Live Coding Machine Learning and Listening: A Survey on the Design of Languages and Environments for Live Coding

Chris Kiefer
Experimental Music Technologies Lab
Department of Music
University of Sussex
c.kiefer@sussex.ac.uk

Thor Magnusson
Experimental Music Technologies Lab
Department of Music
University of Sussex
t.magnusson@sussex.ac.uk

ABSTRACT

This paper reports on a preliminary study of a live coding language for machine learning and machine listening. The study represents early work in a three-year research project that will implement live coding interfaces for novices wanting to apply machine learning and machine listening in their work.

1. INTRODUCTION

The MIMIC (Musically Intelligent Machines Interacting Creatively) project explores how the techniques of machine learning and machine listening can be communicated and implemented in simple terms for composers, instrument makers and performers. The potential for machine learning to support musical composition and performance is high, and with novel techniques in machine listening, we see emerging a technology that can shift from being instrumental to conversational and collaborative. By leveraging the internet as a live software ecosystem, the MIMIC project explores how such technology can best reach artists, and live up to its potential to fundamentally change creative practice in the field.

The project involves creating a high-level language that can be used for live coding, creative coding and quick prototyping. Implementing a language that interfaces with technically complex problems such as the design of machine learning neural networks or the temporal and spectral algorithms applied in machine listening is not a simple task, but we can build upon decades of research and practice in programming language design (Ko 2016), and computer music language design in particular, as well as a plethora of inventive new approaches in the design of live coding systems for music (Reina et al. 2019).

Existing systems and languages are often reported on describing clever solutions as well as weaknesses. Researchers are typically reflective and openly critical of their own systems when analysing them. However, they rarely speculate freely and uninhibitedly about possible solutions or alternative paths taken. Before defining the design of our own system, we were therefore interested in opening up a channel where we could learn

from other practitioners in language design, machine learning and machine listening. We created a survey that we sent out to relevant communities of practice - such as live coding, machine learning, machine listening, creative coding, deep learning - and asked open questions about how they might imagine a future system implemented, given the knowledge we have today. Below we report on the questionnaire and its findings.

2. SURVEY DESIGN

In designing the questionnaire we decided to aim for open ended answers, speculative and cutting-edge ideas, rather than implementation details or specific problematisations. We were interested in how people might see an ideal future system that would be easy to learn, yet be powerful and easy to extend. Questions also involved experiences in learning and teaching programming languages and what best practices in language design might be in the context of education. The survey was split into two parts, a short compulsory section with fundamental questions and a further and longer section with optional questions.

In the compulsory section, we wanted to find out why people use live coding systems, and what language and environment features enable them to live code successfully.

[1] What are the high-level properties and features that you look for in a programming language? Why?

[2] If you have live coded or used dynamical programming environments before, how does it benefit you and what do you use it for?

[3] What are the key features of programming languages or environments that enable you to achieve your objectives?

[4] What features of the language or environment help you to express complicated ideas? Please give an example if possible.

We were also interested in how a new system for live coding could aid novices in learning the fundamental concepts and principles of computer programming:

[5] What key advice would you give to novice programmers?

[6] What key features should the language or environment have to help novices?

[7] In your opinion, what is the key hindrance to novices when learning programming?

Beyond these core questions, we wanted to elicit deeper responses around approaches to live coding, language features and paradigms, and approaches to machine learning and listening:

[8] If you were to design a live coding language from scratch, which features would you implement? Can you give an example?

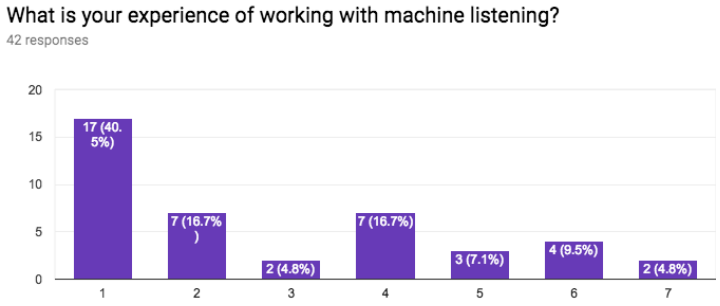
[9] What are the tensions between brevity of code and expression in live coding language design?

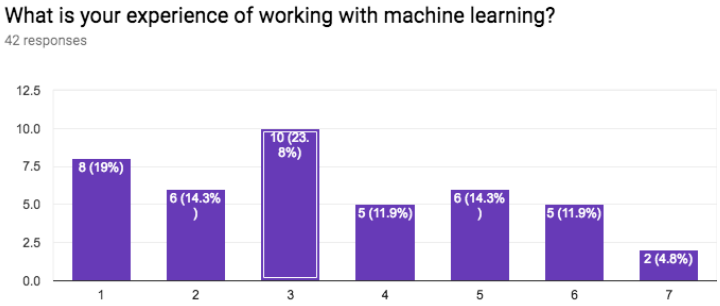
[10] What do you think are good programming paradigms for live coding? (e.g., imperative, functional, OOP, hybrid) Why?

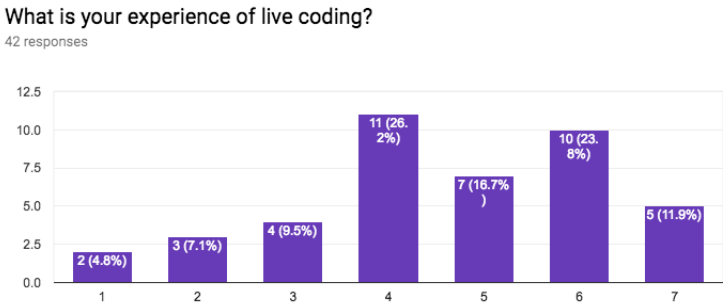
[11] What features would you like to see in a live coding language for machine listening and machine learning?

3. PARTICIPANTS

At the time of writing, 42 participants had completed the survey. The survey continues to be open to future responses. Participants were in age brackets 18-29 (26.2%), 30-49 (40.5%), 40-49 (16.7%), 50-59 (14.3%), 70 and above (2.4%). 39 identified as male, 1 non-binary and 2 preferred not to say. Responses came from a wide range of international locations. Professionally, the respondents were software engineers, musicians, data scientists, academics, students, artists, composers, engineers and business professionals. Participants identified principally with the following roles: machine learning practitioner (39%), machine listening practitioner (24.4%), data scientist (22%), live coder (63.4%), programmer (78%), musician / artist (95.1%), teacher (56.1%). Asked 'which programming languages and environments do you use?', the most commons responses were Python, C, C++, JavaScript, SuperCollider, Pure Data, Max/MSP, Tidal Cycles, Processing, MATLAB and Jupyter Notebook. Participants were asked about their experience in key areas to the survey on a Likert scale from 1 (None) to 7 (expert), responding as follows:







4. FINDINGS

This is the first survey on live coding language design. Live coders exist in a world that is often disconnected, and people evolve their own specific abstractions that reflect the specific way they think and make music (as seen in the responses that highlight the need for customisation and flexibility) or other domains of live coding practice. However, the answers to the survey did demonstrate common threads that emerge from disparate practices, and interestingly show more agreement and convergence around key topics than disagreement.

The survey is ongoing, but these intermediate results were qualitatively analysed by two researchers, with plans for further in-depth coding and analysis at a future date. Summaries of the responses to each question are given below.

What are the high-level properties and features that you look for in a programming language? Why? Brevity, expressiveness, consistency and readability were the terms that most often came up across the participants' replies. People want to talk high-level language with their machines, quickly and it should be in an intuitive language that can be understood by laypeople. The language should support the easy formation of mental models of the program, a fact that signifies the importance of clear metaphors, syntax and semantics. Data structures should be flexible and it should be easy to make abstractions. Such a language needs to support different ways of thinking, and a successful language can be judged *"based on whether the designer has attempted to understand how artists think, and ensure that the language thinks the same way as much as possible."* How "artists think" is clearly a heterogeneous cluster of modes, but the point is solid: the programming language should not force the the artist to think in computer science terms, but rather support alternative ways of thinking, implying that the language should be flexible for diverse methods of domain representations. For example the language should support creation of new language constructs.

If you have live coded or used dynamical programming environments before, how does it benefit you and what do you use it for? It was noticeable how many of the replies suggested that dynamic coding enables a method of sketching or "thinking-out-loud" in code, testing ideas, trying alternative approaches to a problem. "In the moment" engagement with coding was also highlighted as key benefit, not only to enable live musical performance, but also for allowing the development of algorithms to follow an emergent path, without predefined goals. It was clear that for live coding practitioners, live coding is not only a performance method, but rather a general way of conversing with computers and explore emerging ideas. Such emergence is enabled by the ability to test ideas quickly with instant feedback, which might result in unintended but useful outcomes. Live coding has enabled a different method of artistic composition and performance of music compared with traditional music software, such as Digital Audio Workstations, and has also offered useful ways to engage with temporal processes in all kinds of domains (Sorensen & Gardner 2017).

What are the key features of programming languages or environments that enable you to achieve your objectives? The overriding theme in the responses was that coders looked for flexible languages and environments that minimise unnecessary effort and stress for the user. To achieve this, language should be stable, portable and expressive with good libraries and documentation. Participants reported that a flexible dynamic structure would be helpful, mixing programming paradigms, such as OOP and functional programming. Some pointed to the benefit of allowing functions to be first class citizen in the programming language, enabling the user to pass functions around the system, as a

parameter to other functions and objects. If languages in the past brought dogmatic design philosophies (such as OOP or functional programming), the situation today is much more practical: people see the benefits of the diverse paradigms, and would prefer to use a system that enables a hybrid approach.

What features of the language or environment help you to express complicated ideas? Please give an example if possible. Flexibility is seen as a key element in live coding, probably more so than in more traditional programming languages. For this reason, polymorphism (in the form of *ad hoc*, where objects can behave differently depending on the call it gets, or *parametric*, where different set of parameter inputs will result in different operations) is seen as useful: it can be good in a real-time performative situation to be able to copy code but pass different arguments types, change one's mind in the middle of writing a function, or even experience the serendipity of a surprise. This requires, however, that the user understands the structure of the object or function, suggesting that a clear inspection system is built that enables the user to open the black box and study its workings quickly and effectively in the flow of programming. In combination with flexible language features, respondents highlighted the need for expressible and malleable data structures, that are adaptable in emergent live coding processes and across evolving logic. Some mentioned object-oriented approaches were highlighted for code-readability in comparison to functional syntaxes, but others mentioned lambda expressions (or anonymous functions) for the simplicity of writing operations that can be plugged together or passed around the system. Paradigms in functional programming, such as function composition and the avoidance of state, can result in fewer bugs and more streamlined work processes.

What key advice would you give to novice programmers? Many responders gave the advice that deciding upon a simple and clear goal which can be achieved as a small project is an ideal way to learn a programming language, especially if this goal is approached in a simple and incremental manner. There was very clear support for practice-based learning approaches, and also open-ended approaches that involved making mistakes and learning to resolve them through engagement with community support resources. Artists should learn to challenge themselves and go beyond the available resources. To practise programming regularly was seen as necessary in reaching the level of competency required for live coding.

What key features should the language or environment have to help novices? There was a very strong emphasis on good quality documentation and examples that are tightly integrated with the development environment, along with autocompletion. Human-readable error messages are very important for novices to learn new languages. Environments such as Processing and SuperCollider suffer greatly here as the error messages can be quite obscure and it can be hard for the novel user to detect the relevant information in the lengthy dump. Conversely these environment benefit from a large collection of help examples and user communities. Furthermore, participants expressed that ways to visualise complex data structures can be a very useful aid to beginners.

In your opinion, what is the key hindrance to novices when learning programming?

Respondents pointed out pragmatic hindrances, including poor documentation, obtuse terminology and challenging overheads, for example library management and linker configuration. They also highlighted emotional factors that can hold novices back; fear of making mistakes, feelings of intimidation and being overwhelmed, and imposter syndrome. They warned against approaches that involved copying and pasting code without

understanding it fully, and pointed out that understanding of systematic and logical approaches to coding underpins a successful approach. Whilst learning methods that are not practice-based can be a hindrance, the general view was that a hybridity of approaches would be ideal strategy, both as people are different and they might need different approaches depending on where they are in the learning process.

If you were to design a live coding language from scratch, which features would you implement? Can you give an example? Answers clustered around the need for hybridity, representations of time and brevity. Hybrid features, including object-oriented and functional features, give the programmer flexibility to express ideas to match their thought process. Representations of time are not often prioritised in standard languages but are essential for music live coding so a good abstraction is needed here. Concise syntax is needed to allow fast expression of ideas in code. Small, compact, and stable were common replies, but good libraries are seen as important too. There was a tension between building on existing language or implementing something completely new, both strategies having some benefits.

What are the tensions between brevity of code and expression in live coding language design? There was little common agreement in the responses here, probably because this is a challenging problem with multiple constraints to satisfy. Factors included user preference, readability, memorisability, language aesthetics, IDE autocompletion, and consideration of the audience.

What do you think are good programming paradigms for live coding? (e.g., imperative, functional, OOP, hybrid) Why? Participants were almost unanimous in supporting hybrid languages that allow for flexible approaches.

What features would you like to see in a live coding language for machine listening and machine learning?

A common theme linking responses was flexibility, to be able to link processes and data structures across boundaries of format, temporality and category, to be able to plug anything into anything else. Models should be accessible and hackable, and introspective methods (e.g. visualisations) should aid the user in building models. Musicality should be a priority in the design of the language, and there was a suggestion to be able to repeat machine learning transformations to build musical structure, i.e. to instrumentalise the machine learning training API. This is interesting because it proposes to bring typically offline, non real-time processes into real-time performance.

5. DISCUSSION

The survey responses were highly interesting, confirming what we also consider good practice for live coding system design: brevity, simplicity, expressivity, flexibility, adaptability, and plurality. The language should be fast, expressive, high level, and allow for different paradigms of programming, applying OOP principles when that suits and functional programming at other occasions. Some answers hint at challenges ahead: off-line vs. real-time processes, building complex hybrid language features into custom DSLs, or how we design this ‘plug anything into anything’ cross-domain expressive data manipulation system? How do we foreground musicality? (and define it in terms of language design) - what makes a DSL musical? And what about Tanimoto’s levels of liveness (Tanimoto 2013) for example? How will live coding work when intelligent programming languages write code for us, complete tasks, suggest approaches, and so on? How will this affect artistic expression?

The survey replies did not include any paradigm shifting ideas in terms of new language design. There can be many reasons for this, for example that existing paradigms work well, and the combination of them has been suggested, but perhaps also that the survey format is not necessarily the right forum to explain a new vision of language design. However, we have benefitted immensely from the survey and take with us some key findings for the workshops and development sessions that we are planning to further implement our system.

Our survey is largely design and technology focussed, expounding the questions of language design and thought processes in computer programming. However, some responders pointed to the importance of good user communities, where online fora and mailing lists can be extremely helpful in aiding equally the novice and the more experienced user during the development of their project. The real time, and often performative, nature of live coding does make that less relevant in a real-time situation, but such community support is invaluable in the pre-gramming (design) of the live coding system. Perhaps community is the most important thing of any live coding language, but for a community to establish, clearly the environment has to be good.

Acknowledgments

Thank you to all the respondents who give their time to complete the survey. This research is funded by the UK Arts and Humanities Research Council, ref AH/R002657/1.

REFERENCES

Ko, Andrew (2016). “A Human View of Programming Languages (Keynote),” in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2016)*. ACM, New York, NY, USA, 2–2. DOI: http://dx.doi.org/10.1145/2984043.2998390

Reina, Patrick; Ramsona, Stefan; Linckea, Jens; Hirschfelda, Robert and Papea, Tobias (2019). “Exploratory and Live, Programming and Coding A Literature Study Comparing Perspectives on Liveness” in *The Art, Science, and Engineering of Programming Journal*, 3 (1).

Sorensen, Andrew & Gardner, Henry (2017). “Systems Level Liveness with Extempore” in *Proceedings of Onward! 2017*, Vancouver, BC, Canada.

Tanimoto, Steve L. (2013). "A perspective on the evolution of live programming", *Proceedings of the LIVE 2013 workshop on Live Programming.* ICSE conference. San Francisco.