Experimenting with a Generalized Rhythmic Density Function for Live Coding

Renick BELL
Tama Art University
2-1723 Yarimizu
Hachioji, Tokyo 192-0394
Japan
renick@gmail.com

Abstract

A previously implemented realtime algorithmic composition system with live coding interface had rhythm functions which produced stylistically limited output and lacked flexibility. Through a cleaner separation between the generation of base rhythmic figures and the generation of variations at various rhythmic densities, flexibility was gained. These functions were generalized to make a greater variety of output possible. As examples, L-systems were implemented, as well as the use of ratios for generating variations at different rhythmic densities. This increased flexibility should enable the use of various standard algorithimic composition techniques and the development of new ones.

Keywords

algorithmic composition, live coding, Haskell, L-systems, rhythm

1 Introduction

A system for realtime algorithmic composition was first presented in (Bell, 2011) and then improvements were described in (Bell, 2013). The intention of that system was to be able to do realtime algorithmic composition, primarily through a live coding interface. It was concluded that while the interonset interval (IOI) function and density function provided in the Conductive library could yield somewhat useful results, improvements could be made.

This paper describes attempted improvements in this area. First this paper briefly reviews how those functions were implemented previously and describes their output. It then explains the problems with that implementation and output. The paper then proceeds to describe the newly implemented version and its advantages, namely the use of a higher-order function to gain a more modular system. Two example inputs to this higher-order function were implemented and are described. Finally, conclusions are made and directions for future research are proposed.

2 Summary of Previous Rhythm Generation Technique

2.1 A Brief Review of Conductive

Conductive is a library for the Haskell programming language used for managing concurrent processes for realtime music. In addition to providing functions for managing those concurrent processes, it has some features for representing musical time and for algorithmic composition.

Concurrent musical processes are represented by a data structure called a Player. The Player refers to two functions: an action, which can be any IO function, which it runs repeatedly; and an IOI function, which determines the wait times, called interonset intervals (IOIs), that are interleaved between calls to the action function. More information on Conductive and performing with it can be found in (Bell, 2011) and (Bell, 2013). See Figure 1 for a graphical representation, originally included in (Bell, 2011).

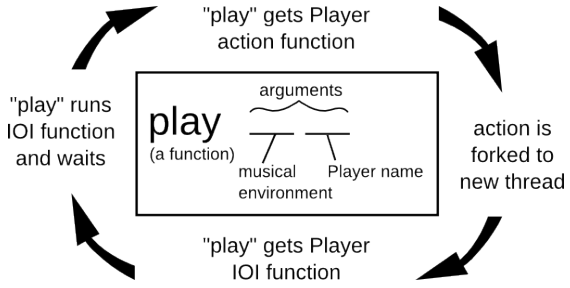

Figure 1: the Play loop, with Player, action function, and IOI function

2.2 Rhythm in Conductive

As described in (Bell, 2013), this author had been experimenting with reading IOI values from something called a density map. This concept involved two parts: the generation of rhythmic figures and the generation of an ordered stack of variations at lesser and greater rhythmic density. Both of these parts used

stochastic methods and were joined finally in a single function.

A higher-level abstraction was developed to generate a list and a set of rhythmically similar lists of greater and lesser density and store them in a table indexed by level of density. Doing so increased the likelihood that two lists would be perceived as having a rhythmic relationship and decreased the chance that an audience would perceive a kind of discontinuous state change when switching from one to another.

The base rhythmic figure was an ordered list of IOI values expressed in terms of beats, whole or fractional. To generate it, a performer selected a core unit which was used to generate potential IOI values. Selection of the core unit, in conjunction with the length of the pattern, largely determined the metrical feel of the pattern. A list of scalars was determined by the performer, from which the function randomly selected a user-specified quantity to multiply with the core unit. The user specified a number of subphrases to generate and the length of those phrases in terms of number of scalars to use. The subphrases were then generated by selecting the scalars and multiplying them by the base unit. Finally, a user-specified number of subphrases were chosen at random by the function. The user determined the length of the final phrase in terms of beats. If the length of the concatenated subphrases did not equal the specified length, the final IOI value was padded. If the length exceeded the specified length, the final IOI value was truncated. The repetition of values and subphrases within the final figure tended to give it a musical quality lacking in a list of purely random numbers. For a complete example, see (Bell, 2013).

Given a particular figure, a series of related patterns was generated in which the rhythmic density was increased or decreased. A large number of such patterns was generated in order so that a stack of patterns from very low rhythmic density to very high rhythmic density resulted, with the original figure somewhere in the middle. Those variations were generated in one of two ways, depending on whether they were to have greater or lesser density. When reducing density, one value from the figure was chosen at random and combined with a neighboring value. That new value was inserted into the figure in the place of those two selected values. The less dense variation was then subjected to the same process recursively until the figure contained only a single item, with the figure at each step added to a list. For increasing density, a value was chosen at random and replaced with two items: an item of lesser value from a list of potential IOI values and the difference between the original IOI value and the lesser value. The resulting figure was subjected to the same process, again recursively and retaining each version, until the figure consisted of a list of the smallest of the potential IOIs. By concatenating the original figure with the lists of greater and lesser density, a table was generated. For a complete example of this, see (Bell, 2013).

An integer value representing the density ranking, with 0 being the least dense pattern, was assigned to each figure in the table. That table, along with the list of potential IOIs and the total length of the figure, was stored in a data structure called an IOIMap.

The function call to execute this process looks like this, containing Ints, Doubles, and lists of each as arguments. The function returned an IOIMap containing the density map based on the generated rhythmic figure:

```
m00 <- iOIandRTfromPhrase 0.25 2
          2 4 [2] 2 [2] 4.0 3
```

Based on a user-specified density value, a particular IOI pattern is chosen from the table. The user queries the table with a value between 0 and 1, and a linear conversion to a list index is done. The value returned is the IOI pattern at that index. Based on the current beat, an IOI value is returned from that pattern.

Density values can vary with time. One method for doing so is employing a TimespanMap. A commonly observed pattern was setting the timing of particular values. It is often desired that values change over time but at different rates. TimespanMaps are structures for handling such cases. Rather than specify the exact timing of a value, it specifies the range of time in which that value can occur. They are maps or dictionaries with intervals as keys to any kind of value. Another parameter of the structure is a specified length at which it loops. When a time is passed to the dictionary, the interval that time falls in is determined to be the key to use, and the corresponding value for that interval is returned. When the time value passed to the TimespanMap exceeds those for which it is defined, it loops to return an appropriate value. For a graphical explanation, see Figure 2 in (Bell, 2013).

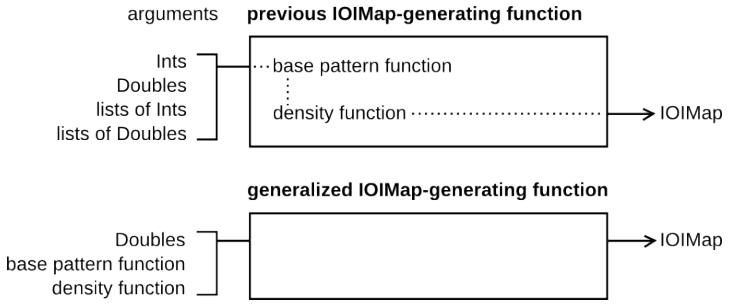

Figure 2: a comparison of the previous method and the new method

2.3 Problems Identified in the Previous Paper

The style of patterns produced was limited, both in the generation of rhythmic figures and their variations at various rhythmic densities. More inconvenient was the fact that other methods could not be tested without rewriting the core functions. A more modular solution was sought.

3 New Method: a Generalized Function for Patterns and Variations

To solve the problem described above, it was determined to rewrite the function that handled density and base patterns, generalizing it to take functions and make the density function a higher-order function. A higher order function is "a function that takes a function as an argument or returns a function as a result is called higher-order." (Hutton, 2010) In this case it takes two functions as parameters: one for the generation of the base pattern and a second for determining how the density of an input should be increased. The function is passed those two functions and a parameter determining how long the rhythmic figures should be. It uses the pattern function to produce a base pattern. It then processes the base pattern with the density function to create the rhythmic variations. It returns an IOIMap, which includes a set of TimespanMaps mapping time intervals to values of next beats ordered according to their density value.The new function has been given the temporary name of newIOIMap2, to be used until a better method of naming it is determined.

The benefit of doing so is that the basic structure is already available before a performance and does not need to be coded at that time or recoded when the current method for generating patterns is longer useful. That means that generating patterns and then making a table of values which can be read from according to a density value can be accomplished more easily and in a greater variety of ways. John Hughes writes in his essay "Why Functional Programming Matters" that higher-order functions are one of two important kinds of "glue" that increase modularity, "the key to successful programming". (Hughes, 1989) The use of higher-order functions in programming for aesthetic output has been described in (McDermott et al., 2010).

As an example of an alternate method for increasing density, a function for increasing density by ratio has been implemented. As an example of a function for generating base rhythms, two types of L-systems were implemented.

3.1 Density by Ratio

This method applies when generating variations of increased rhythmic density.

The user specifies a list of ratios, a lowest target value, and a limit to how small the IOI values in the rhythmic figure can be. A value is selected at random from the rhythmic figure. A ratio is chosen at random from the list of ratios provided by the user. The ratio is applied to the value, which is subtracted from the original value. These two new values are then shuffled and inserted into the rhythmic figure in place of the original value. The new rhythmic figure is stored in a list, and the process is repeated on this figure. This process is carried out recursively until all of the IOI values in the rhythmic figure are equal to or less than the user-specified lowest target value, producing a stack of increasingly dense rhythmic figures. The code for this procedure can be seen in the functions "densifier" and "densifier2".

Consider an example in which a list, [1,1,1,1] is progressively densified according to a ratio of 0.5, with 0.25 being the lowest possible value.

```
let addL unit phraseLength ratios name = do
        ioimap <- newIOIMap2 0 phraseLength (generateDensities2 unit ratios) $ lsysTest4
        rs +@ (name,ioimap)
```

Figure 3: an example of using the new higher order function, newIOIMap2

In this example, "it" is the ghci reference to the output of the previous command.

```
*> densifier 0.25 [0.5] [1,1,1,1]
[1.0,1.0,1.0,0.5,0.5]
*> densifier 0.25 [0.5] it
[0.5,0.5,1.0,1.0,0.5,0.5]
*> densifier 0.25 [0.5] it
[0.5,0.5,1.0,0.5,0.5,0.5,0.5]
*> densifier 0.25 [0.5] it
[0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5]
*> densifier 0.25 [0.5] it
[0.25,0.25,0.5,0.5,0.5,0.5,0.5,0.5,0.5]
```

This process would continue until all of the values in the list are equal to 0.25.

3.2 Explanation of L-systems

Lindemayer systems, abbreviated to L-systems, were developed by Aristid Lindenmayer in 1968 as "a theory of growth models for filamentous organisms" (Lindenmayer, 1968). They have since been used by many for the simulation of plant growth, for visual art, and to a lesser extent, music.

They are string-rewriting systems in which an input string, called an axiom, is transformed according to a set of rules in which each item in the string (a predecessor symbol) is rewritten as a successor string. By inputting this output back through the rule-set, successive generations can be obtained (DuBois, 2003).

Here is a small example of a rule set, input, and seven generations (Supper, 2001):

```
rules: a -> b
       b -> ab

input: a

output: b
        ab
        bab
        abbab
        bababbab
        abbabbababbab
        bababbababbabbababbab
```

L-systems which do not have one-to-one string replacement rules grow in length rapidly as seen above and require users to employ techniques to deal with that size (DuBois, 2003).

3.3 History of L-systems in Algorithmic Composition

L-systems have been used in a variety of ways for algorithmic composition. Some examples from the literature are listed below.

L-systems are frequently used for pitch content. Supper describes the use of L-systems and cellular automata for algorithmic composition (Supper, 2001). Langston used L-systems to choose from previously composed musical phrases (Langston, 1989). Morgan uses a system somewhat similar to Langston in which previously generated pattern fragments are chosen from and assembled according to an L-system-generated template (Morgan, 2007). One of the most complete and useful discussions of using L-systems for music is the dissertation by R. Luke Dubois, which describes various methods for generating patterns of pitches in monophonic melody lines and chords.

Worth and Stepney describe a set of L-system-selected rules by which note duration is progressively transformed (Worth and Stepney, 2005). Use of L-systems for duration or rhythm are described by Kaliakatos-Papakostas, Floros, et. al., including the use of what they call FL-systems, in which L-system output is constrained in length (Kaliakatsos-Papakostas et al., 2012). Kitani and Koike have also described a method of generating rhythms from L-systems in combination with a learning algorithm (Kitani and Koike, 2010). Liou, Wu, and Lee use L-systems to compute the complexity of rhythms (Liou et al., 2010).

For more about L-systems, readers are referred to the dissertations of Dubois (DuBois, 2003) and Manousakis (Manousakis, 2006) first and then the other items listed above.

3.4 The L-system Function Implemented for this System

The initial intention for using L-systems with the higher-order function described above is to generate the base rhythms from which the density table described above can be generated.

The module itself contains functions for generating a string output from an axiom, a rule

set, and the generation number. Using the map function, a set of several generations can be obtained.

The rule set is notated with a colon rather than the traditional arrow for speed of entry. The predecessor symbol and successor string are written without spaces and separated by a colon. Each production rule must be separated by a space. The previous example can be run in ghci as follows:

```
*LSystem> let rules = "a:b b:ab"
*LSystem> getGeneration2 1 rules "a"
"a"
*LSystem> getGeneration2 2 rules "a"
"b"
*LSystem> getGeneration2 3 rules "a"
"ab"
*LSystem> getGeneration2 4 rules "a"
"bab"
*LSystem> getGeneration2 5 rules "a"
"abbab"
```

A more complicated example follows:

```
rules: "a:ab b:acd d:gx e:abc f:ga g:d"
axiom: "abcdefg"
```

A symbol which has no rule is kept as-is. In is the same as if the rule were to repeat the symbol, such as "c -> c".

In this case, the output in the interpreter of the first three generations of this L-system are:

```
*LSystem> getGeneration2 1 "a:ab b:acd
d:gx e:abc f:ga g:d" "abcdefg"
"abcdefg"
*LSystem> getGeneration2 2 "a:ab b:acd
d:gx e:abc f:ga g:d" "abcdefg"
"abacdcgxabcgad"
*LSystem> getGeneration2 3 "a:ab b:acd
d:gx e:abc f:ga g:d" "abcdefg"
"abacdabcgxcdxabacdcdabgx"
```

Two methods have been tested:

- direct output of IOI values
- lists of value-transforming functions applied in sequence to a base value

The direct output of IOI values means that given an axiom, a rule set, the generation number, and a list of potential IOI values, the function will return a list of IOI values. How those IOI values are assigned to the symbols is a matter for which a large variety of options exist. One simple choice is to randomly assign a value to each unique symbol. The string is then rewritten as that list of numeric values. This example illustrates such a method. The list of values ranges from 0.25 to 1.25, containing every step of 0.25.

```
*LSystem> getGeneration2 5 rules "a"
"abbab"
*LSystem> let a = it
*LSystem> randomFinalizer2
[0.25,0.5..1.25] a
[1.25,0.25,0.25,1.25,0.25]
*LSystem> randomFinalizer2
[0.25,0.5..1.25] a
[1.25,1.0,1.0,1.25,1.0]
*LSystem> randomFinalizer2
[0.25,0.5..1.25] a
[0.5,0.75,0.75,0.5,0.75]
```

In the case of using transform, the rules of the L-system are mathematical functions that modify a numerical value. First the output of the L-system is similarly rewritten with one mathematical function randomly chosen for each unique symbol in the string. Given a starting value and that list of mathematical functions, the number is passed through the list so that the output of one function becomes the input of the next. The changes are accumulated so that each step in the transformation of the initial number is kept. That series of numbers is then processed as deltas on which the density function will generate variations. Here is a very simple example of a list of functions processing a value:

```
*> transform 2 [(2 + ),((-1) +),(3 *)]
[2,4,3,9]
```

Here is an application of using the transform function on the output of an L-system.

```
*LSystem> getGeneration2 5 "a:b b:ab"
"a"
"abbab"
*LSystem> let a = it
*LSystem> let b = nub a
*LSystem> b
"ab"
*LSystem> let c = zip (map (\x -> [x])
b) [(1+),(0.5*)]
*LSystem> let d = flatFinalizer c a
*LSystem> :t d
```

```
d :: [Double -> Double]
*LSystem> transform 2 d
[2.0,3.0,1.5,0.75,1.75,0.875]
```

In this example, the variable "d" is the output of the function flatFinalizer, which converts the symbols in a string to their equivalents in a dictionary. In this case, the dictionary is "c", which maps the output of the L-system, "b", to the list of operations above. The ghci command ":t" shows the type of something, and in this case is used to show that d is a list of functions which take a Double and return a Double. The nub function returns a list in which all duplicate items have been removed. In this case, "d" would expand to:

```
*LSystem> transform 2 [(1+),(0.5*),
(0.5*),(1+),(0.5*)]
[2.0,3.0,1.5,0.75,1.75,0.875]
```

The final output in both of these examples, i.e. the list of Doubles, is used as a list of IOI values. Those values are then processed into a density map as described in sections 2 and 3.1.

4 Conclusion

An example of the function call now used to generate the IOIMap is shown below, including the partially applied function *generateDensities2* and the function *lsysTest4* can be found in Figure 3.

A brief example of using the L-system-based method described above can be heard at this URL: http://renickbell.net/sound/renick-bell-fractal-beats-test-140209-b.mp3

In this example, a collection of 100 density maps was created from a single L-System – "a:ab b:c c:abc d:ded e:aabb f:ga g:d" "abcdefg" – using the direct random selection of IOI values described above. Through the performance, 17 of those are auditioned using a variety of audio sample sets as well as live modification of the envelopes which control the density level.

Another brief example can be found at: http://renickbell.net/sound/renick-bell-fractal-beats-140125.mp3

In the near future, this code should be cleaned and added to one of the Conductive packages at Hackage, the Haskell package repository. A rough version of the code can be found in the meantime at this URL: http://renickbell.net/code/generalized-density.zip

With the modifications described above, the system certainly gained an additional degree of freedom. The system should now serve better as a platform for testing various algorithmic composition techniques. That could include more complex L-systems, stochastic systems, and so on.

Using ratios for increasing density works fairly well as long as the ratios are very simple, like 0.5. Other ratios generate patterns that are likely less familiar to listeners, and thus might not be appropriate if the composer has the intention of producing music that neatly fits within most existing genres. However, this was just an example, and more sophisticated methods can now be more easily tested.

The use of L-systems is also interesting, but it will take additional practice to become acquainted with L-systems and how, in the middle of a performance, to write rules and axioms for interesting output. As with the density function above, these L-system functions are also simple examples that can be refined or replaced for future work. They simply demonstrate that generalization was possible.

5 Acknowledgements

I would like to thank Akihiro Kubota and Yoshiharu Hamada for research support.

6 References

Renick Bell. 2011. An Interface for Realtime Music Using Interpreted Haskell. In *Proceedings of LAC 2011*. editions.

Renick Bell. 2013. An Approach to Live Algorithmic Composition using Conductive. In *Proceedings of LAC 2013*. editions.

Roger Luke DuBois. 2003. *Applications of generative string-substitution systems in computer music*. Ph.D. thesis, Columbia University.

John Hughes. 1989. Why functional programming matters. *The Computer Journal*, 32(2):98–107.

Graham Hutton. 2010. *Programming in Haskell*. Univ. Press, Cambridge [u.a., editions.

Maximos A. Kaliakatsos-Papakostas, Andreas Floros, Nikolaos Kanellopoulos, and Michael N. Vrahatis. 2012. Genetic evolution of L and FL-systems for the production of rhythmic sequences. In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference*, pages 461–468. ACM, editions.

Kris M. Kitani and Hideki Koike. 2010. Improvgenerator: Online grammatical induction for on-the-fly improvisation accompaniment. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010)*. editions.

Peter Langston. 1989. Six techniques for algorithmic music composition. In *15th International Computer Music Conference (ICMC), Columbus, Ohio, November*, pages 2–5. Citeseer, editions.

Aristid Lindenmayer. 1968. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299.

Cheng-Yuan Liou, Tai-Hei Wu, and Chia-Ying Lee. 2010. Modeling complexity in musical rhythm. *Complexity*, 15(4):19–30.

Stelios Manousakis. 2006. Musical L-systems. *Koninklijk Conservatorium, The Hague (master thesis)*.

James McDermott, Jonathan Byrne, John Mark Swafford, Michael O'Neill, and Anthony Brabazon. 2010. Higher-order functions in aesthetic EC encodings. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, editions.

Nigel Morgan. 2007. Transformation and mapping of L-Systems data in the composition of a large-scale instrumental work. In *Proceedings of ECAL 2007 Workshop on Music and Artificial Life (MusicAL 2007)*. editions.

Martin Supper. 2001. A few remarks on algorithmic composition. *Computer Music Journal*, 25(1):48–53.

Peter Worth and Susan Stepney. 2005. Growing music: musical interpretations of L-systems. In *Applications of Evolutionary Computing*, pages 545–550. Springer, editions.