Data-Driven Generative Live Coding for Music Creation

Gordan Kreković[1] and Antonio Pošćić[2]

[1] Visage Technologies
[2] Unaffiliated
[1] gordan.krekovic@visagetechnologies.com
[2] antonio.poscic@gmail.com

Abstract. Live coding, as a practice of computer programming used to create music and digital media, represents a novel and relevant form of contemporary artistic practice. The notion of automating the process of live coding entails interesting philosophical, conceptual, and technical questions. While touching upon all these implications, this paper focuses on design decisions and technical aspects as responds to artistic requirements on such a generative, interactive system intended for generating computer code as makeshift music scores. With the existing assortment of techniques for algorithmic composition and text generation, one of the challenges was selecting the most appropriate approach for the task. We noticed a strong interdependence between material (a base corpus of source code) and the process (the generative algorithm) which motivated a novel, hybrid approach that targeted both the required systematization of the corpus and devising a custom algorithmic solution.

Keywords: Live Coding, Artificial Intelligence, Machine Learning, Generative Art, Computer Art.

1 Introduction

Often used in performing arts, live coding is the practice of writing program code concurrently with its execution that enables artists to interact with the computer in real time (Blackwell et al., 2014). While the use of live coding has spread throughout different disciplines – from visual arts to dance – it is most commonly associated with computer music (Magnusson, 2013). By providing a common language between performer and instrument, live coding allows artists to interact with computer systems on a deep level. Here, they manipulate and synthesize sounds, then compose them into music by eschewing traditional notions of "playing", which opens the practice even to musically untrained artists. Or, in Thor Magnusson's words (Magnusson, 2013):

"Live coding is [...] the formalization and encoding of music, often for machine realization, on the one hand, and the open work resisting traditional forms of encoding on the other. Live coding is a form of musical performance that involves the real-time composition of music by means of writing code."

In the most common scenario (Collins et al., 2003), the musician writes program code during a live performance, improvising and creating music based on external influences or moods, often resulting in non-deterministic spontaneous compositions. When performed in clubs, these compositions form events called algoraves (Collins et al., 2014).

To examine the relationship between live coding and generative algorithms, we have created a part performance, part installation work called Anastatica. The base of Anastatica is the algorithm that generates lines of code that manipulate audio samples and create music. This potentially endless generative process is joined by a performer of flesh and blood who improvises and alternately plays with and against the generated music. At a certain point, the performance begins allowing input from the audience via a web-based interface. The audience is given a chance to manipulate the computer-generated code, with the choice between augmentation and erosion left to each individual. While Anastatica opens a number of philosophical, aesthetic, and conceptual questions that we discussed in pervious papers (Pošćić & Kreković, 2020a; Pošćić & Kreković, 2020b), the technical aspect of the generative process is the main focus of this paper.

2 Algorithms and code as makeshift scores

Anastatica contains in itself two strains of influences. On one hand, it draws from the work of live coders such as Alex McLean, Nick Collins, Shelly Knotts, Alice Eldridge and Chris Kiefer, and Orchestra for Females and Laptops (OFFAL). While the music and performances of these musicians pioneered and established an aesthetic framework for live coding, their work has since sprung into numerous directions, exploring both musical and extramusical innovations of live coding, from physical interfaces to internet-aided collaborative practices. As such, Anastatica is indebted to their work as it continues canvassing a similar aesthetic space (e.g. projecting TidalCycles code on a screen), but also retains their sense of research at the boundaries of technologies and techniques in the field.

On the other hand, Anastatica is also heavily influenced by musicians using artificial intelligence and generative systems in their work. In particular, this refers to musicians and computer scientists who have pioneered the use of AI in music outside of academic circles, including Zack Zukowski and CJ Carr's Dadabots (Zukowski & Carr, 2018), a SampleRNN-based system which generates or "invents" new music based on existing samples. Similarly, Holly Herndon and Matt Dryhurst's Proto and James Ginzburg and Paul Purgas's Blossoms (as Emptyset) were one of the first albums that employed AI in the creation of commercial, club and electronic music. Elsewhere, and most similar to Anastatica, Jennifer Walshe's ULTRACHUNK, realized in collaboration with visual artist and researcher Memo Akten, is a piece that focuses on the phenomenology of AI itself.

2.1 Algorithmic manipulation of patterns

TidalCycles offers the concept of patterns and cycles by providing a language for describing heterogeneous sequences (including polyphony, polyrhythms, and generative patterns) together with an extensive library of functions for building, combining, and transforming patterns (TidalCycles Userbase, 2021). Patterns automatically repeated in cycles simplify the coding practice and allowing musicians to focus on creating musical content instead of resolving the infrastructural question of triggering patterns. Manipulation of patterns in TidalCycles is used for generating control messages for SuperDirt, a sampler-based synthesizer in SuperCollider, or other synthesizers and sound effects via Open Sound Control (OSC) and Musical Instrument Digital Interface (MIDI) protocols.

While all the important paradigms of TidalCylces are explained in the introductory tutorial (TidalCycles Userbase, 2021), here we emphasize two main concepts important for understanding formative design decisions behind the generative live coding system. The first one is the notion of parallel connections or channels between TidalCycles and SuperDirt. Using 16 connections named from d1 to d16, it is possible to produce 16 simultaneous independent patterns. These parallel connections are not the only means of producing complexity in TidalCycle, as each pattern within a single connection can be composed of multiple patterns stacked or sequenced together. However, in the context of generative live coding, the existence of parallel connections opened a question of whether and how to distribute and interconnect the musical content among channels in order to achieve control over the vertical compositional aspect. In other words, if patterns in every channel are generated fully independently, theoretically any combination of musical contents from different channels can appear together which prevents both algorithmic and human control over the overall harmonic, polyphonic, and polyrhythmic qualities of the generated music. To address this challenge, we proposed a solution based on specific organization of the building code blocks that is described later in the paper.

The second fundamental concept is the paradigm based on patterns and their repetitive nature that is inherently associated with specific genres of electronic music. In order to make the sonic outcome more distinctive, instead of using the predefined set of samples, we created novel sets based on audio recordings of a violin and electromechanic piano. These analog, organic and imperfect instruments made a deliberate opposition to the rigidity and predictability of the algorithm.

3 Generative live coding

The main idea behind our work on generative live coding is to bring together humans and algorithms on a level playing field providing both organic and inorganic participants the common medium for music inscription. By “writing” TidalCycles code, the computer is no longer just an object. Instead, it becomes a subject that creates music, working on the same semiotic level as human observers and participants. With the code that writes code, the generative process becomes transparent to humans. The aspect of translating code into music – usually the main functionality of computers in music – becomes a corollary. It is the generative part that is key here, set in shapeshifting dynamics of antagonistic or complementary interactions.

One of the requirements on the generative system was to produce consistent sonic results that reflect musical ideas of the composer. At the same time, the variety in the generated music should be broad so that each performance is a unique experience. These apparently opposed expectations lead to an idea that the generative system should operate in a way to preserve musical qualities determined by the composer on the lower compositional time scales (i.e. timbral characteristics of atomic sound objects, sonic textures, and basic musical patterns), while introducing sufficient diversity on the higher level (i.e. phrases and gestures, musical movements, and overall performances). Such an idea is taken into consideration while designing the generative algorithm and the solution architecture.

3.1 Solution architecture

The first design decision was choosing an approach for generating source code among a wide assortment of possible techniques. While the raise of deep learning and advancements in natural language processing significantly contributed to the subfield of source code generation during the recent years, the tendency towards automatic programming is not new (Blazer, 1985). Historically, automation did not refer just to source code generation, but to other aspects more appropriate for their time: 1) translation of source code written in high-level programming languages (Brooker, 1958), 2) improving programming efficiency by using software components (Cointe, 2004), and 3) low-code development platforms that allow creating application software through graphical user interfaces and configuration instead of traditional hand-coded computer programming (Sahay et al., 2020).

Regarding source code generation techniques, the recent surveys (Allamanis et al., 2018; Le et al., 2020) provide an insightful overview on both traditional approaches (such as domain-specific language guided models, probabilistic grammars, n-gram models and simple neural program models) and deep learning techniques, some of which (Oda et al., 2015; Tiwang et al., 2019) have been applied to code generation with encouraging results.

While being superior in tasks like code completion, generating code comments, and translating source code to pseudo-code, current deep learning techniques do not seem to be fully appropriate for generative live coding. The first potential drawback is that the task of generative live coding cannot be formulated either in terms of code completion or any type of translation from a higher-level specification. Instead, the intended functionality is that the generative system automatically produces unlimited numbers of code blocks in a way to achieve the artistic goals: aesthetic consistency and controlled compositional quality at all musical time scales. Another potential drawback is the required amount and diversity of training data that surpass a reasonable amount of code that a single composer can consistently prepare in TidalCycles using a hermetic set of custom samples. Some strategies such as scraping GitHub repositories or weakening requirements on stylistic consistency may simplify the dataset collection process, but would jeopardize the overall goal.

As a solution, we devised and developed a novel, data-driven approach based on a Markov chain, a mature and proven model with a long history in algorithmic composition. The Markov chain enabled the earliest experiments in formalized music during the 1950s (Pinkerton, 1956; Brooks et al., 1957; Hiller & Isaacson, 1957), but still remained a viable, although limited, technique to model melodies and phrases (Pachet & Roy, 2011). Fundamentally, it is a machine learning algorithm whose parameters are induced from a corpus of pre-existing compositions or performances. The main weakness of this approach is the ability of capturing only local statistical similarities, while failing at inducing relations on higher time scales.

For the purpose of generative live coding, we created a Markov model whose states are blocks of TidalCycles code from the predefined corpus, while the transitions between the states are constructed algorithmically. By developing a custom algorithm for determining transition probabilities, we replaced standard statistical learning from pre-existing performances with a data-driven approach that takes into account the textual content of building blocks and their syntactic relations. This way, the generative process does not inherit performative characteristics and the compositional structures from pre-existing examples, but creates them autonomously following a procedural algorithm controlled by several parameters.

The architecture of our solution, therefore, consists of three components: 1) a preprocessing module, 2) the generative process, and 3) the participative interface. The preprocessing module takes the whole corpus of code blocks as an input and generates a set of Markov models. The generative process works in real time during performances or installation. It uses Markov models produced by the preprocessing module to recombine lines of code following Markov transitions, but it also provides an additional level of performative control and indeterminacy. The participative interface is a two-way web interface piped directly into the performance core that allows the audience to interact with the generative process by curating parts of the generated code, but it is not described in more details within this paper. The following sections explain each of the system components in more details.

3.2 Data-driven preprocessing

The preprocessing module runs on the predefined corpus of code blocks and produces multiple Markov chains (one for each channel in TidalCycles) by calculating the transition probabilities in a deterministic way. The resulting probabilities are closely related to the objective difference between code blocks in terms of how many characters have to be changed to obtain one code block from another. This solution favors smaller changes from one block to another in the same manner as a human musician during live performances usually modifies existing code blocks rather than writes new ones (Kreković & Pošćić, 2019).

The reason why the preprocessing module was designed to produce a Markov model for every unique connection used in the corpus (up to the maximum of 16 connections) is the fact that only changes that happen on the same connection affect the pattern repeating on that connection. If two successively generated blocks of code refer to different connections, the first block will continue producing sound and the second one would just additively contribute to the sonic content. In order to ensure that the transitioning between states are reflected in the generated music, the preprocessing module creates multiple Markov chains, one for each connection. The generative process is designed to combine these multiple models by travelling through them in parallel creating complex, potentially polyphonic and polyrhythmic results.

The first step of preprocessing is calculating the Levenshtein distance between all pairs of code blocks that refer to the same TidalCycles connections. The lower distance between two code blocks will lead to a higher transition probability between them. Since the Levenshtein distance is a deterministic and commutative measure (i.e. the distance between code blocks *A* and *B* is the same as the distance between *B* and *A*), this approach could lead to often looping between two most similar code blocks. For that reason, after calculating Levenshtein distance among one code block with all the others, the algorithm is modified to increase distances to all of those blocks for which the distance to the current one has been already calculated. In other words, if the matrix element *distance*(*A*, *B*) already exists, the *distance*(*B*, *A*) will be incremented by the maximal distance between the code block *B* and all other code blocks *X* for which *distance*(*X*, *B*) has not been calculated yet. This will ensure that the generative process favors new transitions before going to previous states.

The second step of the preprocessing algorithm is calculating transition probabilities. For each code block, the distances to other elements are ranked. If the block B is the most similar one to the block A, then it will have the rank 1, while the second closest will get the rank 2. The ranks are then used as negative exponents to the base. Finally, the results are normalized so that transition probabilities from each code block sums to 1. The transitions are permanently stored, so that the generative process can reuse them every time it runs.

3.3 Generative process

In each step, the generative process outputs one code block by making a transition to the next state of the Markov chain of the currently selected TidalCycles connection. Pauses between steps have random durations with limited upper and lower bound. The process of selecting a next connection is based on a unidirectional, cyclic random walk with the gamma distribution. The distribution is parameterized in a way that the step size 1 has the highest probability, while the larger steps sizes have non-linearly exponentially lower probabilities. For example, if the current connection is #5, the highest probability is that the next one will be #6, then #7, and so on. Modulo operations ensure that the process works in a cyclic manner. This approach reduces the number of repetitions of connections successive steps of the generative process, while still maintaining a certain level of indeterminism.

As the generative process executes their respective Markov models fully independently, this approach does not allow a composer to orchestrate selection of code for different connections simultaneously. In order to address this problem, we opted for organizing the corpus in a systematic way. The systematization guideline for Anastatica emerged through iterative trials and eventually reflected the desired distribution of code blocks that produce certain types of sonic qualities in different connections. For each connection we defined an approximate distribution of rhythmic, tonal, and textural contents. Due to the independent nature of the algorithm, there are still possibilities of combining undesired types of sonic materials, but those possibilities are shaped by the compositional process.

When composing Anastatica, we noticed that the musical evolution is usually gradual, as the algorithm tends to transition between syntactically similar blocks that result with a sonically similar content. Sudden changes occur, but they often happen in one of the connections, while all others continue with their patterns mitigating the effect of the change. In order to introduce more diversity on the compositional level, besides the Markov chains, the generative process was extended with a set of special events. Those are events such as tempo changes, hushing all connections simultaneously, hushing one connection, or resetting the states in Markov chains. Such events happen in random, yet rare moments and contribute to the dynamism of the compositional structure.

4 Discussion

The introduction of generative paradigm into live coding results in interesting combinations of human and machinic artistic interventions. Firstly, the generative process can act as an adversary by pushing the musician or performer outside of their comfort zones and pre-learned behaviors, resulting in novel forms of music or art (Kreković & Pošćić, 2019). This phenomenon was also apparent in Anastatica in which the generative part articulated shape-shifting dynamics of antagonistic or complementary interactions between the algorithm, human performer and the audience.

Secondly, the generative process can compose novel music on its own, but under the supervision and direction of the programmer or performer. Even if a generative process powered by AI and machine learning is treated as a post-algorithmic manifestation, the need for human influence is immense both in terms of the used material and the constructed process. The experience with developing the generative system for Anastatica confirmed the striking interconnection between those two aspects and revealed a broad scope of human interventions in both. The situation would be essentially the same (but different in some implementation details) with any other generative method. For example, if we opted for a long short-term memory (LSTM) neural network that is capable of autonomously generating text by learning from a given training set, we will still need to set its architecture, its parameters, and the training set manually. Moreover, the total of 1467 lines and 300 code blocks would be insufficient for the LSTM to converge, so the efforts necessary for preparing and curating the base corpus would be even more prominent.

Finally, generative live coding is inherently indebted to well-known AI techniques, but it can also serve a vessel for problematizing the various dynamics between humans and algorithms in social, artistic, and labor contexts. To emphasize this metaphor in Anastatica, the performance is enhanced with the participative element that extends the original duopoly into democracy and anarchy. The audience is given a chance to manipulate the computer-generated code, with the choice between augmentation or erosion left completely to each individual. However, as with commercial AI, the freedom of choice is an illusion, as the possible choices are formed by the system itself.

5 Conclusion

Generative systems are built on assumptions expressed qualitatively or quantitatively. Usually, when it comes to generative AI, the assumptions are statistical in nature. By making a step backwards from this approach, we opted for a different assumption for the generative system described in this paper. The idea that textual differences between code blocks should influence the transitioning probabilities became the base of the generative process. The development of the custom preprocessing algorithm with its iterative improvements and fine tuning resulted with a convergence of the results towards initial artistic requirements related to aesthetic and conceptual characteristics.

Unveiling different concerns and consequent design decisions, the aim was to transparently showcase a creative practice that results with a versatile autonomous system, but is less elegant itself. The path was never straightforward, as the scope of our iterative interventions expanded both on the material and the algorithm. Even though the focus of those interventions are shifted in comparison to traditional live coding and that they will be shifter further by more advanced forms of AI and other future technologies, the role of the human creator is still prominent and necessary.

References

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, *51*(4), 1-37.

Blackwell, A., McLean, A., Noble, J., & Rohrhuber, J. (2014). Collaboration and learning through live coding (Dagstuhl Seminar 13382). *Dagstuhl Reports*, *3* (*9*). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Balzer, R. (1985). A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, (11), 1257-1268.

Brooker, R. A. (1958). The autocode programs developed for the Manchester University computers. *The Computer Journal*, *1*(1), 15-21.

Brooks, F. P., Hopkins, A. L., Neumann, P. G., & Wright, W. V. (1957). An experiment in musical composition. *IRE Transactions on Electronic Computers*, (3), 175-182.

Cointe, P. (2004, September). Towards generative programming. In *International Workshop on Unconventional Programming Paradigms* (pp. 315-325). Springer, Berlin, Heidelberg.

Collins, N., McLean, A., Rohrhuber, J., & Ward, A. (2003). Live coding in laptop performance. *Organised sound*, *8*(3), 321-330.

Collins, N., & McLean, A. (2014, June). Algorave: Live performance of algorithmic electronic dance music. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 355-358.

Hiller Jr, L. A., & Isaacson, L. M. (1957, October). Musical composition with a high speed digital computer. In *Audio Engineering Society Convention 9*. Audio Engineering Society.

Kreković, G., & Pošćić, A. (2019). Modalities of Improvisation in Live Coding. In *XCoAx 2019: Proceedings of the Seventh Conference on Computation, Communication, Aesthetics and X.* (p. 199).

Le, T. H., Chen, H., & Babar, M. A. (2020). Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Computing Surveys (CSUR)*, *53*(3), 1-38.

Magnusson, T. (2013). The Threnoscope. *Proceedings of the 2013 International Conference on Software Engineering*.
Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., & Nakamura, S. (2015, November). Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 574-584.

Pachet, F., & Roy, P. (2011). Markov constraints: steerable generation of Markov sequences. *Constraints*, *16*(2), 148-172.

Pinkerton, R. C. (1956). Information theory and melody. *Scientific American*, *194*(2), 77-87.

Pošćić, A., & Kreković, G. (2020a). Unboxing the Machine: Artificial Agents in Music. In *XCoAx 2020: Proceedings of the Eight Conference on Computation, Communication, Aesthetics and X.* (pp. 285-298).

Pošćić, A., & Kreković, G. (2020b). On the Human Role in Generative Art: A Case Study of AI-driven Live Coding. Journal of Science and Technology of the Arts, 12(3), 45-62.

Sahay, A., Indamutsa, A., Di Ruscio, D., & Pierantonio, A. (2020, August). Supporting the understanding and comparison of low-code development platforms. *Proceedings of 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 71-178.

TidalCycles Userbase (2021, March 15).
https://tidalcycles.org/index.php/Tutorial

Tiwang, R., Oladunni, T., & Xu, W. (2019, April). A Deep Learning Model for Source Code Generation. In *2019 SoutheastCon* (pp. 1-7). IEEE.

Zukowski, Z., & Carr, C. J. (2018). Generating black metal and math rock: Beyond bach, beethoven, and beatles. *arXiv preprint arXiv:1811.06639*.