application for interactive, real-time performance. We demonstrated that, by separating asynchronous user interaction, synchronous timing, and the application's logical state, the Actor model of concurrency is an appropriate approach to the problem of musical synchronization. We also demonstrated that generator comprehensions in Python can be used to model Euclidean rhythms and other cyclic musical sequences. A Git repository containing the Python source code of the Euclidean rhythm sequencer application is available in BitBucket at `https://bitbucket.org/sperugin/euclidean-rhythm-music-sequencer/`.

To provide more extensive performance options, MIDI control of the application could be offered to allow for hardware control of the Euclidean rhythms instead of control by mouse and keyboard. This level of hardware control would allow musicians to more quickly

orient themselves with the application's controls when they are using multiple pieces of equipment for a performance. The mouse and keyboard interface implemented currently provides an effective proof of concept.

An advantage of the Actor model of concurrency is its ability to scale to large applications with a comparatively small level of code complexity. This property supports the rapid evolution of the the application described here to incorporate many common features of complete digital audio workstations, including audio recording, incorporation of digital signal processing based effects, and communication with a more diverse set of MIDI devices.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Numbers 1712406 and 1712404. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Akka documentation. https://akka.io/. retrieved May 27, 2018.

[2] Pykka documentation. https://www.pykka.org. retrieved May 27, 2018.

[3] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[4] Albin, A., Weinberg, G., and Egerstedt, M. Musical abstractions in distributed multi-robot systems. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 451–458, 2012.

[5] Arellano, D. and McPherson, A. Radear: A tangible spinning music sequencer. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 84–85, 2014.

[6] Braund, E. and Miranda, E. Music with unconventional computing: towards a step sequencer from plasmodium of *Physarum Polycephalum*. In *Proceedings of Conference on Evolutionary and Biologically Inspired Music, Sound, Art and Design; Lecture Notes in Computer Science*, volume 9027, pages 15–26. Springer, Cham, 2015.

[7] Butcher, P. *Actors*. Pragmatic Bookshelf, Dallas, TX, 2014.

[8] Henderson, P. and Morris Jr, J. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, pages 95–103, New York, NY, 1976. ACM Press.

[9] Karmani, R., Shali, A., and Agha, G. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20, New York, NY, 2009. ACM Press.

[10] Kitahara, T., Iijima, K., Okada, M., Yamashita, Y., and Tsuruoka, A. A loop sequencer that selects music loops based on the degree of excitement. In *Proceedings of the 12th Sound and Music Computing Conference*, pages 435–438, 2015.

[11] Li, Z. and Kraemer, E. Programming with concurrency: Threads, actors, and coroutines. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and Ph.D. Forum*, pages 1304–1311, Los Alamitos, CA, 2013. IEEE Computer Society Press.

[12] MIDI Manufacturers Association. The official MIDI specifications. https://www.midi.org/specifications. retrieved May 27, 2018.

[13] Milne, A., Herff, S., Bulger, D., Sethares, W., and Dean, R. XronoMorph: algorithmic generation of perfectly balanced and well-formed rhythms. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2016.

[14] Toussaint, G. The Euclidean algorithm generates traditional musical rhythms. In *Proceedings of BRIDGES: Mathematical Connections in Art, Music and Science*, pages 47–56, 2005.

[15] Wessel, D. and Wright, M. Problems and prospects for intimate musical control of computers. *Computer Music Journal*, 26(3):11–22, 2002.