xCoAx 2021 9th Conference on
Computation, Communication, Aesthetics & X

2021.xCoAx.org

Autonomous Creation of Musical Pattern from Types and Models in Live Coding

Keywords: Live Coding, Musical Artificial Intelligence, Computational Creativity, Algorithmic Pattern, Human-Computer Interaction

Elizabeth Wilson
elizabeth.wilson@qmul.ac.uk
Queen Mary University, London, UK

Shawn Lawson
shawn.a.lawson@asu.edu
Arizona State University, Tempe, USA

Alex McLean
alex@slab.org
Research Institute for the History of Science and Technology, Deutsches Museum, Munich, DE

Jeremy Stewart
jeremy.ste@gmail.com
Rensselaer Polytechnic Institute, Troy, NY, USA

In this paper we describe the implementation of an autonomous agent for live coding—the practice of creating art in real-time by writing computer code. The TidalCycles language (an extension of the strongly typed functional programming language Haskell) is used for the generation of new musical patterns. This is integrated as part of a system which allows automatic suggestion of the agent's patterns to a live coder. We aim for this to be a co-creative system, using machine agents to explore not-yet conceptualised code sequences and support coders in asking new questions.

1. Introduction

This paper investigates the autonomous generation of musical pattern through the practice of *live coding*—a term used to refer to performers creating art by writing computer code, usually in front of an audience (Collins et al. 2003). In live coding, computer language is the primary medium for notation and describing the rules with which to synthesise artworks, in this case we consider the case where the output is musical pattern. The practice of live coding places a strong focus on liveness, embracing error, indeterminism and clear mappings between syntax and output. It constructs a paradigm for musical interaction and forms the basis with which we explore creating an autonomous agent.

1.1. Musical Pattern in TidalCycles

For this system of autonomously generated live coded music, the TidalCycles language is used (commonly denoted as *Tidal*). Tidal is a real-time, domain-specific language (embedded in the strongly typed functional programming language Haskell) used for pattern construction. The Tidal language itself does not produce any audio, rather it produces patterns of Open Sound Control (OSC) messages. These are most commonly sent to a sampler and synthesizer engine in the SuperCollider software, which handles the audio synthesis and rendering. However, it is also applicable to other types of pattern, and has indeed been used to pattern live choreographic scores (Sicchio 2014), woven textiles (McLean 2018), lighting, and VJing. Here is a trivial example of a pattern in the TidalCycles language:

```
d1 $ fast 2 $ sound “bd sn”
```

In the above, ‘`d1`’ stands for a connection between Haskell and SuperCollider. The ‘`sound`’ function specifies outgoing OSC messages and the double quotes denote a pattern of samples to be played using Tidal’s ‘mini-notation’, in this case a bass drum followed by a snare drum sample, played in a loop or *cycle*. The Tidal-specific function ‘`fast`’ speeds up the pattern by the given factor, in this case the pattern would be played two times per cycle. The dollar operator is inherited from Haskell, giving function application with low precedence, therefore passing the ‘sound’ pattern as the second argument of ‘`fast`’. The live-coder evaluates this in their text editor of choice, causing the pattern they have constructed to begin to play, which it does until they decide to edit and re-evaluate the pattern, causing a change in the music on-the-fly.

Tidal was chosen as the target for our creative agent for two main reasons (other than the authors' familiarity). Firstly, although known as a popular live coding environment for human musicians, it originally stemmed from a representation for machine learning and generation, through a project modelling rhythmic continuations based on Kurt Schwitters' sound poem "*Ursonate*" (McLean 2007), and so is designed to be straightforward to parse and manipulate for computers as well as humans. Secondly, in typical models of musical generation, representation of musical structure is usually limited, due to the tendency of these models to use low-level symbolic representations, most often in MIDI format. Although MIDI allows a certain level of expressive completeness in its representation, many generation algorithms reduce these to impoverished note representations of pitch number and velocity, thus losing nuance around timbre, expression and structure. We want a richer representation of music in the generation process for greater depth of musical expression and more learning opportunities for our machine counterparts.

An extensive framework for the description and evaluation of music representation systems suitable for implementation on computers is provided in (Wiggins et al. 1993). Coding languages are a particularly strong way to do this, due to their relation to natural language. Although natural language and programming languages are ontologically distinct, programming languages provide a means for human expression due to the way that syntax can be used to convey musical meaning. Coding presents the musician with the ideal set of tools and performance context for algorithms to be written in the form of instructions (Magnusson 2011).

1.2. Motivation

Many musical generation systems posited as artificially intelligent are often trained on corpuses of musical pieces in which the outcome has already been predetermined. The training corpus provided is usually a set of works by a composer or composers of a certain era or musical genre. Music where some of the elements of the composition are left to chance is often known as *aleatoric*, *stochastic* or *indeterminate music*. Examples include John Cage's *Music of Changes* to determine music structural elements by chance, using methods derived from the I Ching, or procedures of graphic notation scores, used in the works of André Boucourechliev and Sylvano Bussotti, in which drawings, images, or other non-musical symbols are used to suggest musical ideas to the performers (Brown 1986). Indeterministic music is an under-explored area in the field of musical artificial intelligence, mostly due to the inherent challenges

posed by training on a corpus that is not fixed. Live coding provides a conceptual framework for this work to exist, as randomness is often encoded inherently in its structure. This is true of algorithmic music in general, although live coding adds an additional level of indeterminacy, as the notation is designed to be changed while it is followed.

Another of the main motivations in creating this autonomous agent is to provide a way of generating musical ideas that have not previously been conceptualised by human live coders. Perhaps this can be used as a way to combat forms of what Wiggins (2006b) describes as 'uninspiration' by traversing across (and beyond) a search space for novel ideas. Here Wiggins builds on the pseudo-formal definition of creativity provided by Boden as "the generation of novel and valuable ideas" (Boden 2004, 3). We can see how a co-creative system might arise under this definition of creativity, where the machine agent can generate *novel* patterns whilst the human live coder can determine the *value* of these novel ideas. Starting from this point it is clear how to form an interaction loop, where the live-coder generates patterns and a machine agent can also develop a sense of value for these.

Perhaps surprisingly, as practising live coding musicians ourselves, we find that listening to code can be a more important part of live coding than writing it in the first place. In other words if code is a map and music is the territory, then the code can only be read and understood when you listen to the territory that it generates. This is true also of the person who is writing the code, who has the experience of editing the code, hearing the result, and only then being able to fully understand their edit and decide what to change next in response. By writing and editing code, the live coder may be making imperative statements (stating how they want music to be made) or declarative statements (stating what music they want to be made) but they are doing so in order to ask questions - which aren't about *how* or *what*, but *what if*? From this perspective, our project aims to support live coders in asking new questions.

Beyond the practical implementation of such a new interface, we hope this system can augment our understanding of how humans and machines can improvise together. The abstraction of human creativity into computer systems is useful for developing an understanding of how co-creativity with a machine musician aids in the development of methodologies for human-machine improvisation strategies (Wilson et al. 2020). The motivation for creation of this system is to understand more about co-creativity rather than solely machine creativity using search-based techniques and looking at knowledge-based systems exploring conceptual spaces.

Whilst individualistic self-expression is essential to any composer, it should be acknowledged that composition itself does not occur in a vacuum, but rather emerges from community traditions, practices and styles. Likewise, the ethos of live coding is built around community and knowledge-sharing and in turn, the music a live coder makes is interdependent on the communities they exist in. Integrating an autonomous agent serves as an expansion of creative and collaborative practice and thus is hoped to better the live coding community as a result.

Moving further into the territory of autonomous generation of musical pattern, it is important to take stock of the ethical implications of such a system. Music generation systems face ethical minefields around issues of authorship, licensing, data-privacy and inherited societal biases reflected in the music produced. We want to acknowledge that these are potential issues for our system but defer addressing these questions currently, focusing instead on discussion of how our system was developed for its aim of creating musical patterns with code.

2. Background

2.1. Autonomous Agents in Live Coding

Autonomous generation of music has been well explored in recent computer music history, spanning from the first attempt at generating scores, often cited as Hiller's Illiac Suite (Hiller 1957) for the Illiac I computer, through to the deep learning works of Google's Magenta project (Huang et al. 2018) and entries to the AI song contest (Huang et al. 2020). Of particular interest is the work of George Lewis in creating Voyager (Lewis 2000) — an interactive improvisational system with a machine counterpart. Lewis's work was particularly influential as it acknowledged music was more than just data about note relationships but rather music was a product of community and he attempted to encode these aesthetic values into his work. Lewis's work was also particularly relevant as it saw automation as an opportunity for augmentation of the creative process and these ideas align strongly with our motivations.

Given the prolific climate for artificial intelligence and live coding's grounding in human-computer interaction, it is unsurprising that the challenge of co-creation with machine musicians has already been attempted. Co-creation collaborative configurations (human-machine, machine-machine) in various contexts are explored in (Xambo 2017), identifying potential synergies and novel insights of co-creativity applied to collaborative music live coding. Notable examples that generate Tidal code include an autonomous performer, Cibo,

which tokenises Tidal code and uses a sequence-to-sequence neural network trained on a corpus of Tidal performances to generate novel patterns (Stewart and Lawson 2019) or using a defined-grammar and evolutionary algorithms to evolve patterns, using the collaborative live-coding platform Extramuros (Hickinbotham and Stepney 2016).

2.2. Creative Systems

To contextualise this work, we look to some definitions of creative systems. Margaret Boden, a prominent figure in the philosophy of computationally creative systems, defines the notion of a "conceptual space" as a set of artefacts which, in some quasi-syntactic sense, are deemed to be acceptable examples of whatever is being created. From Boden's definition of creativity (2004) arises the ideas of exploratory creativity (the process of exploring a given conceptual space) and transformational creativity (the process of changing the rules which delimit the conceptual space). A formalism of creative systems, the Creative Systems Framework, provided by Wiggins (2006a), defines an exploratory creative system (such as the one here presented) in mathematical representation. This formalism can also be expanded to transformationally creative systems, by considering transformational creativity as exploratory creative on the meta-level. Considering creativity as a search through conceptual space there are clear similarities between this and traditional AI search techniques (Wiggins 2006b). Particularly the notion of a state space (i.e. the space of partial or complete solutions to a particular algorithm) is closely related to Boden's idea of a conceptual space. Many strategies used by humans in creative practice closely resemble algorithms too, artists often use generate and test strategies (Buchanan 2000).

The creative system framework has been applied to create conceptual spaces for possible creative agents in Tidal in (Wiggins and Forth 2018) and a discussion is offered on where creative responsibility in live coding can be shared with a computer. When sharing creative responsibility with a machine agent in Tidal, Wiggins and Forth advocate for three key components. The first is the ability of a computer to relate the meaning of a program to its syntax. Secondly, the computer should have some model for the coder's aesthetic preferences. Finally, the system should have the ability to manipulate the available constructs to take some creative responsibility for the music. This work focuses mainly on the latter aspect of this proposition.

3. Creative Search

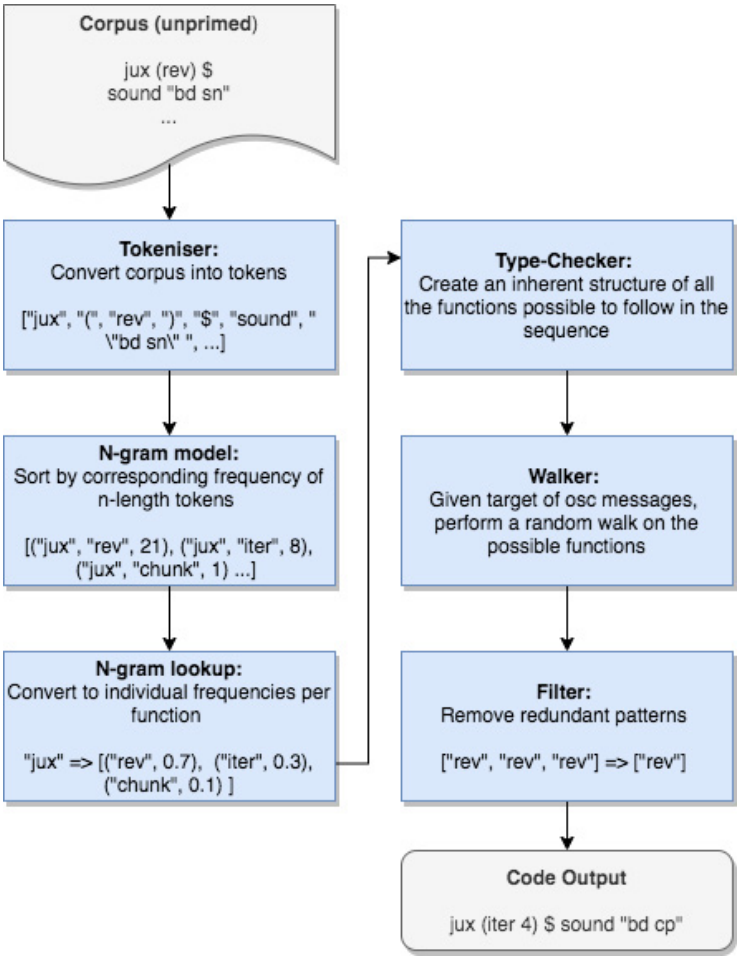

Fig. 1. Flow of the different components of the algorithm needed to produce code sequences, from generating the model from the data to creating syntactically correct code by type-checking.

The search strategy for generating the Tidal agent's outcomes combined a random walk process with Haskell's type system. The possible states for the walker are the various type signatures of functions. The aim was to create a walker agent that could navigate through the conceptual space of all possible syntactically valid Tidal code. Weightings for this walk process were supplied by an n-gram model: a contiguous sequence of n-functions generated from a corpus of existing Tidal patterns. From this model, potentially infinite strings

of code can be generated, providing the search space for the creative agent. However, derived rules and constraints are necessary components of the model to produce useful, executable code. The overall flow from the tokenisation of the corpus, through to creative search and generating code can be seen in Figure 1.

3.1. The Tidal 'Universe'- the Types

Being embedded in the Haskell language means TidalCycles inherits Haskell's system of static, pure types. The type of every expression is known at compile time, leading to safer code. Haskell has type inference which means types don't have to be explicitly specified where they can be inferred by the compiler. Nonetheless in Haskell all functions and other values have an underlying "type signature", defining the types of pure inputs and outputs. Tidal's representation of musical pattern applies the principles of Functional Reactive Programming (FRP), so rather than representing music as data structures, it instead represents it as behaviour—as functions of time.

Tidal's representation of pattern in the Haskell type system has profound impacts on the functionality of the language. For example, instead of representing a sequence as a list of events, it represents sequences as a function, taking a timespan (or rather, a time arc, as time is treated as cyclic) as input, and then returning all the active events during that timespan (McLean 2020). These types are defined as instances of standard Haskell type-classes, including functors, applicative functors and monads. As a result, Tidal patterns are highly composable, at different levels of abstraction. By composable, we first mean in the computer scientific sense — that as functions, patterns can be flexibly composed together into new patterns, but also in the musical sense, in that complex musical behaviour can be composed together from simple parts.

This type system forms the basis with which our autonomous agent can construct new patterns of code. The implementation of this occurs as follows. We start with a dictionary of available functions/values in Tidal, each with a representation of its type signature and the number of times it should occur within any Tidal pattern. An initial target is specified as a pattern of synthesiser control messages as this is the standard output from any Tidal pattern. The walker starts by randomly choosing any function that could produce a pattern of the type of the target. Based on the system's implicit weightings and rules, which are outlined in the following sections, the algorithm recurses through the possible functions and chooses a function that can fit with the next pattern in the sequence, where the possible permutations of functions that fit together are

also explicitly defined. Figure 2 shows the implementation of this. Finally, the recursion ends once the target sequence has been fulfilled. The code generated in the instance of Figure 2 would be:

```
d1 $ jux (rev) $ sound "bd sn"
```

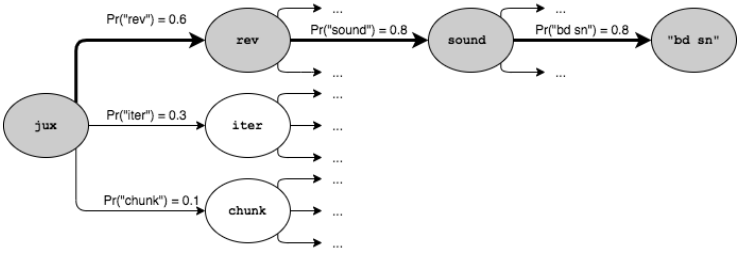

Fig. 2. A directed acyclic graph to illustrate the construction of the simple pattern - `jux (rev) $ sound "bd sn"` - where the highlighted nodes are the functions chosen by the algorithm at each recursive step. The arrows represent the possible transition probabilities to any other possible states.

In this example, the target is 'Pattern Control Map' (a pattern of synthesiser control messages) and the walker randomly chooses the function `'jux'` as a starting point. This is a function with two arguments, the first argument is a function to apply to the pattern given as its second argument, but it does so only in the right-hand audio channel, giving a stereo juxtaposition where one channel is transformed and the other is not. The result of `'jux'` is a new pattern, but in order to arrive at this we must provide the functions inputs. The walker therefore recurses, calling itself with the type of each argument. This recursion allows for one of the arguments to itself be a function that requires further arguments, although that is not the case with this simple example. Note that while `'rev'` is a function, in this case it is treated as a value to pass to `'jux'`; in other words, 'jux' is a higher-order function. The walker continues to recurse and produces a sequence, until it meets the target type signature, where the process terminates.

3.2. Reducing the Search Space

The walker can generate code that is syntactically correct and therefore executable. However, the demands of live coding as a musical practice mean code should be kept concise enough to create a pattern that is both able to be processed by the audio engine without excessive latency, and also with brevity required for both the musician and audience to have some understanding of its relation to the musical output. It was therefore necessary to reduce the options in the search space to those which resembled code a human live coder might produce, although arguably, machine generated code does not have to directly resemble a human's output: human coders have learnt coding behaviours (style,

function choice, sample choice) whereas machine generated code is stylistically agnostic and this agnosticism could prove beneficial for creative ideation.

The generation method has similarities with evolutionary computing's ideas of search and optimisation, and accordingly techniques were borrowed from this field, particularly for reduction of the code. The first search reduction technique incorporated into the algorithm was *bloat*, i.e. where there was an increase in mean program size without improvements in fitness and where the output generated grows excessively due to redundant operations (Luke et al. 2006). For this pattern generation algorithm, function selection was limited to those functions that had not been seen previously. In practice, this reduces bloat by ensuring two functions that have the same action can not be applied in succession, preventing excessive growth. For example, in the TidalCycles language, the 'rev' function will play a pattern in its reverse order, however applying this twice is equivalent to not applying at all and thus adds bloat to the pattern generation.

Further pruning was applied to the search, similar to those seen in search-based algorithms (Garcia et al. 2006). Another of the goals in pruning was removing idempotent functions, i.e. with set E and function composition operator °, idempotent elements are the functions $f : E \rightarrow E$ such that $f \circ f = f$, in other words such that $\forall x \in E, f(f(x)) = f(x)$. This was removed by the algorithm in the case where the function '`every 1`' was applied to another function. This is analogous to the function itself being applied and was thus removed.

3.3. Navigating the Search Space

To navigate a creative search space it was important to be able to steer this walk. To achieve this, weights were applied to all of the possible functions that could occur next in the sequence, corresponding to their respective transition probabilities. A corpus of code patterns created from TidalCycles users was provided as the source for the weightings. The code patterns were used as the source for an algorithm written to tokenise the functions and convert them into an n-gram model. This provided a data structure which, when picking any function at random, can provide the next function to be picked based on its weighted likelihood.

These weights were not static and the weights would redistribute throughout the pattern generation process to ensure excessive and impractical (or potentially infinite) code was not generated. The possible values for the next function in the sequence were chosen using a squared reciprocal factor as the *arity* (how many

arguments the function takes) and *depth* (how many functions have already been chosen prior) parameters increased. A user-controlled environment variable was included in this reweighting factor modifying the overall decay rate of these weights, allowing control over the rate at which the weights decayed to zero and thus the length of sequence generated.

Finally, the walker finishes navigating once it arrives at an expression that meets the target type signature. There are fairly rare cases where it reaches a dead-end - applying functions to functions until it finds a type signature which is not possible to meet with the functions and values available to the walker. Currently if this happens the walker gives up, although in the future we intend to investigate a back-tracking procedure.

3.4. Evaluating Patterns

The current state of the algorithm has no particular faculties for evaluation of its own output, other than the listener's perception. Evaluation is a crucial part of any system that claims computational creativity, yet this is often done with the researcher's subjective claims of creative behaviour (Agres et al. 2016). Human evaluation may still be the best way to judge whether a produced piece sounds aesthetically pleasing, however there are drawbacks to this method. Requiring human participants to rate the algorithm's output over multiple iterations of generated code could take an excessive amount of time for the listener. Furthermore participant fatigue is a commonly noted, yet often ignored, problem with listening tests (Schatz 2012) affecting the reliability of the results.

Additionally, human evaluation might never reach an empirical consensus on what is aesthetically pleasing due to the vast differences in listener preferences. If our goal is simulating some form of artificially intelligent musical behaviours, then the capability of a system to reflect on its productions should be an important functionality of a computationally creative system.

4. Challenges in Code Generation

One of the challenges in creating our autonomous agent was steering throughout the space in a logical way, similar to how a live-coder might produce a coherent musical pattern in Tidal. The first iterations of code generated by the algorithm were often found going down unsolvable or infinite paths. We found weightings from the n-gram model worked well to keep the code produced reasonable, by contextualising code generation in what is likely to happen.

Another of the main problems encountered with this system was with the *mini-notation* within the Tidal language. This is a terse way to represent events within a pattern in Tidal syntax. The mini-notation in TidalCycles is based on the Bol Processor (Bel 2001). Within the mini-notation, polyrhythms, polymeter, repetition, and rhythmic subgrouping can all be described as part of the string passed to the sound function. These additional complexities of notation were omitted in this early version, where mini-notation strings are treated as single tokens. Future work will incorporate generation of mini-notation strings into the process.

5. Outcomes

The overall system as it can be used in performance is seen in Figure 3. This includes building a custom Haskell listener module to request a pattern when a command from the Atom Editor is executed. The listener module, on receiving an OSC message from the Atom editor, then requests a pattern from the walker module. This is then sent back to the listener, parsed into JSON format and then sent to the Atom editor, where the pattern is displayed as a suggestion to the user. This can then be evaluated by the human live coder, which sends the pattern to the SuperCollider sound synthesis engine, rendering the pattern into the acoustic domain for the live coder to listen to, evaluate and then continue to edit their performance.

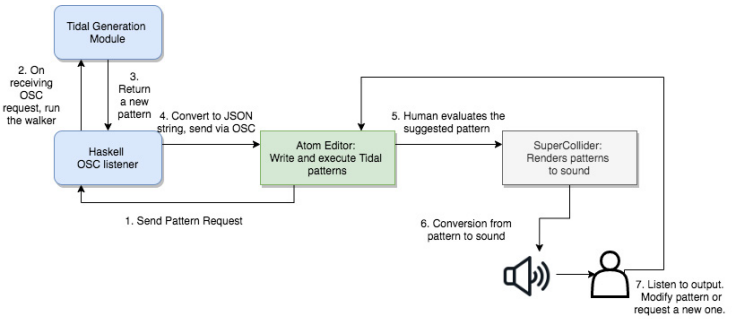

Fig. 3. The overall structure of the system as used in performance.

5.1. Auto-suggestion of Patterns in the Atom Editor

An autocomplete package was written for the Atom text editor software, where Tidal code is usually evaluated, so that the agent could suggest code patterns in real-time. The autocomplete package requests code sequence options from the autonomous agent. These options are returned, displayed to the user in a typical format of a dropdown menu of options to select. Three potential use-cases were thought of during the development using the code generation as a suggestion engine: education, collaborator, and auto-performer.

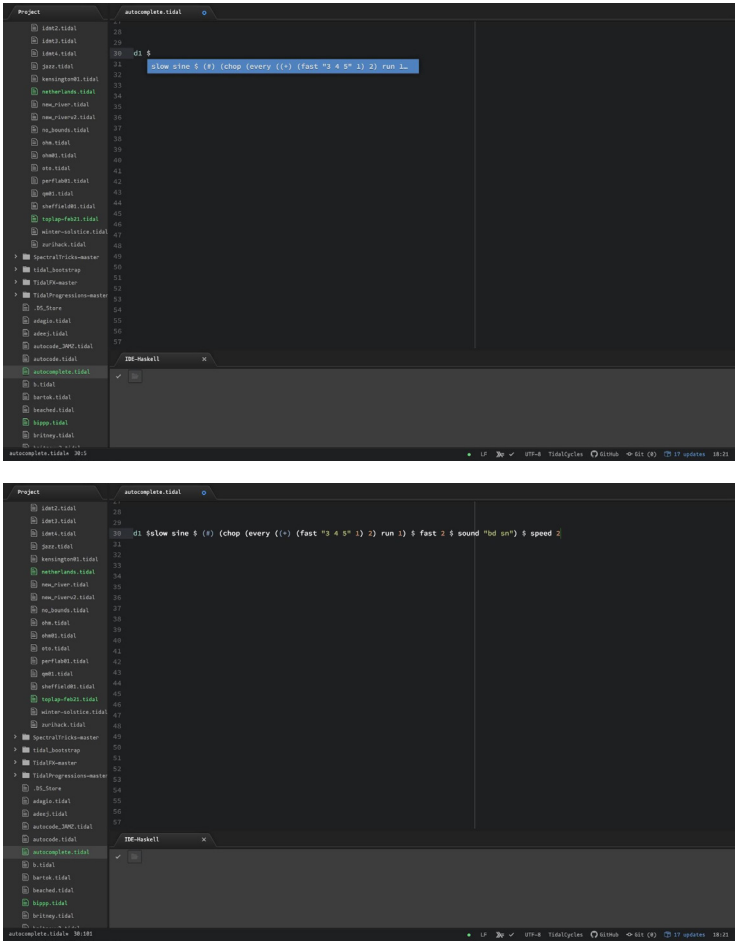

Fig. 4. The Atom Text Editor Package auto-suggesting a syntactically valid code sequence, which would produce a musical pattern when evaluated.

In the education context, the autonomous agent can be called upon to supply someone learning TidalCycles with possible code sequences. The student would be able to see and then select valid code sequences. This helps reduce the learning curve by having small examples at-the-ready, overcoming blank-slate situations of not knowing what to type, and to reinforce learning by doing. The live-coder is allowed control over the length of sequences generated, allowing those using in the learning context to generate manageable pattern suggestions. These suggestions can deepen their knowledge of how the language is used.

For the experienced TidalCycles user, the autonomous agent could be a creative collaborator. In the process of creating new live-coded audio, a user may look to the autonomous agent for inspiration or something completely random that the user may not have thought of. In a sense, consider this as a creative bump or push. Additionally, as a thought experiment, the user may employ the autonomous agent as an antagonistic collaborator. This antagonistic collaborator may insert short sequences of code that the user must work with or work around - not unlike setting up a Surrealist's game.

In the last example of the auto-performer, additional system structure can be added to facilitate the autonomous agent to become a performer for either solo or ensemble member contexts. While similar in outcome to other machine agent performers, the methods of code sequence generation are unique and may lead to new sound producing practices and ideas within live performance.

5.2. Pattern Production

Although this system is still in the early stages of development, its results have been promising with its capabilities to produce syntactically correct and executable code. Sequences where weighting was decaying faster produced results like that below, similar to the trivial short phrases a live-coder might write.

```
fast 2 $ jux (rev) $ sound “bd sn”
```

With an optimal decay rate, patterns that produced some interesting (and sometimes unpredictable) behaviours were produced:

```
(#) ((#) (chop ((+) 1 “3 4 5”) $ vowel $ fast (rev 1) “a e i o u”) $ speed 2) $ slow (rev $ fast “3 4 5” 1) $(#) (shape $ rev $ fast 2 1) $ slow 2 $ slow 2 $ sound $ every (rev 1) rev “bd sn”
```

```
chop (fast “3 4 5” $ (+) (fast 2 $ rev 1) 2) $ slow “3 4 5” $ fast “3 4 5” $ (#) (sound “bd sn”) $ shape 1
```

```
(#) ((#) (rev $ fast 1 $ speed 2) $ shape $ every  (rev $ run $ every (slow 2 1) (every 2 rev) 1) rev 2) $ sound “bd sn”
```

The former creates a combination between Tidal’s granular synthesis function ‘`chop`’ and vowel formant resonances effects ‘`vowel`’, producing a staccato vocaloid pattern. The second pattern takes the same chop function, but alternates the size with which to chop the samples up by, creating a delay effect. The latter creates a pattern out of the shape effect (a form of amplification) on the bass drum and snare drum sample. It does this to such an extent that it produces a pitched sound, by pushing the clipping towards a square wave.

Some more examples of sonic patterns produced by the autonomous live-coding agent can be found at https://soundcloud.com/tidal-bot-main.

Live performances using patterns from the autonomous coding agent can be found at https://www.youtube.com/channel/UCEkXT_natfoK8Kwy3z5hLRw

6. Future Work

The models for generating patterns could be extended in future work as follows. Firstly, although the system has the capability to generate and make use of n-grams, these were restricted to bigrams as a proof-of-concept. When generating the transition probabilities to the next possible functions, the agent only has contextual awareness of one function ahead in the sequence. Whilst this works to some extent, given that some functions in Tidal often are composed of two or more parameters, the agent is not aware of wider context. Extension to tri-grams or n-gram models for any $n \epsilon N^{\star}$ is possible and should be included in future implementations.

Currently the user interaction with the system is limited to only controlling the length of sequences produced. Future versions of this project could look further into the role of the human in such a system, looking for any particular tradeoffs that might occur between how much control is given to the human over the computer. In the Conductive system (Bell 2013) (a Haskell live coding environment), "players" are introduced as a means with which the human can interact, where the programmer is in control of semi-autonomous agents. The autocomplete package is designed to act as a similar mechanism to the players in the Conductive system, where the live coder can retain control over the agents suggestions and ultimately view this as a creative partnership.

For true two-way creative partnership to occur, additional evaluation of the success of the system is required. Moreover, for it to be considered as a computationally creative system some capacity for self-reflection is needed, arguably this a necessary facet of a truly creative system (Agres et al. 2016). As it stands we are not making any assertions on its capacity for creativity, rather are investigating how it can be used as part of a creative process.

Finally, the *live* in live coding suggests that future iterations of this work would benefit from live performance and evaluation of these performances. This would allow the audience to participate in the evaluation by offering feedback and reflections on both the system's outputs and the creative processes that produce them.

7. Conclusion

The work here presents an autonomous agent for live coding in the TidalCycles language. This agent can produce syntactically correct code by piecing together functions from a random walk process. Using Haskell meant that the type system could easily be leveraged to ensure sequences fit together in a way that would produce syntactically correct code. The agent is provided as a proof-of-concept and we have discussed various future extensions to this work. Most importantly, we emphasise the role of automation as an opportunity for augmentation of the creative process.

Acknowledgements. McLean's contribution to this work is as part of the PENELOPE project, with funding from the European Research Council (ERC) under the Horizon 2020 research and innovation programme of the European Union, grant agreement no. 682711. Wilson's contributions were supported by EPSRC and AHRC under the EP/L01632X/1 (Centre for Doctoral Training in Media and Arts Technology) grant and the Summer of Haskell programme, part of the Google Summer of Code.

References

Agres, Kat, Jamie Forth, and Geraint A. Wiggins. 2016. "Evaluation of musical creativity and musical metacreation systems". In *Computers in Entertainment* (CIE) 14(3), 1–33.

Bel, Bernard. 2001. *Rationalizing musical time: syntactic and symbolic-numeric approaches*

Bell, Renick. 2013. "An approach to live algorithmic composition using conductive". In: *Proceedings of LAC*. vol. 2013

Boden, Margaret A. 2004. *The creative mind: Myths and mechanisms*. London: Weidenfeld and Nicolson.

Brown, Earle. 1986. "The notation and performance of new music." *The Musical Quarterly 72*, no. 2 (1986): 180-201.

Buchanan, Bruce G. 2000. "Creativity at the Meta-level," *AI Magazine*. Reprint of keynote lecture to AAAI-2000.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live coding in laptop performance". In: *Organised sound 8*(3), 321–330.

Garcia-Almanza, Alma Lilia, and Edward PK, Tsang. 2006. "Simplifying decision trees learned by genetic programming". In: *2006 IEEE International Conference on Evolutionary Computation*. pp. 2142–2148.

Hickinbotham, Simon, Susan Stepney. 2016. "Augmenting live coding with evolved patterns". In: *International Conference on Computational Intelligence in Music, Sound, Art and Design*. pp. 31–46. Springer.

Hiller, Lejaren, Jack McKenzie, and Helen Hamm. 1957. *Illiac suite*.

Huang, Cheng-Zhi Anna, Hendrik Vincent Koops, Ed Newton-Rex, Monica Dinculescu, and Carrie J. Cai. 2020. "AI Song Contest: Human-AI Co-Creation in Songwriting." *arXiv preprint arXiv*:2010.05388.

Huang, Cheng-Zhi Anna, Ashish Vaswani, Jakob Uszkoreit, Ian Simon, Curtis Hawthorne, Noam Shazeer, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. 2018. "Music transformer: Generating music with long-term structure". In *International Conference on Learning Representations*.

Lewis, George E. 2000. "Too many notes: Computers, complexity and culture in voyager". In: *Leonardo Music Journal*. pp.33–39.

Luke, Sean, and Liviu Panait.
2006. “A comparison of bloat control methods for genetic programming”. In: *Evolutionary Computation* 14, no.3: 309–344.

Magnusson, Thor.
2011. “Algorithms as scores: Coding live music.” *Leonardo Music Journal* (2011): 19-23.

McLean, Alex, and E., Harlizius-Kluck.
2018. “Fabricating algorithmic art”. In *Austrian Cultural Forum*.

McLean, Alex.
2007. “Improvising with synthesised vocables, with analysis towards computational creativity.” PhD diss., Master’s thesis, Goldsmiths College, University of London.

McLean, Alex.
2020. “Algorithmic pattern”. In *NIME*.

Schatz, Raimund, Sebastian Egger, and Kathrin Masuch.
2012. “The impact of test duration on user fatigue and reliability of subjective quality ratings.” *Journal of the Audio Engineering Society 60*, no. 1/2 : 63-73.

Sicchio, Kate.
2014.“ Data management part iii: An artistic framework for understanding technology without technology”. In: Media-N: *Journal of the New Media Caucus 3*(10).

Stewart, Jeremy, Shawn Lawson.
2019. “Cibo: An autonomous tidalcycles performer”. In: *International Conference on Live Coding*.

Wiggins, Geraint, Eduardo Miranda, Alan Smaill, and Mitch Harris.
1993. “A framework for the evaluation of music representation systems”. In: *Computer Music Journal 17*, no.3. pp 31–42.

Wiggins, Geraint A.
2006a. “A preliminary framework for description, analysis and comparison of creative systems”. In: *Knowledge-Based Systems 19*, no.7. pp 449–458.

Wiggins, Geraint A.
2006b. “Searching for computational creativity”. *New Generation Computing 24*, no.3 (2006): 209–222.

Wiggins, Geraint A, and Jamie, Forth.
2018. “Computational Creativity and Live Algorithms” In: Alex McLean and Roger Dean, eds. *The Oxford Handbook of Algorithmic Music*. Oxford, UK: Oxford University Press.

Wilson, Elizabeth, György Fazekas, Geraint Wiggins.
2020. “Collaborative human and machine creative interaction driven through affective response in live coding systems.” In: *International Conference on Live Interfaces* (2020)

Xambó, Anna, Gerard Roma, Pratik Shah, Jason Freeman, and Brian Magerko.
2017. “Computational Challenges of Co-Creation in Collaborative Music Live Coding: An Outline.” In *Proceedings of the 2017 Co-Creation Workshop at the International Conference on Computational Creativity*.