UiO : University of Oslo

Qichao Lan

Exploring Collaboration in Computer Music Systems for Live Coding

Thesis submitted for the degree of Philosophiae Doctor

Department of Musicology
Faculty of Humanities

RITMO Centre for Interdisciplinary Studies in Rhythm, Time and Motion

2022

Thesis submitted 9 November 2021
Thesis defended 2 June 2022

Advisors:

Professor Alexander Refsum Jensenius, University of Oslo
Professor Jim Tørresen, University of Oslo

Committee:

Professor Thor Magnusson, University of Sussex
Professor Rebecca Fiebrink, University of the Arts, London
Professor Rolf Inge Godøy, University of Oslo

Series of dissertations submitted to the
Faculty of Humanities, University of Oslo

Cover: Hanne Baadsgaard Utigard.
Print production: 07 Media, Oslo.

Abstract

This thesis investigates how our understanding of music collaboration in computer music systems can be shaped by different levels of abstraction in sound and music computing. Two types of collaboration have been studied: (a) human–computer collaboration where the computer acts as a composer or performer; (b) human–human collaboration where the computer serves as a tool. The research has been practice-based, focused on four projects: (1) RaveForce, a music generation environment using reinforcement learning, (2) QuaverSeries, a live coding language following a functional programming paradigm and its browser-based collaborative live coding environment, (3) Air Guitar, based on modelling action–sound mappings using data from an empirical study of guitarists, (4) Glicol, a redeveloped live coding/music programming language from audio sample level using the Rust programming language. The different projects have been developed iteratively through literature studies, theoretical reflections, design, implementation, performing, user studies, and self-reflection. This has resulted in functional prototypes that have been used in numerous artistic and workshop settings. Knowledge from these applications has funnelled back to a new understanding of collaboration in musical human–computer interaction. Particularly novel is the focus on the audio sample level, rethinking collaboration from a low-level audio programming aspect. This reveals an alternative way to build collaborative relationships between humans or between humans and machines.

Sammendrag

Denne avhandlingen utforsker hvordan kunnskap om musikalsk samhandling i datamaskinbaserte systemer kan formes av forskjellige abstraksjonsnivåer i lyd- og musikkprogrammering. To samhandlingstyper har blitt studert: (a) menneske–maskin-samhandling hvor maskinen fungerer som komponist eller utøver; (b) menneske–menneske-samhandling hvor maskinen er et verktøy. Forskningen har vært praksisbasert, med fokus på fire prosjekter: (1) RaveForce, et musikalsk utviklermiljø basert på forsterkende læring, (2) QuaverSeries, et live-kodespråk som bygger et funksjonelt programmeringsparadigme, (3) Air Guitar, basert på modellering av handling–lyd-mappinger som bruker data fra en empirisk studie av gitarister, (4) Glicol, et nyutviklet språk for live-koding/musikkprogrammering på lydsamplenivå basert på programmeringsspråket Rust. The ulike prosjektene har blitt utviklet iterativt gjennom litteraturstudier, teoretisk refleksjon, design, implementering, utøving, brukerstudier og selv-refleksjon. Dette har resultert i fungerende prototyper som har blitt brukt i et flere kunstneriske sammenhenger, samt workshops. Kunnskap fra disse anvendelsene har blitt brukt til å utvikle en bedre forståelse av musikalsk samhandling i menneske–maskin-interaksjon. Fokuset på lydsamplenivået er spesielt nyskapende, og muliggjør å tenke nytt om samhandling fra perspektivet til lavnivåprogrammering. Dette viser en alternativ måte å bygge samhandlingsrelasjoner mellom mennesker og mellom mennesker og maskiner.

Acknowledgements

Through the journey of my music technology research, there are so many people that I would like to thank. First and foremost, I am extremely grateful to my main supervisor Alexander, who gave me the opportunity to begin my PhD study. Studying in a different country far from home can be very challenging, and without your constant help, guidance and understanding, it would be impossible for me to complete this thesis, especially under the COVID-19 pandemic. I will never forget the time we spent together on meetings, paper editing, and performances. Your thoughts on music technology and the way you carry out research have been and will always be invaluable inspiration for my research.

Also, I want to thank my co-supervisor Jim for your insightful comments and suggestions on computer science. I am also grateful to Rolf Inge for coaching me on musicology. Your immense knowledge and great experience have made my understanding of humanities to the next level. I would like to extend my sincere thanks to my mentor Kyrre who has been offering me help since I started my PhD study. I should also mention Peter and Stefano from the Department of Musicology for all the help and communications.

I would like to thank the Research Council of Norway and the Nordic Sound and Music Computing Network (NordicSMC) for the funding of my research. I also want to thank Anne, Ancha, Jonna, Ragnhild and other leaders in RITMO Centre for the interdisciplinary and multi-culture working environment. Many thanks to Victor, Charles, Mike, and Olivier for your experience sharing during your time at RITMO as postdocs. It is also a valuable and memorable experience for me to work with Cagri, Tejaswinee, Agata and Dongho both as collaborators and friends. I also had a good memory to take PhD courses together with Benedikte, Julian, Kjell Andreas, Ulf, Martin, Dana and other colleagues in RITMO and the Department of Musicology.

During my Master's study, I was honoured to be supervised by Adrian Moore and Adam Stansbie at the University of Sheffield. Without your guidance, it would be impossible for me to start music research. Also, I would like to thank James Surgenor for showing me different music programming languages. I want to thank Alejandro Albornoz for introducing me to many Algorave scenes in Sheffield. Thanks should also go to Alex McLean for your generous help during the first live coding summer school. I would like to mention Ryan Kirkbride, Lucy Cheesman, and other Algorave community members in Yorkshire for your warm welcome at the AlgoMech Festival 2017. It was this period in Sheffield that motivated me to start my PhD study.

During my PhD study, I have also received lots of help around the world. I would like to thank Thor Magnusson, Chris Kiefer, and Francisco Bernardo for your hospitality during the Sema workshop at the University of Sussex. It was

one of the best memories before the pandemic. Special thanks to Anna Xambó for all your comments and suggestions during my PhD midway assessment. I am also thankful to Tero Parviainen, Charles Roberts, and Jack Armitage for all the communications on music technology with me. I should also mention the open-source communities. I am deeply indebted to Mitchell Nordine, Paul Adenot and other contributors for your work on the libraries and dependencies used in this thesis.

Finally, I want to thank my partner Gaoyang Li who has accompanied me throughout my PhD study. And I would like to express my deepest gratitude to my mother Yawen He. Without your tremendous support, understanding and encouragement in the past few years, it would be impossible for me to complete my study.

Qichao Lan
Oslo, May 2022

Papers

Paper I

Qichao Lan, Jim Torresen and Alexander Refsum Jensenius "RaveForce: A Deep Reinforcement Learning Environment for Music Generation". In: *Proceedings of the SMC Conferences. Society for Sound and Music Computing.* (2019), pp. 217–222.

Paper II

Qichao Lan and Alexander Refsum Jensenius "QuaverSeries: A Live Coding Environment for Music Performance Using Web Technologies". In: *Proceedings of the International Web Audio Conference (WAC).* NTNU, (2019), pp. 41–46.

Paper III

Cagri Erdem, Qichao Lan, Julian Fuhrer, Charles Patrick Martin, Jim Torresen and Alexander Refsum Jensenius "Towards Playing in the 'Air': Modeling Motion-Sound Energy Relationships in Electric Guitar Performance Using Deep Neural Networks". In: *Proceedings of the 17th Sound and Music Computing Conference.* (2020), pp. 177–184.

Paper IV

Cagri Erdem, Qichao Lan, and Alexander Refsum Jensenius "Exploring relationships between effort, motion, and sound in new musical instruments". In: *Human Technology.* (2020), pp. 314–347.

Paper V

Qichao Lan and Alexander Refsum Jensenius "Glicol: A Graph-oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet". In: *Proceedings of the International Web Audio Conference (WAC).* (2021)

Contents

Chapter 1

Introduction

Philosophers have hitherto only interpreted the world in various ways; the point is to change it.

—Karl Marx

1.1 Motivation

How can new technologies support and encourage musical collaboration? I still recall the first time I tried Troop (Kirkbride, 2017), a text editor developed for collaborative coding. It can be used with music live coding languages such as TidalCycles (McLean, 2014) and FoxDot (Kirkbride, 2016). This experience made me realise that live coding could be a future of music collaboration. In live coding, performers write code that is directly 'translated' into musical sound. In performance, it is common to display the code on a screen behind the performer so that the audience can follow the coding while listening to the resultant sound (Collins and McLean, 2014). However, after testing these systems for some time, I never felt fully satisfied with the coding syntax. Eventually, I began to investigate how I could build my own live coding language. While many other live coding environments started out as single-performer tools, my ambition was always to develop a collaborative solution.

Parallel to my interest in collaborative live coding, I was also curiously following the rapid growth of AI applications. Andrew Ng (2016) introduces artificial intelligence (AI) as 'the new electricity'. I was, therefore, keen to also explore how AI could support live coding. My PhD project has grown out of my interest in exploring these two types of technologies: collaborative live coding and AI-based musical interfaces.

Hence, in this thesis, I am motivated to take a bidirectional approach. One is bottom-up, in which I will explore the possibles to apply the new technology I learn into the design of new musical interfaces. The other is top-down, indicating that I will try to find necessary technology for my artistic ideas. Eventually, I aim to merge these two perspectives into one system that can be used in the future and provide a new understanding of the concept of music collaboration in computer music systems.

1.2 Research Questions

The main research objective of this dissertation is to investigate how the concept of collaboration in computer music systems can be shaped by different technologies.

The overarching research question, as stated above, is:

> **How can different levels of abstraction in music/audio programming influence collaboration in computer music systems?**

Here, the term *computer music system* is used over terms like *digital musical instrument* (DMI) or *new interfaces for musical expression* (NIME). That is because the focus of this thesis is on how the computer should be programmed. In a programmed computer, the concept of abstraction is widely used, which will be elaborated in Chapter 2. My main focus is on live coding, which on the one hand could be seen as a real-time programming activity, hence having some instrument-like qualities. However, my project also taps into non-real-time programming and composing-like tasks. These do not necessarily fit equally well within the context of 'instrument'. Furthermore, investigations into DMIs or NIMEs often involve hardware design and discussion, while my focus is primarily on the software side. The aim is to investigate how new technologies can enable new collaboration paradigms and generalise this process. More specifically, the main research question can be broken into three sub-questions:

> **RQ1: What kinds of relationships can be found in collaborative computer music systems and how can new relationships be designed?**

The term *collaboration* will be discussed in more detail in Chapter 2. I will use it to describe both human–human and human–computer types of collaboration. The goal is to get an overview of the current understanding of collaborative music-making and discuss how the paradigm of collaboration can be expanded. Chapter 3 introduces my practice-based method to explore this question and demonstrate how some new types of collaborations can be established in computer music systems.

> **RQ2: How can different kinds of relationships in collaborative computer music systems influence the performing practices?**

After defining and categorising the concept of collaboration and various relationships during collaboration, it is natural to analyse the relevant impact of these relationships on musical performances. Since this thesis is practice-based, this question will be answered mainly by analysing the results of a set of performances carried out with the developed computer music systems.

> **RQ3: What can new collaborative paradigms bring to the design of computer music systems?**

The last question points out a future goal of the research conducted in this thesis. The aim is to provide a new analytical framework on the design of collaborative computer music systems, which could ideally also be generalised to other types of computer music systems.

1.3 Approach

This is a highly interdisciplinary project. It is based on my dual background and interest in music performance and composition on one side, and computer science and technology on the other. In addition, I have been highly inspired by recent research in (embodied) music cognition during my years at RITMO Centre for Interdisciplinary Studies in Rhythm, Time, and Motion during my PhD studies at the University of Oslo. This has made me approach the questions with both 'soft' and 'hard' perspectives. More concretely, three aspects have shaped the background of the project:

- the design and development of previous computer music projects
- the theories that summarise and reflect on such projects
- the philosophical discussion on the experience of using computer music systems

On the practical side, the projects included in the thesis follows on a more than half-century-long path of explorations with text-based computer music languages (Wang, 2007), including Csound (Boulanger et al., 2000), SuperCollider (McCartney, 2002), and Chuck (Wang et al., 2015). Also, Kirkbride (2020)'s projects FoxDot and Troop have been an inspiration. So has Roberts and Kuchera-Morin (2012)'s Gibber, one of the leading browser-based live coding environments. The developments within network music, as reviewed by Ogborn (2018), has also been inspirational.

Based on the above-mentioned programming practices, I have also been inspired by various theoretical discussions and reflections on computer music systems, performance, and composition. This includes the reflections by Magnusson (2009) on how music technologies influence our understanding of music. More specific, McPherson and Tahıroğlu (2020) writes on how different computer (music) languages can make us think differently about music-making. Looking more specifically at instruments, Jensenius (2022) reflects on what an instrument is and the importance of considering the temporal aspect in both performance and perception of music. This is particularly relevant for my interest in live coding, which can be thought of as situated somewhere between composition and performance.

Although my focus is less on instrumental practice, I have still been inspired by the NIME literature (Jensenius and Lyons, 2017). In their description of the process for building new interactive musical instruments, Miranda and Wanderley (2006) define three basic subsystems: (1) the sensor input, (2) mappings, and (3) sound synthesis. Mappings, in particular, has emerged as a critical research question in the community. Mappings relate to how the input to a system is connected to the outputs. This ultimately also defines the *affordance* of a system, to use a term arising from Gibson (1977)'s seminal work in the psychology of perception which was introduced to the design sphere by Norman (2013). The

affordance of an object is based on the potential action relationships between subject and object based on qualities of the involved objects, what Jensenius (2022) calls action–sound mappings. His account is focused on instruments in particular, but I find it interesting to also investigate the concepts of mapping and affordance from a computer music system perspective.

When zooming out from the computer music domain to a more general technology of philosophy, two branches can be found. One is what Mitcham (1994) calls the 'humanities philosophy of technology' and begins with the experience of humans with technology. The other focuses on the technology itself. Mudd (2019) draws a distinction between communication-oriented and material-oriented instrument design. The former regards music technology as a tool to achieve artistic ideas while the latter views music technology as part of the instrument. In this thesis, I will have no pre-established preference for these two directions. Instead, I will discuss and compare these directions based on my practices. One is to design things viewing the tool I use as a black box. The other is to start from new technology and arrive at a complete computer music system.

1.4 Main Contributions

As Figure 1.1 shows, this thesis is based on the design, implementation, performance, and evaluation of four different computer music systems:

1. RaveForce: a music generation environment using reinforcement learning;
2. QuaverSeries: a live coding language following a functional programming paradigm and its browser-based collaborative live coding environment;
3. Air Guitar: an empirical study of 36 guitarists, with muscle data (EMG), motion capture, audio and video recordings. The collected data was used to train a machine learning model that can predict music information from muscle signals;
4. Glicol: a redeveloped live coding/music programming language from audio sample level using the Rust programming language.

The latter project, Glicol [1], can be seen as the culmination of ideas explored in the other projects. Glicol is first live coding system written in the highly efficient Rust programming language. It is also the first graph-oriented live coding language, and it brings a new design pattern for audio programming by combining LCS (longest common sub-sequence) algorithm and code pre-processing for efficient real-time audio graph updating.

Besides the concrete development results, the thesis also provides more general knowledge about developing at the audio sample level and interaction design in

[1] https://glicol.org

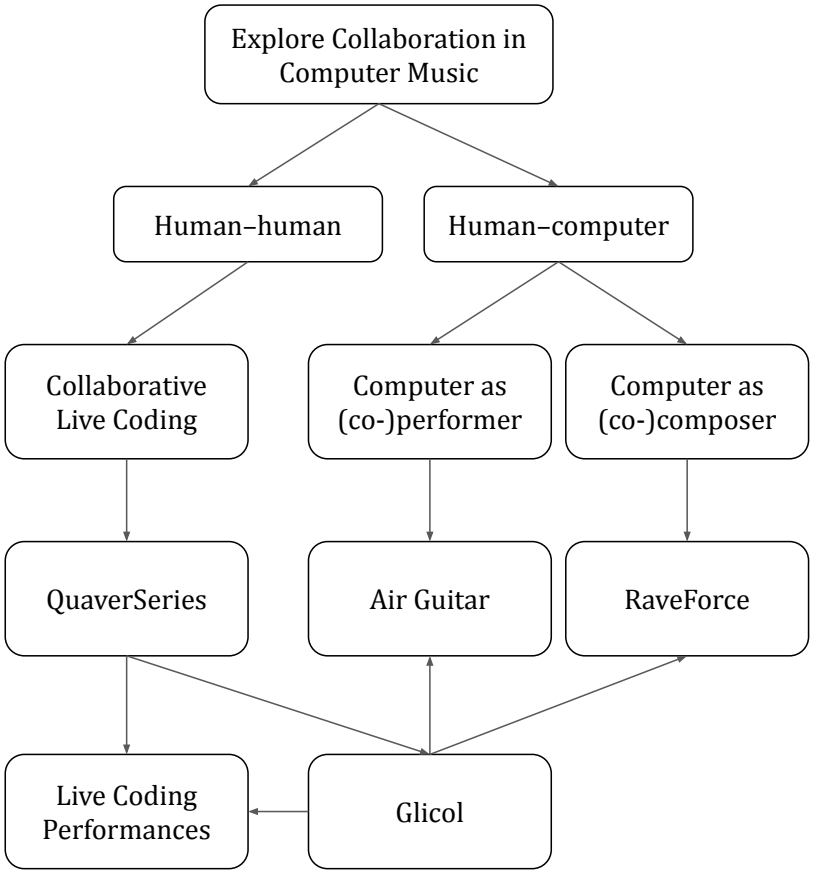


Figure 1.1: Road map of the method used in this thesis.

collaborative live coding. The results from my own and other users' experiences with the system have been used to reflect on the difficulties, uncomfortableness and errors.

1.5 Thesis Outline

This thesis comprises two main parts. The first part contains theory and method chapters (Chapters 2 and 3) and discussions about the research contribution (Chapters 4 and 5):

- Chapter 1 introduces the project and explains its core ideas, terms and research questions;
- Chapter 2 explains the background of some key concepts and terminology. The aim has been to write this part so that it is useful for readers coming both from music and computer science perspectives;
- Chapter 3 covers the methodology of this thesis, where I elaborate on the chosen methods in each project. These discussions will expand on the method sections presented in the related papers and also contextualise more broadly some of the benefits and weaknesses of the approach;
- Chapter 4 summarises the findings in each paper, and discusses the links between these findings and some of the related theories;
- Chapter 5 discusses the general findings with respect to the research questions and identifies possible avenues for future research.

The second part of the dissertation contains the five published research papers that are included in the thesis:

- Paper I introduces RaveForce, a music generation environment inspired by live coding;
- Paper II presents the live coding programming language QuaverSeries and its browser-based collaborative live coding environment;
- Paper III presents a machine learning-based musical interface that takes the muscle signal sensor data as input and predicts musical features as output;
- Paper IV further expands Paper III, and explore the relationship between the motion and the sound;
- Paper V is about the new music programming language Glicol, its web-based collaborative music performing environment.

The accompanying web page[2] contains links to the various projects, source code, data, and audio/video material of some performances.

[2] https://github.com/chaosprint/phd

Chapter 2

Background

In this chapter, I will elaborate on the background of this thesis, both from theoretical and practical aspects. The aim is to provide an overview of how the current understanding of collaboration in computer music systems is established. Given the interdisciplinary nature of this thesis, I have attempted to describe the various parts so that they are understandable from the two perspectives that I have been mainly working between: musicology and computer science.

2.1 What is computer music?

According to Keislar (2009), the field of computer music focuses on studying the algorithmic nature of computers for music composition and performance. In this thesis, the focus is on the algorithmic nature of the computer as well as a relatively young music practice called live coding, the music practice where musicians write programming codes to generate music. I will get back to a presentation of live coding. First, it is important to understand some basic components and principles for computer music, from how it is programmed to the representation of music.

2.1.1 The Computer and Digital Signal Processing

Modern computers are very versatile and can, for example, be used to simulate and record/playback the sounds of physical and analogue instruments. Compared with analogue devices, a digital computer can offer features such as programmability, algorithms, and reproducibility. To better understand these features, it is necessary to know what exactly a 'computer' is. The first computers were actually humans doing computations manually (Kvifte, 2008). Eventually, these human computers were overtaken by machines—first analogue, then digital—that could do the computations automatically. Today, the computer is a term used to denote machines that can perform logical calculations. They all have a CPU (Central Processing Unit), RAM (Random Access Memory), ROM (Read-Only Memory), etc., under the Von Neumann architecture (Bryant et al., 2003). On top of the hardware is the operating system (OS) software. And on top of the OS runs the common software such as a digital audio workstation (DAW).

The job of most music software is to process audio based on user interactions or commands (Roads et al., 1996). The audio processed by computers is represented by discrete signals. These signals can be modified using digital signal processing (DSP) techniques. Recent computers can all handle common DSP tasks in real-time. In contrast, non-real-time means that the user should wait for the

audio task to be done in a certain amount of time and cannot get the 'on-the-fly' audio immediately. This is often realised by calculating the audio output in blocks (Reiss and McPherson, 2014).

2.1.2 Programming, Interface and Abstraction

All software is written in a programming language that will eventually be compiled into machine code that controls the computer's CPU and memory. One can program on the hardware directly, which is often known as bare metal programming (Simmonds, 2012). The most primitive way is to write the machine language or machine code, composed of binary numbers (0 and 1). Assembly language is one of the most low-level languages that humans can understand while it is still relatively close to machine code. Nevertheless, Assembly syntax is still rather obscure. Therefore, Assembly is mainly used to write translation tools to translate some higher-level languages into machine code, and this process is known as *compilation* (Friedman et al., 2001).

The C programming language plays an important role in computer music (Boulanger and Lazzarini, 2010). Compared with Assembly, C is already closer to natural language, although it is still structural and abstract. The C programming language can directly control memory addresses and directly read or write values in ROM and RAM. C is generally considered to be a low-level language (Frampton et al., 2009), although in some cases when its lower architecture is discussed, C can also be viewed as a high-level programming language (King, 1996). Sometimes it is also called a 'system-level language', as C makes a good balance on the readability and the hardware accessibility (Ritchie et al., 1988).

Compared with the C programming language, its modern evolution C++ has added modern language features such as *Classes* while still retaining the low-level memory handling capabilities of C (Stroustrup, 2018). On top of these system-level languages, there are many high-level languages such as JavaScript or Python. In so-called high-level languages, operations are encapsulated, and the syntax becomes simpler and closer to natural language. Some of these languages can do JIT (just-in-time) execution, which means the program can be executed without having to wait for the compilation (Aycock, 2003). But the speed will generally slow down because of a mechanism called *garbage collection* that is designed to clean the unused memory by the program (Blackburn et al., 2004). In contrast, in C or C++, the programmers often have to remember to handle the memory cleaning manually (Nagarakatte et al., 2015).

The hierarchical relationship from low-level languages to the high-level ones is called *abstraction* (Kahanwal et al., 2013). Figure 2.1 shows an overview of the languages used for the projects included in this thesis. RaveForce, Air Guitar, and QuaverSeries are all based on C/C++ and three different higher-level languages(SuperCollider, Python, Javascript, respectively). As we will get back to later, I decided to explore Rust in the implementation of Glicol.

In addition to the abstraction in computers, a program typically also need an *interface* that users can interact with. This can be a command-line interface (CLI) where the interaction is happening by typing characters. It could also

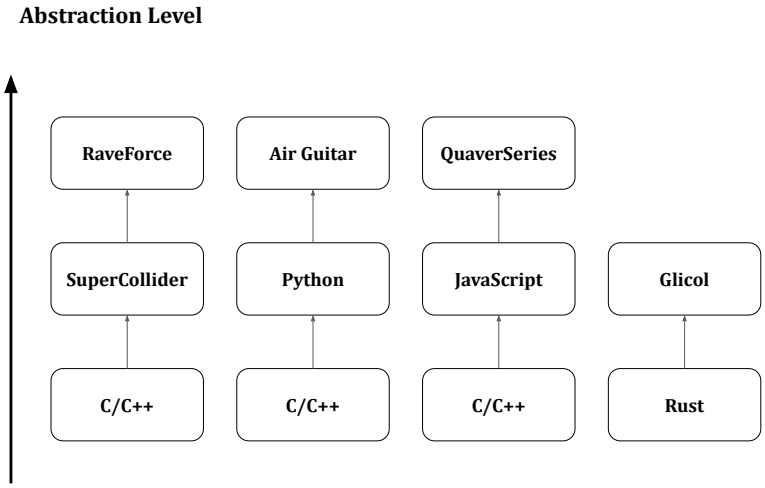


Figure 2.1: Abstraction level of different programming languages and projects involved in this thesis.

be a graphical user interface (GUI) operated by a mouse or touch screen. A computer program may also act as a library that other programs interact with through an application programming interface (API). A lot of contemporary web applications interact through APIs.

2.1.3 Representations of Music

Magnusson (2019) writes about how music has been encoded through the centuries using various types of representations. We can broadly classify the various music representations into what it often described as as symbolic or sub-symbolic representations in artificial intelligence studies (Calegari et al., 2020). A symbolic representation refers to the categorical description of an object while sub-symbolic representations are often not human-readable (Clarà and Barberà, 2014).

Western music notation using scores with notes is an example of a symbolic representation (Rowe, 2009). Sub-symbolic representations aim to represent continuous musical features that are not captured through symbols such as notes. A digital sound file is an example of a signal-based sub-symbolic representation of music.

Some of the popular computer-based symbolic representation methods include MIDI (Musical Instrument Digital Interface), musicXML, and MEI (Music

Encoding Initiative) (Fujinaga et al., 2018). MIDI is the oldest of these, and is one of the most popular data formats used in music technology research. In some particular styles of music, and particularly the ones based on traditional music notation, MIDI data can be an efficient representation. However, the MIDI protocol was widely criticised from its inception (Loy, 1985; Moore, 1988; Jack et al., 2018) and Cook (2017) described the standard as: 'MIDI: Miracle, Industry Designed, (In)adequate'. For example, the compact representation leaves out all information but the pitch, velocity, and duration of notes. If the aim is to capture more information, it is better to work with musicXML and MEI. Still, MIDI remains one of the most popular standards around, largely because of its ease of use, widespread accessibility, and a large number of music corpora. As such, the MIDI format remains a common format also in music research.

Another challenge with MIDI—which it shares with the other symbolic representations—is that it is unable to capture the richness and nuances of the sounding music, such as timbre and texture. This is where sub-symbolic representations, such as audio recordings, come into play. In the digital era, audio is stored with sampling techniques (McGuire and Pritts, 2013). Sound, as a physical phenomenon, is caused by the vibration of the source (Kinsler et al., 2000). The vibration is periodic with an origin centre. Accordingly, if we can sample its relative position to the origin centre fast enough, we can record the vibration of the continuous sound signal and save the samples in a digital storage. The rate of sampling per second is called the sample rate. For CD-quality audio, each sample has a *bit depth* of 16, which means there can be 2^{16} equally distributed float numbers between minus one to one. If the quality/accuracy is compressed to half, i.e. making it 8 bit, the music will sound *Lo-Fi* (low fidelity). Raw audio files can take up a lot of memory and disk space. To overcome this problem, audio files are often compressed using a lossy or lossless compression (Roads et al., 1996). The lossy compression formats are based on psycho-acoustics, including MP3 (formally MPEG-1 Audio Layer III or MPEG-2 Audio Layer III) or ACC (Advanced Audio Coding). The underlying compression algorithms utilise the fact that the human ear does not perceive all frequencies equally. Therefore, we can lower the quality of certain frequency ranges that are insensitive to humans. In this way, the audio file can be compressed to a smaller size with negligible audible effects (Bosi and Goldberg, 2012). This is efficient for file storage and sharing, although it requires more CPU processing both for encoding and decoding the signal.

2.2 Music Programming v.s. Audio Programming

In the previous section, I differentiated between low-level and high-level programming languages. Another way of categorising programming languages is to separate between general-purpose and domain-specific languages (Van Deursen et al., 2000). The general-purpose languages include C, C++, JavaScript, and Python, etc. All of these can be used for audio programming, although most

music technologists would probably rather prefer at domain-specific language. The benefits of a domain-specific language is that it is designed for a certain task (Visser, 2007), and will in most cases accomplish the goal more quickly and easily than a general-purpose language. Examples of domain-specific music languages include text-based languages (such as Csound (Boulanger et al., 2000), SuperCollider (McCartney, 2002), FAUST(Orlarey et al., 2009) and ChucK (Wang, 2008)) and visual languages (such as Pure Data (Puckette et al., 1997) and Max/MSP (a commercial project)). All of these have their benefits and limitations and attract different user bases.

2.2.1 Rust and Web Audio

In this thesis, I decided to venture into the exploration of some relatively new programming technologies: the Rust programming language (Jung, 2020) and the stack of Web Audio technologies (Smus, 2013). As mentioned above, the C and C++ programming languages are commonly viewed as low-level languages since they can access the computer memory pointer directly. These languages are also popular in low-level audio programming (Boulanger and Lazzarini, 2010; Pirkle, 2019). Rust programming language offers an effective alternative to C and C++. Rust is special in terms of its memory safety and speed (Blandy and Orendorff, 2017). This system-level programming language is becoming increasingly popular for its robustness in memory safety and computing performance (Beingessner, 2016). These advantages are realised by the *ownership* mechanism in Rust, and this mechanism also further facilitates a 'zero-cost abstraction'.[1] In the Rust Audio community, many audio libraries have been published, e.g. the `FunDSP` project[2] and the `dasp_graph`[3] library.

In recent years, browser-based audio programming has become increasingly popular. Thanks to new technologies such as WebAssembly (Haas et al., 2017) and AudioWorklet (Choi, 2018), browsers such as Chrome and Firefox can now support high-performance audio at a near-native speed. Several audio programming languages and libraries, e.g. Csound (Yi et al., 2018) and Faust (Letz et al., 2017), together with the recently developed Sema live coding environment (Bernardo et al., 2019), have all adopted these technologies.

2.2.2 Live Coding

As forerunners of live coding, Collins et al. (2003) describes live coding as 'coding music on the fly' and 'tweaking or writing the programs themselves as they perform'. In a broad sense, the term 'live coding' can mean any form of live programming practice. In this thesis, however, I will use live coding to describe music performance in which performers produce music by writing code rather than playing a traditional musical instrument.

[1] https://boats.gitlab.io/blog/post/zero-cost-abstractions/
[2] https://github.com/SamiPerttu/fundsp
[3] https://docs.rs/dasp_graph

Many live coding environments are installed locally and use SuperCollider as the sound engine (McCartney, 2002). SuperCollider consists of a programming language called *sclang* and also has an integrated development environment (IDE). In the IDE, users can boot an audio server called *scsynth* in the background, and write the code following the syntax of *sclang*. The code, when executed, will be sent as Open Sound Control (OSC) messages to the *scsynth* server to control the music sequence. One typical workflow in SuperCollider is to define the synthesiser architecture with the keyword *SynthDef*, and then play the *Synth* in SuperCollider *Pattern*, i.e. music sequences. The concept of pattern has deeply influenced this thesis, inspired by the live coding practice of McLean (2014).

Although some live coders use SuperCollider directly, other live coding artists choose to design domain-specific languages that interact with SuperCollider in the background. For instance, TidalCycles (Tidal) is a domain-specific language written in the Haskell programming language (McLean, 2014). As a functional programming language, everything in Tidal should be viewed as a function. In live coding, Tidal code will be interpreted to OSC messages to control the sound engine called *SuperDirt* running in SuperCollider (McLean, 2014). FoxDot, a Python live coding library, follows a similar architecture but uses Python as the host programming language (Kirkbride, 2016).

Nowadays, the web platform is becoming more and more popular for live coding. Web-based or browser-based live coding environments only require an up-to-date browser to get started with live coding. With the rapid progress of the Web Audio API, the sound synthesis possibilities and timing capabilities for browser-based live coding have matured quickly. Two good examples of this are the JavaScript-based Gibber environment (Roberts et al., 2015a), and the Lisp-style language Slang.js (Stetz, 2018). Although the latter currently does not support collaboration, its parser, written in Ohm.js, provides a valuable example for me. Another inspiration for me is from EarSketch (McCoid et al., 2013), a music producing environment mainly designed for normal programming education, and its use of the Firebase real-time database pointed us in the direction of a collaborative live coding solution (Xambó et al., 2017). Some other web-based environments serve as interfaces for other languages. Estuary is a system built for live coding with Tidal in browsers (Ogborn et al., 2017). It has several unique features: collaboration in four different text fields, the support for both SuperCollider and the Web Audio API, and so on. Estuary makes it possible to live code together from different locations, and has been shown to work reliably in cross-continental live coding.

Some music communities focus on 'hiding' the production techniques for the end-user. The live coding community is the opposite (Mori, 2015). Performers show the code during performances and the errors is widely accepted in the community (Magnusson, 2015). The performances adopt the computer's algorithmic capabilities. As such, live coding has developed into an algorithmic culture (McLean and Dean, 2018). Making errors is accepted (McLean, 2014). In fact, errors are often embraced and used actively in performance (Griffiths and McLean, 2017).

2.3 Artificial Intelligence

Another core part of this project, is the exploration of artificial intelligence (AI) in music. There are many types of AI, but we may generally split them into two core directions: rule-based and machine-learning approaches. Rule-based approaches have been used in music for centuries, but it was first at the end of the 20th century that AI and music became more systematically investigated (Roads, 1980). During the 1980s, there had been a profound discussion on AI and music (Roads, 1985), but the development of AI arrived at a bottle-neck during 1990s due to hardware limitations. More powerful computers and the development of advanced deep learning methods have made AI a cornerstone of many applications in the last decades (Russell and Norvig, 2016). AI technologies are currently used in many fields, including facial recognition, speech transcription, etc. We have also seen examples of how deep learning has surprised humans at many tasks, including the success of AlphaGo beating the world champion in Go (Silver et al., 2016).

2.3.1 Rule-based Approaches

Probabilistic and statistical models have been widely used in generative music (Rohrmeier and Pearce, 2018; Pearce and Rohrmeier, 2018). *Illiac Suite*, developed in 1957, is still worth mentioning today for its usage of Markov chains (Hiller and Isaacson, 1957). In 1972, James Anderson Moorer developed a heuristic and generative grammar for composition (Moorer, 1972). David Cope's EMI (Experiments in Musical Intelligence) system is another well-known example (Cope, 1996), which required human performers to play the generated score. Ebcioğlu's (1988) CHORAL software is based on 350 rules for choral melody generation. Genetic algorithms have been studied by Miranda (Miranda, 2001), inspired by biological concepts such as mutation, crossover and selection, which is commonly used to find solutions in optimisation problems (Mitchell, 1998). As evidenced through the comprehensive review of generative music models by Tatar and Pasquier (2019), there is no standard approaches taken: individual artists design systems based on their needs and preferences. I also find it particularly interesting that the systems are based on both symbolic and sub-symbolic representations.

2.3.2 Machine Learning

Machine learning is one of the most important AI approaches (Russell and Norvig, 2016). In general, machine learning refers to the process in which a machine is trained to learn a certain pattern from data. Machine learning can typically be divided into three categories: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning means that the learning function maps an input to an output based on example input-output pairs. The model compares the generated result with the ground truth answer and calculates an error value. The goal

of the training is to minimise the error. This is different from unsupervised learning in which there is no such ground truth in the datasets. Here the goal is to find some particular patterns within the datasets, through, for example, clustering techniques. Reinforcement learning is different from both supervised and unsupervised learning in that its updating strategy relies on the interaction between an agent and the environment rather than a function gradient. In a given period—that is, an *episode* in reinforcement learning—the agent will try to maximise the reward it can get. The reward is calculated in each episode, and it is used to update the parameters of the agents (Sutton and Barto, 2018).

A core topic in machine learning is that of *artificial neural networks* (ANN). Such ANNs were inspired by how one thought the human brain worked, that is, through neurons and activation functions (Goodfellow et al., 2016). When a sequence of neurons form a neural network, the calculation can be simplified with a *matrix* addition and multiplication. Such matrix calculations require a lot of resources. This is often done on computers with powerful graphics processing units (GPU). Although the GPU was not originally designed for neural network calculations, they are built for processing a lot of matrices. Thus, the GPU is much more efficient in neural network calculation than the CPU.

The advancements in GPU performance has opened a new field called *deep learning*. The word *deep* refers to the deeper neural network architecture of these models, which can be difficult to train without the help of GPUs. As a specific type of machine learning, deep learning can also be supervised, unsupervised or reinforced.

Deep learning has already been widely explored in music generation tasks (Briot et al., 2017). One example of supervised deep learning is the piano score generation in the DeepBach project (Hadjeres et al., 2017). It uses MIDI files of the works of Johann Sebastian Bach and trains a neural network to make new Bach-like compositions based on the datasets.

Unsupervised learning techniques, such as autoencoder and generative adversarial network (GAN), are frequently adopted in raw audio generation (Mehri et al., 2016; Donahue et al., 2018). For example, Roche et al. (2018) use auto-encoders to train neural networks to represent sound samples. By tuning the parameters inside the neural network, users can generate a new sound that is slightly different from the original. Unsupervised learning has also been used for music genre classification (Barreira et al., 2011). One of the most well-known applications of GAN in music is the GANSynth (Engel et al., 2019).

Recently, deep learning technology has brought new possibilities to reinforcement learning as it allows the agents to examine higher-level information. In deep reinforcement learning, the agent can be represented by a neural network, which makes it capable of evaluating the raw audio signal and then outputting the decision. Deep reinforcement learning has been successful recently since it shows that a virtual agent can surpass human beings in several tasks, e.g. Atari games (Mnih et al., 2013). One powerful algorithm is Proximal Policy Optimization (PPO) (Schulman et al., 2017). For testing this and other algorithms, there are many simulation environments, of which OpenAI Gym is one that I have explored in my project (Brockman et al., 2016).

Besides being used for generating music sequences, machine learning can also be used for creative interaction design for music. Machine learning has been a part of the design of what is often referred to as new interfaces for musical expression (NIME) since the early 1990s (Lee et al., 1991). Well-known examples include the *Wekinator* (Fiebrink, 2011), Gesture Follower (Bevilacqua et al., 2009), ml.* library (Smith and Garnett, 2012), Gesture Recognition Toolkit (GRT) (Gillian and Paradiso, 2014), Gesture Variation Follower (GVF) (Caramiaux et al., 2015), and ml.lib (Bullock and Momeni, 2015). These (and other) tools allow for using machine learning algorithms through either a graphical user interface (GUI), or, in the form of external libraries for audio programming platforms, such as Max/MSP and Pure Data. A number of new musical interfaces have employed such systems, such as, *The Birl* (Snyder and Ryan, 2014), Echo State Networks (ESNs) (Kiefer, 2014), and *Double Vortex* (Schacher et al., 2015).

In recent years there has been an increasing interest in applications of deep neural networks (DNNs) for symbolic music generation or audio modelling such as the DDSP project (Engel et al., 2020). There are fewer musical examples of physical interaction (see, for example, (Martin et al., 2018a) for an overview of deep predictive models in interactive music). A recent interactive music framework for deep learning is *IMPS*, which uses a mixture density network (MDN) over long short-term memory (LSTM) layers, and provides a low-entry-fee for a musical exploration of DNNs (Martin and Torresen, 2019). Within instrument design, the intelligent mapping structure of Gregorio and Kim (2019), and the human–machine collaborative improvisation system of McCormack et al. (2019) are some relevant examples.

2.4 Network and Collaboration

So far, I have covered topics from computer music related to this thesis, including digital signal processing, audio programming, live coding, music representations, and musical AI applications. In this section, I will focus on two concepts that serve as a theoretical framework of this thesis: networks and collaboration.

I will argue that networks and collaboration are almost inseparable in computer music. Networks are based on creating collaboration between separate entities and collaboration necessitates the formations of networks (Budner and Grahl, 2016). Despite the inherent relationship between these two terms, they are often separated in theory and practice. As Ogborn (2018) quotes from Small (1998):

> Networking, then, can be productively elided with musicking, understood by Christopher Small as the establishment of relationships that 'model, or stand as metaphor for, ideal relationships as the participants in the performance imagine them to be: relationships between person and person, between individual and society, between humanity and the natural world and even perhaps the supernatural world'.

In music technology, we often refer to 'network technology' as the cables, devices, and software that makes communication happen (Rohrhuber, 2012). A more conceptual understanding of could start from considering different elements in musical engagement. A conceptual network is very close to the concept of collaboration. Even the technical aspect of a network can be seen as a means to support collaboration (Roberts et al., 2015b). Since my aim in this thesis is to examine the concept of collaboration in live coding, I will think of 'network' as both a technological and conceptual concept. In the following, I will elaborate on two aspects of collaboration: human–human and human–computer.

2.4.1 Human–human collaboration

Several things are important in human–human networks: communication, roles, power structures, and errors (Born, 2013). All of these can significantly influence the design of computer music systems. For example, the designer needs to consider the communication nature of the musical work in question (Kendall and Carterette, 1990). The designer should also decide on the roles and power structure between agents to ensure the order in performances (Mahon, 2014). Potential errors should also be considered, both technically and culturally (Knotts, 2020).

Baalman (2015) points out that live coders focus more on the process than the end result (the 'work'). One feature that makes live coding interesting is the openness of the music-making process. Unlike other programming practices, live coding, as an open-end art activity, seeks not to address some particular problem but to explore code through music-making. An important part of the performance is to display the code. This open-ended artistic nature of live coding emphasises social communication over problem-solving. The visual display of the code is not just for musicking but also comes with communication possibilities. Aaron (2016) indicates that the live coding language Sonic Pi is not only for musicking but also for teaching programming. Then it is essential for the code to be meaningful also for the audience.

Jensenius (2022) discusses different roles and actors in musicking, from instrument builder to music critics. He also argues that new technologies open for more overlap—and even merging—of roles. I think live coding is a good example of a musical practice in which the roles are in play. Live coding is perhaps the only music practice so far that can offer someone to talk on all the different roles: from (literally) developing their own instrument to composing and performing. They can pause and listen to the machine play their code, thereby also acting as listeners and critics of the unfolding musical sound.

We may view such a role shift as a reallocation of power. For example, when the performer is given the power to modify the instrument, it empowers the performer to break written or unwritten rules. Of course, we need to distinguish between modifying and using an instrument. An acoustic instrument can only be modified to a certain extent on stage. Construction changes need to be done off-stage by a luthier. A live coder, on the other hand, may (at least in theory) reconfigure everything on stage. Still, there are some intrinsic limitations posed

by developing computer-based instruments. In 'iMac Music', Jonathan Reus challenges this limitation: when the program runs, the performer alters the circuitry so that sounds created by the iMac is changed (Norman, 2016).

2.4.2 Human–computer collaboration

A human and a computer can also form a conceptual network, something that has been explored generally in human–computer-interaction (HCI) design, and also specifically from a musical perspective in the community surrounding the NIME Conference (Jensenius and Lyons, 2017). In the following, I will discuss some aspects relating to human–computer collaboration.

Music is typically made at the meeting point between a performer and an instrument. So when we want to explore human–computer collaboration from a live coding perspective, we need to start by identifying the 'instrument'. Many live coders view the language they work with as an 'instrument' (Blackwell et al., 2014). This definition may be dubious when comparing it with a physical instrument, such as the guitar, in which there are physical action–sound couplings between the performer and the instrument (Jensenius, 2022). In the case of live coding, it may make sense to think that musicians 'play the computer' as an instrument (Cook, 2017). It is the computer (nowadays usually in the form of a laptop in live coding) that makes sound and the performer is the one who modifies the way the computer plays music. What differentiates live coding from other types of computer music, is that the programming language serves as an interface for the instrument.

Live coding can often be seen as a 'disembodied' performance practice. But how is actually the relationship between the human body and the instrument in live coding? The term 'embodiment' comes from cognitive science and has more recently been introduced in music research (Leman et al., 2018). On physical instruments, such as the piano and guitar, performers make sound by physically touching the instrument. The notion of embodiment is typically built on top of our understanding of such 'traditional' instruments with which performers need to make contact. Even for scholars outside musicology, such as Ihde (2004), physical instruments are often used to illustrate the 'embodied relationship' that we build with technology. However, as a counterexample, the emergence of analogue electronic instruments brings the notion of embodiment to a different level. For instance, with the Theremin, the performers do not need to touch the instrument to control the sound with their body (Theremin and Petrishev, 1996).

Live coding inherits performance features found in contact-less instruments and musical styles with loose performance timing. Performers write programs to change the behaviour of the computer algorithmically. Delving deeper into the timing in live coding, we may find that the control unit is relatively different from physical instruments. Godøy (2008) has proposed the concept of 'sound-–action chunks' in music. Here a chunk is seen as a meaningful unit in music practice. However, in many live coding languages, the minimal unit is typically set to

one bar. This unit is so large that it may completely alter the cognition mode performers build with the instrument.

Considering timing, Goldman (2019) argues that live coding tends to be 'disembodied and propositional'. But we should not forget that the performers are still under a certain degree of time pressure. Erdem and Jensenius (2020) have reported the difference in time pressure in performances with embodied instruments and live coding. The context they mention is a collaboration between live coders and embodied digital instrument musicians. They indicate that different roles in sound practices tend to arrange time differently. As opposed to regular music programming, which usually happens outside of a performance context, in live coding there is, indeed, a time pressure.

I would argue that live coding is an example of a quite unique body-–instrument relationship. There is no direct action-–sound feedback nor strong time pressure that you would find when performingon an acoustic instrument. The time is loosely controlled within the unit of a bar. But the performer can choose to wait as the music will continue to play even without modifications. Still, there is a need to update the music once in a while. Thus performers are not entirely free from time pressure. They are not composers and producers, working in a non-real-time mode. As we will get back to later in this thesis, live coding is situated somewhere between composition and performance.

Another important concept from the embodiment literature, is that of *affordance*. This term was introduced by (Gibson, 1977), denoting the action potential of an object. The concept was later introduced to the design community by Norman (2013) and has become influential both in the general field of HCI, and also in the NIME community. How can we think about affordances within live coding? Or, how would live coding differ from other 'programmable' instruments, say drum machines? In his taxonomy of musical instruments, Jensenius (2022) describes instruments along an axis going from highly embodied to completely disembodied. Examples of the latter are automatic or even imaginary instruments. Programming languages have enabled the computer to work automatically. In some cases, one can even say that they get closer to the imaginary domain. For example, the audio effect 'reverb' refers to the simulation of space feelings. In artificial reverb, we are limited by the electric circuits and can at most adjust within a specific range (Valimaki et al., 2012). On the other hand, in programming languages, we can have a huge parameter for the room space. As such, affordances can be programmed into the system, something that can also actively be used in live coding practice.

2.5 Music (and) Technology

I will end this background chapter with a brief reflection on the term 'music technology'. The word 'technology' originates from the Greek word *techne*, which means the skill to build things (Tabachnick, 2004). Reflections of technology have nearly the same long history as ancient Greek philosophy. In modern ages, the philosophy of technology began to emerge as an independent discipline in

recent decades (Dusek et al., 2006). Two diverging branches can be found in this discipline. The first one is what Mitcham (1994) calls 'humanities technology of philosophy'. This philosophy is inherited from the phenomenology of Husserl, and later Heidegger, and reflects on the impact of technology on society and culture (van Mazijk, 2019). Most of the philosophers from this branch hold a critical attitude towards technology (Wierzbicki, 2015). The second branch is often called 'analytic philosophy of technology' or 'engineering philosophy', and aims to build philosophical reflections from the technology itself (Rogers, 1983).

As for the definition of music, from a more theoretical perspective, music can be defined as the exploration of melody, harmony, rhythm and timbre. Another famous definition of music is *organised sound* by Varèse and Wen-Chung (1966). Nevertheless, music is beyond the sound itself but the whole process of artistic expression. Jensenius (2007) argues that music is *movement*, which is in line with the concept of *musicking* presented by Small (1998). Similarly, Bunge (1966) argues that technology, as 'applied science', is about action, but an action heavily underpinned by theory. He continues to indicate that technology and music are all a form of action based on theories, while music can break the rules and create new theories. However, what Bunge has not covered is that technology, and especially music technology, can also sometimes break the rules. One example is how Jimi Hendrix broke the rules by using a distortion effect on his electric guitar. This broke with the audio engineering conventions at the time, but it ended up creating a new aesthetic (Fricke, 1998).

I find in interesting to consider music from the two pathways in the philosophy of technology. One is to analyse how current music technologies influence the way we understand music production and performance. This can be seen as a communication-oriented method (Mudd, 2019), which sees music technology as only a means to achieve artistic ideas. The other is to analyse the possibilities afforded by new technologies and how they alter the way we understand music. Such a material-oriented approach can be exemplified with some practice-led computer music projects. In the context of developing the Chuck programming language, Wang (2008) quotes from the great computer scientist Alan Perlis that 'a programming language that doesn't change the way you think is not worth learning'.

To sum up, it is important to remember that music technologies are often creative in both scope and nature. Developing music technologies is different from other technologies that aim for a particular goal. The artistic nature of music means that the technologies also need to embody the same artistic ambitions. And, as I will discuss in the coming chapters, live coding environments become interesting due to their peculiarities.

Chapter 3

Method

3.1 Introduction and Rationale for Research

The research undertaken in this thesis is based on a practice-based approach. Since technologies are constantly in development, it is hard to simply review past accomplishments and yield knowledge that can guide the future. I find that the new artefacts may completely contradict previous concepts, hypotheses, and ideas. I have, therefore, found it useful to employ what may be thought of as an iterative design–research method: looping between designing, implementing, performing and reflecting. The goal has been to extract new knowledge while continuously building and testing. In this chapter, I will elaborate on the various methods used, why I chose them and how I solved issues during the process.

3.2 Design Considerations

3.2.1 RaveForce

In RaveForce, I was interested in exploring musical representations somewhere between symbols and signals. One way to address limitations of symbolic representations is the use of sample-level music generation, as demonstrated in WaveNet (Van Den Oord et al., 2016) and WaveRNN (Kalchbrenner et al., 2018). However, although some progress has been made, raw audio generation requires a lot of computational resources. The data format can also influence the design of the neural network. In symbolic representations, supervised learning can be found in many applications (Sturm et al., 2019). For raw audio signals, unsupervised learning techniques such as autoencoder and generative adversarial network (GAN) are frequently adopted (Mehri et al., 2016; Donahue et al., 2018).

Reinforcement learning is different from supervised or unsupervised learning techniques in that its updating strategy relies on the interaction between an agent and the environment. For example, the agent can be a neural network that reads the state from a video game (the environment), be it the current score of even the raw screen pixels. In a given period—that is, an *episode* in reinforcement learning—the agent will try to maximise the reward it can get, e.g. to survive as long as possible in a game. The reward is calculated in each episode, and it is used to update the parameters of the agents (Sutton and Barto, 2018). The connection between reinforcement learning and music generation goes back to the use of Markov models in algorithmic composition (Bell, 2011). As one of the pioneers in automated music generation, in the piece called *Analogique A*, Iannis Xenakis uses Markov models for the order of musical sections (Xenakis, 1992). The use of Markov models in composition reveals its connection with reinforcement learning as the action of the agent only depends on the current

state. However, in previous research on reinforcement learning in computational music generation (Collins, 2008), the observation is not built on a deep neural network which only becomes prevalent in recent years. In deep reinforcement learning, the agent can be represented by a neural network, which makes it capable of evaluating the raw audio signal and then outputting the decision. Deep reinforcement learning has been a success during the past few years since it shows that a virtual agent can surpass human beings in several tasks, e.g. playing Atari games (Mnih et al., 2013). For music, deep reinforcement learning has been used for the score following (Dorfer et al., 2018). However, there is still no environment designed for music generation. For testing these algorithms, there are many simulation environments. In RaveForce I decided to work with OpenAI Gym[1] as it is one of the most popular benchmarks for training agents.

Though symbolic representations have shown some limitations, generating music at the audio sample level can be computationally expensive. Therefore, in RaveForce I propose to first generate some symbolic representations and then use these representation to synthesise audio for evaluation. The first step is to choose a proper method to convert a symbolic representation to an audio file. Three options were considered:

1. to send the generated sequence to an instrument and record the sound for evaluation,

2. to use a general-purpose programming language such as C++ for the sound synthesis,

3. to use a music programming language like Max/MSP, Pure Data, Csound and SuperCollider for non-real-time synthesis.

I quickly excluded the first option because it would be too time-consuming. The deep learning training process requires a considerable number of iterations. The second option is the most efficient in terms of computational speed and flexibility. However, a general-purpose programming language lacks the extensibility from a music perspective and users would have to be familiar with a C-style programming language. Then the third option better balances efficiency and usability as music programming languages are ubiquitous in the electronic music field (Wang, 2007).

I decided to try deep reinforcement learning because 1) it is still relatively less explored for music generation, 2) the success of AlphaGo (Silver et al., 2017) is motivating, and 3) it provides a possibility to connect with music programming languages that are familiar to music programmers. The idea is to train a neural network to output a sequence of symbolic music notation (such as the parameters for a synthesiser) and send the information to an audio programming language for non-real-time synthesis. Then, I compare the synthesised audio file with the target file, or I can use a neural network to grade the audio file directly (see

[1] https://gym.openai.com

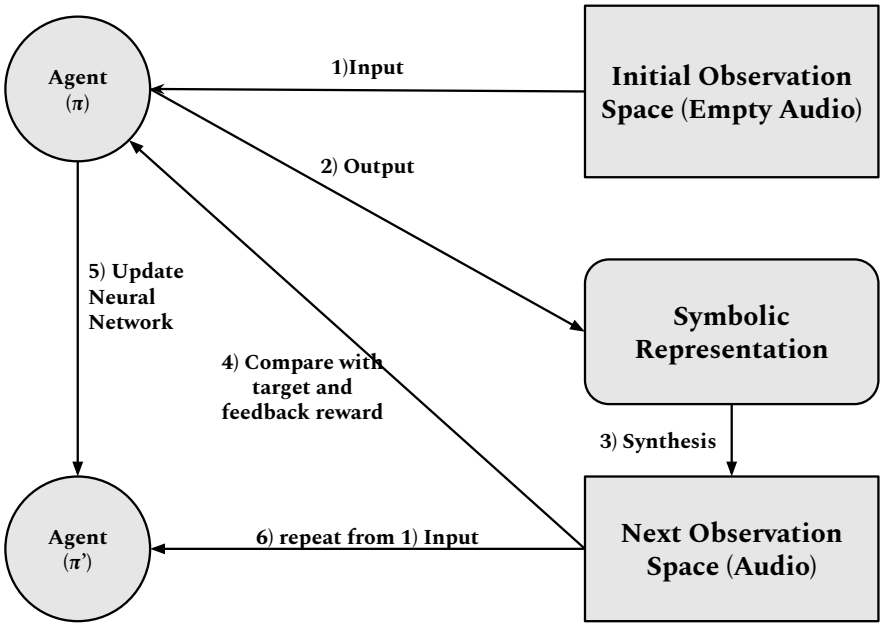


Figure 3.1: RaveForce workflow.

Figure 3.1). When an action brings a positive reward, the probability of the action should increase, and vice versa.

There are several important concepts in deep reinforcement learning that need to be defined in the music context (see Figure 3.2):

1. *Step* refers to the process of executing what has been decided to do in the next 16 th note or rest.

2. *Episode* refers to a series of continuous interactions before the *done* attribute turns to *true*, e.g. the end of a game. In a musical context, I use a *total-step* attribute to decide the length of an episode. Thus, it can vary from one single note to a note sequence.

3. *Observation-space* refers to the current state. In the musical context, I set the currently synthesised audio file to be the observation-space. In other words, the agent should be 'aware' of the previous state (synthesised audio) and take the next step accordingly.

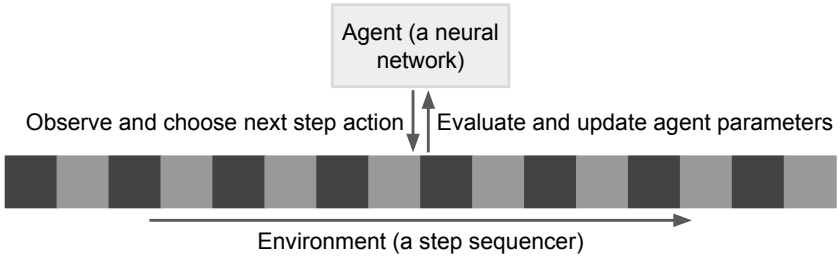


Figure 3.2: RaveForce architecture: in each note (step), the agent (neural network) will choose an action according to its observation on the current state (currently synthesised audio).

4. *Action-space* refers to the set of action choices for the agent. In a musical context, the action-space can be discrete (e.g. a note pitch) or continuous (e.g. the amplitude).

With this design, the expected result is to build an agent that can help to detect synthesiser parameters. With an audio file and neural network structure provided in Python and a synth structure written in SuperCollider, the agent should be able to tweak the synth parameters by itself.

3.2.2 Air Guitar

My inspiration for this project comes from the theoretical analysis on the 'air guitar' playing from a music cognition perspective (Godøy et al., 2005). I was motivated to use machine learning knowledge to make the concept a real instrument. The central idea of this project was to investigate what Jensenius (2022) refers to as the action–sound couplings found in guitar performance and use these for the creation of action–sound mappings in an 'air instrument' controller. Here couplings refer to the physical relations between action and sound found in acoustic instruments, while mappings are used to describe the connections created by input and output parameters in a computer music system. Creating an air instrument can be thought of as a way of letting a user explore sonic interaction without physically touching an instrument (Jensenius, 2017).

There are many different options for providing the gestural information as the input of the neural network, e.g. the Leap Motion sensor (Han and Gold, 2014). But what I view as the most intuitive one is the Myo muscle signal sensor that has been used in RITMO Centre for years (Martin et al., 2020; Erdem et al., 2020). Muscle signals can be captured using electromyography (EMG), which essentially records the electrical activity of muscles (Phinyomark et al., 2020). The idea was to create a model of relationships between extracted muscle activity and sound features. The model is trained on raw EMG signals and the

RMS of the resultant sound. Finally, the system should be tested with the EMG input from freely improvised recordings.

3.2.3 QuaverSeries

After the explorations with RaveForce and Air Guitar, I moved on to consider collaborative paradigms. Many existing live coding environments are installed locally and use the SuperCollider music programming language as their sound engine (McCartney, 2002). With the advent of the Web Audio API, there has been a shift towards developing live coding environments with web technologies. In the following, I will reflect on these two approaches to live coding.

SuperCollider consists of a programming language called *sclang* and an integrated development environment (IDE). In the IDE, users can boot an audio server (*scsynth*) in the background, and write the code following the syntax of *sclang*. In general, the code, when executed, will be converted into Open Sound Control (OSC) messages (Wright, 2005), and sent to the *scsynth* server to control the music sequence. One typical workflow in SuperCollider is to define the synthesiser architecture with the keyword `SynthDef`, and then play the *Synth* in a SuperCollider music sequence (*Pattern*).

Several live coders have chosen to design their syntaxes on top of SuperCollider. For instance, TidalCycles (Tidal) is a domain-specific language written in the Haskell programming language (McLean, 2014). During live coding, the Tidal code will be interpreted and sent as OSC messages to control the sound engine called *SuperDirt* running in SuperCollider. FoxDot follows a similar architecture but uses Python as the host programming language (Kirkbride, 2016). Additionally, Troop is a collaborative environment developed for both Tidal and FoxDot, which allows users on the same network to co-edit and share the code on the screen (Kirkbride, 2017).

An inconvenience with the above-mentioned environments, relying on one or more programming languages in addition to SuperCollider, is that it requires several steps of installation. This can be a hassle for users and in my experience it often leads users to give up before they even get started. A more user-friendly solution, then, is Sonic Pi (Aaron, 2016). This environment is also built on the SuperCollider audio engine, but it offers a single, complete installation package. Even though several operating systems are supported, Sonic Pi does not currently run on desktop Linux or Chrome OS. The latter is particularly problematic, because it makes the environment less usable for schools that have decided to use Chromebooks in their education. More generally, have found that having to rely on software installs is less than ideal in a teaching situation. Then working with a web-based solution is more feasible and scalable.

Web-based or browser-based live coding environments only require an up-to-date browser to get started with live coding. With the rapid progress of the Web Audio API (Smus, 2013), the sound synthesis possibilities and timing capabilities for browser-based live coding have matured quickly. Two good examples of this are the JavaScript-based Gibber environment (Roberts et al., 2015a), and the Lisp-style language Slang.js (Stetz, 2018). Although the latter currently does

not support collaboration, its parser, written in Ohm.js [2], provides a valuable example for the development. Another inspiration was EarSketch (McCoid et al., 2013), a music producing environment mainly designed for normal programming education. Its use of the Firebase real-time database pointed us in the direction of a collaborative live coding solution (Xambó et al., 2017).

Some other web-based environments serve as interfaces for other live coding languages. Estuary is a system built for live coding with Tidal in browsers (Ogborn et al., 2017). It has several unique features: collaboration in the different text fields, the support for both SuperCollider and the Web Audio API, and so on. Estuary makes it possible to live code together from different locations, and has been shown to work reliably in cross-continental live coding.

As can be summarised from the brief review of existing live coding environments, the programming languages, syntaxes, and interfaces are diversified. In the development of QuaverSeries, I borrowed parts from many of these when designing the syntax and environment.

3.2.4 Glicol

The iterative development and testing of QuaverSeries eventually led to the development of Glicol.[3] The key philosophy in Glicol is to make music by using a graph-oriented syntax. Such a design is based on both technical and aesthetic considerations. As mentioned in the introduction, the aim of Glicol was to keep the syntax simplicity of QuaverSeries. However, the development of a new audio engine necessitated some changes in the language.

QuaverSeries is a functional programming language that runs in a web browser. Its audio engine relies on Tone.js (Mann, 2015), a JavaScript library built on top of the Web Audio API. While this allows for concise customised syntax it is less flexible for sample-level sound synthesis. It is also hard to handle run-time errors from a memory aspect. These issues make it hard to guarantee the run-time environment's robustness in musical performances. These challenges motivated me to rethink live coding languages from the lower level in the development of Glicol.

One major change was the decision to develop Glicol in the Rust programming language (Balasubramanian et al., 2017). Rust has been developed as an effective alternative to C and C++ and this system-level programming language is becoming increasingly popular for its robustness in memory safety and computing performance. These advantages are realised by the *ownership* mechanism in Rust, a mechanism that facilitates a 'zero-cost abstraction'[4]. This means that, in theory, we can write a parser in Rust and convert it into audio streams without losing performance.

Developing a live coding language from the low level has several benefits: (1) better audio performance can be obtained; (2) errors can be handled in the memory level; (3) the language is easier to port to different platforms, including

[2] https://github.com/harc/ohm

[3] https://github.com/chaosprint/glicol

[4] https://boats.gitlab.io/blog/post/zero-cost-abstractions/

embedded platforms such as Bela (McPherson, 2017); (4) the language can serve as an intermediate structure to support higher-level customised languages.

I began the design considerations with the programming paradigm, which largely determined the interface I interacted with in live coding. The two most popular programming paradigms in the live coding community are *functional* programming (FP) and object-oriented programming (OOP). There are benefits of both these approaches, but I have chosen a third: *graph-oriented* paradigm. One reason is to experiment with this new idea from a sound and music computing perspective. Another is to fully leverage some unique features in Rust, such as the 'ownership mechanism' (Levy et al., 2015).

The term *graph* means the abstract collection of a series of *nodes* connected by *edges* (Shirinivas et al., 2010). In audio programming, the concepts of graph and nodes are ubiquitous, such as in the Web Audio API and SuperCollider. In the Rust Audio community, the concept of graph has been widely adopted, e.g. the in the FunDSP project[5] and the dasp_graph[6] library. They both take advantage of Rust's *trait* feature by offering a template for implementing the `node` *trait* for different structures.

Though the concept of graphs and nodes is widely used in audio programming, few languages adopt a graph-oriented programming paradigm in the syntax design. In the experiments with QuaverSeries, I found that its syntax based on the functional programming paradigm can also be used for a graph-oriented paradigm with some minor modifications. I believe this paradigm can be easier for a beginner to master due to its 'linear' and straightforward logic. It can also be used by advanced programmers as an intermediate structure for developing higher-level languages.

In Glicol, a *node* is represented by using specific keywords such as `sin`, `mul` and `add`, followed by its required parameters. A *chain* can be created by connecting nodes in series with the double greater-than sign (>>), and a *reference* can be used to denote this chain of signal flow. In the example, both `lead` and ~`mod` are *references*. Using the *reference* as the node's parameter means that this parameter is controlled by another signal chain, which is also called a *side-chain* in signal processing. Note that only the *reference* that comes without a tilde (~) will be sent to the audio interface. This is the syntax for separating control signals and audio signals, although they both run at audio rate. The reference of a signal chain can also be used as a node in another chain.

Despite their similarities in appearance, there is some intrinsic discrepancies between these two syntaxes. The nodes in Glicol are not functions, but Rust graph data structures. This data structure can be converted from the input code *String* smoothly. In contrast, QuaverSeries is based on 'impure' functions that can access global variables such as the Tone.js *Object* directly. As a result, the implementation in QuaverSeries is not as robust as the one in Glicol in terms of memory safety, as there may be potential data conflicts in these impure functions. In Rust, implementing Glicol in a functional programming manner is

[5] https://github.com/SamiPerttu/fundsp
[6] https://docs.rs/dasp_graph

limited by the ownership mechanism, which is used to guarantee memory safety. For example, it is complicated for a global variable, such as the clock, to be 'moved' inside and outside the *scope* of different functions. Therefore, I believe that the graph-oriented paradigm fits the design of Rust better when it comes to implementing an audio programming language.

3.3 Implementation Challenges

3.3.1 RaveForce

As discussed above, the key to the proposed RaveForce method is an environment that can handle non-real-time synthesis and evaluate the result. In the implementation[7], I follow the OpenAI Gym interfaces in the Python module. Then in SuperCollider, I create a quark to execute the non-real-time audio synthesis.

For the sound synthesis part, I am inspired by how live coders use the client-server architecture in SuperCollider. The client-server architecture that contains two parts: the *scsynth* (SuperCollider Synthesiser) and the *sclang* (SuperCollider Language). The *sclang* is combined in real-time through a simplified type of Open Sound Control (OSC) messages (Wright, 2005) and sent to the *scsynth* to control the sound. This architecture allows the *scsynth* server to run alone, while *sclang* can be replaced by other domain-specific languages (DSLs) like TidalCycles[8]. In short, in a live coding session, the coders use DSLs as a client to control the real-time sound synthesis in the SuperCollider server. In RaveForce, I have used SuperCollider in a similar client–server setup, although as non-real-time audio synthesis engine.

I decided to develop the RaveForce client in Python. The main reason for that was to leverage the many deep learning frameworks (such as PyTorch [9]) that have been implemented in Python. Also, the Python module Gym is one of the most important benchmarks for deep reinforcement learning. By designing the client part in Python, I could follow the Gym interface and connect with a deep learning framework. With the help of Open Sound Control messages, I could link the neural network training with the audio synthesis in SuperCollider (see Fig. 3.3).

The pseudo-code of this implementation strategy is as follows:

1. Use a *make* function in the client to create the required environment, which will send a message to the server, asking the server to load related music patterns, synthesise an empty file and return the address of the file to the client-side. On receiving the returning message, the client should read the action space and the observation space.

[7] The code is openly available at https://github.com/chaosprint/RaveForce
[8] https://tidalcycles.org
[9] https://pytorch.org

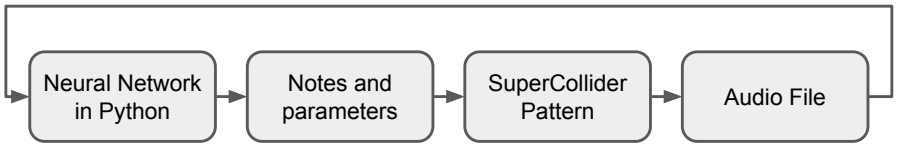


Figure 3.3: Python-SuperCollider communication in RaveForce: a neural network (agent) is trained in Python; it sends symbolic music representations(e.g. notes and synthesiser parameters) as Open Sound Control messages to the SuperCollider pattern; then the pattern will be synthesised to an audio file in non-real-time and sent back to Python as the input of the neural network, forming an iteration.

2. Send a *reset* message to the server. Empty the observation space if it is not.
3. According to the observation space, decide what action to take. Send the *step* message to the server-side with chosen actions in each step. The server will do non-real-time synthesis in each step according to the given action message. Also, the server should return the client with the synthesised file address.
4. The client should use the address to load the currently synthesised sound file and set it as the observation space. Calculate the reward by comparing the generated audio file with the target audio file.
5. Send the reward back to the client for updating the neural network.
6. Repeat from Step 3 until the limit of episode length is reached.

In the implementation, a unique strategy is designed for the observation space. As neural networks typically require a fixed-length input, the observation space needs to be padded to have the same length in every step. Hence, in the initialisation stage, I require SuperCollider to generate an empty full-step (16-step by default) long audio file corresponding to the beats per minute (BPM) parameter. The length of this empty file will be set as the *total-length* attribute. In the following steps, though the actual output of the audio file varies in length, it will be padded with zeros to match the total-length attribute. With this strategy, the observation spaces in each step can share the same length.

3.3.2 Air Guitar

The implementation of the Air Guitar project began with the data collection. A total of 36 music students and semi-professional musicians took part in the study, three of which were excluded due to incomplete data. Thus, the dataset consists of data from 33 participants (32 male, 1 female, mean age and the standard

deviation is 27 ± 7 years). All the participants had some formal training in playing the guitar, ranging from private lessons to university-level education. The recruitment was done through an online form published on the website of the University of Oslo, which was announced in various communication channels targeting music students. Participation was rewarded with a gift card (valued approx. €30). The study obtained ethical approval from the Norwegian Centre for Research Data (NSD), with project number 872789.

Recordings took place in the fourMs motion capture lab at the University of Oslo. The audio was recorded at 16-bit 48 kHz using a Universal Audio Apollo Twin audio interface. All participants used the same performance setup: A Sadowsky Semihollow guitar with 11-49 gauge roundwound strings, a 1.5mm Jim Dunlop Tortex plectrum, a Roland AC-40 acoustic guitar amplifier (clean tone with all-flat equalizer settings) connected into the audio interface through the line output. The sound level was set to be comfortably loud for the participant.

Figure 3.4: A participant during the recording session. Motion capture cameras can be seen hanging in the ceiling rig behind, and on stands in front of, the performer. The monitor with instructions can be seen below the front left motion capture camera.

The participants' muscle activity as surface EMG was recorded with two systems: consumer-grade Myo armbands and a medical-grade Delsys Trigno system. The former has a sample rate of 200 Hz, while the latter has a sample

rate of 2000 Hz. Overt body motion was captured with a twelve-camera Qualisys Oqus infrared optical motion capture system at a frame rate of 200 Hz. This system tracked the three-dimensional positions of reflective markers attached to each participant's upper body and instrument. A trigger unit was used to synchronise the Qualisys and Delsys Trigno systems. I have also developed custom software for recording data from the Myo armband in synchrony with the audio. The regular video was recorded with a Canon XF105, synchronised with the Qualisys motion capture system.

Figure 3.5: Placement of the EMG sensors on the arms of the guitarists. Two delsys EMG sensors were placed on each side of the arm, right below the Myo armbands.

In the paper included in the dissertation, only EMG data from the Myo armbands was considered, since the aim was to use the trained model in an interactive music system. Data from the Delsys system, as well as the motion capture data and audio/video recordings, were only used for reference only.

The participants were recorded individually and were asked to carry out the following set of tasks:

0. A warm-up improvisation with a metronome at 70 bpm
1. Task 1

   a) Softly played *impulsive* notes
   b) Strongly played *impulsive* notes

2. Task 2
   a) Softly played *iterative* 16th notes
   b) Strongly played *iterative* 16th notes

3. Task 3
   a) Softly played hammer-ons and pull-offs
   b) Strongly played hammer-ons and pull-offs

4. Task 4
   a) Softly played *sustained* semi-tone bending
      i. 'As fast as possible'
      ii. 'As slow as possible'
   b) Strongly played *sustained* semi-tone bending
      i. 'As fast as possible'
      ii. 'As slow as possible'

5. A free improvisation (the tone features and the use of metronome are at the participant's discretion)

All the given tasks (1–5) focused on the notes B3 and C4 on the 4th (D) string played by index and middle fingers. Each task was recorded as a fixed-form track of duration 2'16", where participants were instructed to play for 4 bars, rest for 2 bars, and repeat the same pattern for 5 more times. All tasks are prompted through a Max/MSP patch on a screen, which allows for a consistent and efficient experiment process.

I developed a custom Python interface[10] to record synchronised sensor data and audio. This was using the previously developed *myo-to-osc* bridge (Martin et al., 2018b), which implements low-latency support for multiple Myo armbands connected via separate Bluetooth Low Energy (BLE) adapters. This was our chosen solution to overcome challenges related to having multiple armbands connected to the same Bluetooth receiver. It also prevented possible data loss due to bandwidth limitations and latency problems.

The data acquisition interface contains three parts: (1) data collection from the two Myo armbands, (2) generation of a metronome sound for the performers, and (3) audio recording using PyAudio. Audio and metronome timeline information was captured alongside the EMG data to simplify the segmentation and organisation of the training dataset.

[10] https://github.com/chaosprint/dual-myo-recorder

To prepare data for the model, I first aligned EMG and audio arrays based on the recorded metronome timeline. Then I applied interpolation on the EMG data and calculated the root mean square (RMS) from the audio signal.

Data recorded from Myo armbands needs to be pre-processed before it can be used for further analysis. This is to compensate for noise and possible data loss during recording. I solved this by performing a linear interpolation on the data. Since the data was recorded at a frequency of 200 Hz, the data loss is usually not more than a few samples. Thus, this additional step to account for the lost data should not create much of an error.

The root mean square (RMS) of the EMG data from the Myo armbands was calculated to reduce the dimension of the discrete signals and to characterise the signal. The RMS of a discrete signal $x = (x_1, x_2, \dots, x_n)^\intercal$ with n components is defined by

$$\mathrm{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + \cdots + x_n^2}{n}}\,. \tag{3.1}$$

Even though it is a simple measure, the RMS arguably has both a physical and perceptual significance. Its physical significance is related to its proportionality to the effective power of the signal. On average, the RMS is also correlated to perceptual loudness. The brain can judge whether a signal is loud, soft or in between, but it cannot infer where a periodic signal is peaking or is at a zero-crossing (Beranek and Mellow, 2012; Ward, 1971). Thus, RMS is a better feature than simply taking just the peak value within a given time interval.

The aim of the developed model is to map the EMG data (raw muscle signals) to the RMS of the instrument's audio signal. Concretely, the input to the neural network is every 50 samples of the EMG recorded from all 16 channels of the two Myo devices (e.g. sample N=0–49, sample N=1–50, etc.). As I use the data from both hands, and each Myo has 8 analogue channels, there are 16 channels for each sample. The output of the neural network is the predicted sound RMS energy on the guitar.

The system was developed based on a Long Short-Term Memory (LSTM) recurrent neural network (RNN) model built in PyTorch (Paszke et al., 2017), a popular model for time-series prediction.[11] As depicted in Figure 3.6, the LSTM network receives the aligned raw EMG data and audio RMS and produces a predicted audio RMS. The training loss function was defined as

$$\begin{aligned}\mathcal{L}(x_{\mathrm{RMS}}, \hat{x}_{\mathrm{RMS}}) &= \frac{1}{n}\|x - \hat{x}_{\mathrm{RMS}}\|_2^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}(x_{\mathrm{RMS,i}} - \hat{x}_{\mathrm{RMS,i}})^2\,,\end{aligned} \tag{3.2}$$

where x_{RMS} are the recorded values, and $\hat{x}_{\mathrm{RMS}}$ are the values to be predicted. The sliding window has size n. The predicted RMS is computed according to Equation 3.1.

[11] https://github.com/cerdemo/air_model

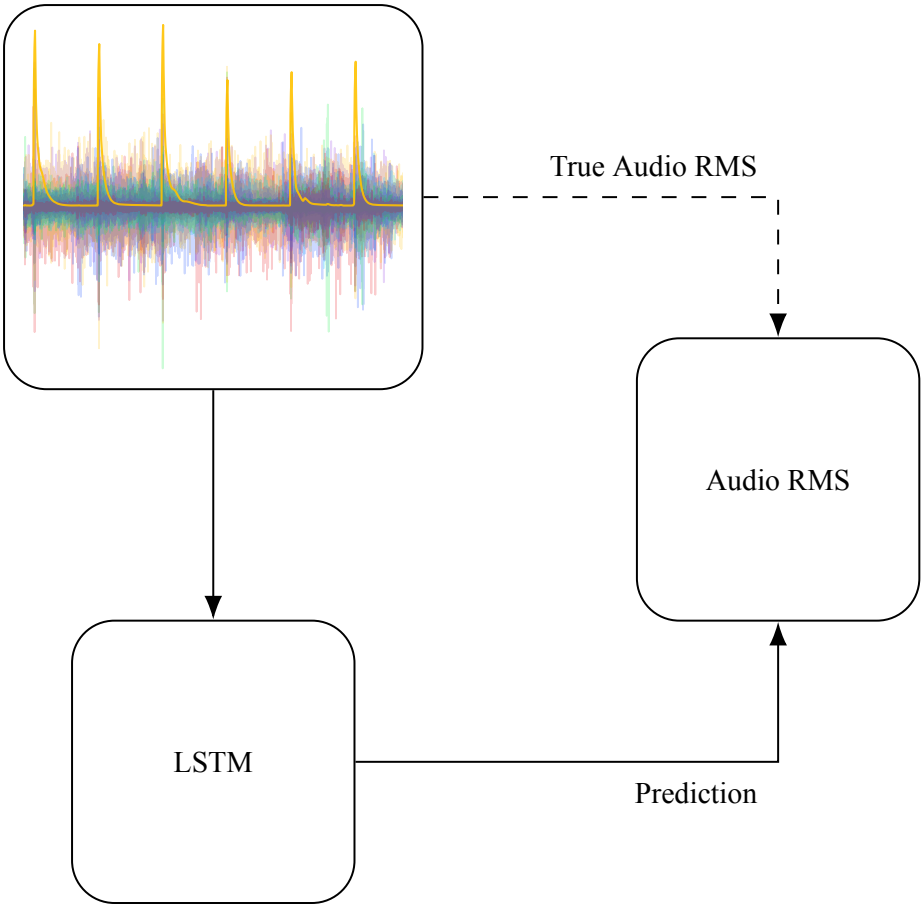


Figure 3.6: Simplified signal flow diagram of the system.

A relatively small RNN was used for the training, consisting of five hidden layers and with 32 LSTM units in each layer. The window size of the input was 50, which is in line with the size of the input layer that was also 50. For the training, I used the data (excluding the improvisations) of 15 subjects out of 20 and validated it on the remaining subjects. I chose a batch size of 100 for determining the gradient of the cost function. Typically, at the first 5 epochs, the loss dropped quickly and became stable after 10 epochs, which took around 3 hours. Overall, I managed to finalise the training within the 12-hour limit of *Google Colab*'s graphics processing unit (GPU) resources.

It turned out that the model was generally capable of predicting the RMS of the audio. This can be seen in the figures of the recorded versus predicted RMS of the tasks of playing impulsive notes (Figure 3.7) and iterative 16th notes (Figure 3.8). For the latter, the model can generate a similar consecutive

energy shape as a series of attacks. The collaborator of this project and I were all positively surprised to see that the model could predict the general trend of the sound energy in free improvisation tasks (Figure 3.9). This is the task that is most relevant for the ultimate goal of creating an 'air instrument' to perform with.

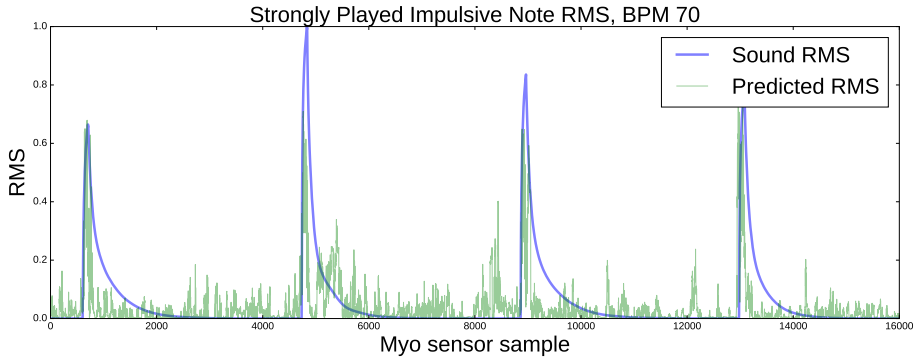


Figure 3.7: The RMS of the recorded sound and the model prediction for the impulsive note playing task.

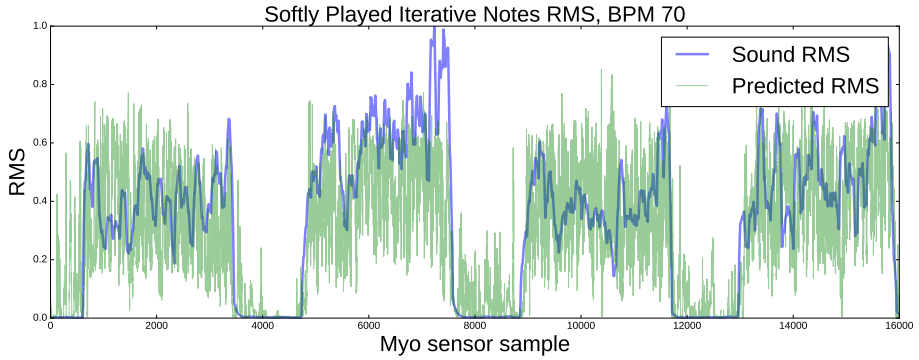


Figure 3.8: The RMS of the recorded sound and the model prediction for the iterative notes playing task.

3.3.3 QuaverSeries

The implementation of QuaverSeries begins with the syntax. As mentioned above, the syntax design of QuaverSeries is based on a functional programming paradigm.[12] The following sections will describe its syntax and how Ohm.js and Tone.js have been used to implement the parsing and semantics.

[12] https://github.com/chaosprint/QuaverSeries

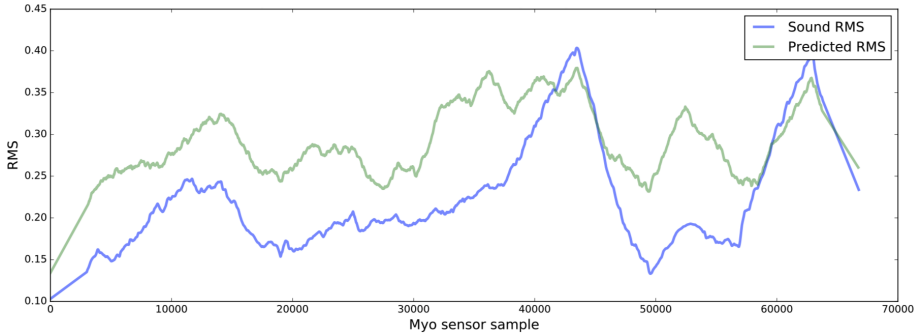


Figure 3.9: The RMS of the recorded sound and the model prediction. Both curves are processed with a Savitzky-Golay filter to reflect the general shape of the RMS comparison.

The representation of musical notes is probably the element that varies the most among live coding languages. QuaverSeries is based on the same concept as most music sequencers. An example note *sequence* looks like this:

```
60  _62  63_64_65_  66_67_68_69
```

This sequence has three elements: numbers, underscores and blank spaces. The numbers refer to MIDI note values, with 60 being the 'middle C'. A blank space indicates a separation of individual *notes*, while an underscore denotes a musical *rest*.

A sequence will always occupy the duration of a whole note, and all the notes will be divided equally. To illustrate, the one-line sequence above will be divided into the notes: `60`, `_62`, `63_64_65_`, and `66_67_68_69_`, with each of them occupying a quarter note duration. Each note can be further divided (equally) by the total number of MIDI notes and rests. In the example above, `_62` means that an eighth MIDI note 62 will be played on the off-beat, after an eighth rest. Likewise, `63_64_65_` means eighth note triplets.

As can be seen from the examples above, the syntax rule refers to the musical sequencer. I have added extra programmability to the syntax using the dividing algorithm invented in TidalCycles (McLean, 2014). One direct influence here is that I form a left-to-right typing flow. For the sake of consistency, this flow is kept in other parts of the syntax design as well. Hence, there is no pairing symbol such as parentheses and quotation marks in the syntax.

The sequence can then be connected to a sound generating module using the double greater-than sign (») which is prevalent in programming languages, e.g. C++. In QuaverSeries, this symbol indicates a signal chain flow, from left to right:

```
loop 20 20 20 20 >> membrane >> amp 0.8
```

The one-line code above will create a 'flat four' kick drum sequence using the oscillator function (`membrane`), and with an amplitude of 0.8 (`amp`). This syntax style naturally leads to the choice of using a functional programming paradigm.

Functional programming languages are prevalent in musical applications. One example is the above mentioned Tidal; Overtone is another, following the syntax of Clojure, a dialect of Lisp language (Aaron and Blackwell, 2013). In a functional programming language, everything is a function in the mathematical sense. For example, for the function `y = f(x)`, `x` refers to the *independent variable*, `y` refers to the *dependent variable*, and `f` refers to the transformation. In a similar manner, to express an electronic music signal flow, I can, for example, write the pseudo code:

```
(lpf (square 440 1) 1000 1)
```

Here the `square` function, followed by `440` and `1` means a square wave oscillator function with two parameters as independent variables (inputs): frequency and amplitude. In this example, `lpf` is the abbreviation of *low-pass filter*, and refers to another function that receives three parameters: the audio signal for filtering, the cut-off frequency, and the Q-value. A pair of parentheses is here used to wrap the function that is passed to the next function as its input. This is typical in functional programming languages such as Lisp.

The same synthesis architecture can be rewritten in an object-oriented programming style as:

```
osc = Square(440, 0.8)
osc.connect(LPF(30, 1500))
```

Here `Square` refers to a *class* with a *constructor*. When the constructor is called, an *instance* will be created, and I can save it as in a variable called `osc`. The instance can call its *methods* predefined in the class.

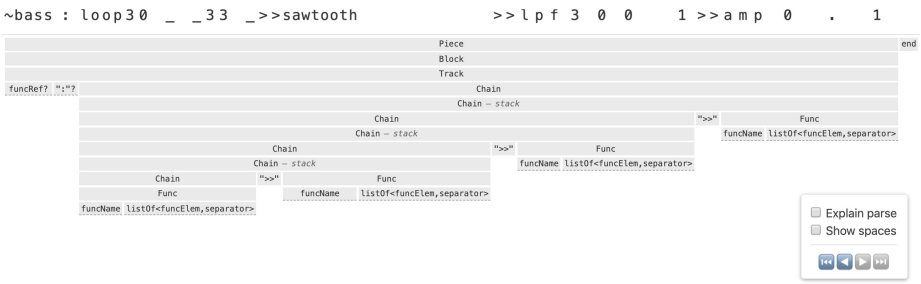


Figure 3.10: An example of an Ohm.js parsing tree. The hierarchy from top to bottom mainly includes: *Piece*, *Block*, *Chain*, *Func*, etc. In this example, the one-line code makes a *Piece*. This *Piece* contains one *Block*, and the *Block* is a *Track* rather than a *comment*. This *Track* can be further divided into a function reference name (*funcRef*) and a function chain (*Chain*). The function chain will be parsed using left recursion, and the semantics definition is written in JavaScript using Tone.js.

Both functional and object-oriented programming paradigms have their pros and cons. However, since I have chosen to start the syntax with a sequencer as the main defining element, I have found it most practical to use functional programming in the syntax design.

In the one-line functional programming code mentioned above, `loop`, followed by a sequence of MIDI numbers is the first function. When followed by `membrane`, the function on the left should become the frequency parameter of `membrane`, with an implicit conversion from MIDI notes to frequency. The `amp` is a function that sends the audio signal generated by the function chain to the audio interface, with a sound level scaling of 0.8. The equivalent Lisp-style would be:

```
(amp (membrane (loop 20 20 20 20) 0.8))
```

The parser in QuaverSeries is built from scratch with the help of Ohm.js. This requires to first program in its domain-specific language, describing how the parser should act. The parser will then generate a parsing tree, identifying the structure of the code (see Figure 3.10). An example of the QuaverSeries syntax may help to explain how it works:

```
bpm 120

~bass: loop 30 _ _33 _
>> sawtooth >> adsr 0.04 0.3 0 _
>> lpf ~cutoff_freq 1
>> amp 0.1

~cutoff_freq: lfo 8 300 3000
```

The whole *piece* in this example can be divided into different *block(s)*, with each block containing at least one function separated by an empty line. The first line is a function for setting the tempo of the piece (120 beats per minute). All the function names are typed in lower case, with an optional underscore in between.

Each function is typically followed by the function elements (*funcElem*). For example, the `adsr` function has four parameters: *attack*, *decay*, *sustain* and *release* of the audio envelope. The usage of the underscore is flexible. Apart from its usage in the note representation (to denote a rest), an underscore can also be used as a Python-style placeholder to keep the default value of a parameter. For instance, the `adsr` function has a value of zero for the *sustain* parameter, which means that there is no need to write the *release* value. Hence, I can use an underscore to represent the release.

The second block in the example above demonstrates a concept called *reference*. With a tilde-prefix (∼), a function name becomes a reference that can link two signal chains with one signal chain modulating a parameter of the other. In the example above, the cut-off frequency of the low-pass filter is modulated by a low-frequency oscillator (`lfo`). Hence, to keep the consistency, it is suggested users add a reference at the beginning of every function chain.

The semantics part of QuaverSeries defines how the code should be executed after being parsed. In Ohm.js, the parsing and semantics definitions are separated. Thus after the parser reads through the code and identifies several valid functions,

Ohm.js needs further instructions on how to deal with these functions. For instance, when the parser detects a number, the parser will return the number as a string character. It is therefore necessary to write the semantic action as a JavaScript function that converts the string to a float in JavaScript, so that it can be used for numerical operations.

The semantics definition of QuaverSeries is written with Tone.js, a JavaScript audio and sound synthesis library based on the Web Audio API (Mann, 2015). Currently, the functions are categorised into three parts: *control*, *effect*, and *synth*. In the semantics definition, each function is organised into different tracks. Each track has its attributes, including *note*, *synth*, and *effect*.

Once a `run` message is received, the parser will read through the whole page, and convert every function to Tone.js code based on the semantics definition. For instance, when the `loop` function is detected, a Tone.js *sequence* instance will be created. Likewise, if a `synth` function is identified, a Tone.js *synth* instance will be created. If audio effects are found, the relevant Tone.js effect instances can be created. Finally when the `amp` is detected, the `connect` method of the synth instance will be called to connect all the effects, with the amplifier (`Tone.Master`) at the end of the effect list.

As a summary, when the `run` command is given, the parser will read through the whole page and identify the functions. Next, a semantics action, i.e. how the parsed code should be processed next, will be executed by constructing Tone.js instances and calling their methods. The `update` command also reads the whole page, and updates each node that is playing, although it will first be effective at the beginning of the next bar.

Collaborative live coding was an important motivation when developing Quaverseries. The aim has been to create a web application that live coders can use to collaborate in different *virtual rooms*. This approach also benefits an audience, who can go to a particular room to watch an ongoing performance, albeit with a different access level (see Figure 3.12).

The implementation of real-time code sharing was simplified through tools and algorithms such as Firebase and Operational Transformation (Ellis and Sun, 1998). Firepad is an open-source tool that mainly uses a Firebase real-time database and the Operational Transformation algorithm. Thus, it provides a solution for synchronising code and sharing the cursor position between clients. In QuaverSeries, Firepad is used to share the code, while a customised strategy is designed to broadcast the related `run` and `update` commands to every client connecting to the database. In this way, a live coder can control the sound running in all the browser clients. This is a similar strategy to what can be found in the Hydra synth, an environment developed for sharing visuals in the browser (Jack, 2019).

In the server database, two entries are storing the states of the `run` and `update` commands. Hence, once a user sends `run` by clicking the button or using the keyboard shortcut, the value of the entry `run` in the database will be set to the Boolean value `true`. As each client connecting to the database has its monitoring function for the value, once the Boolean value `true` is detected, each client will execute a relevant handling function. This function will do two things:

1) execute the code in the editor, 2) set the `run` entry back to the Boolean value `false`. Here is an example to illustrate this in pseudo-code:

```
# the server
if Server get "run":
    send "run" to every client

# the client
if Client get "run":
    execute Music Code

# the interface
if Button "run" is pressed:
    sent "run" to Server
```

The principle of `update` is almost the same as `run`. The only difference is that `update` is used to renew the piece while the music is already on, that is, to calculate the current time and schedule what to play from the beginning of the next bar.

Since only code is transmitted between clients, it is possible to run the system over connections with very limited bandwidth. Furthermore, since the system is based on looped sequences, and an updating strategy per bar, it allows for a considerable network delay without necessarily influencing the final musical result. This system design can, of course, be problematic if an `update` message is sent at the end of a bar. Still, the worst-case scenario is a one-bar offset among different locations. However, this has not been a problem in real-world testing so far.

QuaverSeries is novel in that it focuses on *code streaming*. Thus, instead of watching the audio/video of a performer's screen, the audience can enter a virtual room, watch new code appear on the screen, and have the musical sound rendered locally in the user's own browser. The result is sound with a higher level of fidelity than when streaming compressed audio. The audience can unidirectionally receive the `run` and `update` message from the server. This makes it possible to stop the rendering of music in the local browser at any time without influencing any other instances running on the machines of other performers or audience members.

The main difference between the performer and audience modes is that in the latter the code is not editable. Technologically speaking, though, each audience member is running a complete local version of the instrument, with the performer triggering the code. Thus every audience member could be seen as a collaborator and partaker in the musicking.

3.3.4 Glicol

Developing Glicol in Rust, I had to build different node structures and a language engine that could convert the code text to an audio graph. Also, I had to consider how I could dynamically manage the nodes in a graph during a live coding session. In the audio engine implementation, I ended up using a customised version of

the `dasp_graph` library.[13] The default buffer size in the library is hard-coded to a constant value 64, while in the customised version, the new Rust feature *min_const_generics* was used so that I could set the buffer size to any valid number, e.g. 128 in the Web Audio application. The library provides a template for implementing a *trait* called `Node` for all these node structures (`SinOsc`, etc.). These node structures are all embedded with a method called `process`, which takes an input *buffer* array and outputs a *buffer* array. Within its definition, I have written the DSP code to determine how the output *buffers* should be calculated from the input *buffers*. Thus, the update rate is based on the buffer size, while the control of some nodes can be at the sample level such as in the `delayn` node.

The nodes that support sample playback required some special consideration during the development. In Glicol, I define the behaviour of the`sp` (sample playback) node like this: once it receives a non-zero value from its input—which should be placed at the first position of the incoming block signal array—it will schedule a sample playback inside the node. The playback rate is determined by the trigger's value, which will consequently alter the playing pitch of the audio sample. For instance, the value 1.0 triggers the default playback rate of the audio sample and a value of 2.0 will play the sample one octave higher.

Many nodes can be used to trigger audio sample playback. For instance, the `imp` node (an 'impulse signal node') can send out an impulse signal that triggers a sample playback periodically. The node `seq` takes a sequence of MIDI note values or underscores as parameters. Notes are represented by integers while underscores denote rests (silence). The parser will divide one bar into equal lengths based on the spaces. The default bar duration is 2 seconds, equivalent to 120 beats per minute with a time signature of 4/4. Then, each segment can be further divided into smaller, equidistant sub-segments based on the number of MIDI notes and rests.

To convert a code string to an audio graph, I had to build a parser to process the code. In Glicol, I chose Pest.rs[14] as the parsing tool. It allows defining the language rules in a PEGs (parsing grammar expressions) paradigm (Laurent and Mens, 2015). Next, I call its API to parse the code to node information such as the *reference* of a *chain* of nodes, the name of a single node, and its parameters.

To maintain the lazy evaluation manner, that is, writing first and defining later, the code is parsed first, and the node information is kept in a *Vector* structure (re-sizable arrays in Rust) chain by chain. Then, these vectors are saved in a *HashMap* structure (a dictionary-style data structure in Rust). When parsing each node, the side-chain information (the node index and reference name tuple) is stored in another vector called `sidechain_list`. Finally, the edges are handled only after all the nodes are parsed and the relevant information is stored.

As for the clocking, a `clock` node is connected to all the user-created nodes to ensure synchronisation. The `clock` node is invisible to the users but plays

[13] https://bit.ly/3n2ehfI
[14] https://pest.rs/

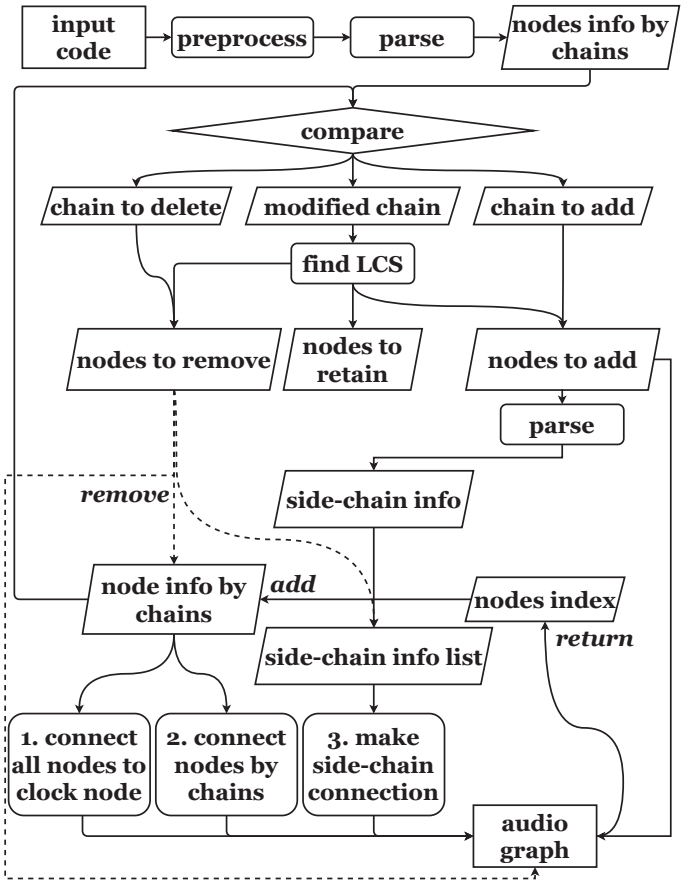


Figure 3.11: The process of converting code string into an audio graph. I come up with a dynamic node management strategy with the LCS algorithm.

a vital role in avoiding over-processing for some nodes used as *references* in more than one place. When the `process` method is called within each node, the internal clock of that node will be compared with the input buffer of the clock node that contains the current clocking information. If it is already processed once, the node should yield a stored output buffer rather than calculate a new one.

In live coding, the audio graph needs to be updated in real-time. In Glicol, I have chosen a WYSIWYG (what you see is what you get) approach as I believe this can help the audiences better understand the code–sound relationship. This means that every time the user runs the code, the audio graph is always dependant on all the current code. However, resetting the entire audio graph every time the update is scheduled would be a dramatically reduce the performance efficiency. It

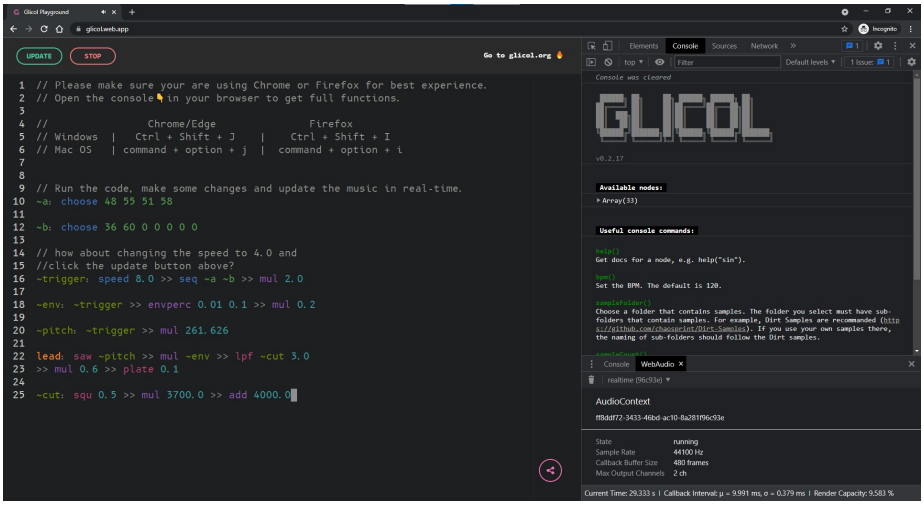


Figure 3.12: The interface of Glicol Web-IDE. The editor has syntax highlights implemented with regular expressions. The browser console is creatively used to provide help documentation and command-based interaction. The WebAudio tab in Chrome DevTools can be used to monitor real-time audio performance.

would also be prone to audio clicks. The solution has been to manage the nodes dynamically using the longest common subsequence (LCS) algorithm (Bergroth et al., 2000). First, the new code is parsed and processed chain by chain. When dealing with a chain, I compare it with the node by chain *HashMap* stored previously. The comparison has three possible outcomes. In the first case, the chain shows in the previous code but not in the new one. Then I will remove all the nodes in this chain from the graph and the side-chain information list. The second case is that this chain is a new one, so I can simply add all the node information to the graph. The last case is that this chain is a modified version in the new input code. Then I use the LCS algorithm to find out which nodes to add and remove, while keeping most of the nodes untouched in the graph.

Taking advantage of the dynamic node management algorithm, I further added a strategy to optimise the audio performance. In the pre-processing stage, the parameter of all the `mul` nodes, and oscillator nodes such as `sin` (sine wave oscillator), is replaced by a control signal that contains a single `const` node. In this way, when the user changes the parameter of these nodes, it is the `const` node that will be removed from the graph and a new `const` node will be added, while the `mul` or `sin` node will remain in the graph. Then, within the `mul` or `sin` node, the previous audio state, e.g. the phase of an oscillator, can be retained. Thus, a smooth transit can be created.

The language and audio engine I have built with Rust is compiled to a WebAssembly module that can run in a browser-based IDE (integrated development environment). To improve the user experience of the environment,

I have implemented syntax highlighting in the code editor. I have also developed documentation in the browser and added support for collaborative coding similar to the approach developed in QuaverSeries.

When users click the `run` icon in Glicol, the code string will first be encoded into UTF-8 format. Then it will be sent to the AudioWorklet thread as an *Uint8Array* with a label `run`, using the `SharedArrayBuffer` feature in browsers such as Chrome or Firefox. Once the AudioWorklet thread gets the array buffers, it will call a function `'alloc_u8'` exported from the WebAssembly module. This function will create memory space for the code string. On the JavaScript side, I use the `array.set()` method to write the array data to the allocated memory location. Then I pass the pointer and the array size to the `run` function exported from Rust/WebAssembly. This function will call the Glicol engine inside to process the code string.

Similarly, the audio samples can be passed to the WebAssembly/Rust module as array buffers. First, users can use the command function `sampleFolder()` in the browser console to load their own local samples. These audio samples will be stored temporarily as a JavaScript *Float32Array*. Then, the sample arrays can also be passed to the Rust/WebAssembly engine using the `SharedArrayBuffer`. Here, sending the audio samples requires the WebAssembly/Rust side to allocate memory locations for both the sample names and the sample data. The audio samples will be stored in a *HashMap* and can be later used in the audio engine.

Users can also use the browser console for communication to the WebAssembly module. Adding samples can be done with the `addSamples(name, URL)` function export to the browser window. Beats per minutes can be set with the `bpm()` function. And the amplitude of each audio node chain can be set with the `trackAmp()` function.

Besides the interface for solo testing and performing, I have also built a decentralised environment to support collaborative live coding. The user only needs to create a 'room' and share the generated link to friends, and then they can start the collaboration. The code is synchronised using CRDT algorithm (Lv et al., 2017). The interface is built with the dependencies of the Yjs project [15]. This interface is later used in the performance mentioned in 4.3.

3.4 General Reflection

The RaveForce project's initial self-testing shows that the environment can report a reasonable reward given the audio files and a proper synthesiser architecture. However, as I have mentioned above, the non-real-time rendering is too slow that it limits the number of iterations the neural network can get in a certain period of time, say 12-hour training. Since the reward feedback is working, shifting to an evolutionary algorithm can be an alternative to reinforcement learning. But still, it may be much more helpful if the non-real-time rendering can be optimised. This is why I chose to shift towards web browsers. It offers

[15] https://github.com/yjs/yjs

an integrated environment and can also be used to explore collaborations with easier access.

The guitar project is mostly a subbranch that is used to try a different angle of machine learning-based AI music research. Instead of building a tool for composition, I was about to build a tool for performing. But limited by the data and the black-box nature of the neural network, only the RMS can be predicted based on the model I built. In fact, the sound RMS already shows some statistical connection with the EMG signal. As a reflection, it may be seen as a detour using deep learning here, especially when it significantly hinders the real-time implementation.

But eventually, I realised that it is essential to build something on the low-level audio for both live coding and AI research and in particular, for their combination in the future. I picked up the Rust programming language, but I did take some detours as I tried to develop with a functional paradigm similar to achieve the syntax of QuaverSeries. I was designing the parser in Rust to convert the node name to functions in Rust and then chain those functions. But this path did not work based on my testing. This is because Rust has a strict restriction on the ownership of different variables, which makes it much harder to use a global variable, or pass around the state, such as the clock, to different functions.

Another issue is that I can do almost anything from the low level, but how should I design the whole language? There are a lot of idioms from the previous languages, while there is also a rule in the programming domain that we should not 'reinvent the wheels'. The Rust project *dasp* [16] inspired me a lot, so that I was then able to solve this issue. I realise that I should follow Rust's design philosophy to explore a new angle for live coding and audio programming. In this way, I landed on the graph-oriented paradigm. This new idea brings questions such as how the audio graph can be efficiently updated in real-time. Fortunately, the application of LCS (longest common subsequence) algorithm application solved the issues.

As a practice-based or artistic research (Balkema and Slager, 2004), the method used here is quite unique from the previous ones in that a considerable amount of time is invested into the audio programming, data gathering and machine learning model training. The time for performing and workshops are relatively traded off. Nevertheless, the influence of COVID-19 should be taken into consideration as well. In general, the interactive method in this thesis can be viewed as a process for gaining the 'entry ticket' for collaborative performance, which can be crucial for the reflections in the later chapters. In this process, the main takeaway from the methodological reflection is that for the collaboration, either human–human or human–computer, I found it significant to rethink music interaction design from the low-level audio programming aspect. Understanding the whole process of how signals and symbols work can support the design of the higher-level interaction. Otherwise, risks and unpredicted timing issues will

[16] https://github.com/RustAudio/dasp

emerge in performing practices, which will be further discussed in the coming chapters.

Chapter 4

Research summary

4.1 Introduction

In this chapter, five papers on the four projects in this thesis will be summarised and discussed. In addition to the papers included in the thesis, I will present and discuss some performances using the developed QuaverSeries and Glicol live coding environments.

4.2 Papers

Paper I

Qichao Lan, Jim Torresen and Alexander Refsum Jensenius "RaveForce: A Deep Reinforcement Learning Environment for Music". In: *Proceedings of the SMC Conferences. Society for Sound and Music Computing.* (2019), pp. 217–222.

Abstract

RaveForce is a programming framework designed for a computational music generation method that involves audio sample level evaluation in symbolic music representation generation. It comprises a Python module and a SuperCollider quark. When connected with deep learning frameworks in Python, RaveForce can send the symbolic music representation generated by the neural network as Open Sound Control messages to the SuperCollider for non-real-time synthesis. SuperCollider can convert the symbolic representation into an audio file which will be sent back to the Python as the input of the neural network. With this iterative training, the neural network can be improved with deep reinforcement learning algorithms, taking the quantitative evaluation of the audio file as the reward. In this paper, we find that the proposed method can be used to search new synthesis parameters for a specific timbre of an electronic music note or loop.

Discussion

This paper presents Raveforce as a new conceptual framework in sound design with artificial intelligence. In addition, the relevant software is provided as proof of concept. The paper begins with a research gap I identified in 2018: that research on music generation was limited to either symbolic (MIDI notes) or sub-symbolic (audio) generation. In real-world contexts, we often see a combination of symbolic and sub-symbolic approaches. Therefore, I found it meaningful to develop a solution in which AI-based sound/music generation system could

work with both approaches. The challenge, then, was to consider the forms of representation and choose relevant algorithms for implementation.

The main limitation of RaveForce comes from a technological point of view and is related to this thesis' second sub-question (RQ2). Consider the time spent on one iteration for sound generation: first sending an OSC message from Python to SuperCollider, then waiting for SuperCollider to render a sound file on the hard disk, sending back the file's path to Python, and finally reading the generated audio from the hard disk in Python. This can be a very time-consuming process, especially when writing on and reading from the hard disk. This process can be greatly accelerated if the rendering is done in memory (RAM) rather than on the hard disk (ROM). However, this requires more knowledge about the inner workings of music programming languages, and even the programming languages that the music programming languages build on. It was this realisation that paved the way for much of the other research conducted during my PhD research.

Another challenge with RaveForce is the non-real-time nature of the environment. Considering the current state-of-the-art of libraries and platforms, it is inevitable to build such a system for non-real-time processing. Even with higher computational power, the uncertain number of iterations for sound shaping and generation can still make the waiting time uncertain. However, it may be possible to integrate this uncertainty and waiting time into the creative process, e.g. using it in a 'semi-real-time' mode in loop-based music.

Finally, looking back at RaveForce some years later, I realise that its scope and usage could have been better described in the original paper. Calling it a 'music generation environment' was more based on the long-term aim than its current realisation. Perhaps it could be better described as an 'OpenAI Gym-style sound synthesis parameter finder'. Also, the system should not be limited to reinforcement learning. There are other algorithms (such as evolutionary algorithms) that can also be used to find parameters. In the use case presented in the paper, I pointed out that finding parameters for some particular timbre with a given synth structure can be a very concrete and realistic goal. On the other hand, searching for machine creativity can be hard to evaluate. For example, if we compare the machine learning model with a fine-tuned rule-based system, which one should be viewed as more creative?

The development of RaveForce made me reflect on the possibilities and limitations of selected platforms. The combination of Python and SuperCollider worked well as a prototype. However, it turned out to be cumbersome when thinking about scaling up for collaborative usage. This led me to look for a more integrated environment, which ended up being the web browser solutions that I later explored in QuaverSeries and Glicol. Also, RaveForce encouraged me to explore the combination of symbolic and sub-symbolic representations of sound and music.

Paper II

Qichao Lan and Alexander Refsum Jensenius "QuaverSeries: A Live Coding Environment for Music Performance Using Web Technologies". In: *Proceedings*

of the International Web Audio Conference (WAC). NTNU, (2019), pp. 41–46.

Abstract

QuaverSeries consists of a domain-specific language and a single-page web application for collaborative live coding in music performances. Its domain-specific language borrows principles from both programming and digital interface design in its syntax rules, and hence adopts the paradigm of functional programming. The collaborative environment features the concept of 'virtual rooms', in which performers can collaborate from different locations, and the audience can watch the collaboration at the same time. Not only is the code synchronised among all the performers and online audience connected to the server, but the code executing command is also broadcast. This communication strategy, achieved by the integration of the language design and the environment design, provides a new form of interaction for web-based live coding performances.

Discussion

In the previous paper on RaveForce, I mentioned that it was inspired by the process of music live coding: an AI agent was trained to imitate what humans do in a live coding setting. However, using SuperCollider as the audio engine in RaveForce limited its potential for further experiments. Thus in this paper on QuaverSeries, I introduced how I developed a new live coding environment. The web platform was chosen for its easy access and the potential for collaborative music-making. The successful implementation of QuaverSeries made me able to perform several concerts and run workshops to test it in real musical settings.

Looking back at QuaverSeries, I see that it came out of needs found when developing RaveForce. I also learned a lot that eventually led to the development of Glicol. This in many ways summarises the iterative process I have employed throughout all my thesis work. As a middleware towards the realisation of Glicol, QuaverSeries seems to be in an unnecessary position. However, by the time when QuaverSeries was developed, some important dependencies for Glicol were not published yet. Thus, using the raw Web Audio API appeared to be the best way to provide a proof-of-concept for my idea.

When designing a live coding environment, there is a lot of consideration on the aestheticism of the code. I was designing the language similar to how one would build a musical instrument. However, instead of putting together physical objects, I thought about the syntax as ergonomics. For example, I decided to abandon the use of parentheses which minimises the characters to be typed. I also provided a left-to-right writing style imitating the signal flow found in signal processing chains. Still, I tried to maintain readability of the code by using semantic symbols. However, the biggest challenge is that although such a system is designed to enhance the user experience, if it does not align well with the programming paradigm and the audio engine on the low level, such a syntax can be problematic. For example, in functional programming, the meaning of each parameter can be unknown to most users, which will make it hard to learn.

Another experimental part of QuaverSeries is its collaboration environment. I designed it as a 'flat' one in which all users can control the sound and the code of each other. This raised some questions after the publication of this paper and after more experience with using the environment in real-world settings: are some music programming languages more suitable for online collaboration than others? In online collaboration, different performers need to consider the actions of one another. In the context of live coding, the actions include the code input and the code execution. Errors in the typed code will inevitably occur. An important question, then, is how the environment handles such errors. This became an important topic leading to the discussion part of Paper V.

Paper III

Cagri Erdem, Qichao Lan, Julian Fuhrer, Charles Patrick Martin, Jim Torresen and Alexander Refsum Jensenius "Towards Playing in the 'Air': Modeling Motion-Sound Energy Relationships in Electric Guitar Performance Using Deep Neural Networks". In: *Proceedings of the 17th Sound and Music Computing Conference*. (2020), pp. 177–184.

Abstract

In acoustic instruments, sound production relies on the interaction between physical objects. Digital musical instruments, on the other hand, are based on arbitrarily designed action–sound mappings. This paper describes the ongoing exploration of an empirically-based approach for simulating guitar playing technique when designing the mappings of 'air instrument' designs. We present results from an experiment in which 33 electric guitarists performed a set of basic sound-producing actions: impulsive, sustained, and iterative. The dataset consists of bioelectric muscle signals, motion capture, video, and audio recordings. This multimodal dataset was used to train a long short-term memory network (LSTM) with a few hidden layers and relatively short training duration. We show that the network is able to predict audio energy features of free improvisations on the guitar, relying on a dataset of three distinct motion types.

Discussion

See joint discussion for Paper III and IV below.

Paper IV

Cagri Erdem, Qichao Lan, and Alexander Refsum Jensenius "Exploring relationships between effort, motion, and sound in new musical instruments". In: *Human Technology*. (2020), pp. 314–347.

Abstract

We investigated how the action–sound relationships found in electric guitar performance can be used in the design of new instruments. Thirty-one trained guitarists performed a set of basic sound-producing actions (impulsive, sustained, and iterative) and free improvisations on an electric guitar. We performed a statistical analysis of the muscle activation data (EMG) and audio recordings from the experiment. Then we trained a long short-term memory network with nine different configurations to map EMG signal to sound. We found that the preliminary models were able to predict audio energy features of free improvisations on the guitar, based on the dataset of raw EMG from the basic sound-producing actions. The results provide evidence of similarities between body motion and sound in music performance, compatible with embodied music cognition theories. They also show the potential of using machine learning on recorded performance data in the design of new musical instruments.

Discussion

Papers III and IV were based on a collaborative project that ran parallel to my development of QuaverSeries. In the Air Guitar project I continued with ideas explored in the development of RaveForce relating to the use of AI in music. This included some fundamental questions about real-time v.s. non-real-time processing and symbolic v.s. sub-symbolic representations.

Live coding practice is often seen as a 'disembodied' way of performing music. I agree that typing on a keyboard is different from playing with large gestures on stage. There is also less direct control—hence perceived causality—between bodily actions and musical sound. This made me curious to understand more about relationships between sound-producing actions and the resultant sounds on a semi-acoustic instrument: the electric guitar.

In the Air Guitar project, we explored relationships between action and sound by looking at correlations between the muscle signals (captured as EMG) and the resultant sound. In terms of signals, this a relationship between two sub-symbolic signals, both of which are highly complex. The question is how to work with such sub-symbolic signals and utilise advanced machine learning techniques to create mappings between muscle data and sound? And, when we use machine learning in this way, how does it differ from a rule-based system with artificial randomness? For example, we could have read the RMS shape directly from the Myo and added some randomness to it, which would still provide some kind of creative 'unexpectedness'.

From the very beginning of this project, in the data-gathering phase, the latency was a main challenge. The Myo receivers are not capable of reliably reading two sensors simultaneously. Based on my newly acquired programming knowledge from RaveForce, I successfully made a Python program that could do multi-thread reading. Without it, there would not have been the later experiments.

One core challenge that emerged after training the neural network with the guitar sounds, was how to use the model for musical interaction. By the time paper III was written, the model worked only in non-real-time. There are two solutions to this problem. The first is to write better code at a low level. The other is to use 'creative' mappings, and build the waiting time into the musical process. Both of these solutions inspired the development of Glicol.

Paper V

Qichao Lan and Alexander Refsum Jensenius "Glicol: A Graph-oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet". In: *Proceedings of the International Web Audio Conference (WAC)*. (2021)

Abstract

This paper introduces the new music live coding language Glicol (graph-oriented live coding language) and its web based run-time environment. As the name suggests, this language is designed to represent directed acyclic graphs (DAG), using a syntax optimised for live music performances. The audio engine and the language interpreter are both developed with the Rust programming language. With the help of WebAssembly and AudioWorklet, this language can run in web browsers. It also supports co-performance with the support for collaborative editing. Taking advantages of the Rust programming language design, the run-time environment is both safe and efficient. Documentation and error handling messages can be accessed in the web browser. All in all, we see Glicol as an efficient and future oriented language for collaborative text-based musicking.

Discussion

Glicol can in many ways be seen as the 'conclusion' of this thesis. It embeds knowledge from all the previous projects in its pursuit for tools for collaborative live coding. In addition, as the development of Glicol continues, it pushes the exploration of various conceptual aspects to new levels.

Glicol provides a new approach to low-level audio programming and designs the collaborative live coding environment based on this low-level architecture. This could not have been achieved using methods from previous projects. This new approach allows for collaboration where all participants share the same coding spaces and can modify the whole audio graph together.

With more possibilities, such a low-level method also brings challenges. To design a live coding environment from the low level meant that I had to decide whether to use the client-server architecture of SuperCollider or use a brand new paradigm for the audio engine. Still, from today's point of view, a WYSIWYG approach is better for solo performance. It is easier for the performers and the audience to establish the visual connection between the code and the sound.

On the other hand, the WYSIWYG paradigm requires more work on the low-level side. Applying LCS algorithms to dynamic audio graph updating is a

milestone for this project, and it is also innovative from an audio engineering perspective. Not only does it solve the updating glitch issue, but it can also help maintain the oscillator phase and create a smooth transition if the changes happen only to the *mul* (multiplication operation) node. Without dynamic audio graph updating, the paradigm of WYSIWYG cannot be realised without causing audio glitches.

Error handling is also a concern for online collaboration. With the robust error handling mechanism in Rust, the errors from the programming aspect can mostly be captured and handled. But there may be issues in online collaboration if each collaborator can run the code at any time. For example, imagine one performer that wishes to run a piece of code while another performer is still typing. The latter may plan to type a frequency of 70, but the code is executed when only the number 7 is entered. This kind of situation may have unexpected consequences and will need to be explored both technologically and conceptually in the future.

4.3 Performances

As part of my iterative development cycle, all the tools mentioned above have been used in both small and large performances and in several teaching and workshop contexts. In the following, I will reflect on three performances.

Oslo World Music Festival, Gestural Control with SuperCollider and Live Coding with QuaverSeries, November 2019

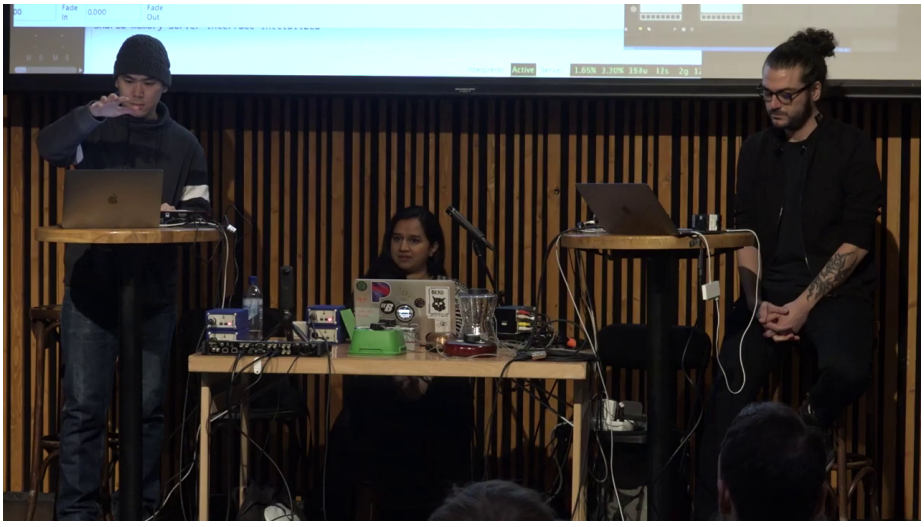


Figure 4.1: Performing at Oslo World Music Festival, Gestural Control with SuperCollider and Live Coding with QuaverSeries.

This performance was in November 2019, at the Oslo World music festival[1], in collaboration with two other artists, Tejaswinee Kelkar and Cagri Erdem. Kelkar performed with vocals and a set of acoustic objects. Erdem played with a muscle-controlled new digital instrument building on ideas from the Air Guitar study. I performed with a gesture-controlled SuperCollider interface I built and live coded with QuaverSeries.

This was the debut of QuaverSeries, at a time when the collaborative part of it was not yet developed. Even though there was no collaborative coding in this performance, the browser-based environment still required an Internet connection. Unfortunately, there was a Wi-Fi dropout during the performance. The instant solution was to fade out the sound using the physical buttons on the audio interface. The episode can be compared to a guitar string breaking during an improvisation. In the case of QuaverSeries, the failure exposed the 'materiality' of the network. As a performer on stage, I had to respond to the problem just as if it had been a physical problem.

WAC 2019, Collaborative Live Coding with QuaverSeries, December 2019

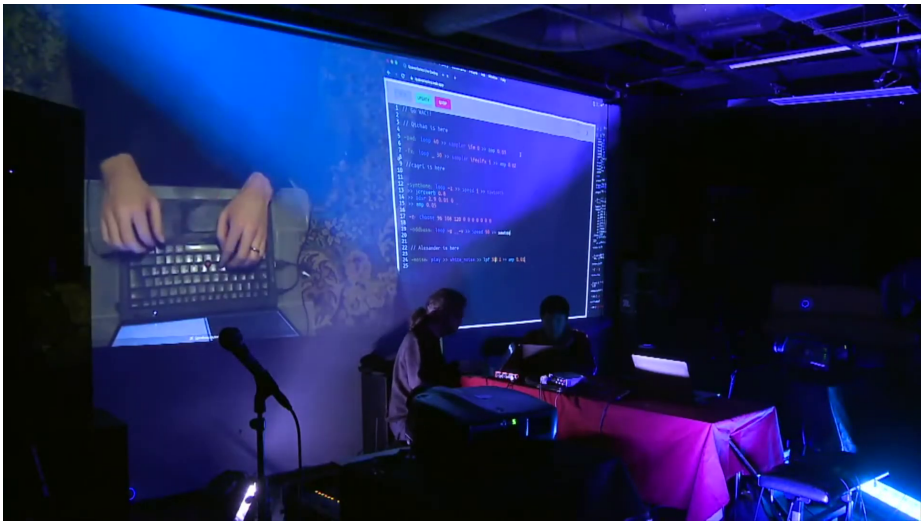


Figure 4.2: Performing at WAC 2019, Collaborative Live Coding with QuaverSeries.

This performance was at the Web Audio Conference 2019[2]. The performance was a pure music live coding session without any visuals, based on a real-time collaboration between Stockholm and Trondheim. This was the debut of

[1] https://youtu.be/dYu55YZJH_s
[2] https://youtu.be/qnEiHg6ljTk

QuaverSeries' collaborative functions, and we had planned and rehearsed the general structure of the performance. It went well for the first 20 minutes but ended with a software bug that caused the music to stop suddenly. Incidentally, the crash happened at a time in which it was quite unnoticeable for the audience. Still, it influenced the performers and the final musical result.

I later solved the problem leading to the crash. However, it brings about a relevant question: is it necessary—or even wanted—to fix all such errors? One view is that mistakes should be accepted in a performance. Some music genres are focused on demonstrating virtuosity by avoiding errors. Others can tolerate mistakes and even build the performance around them. If we develop software that effectively rules out all possibilities of errors to happen, we may also loose out on both 'speed' and 'resolution'. This may eventually also make the environment less expressive and fun to perform.

The exploration of the collaborative aspects of QuaverSeries in various performances also exposed more instances of the material nature of the network. It is possible to explore this and even encourage errors. For example, in ixiLang (Magnusson, 2011), there is a 'suicide' function that comes with a probability to crash the whole program. As live coding is a relatively young music practice (Kirkbride, 2020), the borders between 'accepted' and 'unaccepted' mistakes can be explored in the future.

WAC 2021, Collaborative Live Coding with Glicol, July 2021

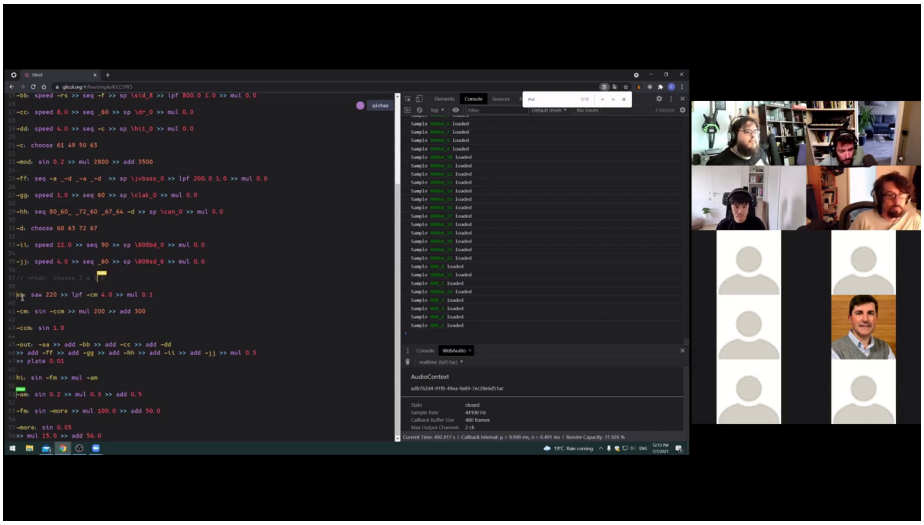


Figure 4.3: Performing at WAC 2021, Collaborative Live Coding with QuaverSeries.

This performance for Web Audio Conference 2021 was held online via Zoom[3]. In this performance, the original plan was to invite participants from the Glicol workshop held at the conference to join this performance (Lan and Jensenius, 2021). However, due to technical errors on Zoom, the workshop link was not accessible to the participants. As a compromise, I was performing solo for the first half of the session and made the rest as an 'open-mic' one.

There are two reasons this performance was special. First, like other performances during the COVID-19 pandemic, it was carried out completely virtually. Secondly, in this performance, I as the performer, allowed the audience to modify the code in real-time. I utilised the new functionality of Glicol with two levels of performers: primary and secondary. Everyone can add code, but only the primary performer can run it. As the primary performer, I was in charge of when and whether to run the code. It was an interesting experiment, but since most of the audience were new to the syntax the performance turned out to be closer to a tutorial in which the audience introduced errors in the code. It could be compared to playing on an open grand piano in which the audience insert random objects that modify the sound.

Even though the performance ended up differently than expected, it showed the potential of this type of musical collaboration. In past performances, I have often co-performed with people that know the syntax and where it is possible to collaborate on equal terms. In a setting where people with limited experience are allowed to participate, the dynamics change radically. I find it intriguing to explore how people can participate in a musical performance in such a novel way. It was also interesting to notice the different errors that may emerge and how I can improve the system to account for such errors. As discussed above, it may also be that such errors could actually inspire the primary performer in the moment. As such, one can see the audience participation as a 'random number generator'.

[3] https://youtu.be/Ep5J97MDt7A

Chapter 5

Discussion

After going through the background of the projects (Chapter 2), my methodological approach (Chapter 3), and research contributions (Chapter 4), it is time to analyse the results and revisit the research questions posed in Chapter 1.

5.1 Addressing the Research Questions

The main research objective of this thesis has been to investigate how new technologies can alter the concept of collaboration in computer music systems. In this section, I will begin by addressing the sub-questions first and then conclude by reflecting on the main research question.

RQ1: What kinds of relationships can be found in collaborative computer music systems and how can new relationships be designed?

Two topics have emerged from the exploratory work I have carried out over the last few years: time and control. Based on these two criteria, different relationships can be identified (See Figure 5.1). Let us start with a case where at least two humans are performing in collaboration with a computer music system. They can establish either a *co-performing* or a *hierarchical* relationship. In a co-performing relationship the performers can be seen as being on an equal level in musical and technical skills. In a hierarchical relationship, on the other hand, there may be one primary performer with more knowledge than the secondary performer(s). As discussed in the context of Glicol, these two models show differences in terms of the order that commands are sent to the computer. In a co-performing relationship, each performer can have an equal opportunity to send any command at any time to the computer. In the hierarchical relationship, the primary performer tends to send the executive command, which is always the latest one on the timeline. Yet, since there is a hierarchical order, the primary performer can decide whether to send the command directly. For example, the primary performer can only give oral commands to all the secondary performers who share a sub-co-performing relationship. In this case, the primary performer can lose some control: the secondary performers may misunderstand the idea and lead the performance to an unwanted direction from the primary performer's point of view. Thus, in computer music systems, the performer who is in charge of the time and order of command tends to have more control over the performance.

When it comes to co-performance between humans and computers, the relationships become trickier. Still, time and control are important here as the computer can be on the right of the timeline. To have the final control, a human

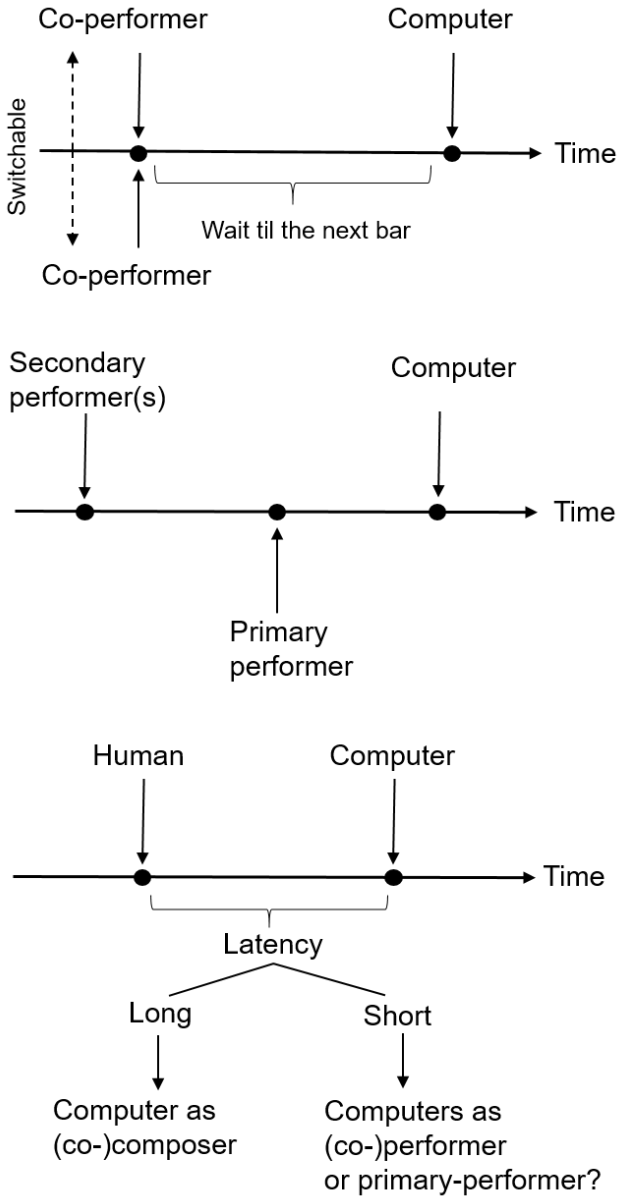


Figure 5.1: Three models for rethinking collaboration.

performer can add another step to the chain and listen to the sound of the computer before deciding whether to use it or not. This will put the human performer on the right side of the timeline again. However, in most cases, one can think of an iterative loop within which humans and computers interact and make the sound in turn. This could be characterised as a flat co-performing relationship. When the latency between the human action and the computer execution becomes longer, the relationship can gradually shift to a co-composing one, as can be exemplified from the live coding practices in this thesis and previous literature (Magnusson, 2015).

RQ2: How can different kinds of relationships in collaborative computer music systems influence the performing practices?

Different human–human relationships and human–computer relationships should be combined to analyse the benefits and the risk of each relationship. In general, a flat relationship between humans tends to be riskier than a hierarchical relationship. QuaverSeries can be an example here. In this environment, all the performers can have equal power to send commands to the remote server to control all the clients. This leads to a flat relationship. To control the risk, the performers may choose to form a hierarchical relationship by themselves. This is what was planned in the WAC 2019 performance, with one performer taking the lead.

The same rule goes for the relationship between humans and computers. As discussed in relation to RQ1, instance feedback tends to create a flat relationship between humans and computers. Making sound in turn between humans and the computer can also be considered a flat relationship. In live coding, the humans give a command and wait until the command is processed by the computer. In this case, the computer shows dominance. When the computer receives the command, it can determine the final sonic result, be it a smooth playback or an error. The designer of the error handling mechanism also has a strong impact in this case.

RQ3: What can new collaborative paradigms bring to the design of computer music systems?

In answering RQ1 and RQ2, I have elaborated on how time and control can determine the paradigm for collaboration. When designing new computer music systems, it is necessary to consider how different time relationships correspond to power distributions. Here the executing time is pivotal. In live coding, the tradition is to execute the command at the beginning of a new bar. This makes sense from a human perspective. However, from a low-level audio aspect, the computer's unit of operation can be as little as one audio block.

Time can also influence the control. For example, if the collaborative computer music system is designed with an instant feedback, then executing order does not matter anymore, and all the performers are co-performers. This type of performance can also be thought of more as a case of 'instantaneous composition'.

The performance is not so much about *performing* in real-time but about creating sonic/compositional structures that can be executed within the blink of an eye.

Both the RaveForce and Air Guitar projects considered temporal limitations. The non-real-time rendering speed limits the former, and the real-time calculation of neural networks restrains the latter. Both issues can be alleviated in the future with increasing computational power and more efficient algorithms. Today, these systems currently have an unacceptable delay from a real-time perspective. However, one can imagine that this delay can be shortened to one or two musical bars and, finally, to a duration that is unnoticeable to the human ear. As the latency level change, the collaborative paradigm will change accordingly.

The accessibility on the software level can also influence control. As the designer and developer of the various systems presented in this thesis, going to the low level has already given me a new level of control that also influences my relationship with the computer. As a developer–performer, I am able to perform differently than someone without as detailed knowledge of the inner workings. In some cases, I could even switch to solving low-level problems during a performance. Yet, this power of control comes with a great deal of risks.

5.2 Reflections on the General Research Question

After discussing the specific research questions (RQ1–3), let us return to the main research question of this thesis:

> **How can different levels of abstraction in music/audio programming influence collaboration in computer music systems?**

I will try to answer this question by using the 5W1H model often found in journalism (Waisbord, 2019): Who, What, When, Where, Why, and How.

Who?

I would argue that the designer of a computer music system—such as a collaborative live coding environment—should be viewed as a musical actor akin to instrument builders, composers, and performers. Music software designers often make many choices that influence the aesthetics of the final musical output. Making the overall syntax and sonic possibilities of a music environment, are some example, as well as the error handling in a live performance. The error handling can be seen as a real-time performance, in which the computer 'rescues' a situation based on algorithms. These algorithms have been programmed prior to the performance, but they act during the performance. Music software designers, therefore, become more active as a collaborator also in the performance. The evolution of technology facilitates this trend. The computer embeds more and more complex abstraction of programming (and musical) knowledge and the low-level implementation is hidden to most of the users.

	Input	Output
RaveForce	Symbolic (synth structure) and sub-symbolic (audio for environment)	Symbolic (synth parameters) and sub-symbolic (sound)
Air Guitar	Sub-symbolic (mucsle signals)	Sub-symbolic (Predicted sound RMS)
QuaverSeries	Symbolic (code)	Sub-symbolic (sound)
Glicol	Symbolic (code)	Sub-symbolic (sound)

Table 5.1: An overview of different representation types in the various environments.

Live coding is a special case, in which the designer of the computer music system may also perform live. This allows for different types of musicking than when only being the user of a system. For example, since I know Glicol and QuaverSeries so well, I can even commit errors, expecting the computer to take control and rescue the situation.

What?

Designing digital instruments is about creating action–sound mappings (Jensenius, 2022), that is, relationships between input actions and output sounds. Designing other types of computer music systems can be seen as a kind of *meta-mapping* in which there are more loose relationships between what you do and what you get. The systems I have worked on in this thesis can be seen as such a type of meta-mapping. As opposed to developing an instrument with a high level of causality between action and sound, I have worked on creating musically relevant mappings between various types of inputs and outputs (see Table 5.1).

I have been particularly interested in exploring the possibilities (and limitations) of working between symbolic and sub-symbolic representations. Both these types of representations have been used in musical AI over the years. In the beginning, researchers and composers often worked with symbolic representations (music as scores). In later years, machine learning has allowed for working more directly also with sub-symbolic representations (music as signals). I have worked with both approaches in the different projects.

Working with both symbolic and sub-symbolic representations raises some interesting questions. For example, when using the shape of the muscle signal in the Air Guitar project , it can be questioned how much it differs from a direct rule-based mapping with some artificial randomness added to the output. However, when the sub-symbolic is mapped to a symbolic output, such as to categorise the muscle signal into musical notes, then it can be hardly replaceable with rule-based ones. What I have come to see is that there are fuzzy borders between working with symbolic and sub-symbolic representations. Instead of insisting on separating the two approaches I think there is a great potential in

exploring how they can be used together. No matter how technology evolves, however, it is essential to always examine the relationships between the inputs and outputs of a computer music system.

When?

As discussed earlier, it is crucial to evaluate the timing of events, since the timing is important to understand the relationships in a collaborative setting. When it comes to timing, I believe that order and latency are the most two important factors. In answering RQ1, I illustrated how different order can determine the control in the collaboration. At the same time, the latency can influence the power structure. The latency can be related to technical limitations, which I experienced in RaveForce and the Air Guitar projects. It can also relate to design considerations, such as in the way time is handled in QuaverSeries and Glicol. When the latency is very short, an action may have an instant influence on the music and therefore have a higher level of control. As technology improves, the latency may become shorter, and may eventually reach a threshold that may be lower than our perceptual threshold.

The discussions about time also boil down to a question of the role one takes in the music-making process. As illustrated in Figure 5.1, when there is a short latency one acts as a performer. However, if the latency becomes long, the musical control may be closer to that of a composer. A composer typically creates structures that are later performed. Live coding is about exploring such differences between 'real-time composition' and 'non-real-time performance'.

Where?

One of the aims of developing QuaverSeries and Glicol was to allow for distributed collaborative performance. During the pandemic, many musicians began exploring performing together online. Most have streamed audio/video over the network and have struggled with long latencies and poor sound quality due to compression. My approach to combining symbolic and sub-symbolic representations allow for an entirely different approach to network-based musicianship. Combining cutting edge audio solutions in web browsers with the transmission of only packages of symbolic information, has demonstrated a different approach to musical collaboration. I think there is a promising future for such live coding platforms. Web browsers also allow for new types of collaboration, such as the decentralised state sharing that I have experimented with in the Glicol interface. Yet, how to communicate with performers in remote locations is worth exploring more in the future.

Why?

I have already mentioned many aspects why I believe it is interesting to explore collaboration in computer music systems. One more aspect is related specifically to the practice of live coding. This performance practice is often considered

to be slow, as performers often spend several minutes of building up the sonic and musical structures (Kirkbride, 2020). With more collaborators, there will be more layers typed into the engine at the same period of time. This can also result in more risk-taking and unexpected musical results, although this fits the improvisation nature of live coding quite well. Live coding is an experimental art form that continues to push the borders of new technologies. It is therefore in a good position to explore new directions, such as the use of AI in music. In my project, I was unable to embed the deep learning approaches used in RaveForce and Air Guitar into QuaverSeries and Glicol. Still, I have done much of the leg work that is necessary to succeed in making such connections in the future. As technology improves, there will be many possibilities of using AI to collaborate in real-time for live coding.

How?

Throughout the thesis, I have shown the possibilities of some new and powerful technologies, such as the Rust programming language, the Web Audio API, and the collaborative (and decentralised) text editing made possible in modern web browsers. The main takeaway is to understand how low-level software development can greatly facilitate the collaboration. Whatever new stack of technology emerges, the key for designing collaborative computer music systems is always to understand as much as possible of the abstractions made by the computer.

5.3 General Discussion

This thesis has described my journey from exploring a live coding inspired human-computer collaboration system to the development of browser-based live coding systems designed from the audio sample level. I began with a question about how technology can encourage collaboration. This does not imply that is impossible to encourage musical collaboration with new technologies. However, new technologies allow for investigating new approaches to music-making. My contribution has been the development of various prototype systems that explore new paradigms through trial-and-error experimentation. Glicol can be seen as the final 'product' coming out of the research. However, I would not have gotten the motivation and direction for developing Glicol without RaveForce and QuaverSeries.

Although I have spent much time and effort on programming, I still see this project as humanistic and artistic in nature. I have been inspired by Magnusson (2019), who writes in his book *Sonic Writing*:

> ...from the perspective of musicology, we ought to establish a discipline that builds up methods, concepts, and language to analyse algorithmically generated musical works. Such musicology would study algorithms as musical material, inventions notated through the medium of code, and apply diverse techniques, ranging from early

> symbolic AI, rule-based systems, and expert systems to artificial life and contemporary deep learning neural networks.

My work has been trying to tackle this divide between different musical representations and explore how it can be used in real-world contexts. For example, when I began developing Glicol, one question was on my mind: why should we work with functional or object-oriented paradigms? On the one hand, there is a need for readability and idioms. On the other hand, the architecture of the computer needs to be considered, such as the memory efficiency, memory safety, etc. After delving into the details of low-level audio programming, I have seen that the audio graph may actually work better, both technically and conceptually. I would not have been able to think outside the box in such a manner without getting into a lot of technical detail. Yet, the trade-off with this approach is the massive volume of audio engineering that it requires. I am happy that I succeeded in the end, but there were times where I questioned the risk of such an approach.

One of the reasons I believe this project has been successful, is the iterative approach taken. By employing an agile development practice, I have been able to continuously test the various software tools in performances and at workshops. Since collaboration has been at the forefront, being able to test with others have been vital. I am now more confident to conclude that the future will be collaboration-oriented.

5.4 Limitations and Future Work

Live coding can be a useful tool for musical collaboration but it is certainly not the only. Still, the web browser-based examples demonstrated in this thesis can be taken further in other types of systems. It is essential to consider the application context, and think of the input and output. When it comes to using AI in music, I believe it necessary to explore how both machine learning-based and rule-based alternatives.

Software development was primarily a method and not a goal in itself in this project. Thus, the audio library developed in Glicol is still not as comprehensive as other audio libraries such as the FAUST project (Orlarey et al., 2009). There are also many loose ends in the syntax and interface. Even though I have been able to perform and do some workshops, these activities were severely limited due to the COVID-19 pandemic. Fortunately, my approach works well in an online context. Still, the experience is much different from the on-scene ones. Therefore, I foresee more active exploration of the tools also in physical concerts in the time ahead.

One of the main takeaways from this thesis has been that low-level development can significantly influence collaboration in computer music systems. This is clearly embodied in the evolution from QuaverSeries to Glicol. The reflection on RaveForce and Air Guitar also shows how these two projects are hindered due to inaccessibility to the low-level architecture. As an ‘answer’ to

these issues, Glicol's low-level capabilities may restart RaveForce and Air Guitar and thus will be the focus of my future work.

I have a long to-do list for the future development of Glicol. The primary thing is to make a complete audio library in Rust. A more long-term goal is to see how Glicol can be connected to some of the deep-learning approaches explored in the RaveForce and Air Guitar projects. This will create entirely new possibilities when it comes to thinking about computers as collaborative performers or composers.

From a conceptual point of view, I see the various tools and technologies that I have worked on as a platform for future musicological studies. It is easy to record all the steps that live coders use in performance. This makes it possible to analyse how they type, how they develop musical ideas, and how they build up musical layers. This can also inform music cognition research about attention and prediction. The mechanism of collaborating with a recorded player can also be explored, which is in line with the concept that music is an *action*, or a *process* introduced by Small (1998).

Analyses of how the tools are used in real-world musicking can also feedback to the developments. From a pedagogical perspective, I am curious about how the syntax can be improved to make it easier to use, such as in schools. For example, is 'lpf' the best abbreviation of a low pass filter? And how does one teach a school kid what a low pass filter is and does? I foresee many exciting development iterations as I—and hopefully others—continue with the development of such tools.

Finally, I hope that my work can inspire others to explore collaborative practices more in music. Then I am not only thinking about collaboration as playing separate instruments together. Rather, I am curious about how people can make music together on the *same* instrument or system. Lastly, I believe that such encounters can be greatly facilitated to the use of AI-based methods, both symbolic and sub-symbolic.

Bibliography

S. Aaron. Sonic pi–performance in education, technology and art. *International Journal of Performance Arts and Digital Media*, 12(2):171–178, 2016.

S. Aaron and A. F. Blackwell. From sonic pi to overtone: creative musical experiences with domain-specific and functional languages. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 35–46. ACM, 2013.

J. Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35 (2):97–113, 2003.

M. Baalman. Embodiment of code. In *Proceedings of the First International Conference on Live Coding*, pages 35–40, 2015.

A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161, 2017.

A. W. Balkema and H. Slager. *Artistic research*, volume 18. Rodopi, 2004.

L. Barreira, S. Cavaco, and J. F. da Silva. Unsupervised music genre classification with a model-based approach. In *Portuguese Conference on Artificial Intelligence*, pages 268–281. Springer, 2011.

A. K. Beingessner. *You Can't Spell Trust Without Rust*. PhD thesis, Carleton University, 2016.

C. Bell. Algorithmic music composition using dynamic markov chains and genetic algorithms. *Journal of Computing Sciences in Colleges*, 27(2):99–107, 2011.

L. L. Beranek and T. J. Mellow. Chapter 1 - introduction and terminology. In L. L. Beranek and T. J. Mellow, editors, *Acoustics: Sound Fields and Transducers*, pages 1–19. Academic Press, 2012. ISBN 978-0-12-391421-7.

L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48. IEEE, 2000.

F. Bernardo, C. Kiefer, and T. Magnusson. An audioworklet-based signal engine for a live coding language ecosystem. In *Proceedings of Web Audio Conference (WAC-2019)*, 2019.

F. Bevilacqua, B. Zamborlin, A. Sypniewski, N. Schnell, F. Guédy, and N. Rasamimanana. Continuous realtime gesture following and recognition. In *International gesture workshop*, pages 73–84, Bielefeld, Germany, 2009.

S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

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

J. Blandy and J. Orendorff. *Programming Rust: Fast, Safe Systems Development*. O'Reilly Media, Inc., 2017.

G. Born. *Music, sound and space: Transformations of public and private experience*. Cambridge University Press, 2013.

M. Bosi and R. E. Goldberg. *Introduction to digital audio coding and standards*, volume 721. Springer Science & Business Media, 2012.

R. Boulanger and V. Lazzarini. *The audio programming book*. the MIT Press, 2010.

R. C. Boulanger et al. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. MIT press, 2000.

J.-P. Briot, G. Hadjeres, and F. Pachet. Deep learning techniques for music generation-a survey. *arXiv preprint arXiv:1709.01620*, 2017.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

R. E. Bryant, O. David Richard, and O. David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

P. Budner and J. Grahl. Collaboration networks in the music industry. *arXiv preprint arXiv:1611.00377*, 2016.

J. Bullock and A. Momeni. Ml. lib: robust, cross-platform, open-source machine learning for max and pure data. In *Proc. Int. Conf. on New Interfaces for Musical Expression*, Baton Rouge, LA, 2015.

M. Bunge. Technology as applied science. In *Contributions to a Philosophy of Technology*, pages 19–39. Springer, 1966.

R. Calegari, G. Ciatto, and A. Omicini. On the integration of symbolic and sub-symbolic techniques for xai: A survey. *Intelligenza Artificiale*, 14(1):7–32, 2020.

B. Caramiaux, N. Montecchio, A. Tanaka, and F. Bevilacqua. Adaptive gesture recognition with variation estimation for interactive systems. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 4(4):18, 2015.

H. Choi. Audioworklet: the future of web audio. In *ICMC*, 2018.

M. Clarà and E. Barberà. Three problems with the connectivist conception of learning. *Journal of Computer Assisted Learning*, 30(3):197–206, 2014.

N. Collins. Reinforcement learning for live musical agents. In *ICMC*, 2008.

N. Collins and A. McLean. Algorave: Live performance of algorithmic electronic dance music. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 355–358, 2014.

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

P. Cook. 2001: Principles for designing computer music controllers. In *A NIME Reader*, pages 1–13. Springer, 2017.

D. Cope. *Experiments in musical intelligence*, volume 1. AR editions, 1996.

C. Donahue, J. McAuley, and M. Puckette. Synthesizing audio with generative adversarial networks. *arXiv preprint arXiv:1802.04208*, 2018.

M. Dorfer, F. Henkel, and G. Widmer. Learning to listen, read, and follow: Score following as a reinforcement learning game. *arXiv preprint arXiv:1807.06391*, 2018.

V. Dusek et al. *Philosophy of technology: An introduction*, volume 90. Blackwell Malden, MA, 2006.

K. Ebcioğlu. An expert system for harmonizing four-part chorales. *Computer Music Journal*, 12(3):43–51, 1988.

C. A. Ellis and C. Sun. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. Citeseer, 1998.

J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts. Gansynth: Adversarial neural audio synthesis. *arXiv preprint arXiv:1902.08710*, 2019.

J. Engel, L. Hantrakul, C. Gu, and A. Roberts. Ddsp: Differentiable digital signal processing. *arXiv preprint arXiv:2001.04643*, 2020.

C. Erdem and A. R. Jensenius. Raw: Exploring control structures for muscle-based interaction in collective improvisation. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 477–482. Birmingham City University, 2020.

C. Erdem, K. H. Schia, and A. R. Jensenius. Vrengt: a shared body-machine instrument for music-dance performance. *arXiv preprint arXiv:2010.03779*, 2020.

R. A. Fiebrink. *Real-time human interaction with supervised learning algorithms for music composition and performance*. Citeseer, Princeton, NJ, 2011.

D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 81–90, 2009.

J. J. Fricke. Jimi hendrix'use of distortion to extend the performance vocabula˜ of the electric guitar. 1998.

D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of programming languages*. MIT press, 2001.

I. Fujinaga, A. Hankinson, and L. Pugin. Automatic score extraction with optical music recognition (omr). In *Springer Handbook of Systematic Musicology*, pages 299–311. Springer, 2018.

J. J. Gibson. The concept of affordances. *Perceiving, acting, and knowing*, 1, 1977.

N. Gillian and J. A. Paradiso. The gesture recognition toolkit. *The Journal of Machine Learning Research*, 15(1):3483–3487, 2014.

R. I. Godøy. Chunking sound for musical analysis. In *International Symposium on Computer Music Modeling and Retrieval*, Copenhagen, Denmark, 2008. Springer.

R. I. Godøy, E. Haga, and A. R. Jensenius. Playing "air instruments": mimicry of sound-producing gestures by novices and experts. In *International Gesture Workshop*, pages 256–267. Springer, 2005.

A. Goldman. Live coding helps to distinguish between embodied and propositional improvisation. *Journal of New Music Research*, 48(3):281–293, 2019.

I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

J. Gregorio and Y. Kim. Augmenting parametric synthesis with learned timbral controllers. In *Proc. Int. Conf. on New Interfaces for Musical Expression*, Porto Alegre, Brazil, 2019.

D. Griffiths and A. McLean. Textility of code: a catalogue of errors. *TEXTILE*, 15(2):198–214, 2017.

A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

G. Hadjeres, F. Pachet, and F. Nielsen. Deepbach: a steerable model for bach chorales generation. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1362–1371. JMLR. org, 2017.

J. Han and N. Gold. Lessons learned in exploring the leap motion™ sensor for gesture-based instrument design. Goldsmiths University of London, 2014.

L. Hiller and L. M. Isaacson. *Illiac suite, for string quartet*, volume 30. New Music Edition, 1957.

D. Ihde. Philosophy of technology. In *Philosophical problems today*, pages 91–108. Springer, 2004.

O. Jack. Livecoding networked visuals in the browser. https://github.com/ojack/hydra, 2019.

R. H. Jack, J. Harrison, F. Morreale, and A. P. McPherson. Democratising dmis: the relationship of expertise and control intimacy. In *NIME*, pages 184–189, 2018.

A. Jensenius. *ACTION – SOUND - Developing Methods and Tools to Study Music-Related Body Movement*. PhD thesis, University of Oslo, 01 2007.

A. R. Jensenius. Sonic microinteraction in "the air". In M. Lesaffre, P.-J. Maes, and M. Leman, editors, *The Routledge Companion to Embodied Music Interaction*, chapter 46, pages 431–439. Routledge, New York, NY, 2017.

A. R. Jensenius. *Sound Actions: Conceptualizing Musical Instruments*. The MIT Press, 2022.

A. R. Jensenius and M. J. Lyons. *A NIME Reader: Fifteen Years of New Interfaces for Musical Expression*, volume 3. Springer, New York, NY, 2017.

R. Jung. Understanding and evolving the rust programming language. 2020.

D. Kahanwal et al. Abstraction level taxonomy of programming language frameworks. *arXiv preprint arXiv:1311.3293*, 2013.

N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. v. d. Oord, S. Dieleman, and K. Kavukcuoglu. Efficient neural audio synthesis. *arXiv preprint arXiv:1802.08435*, 2018.

D. Keislar. A historical view of computer music technology. In *The Oxford handbook of computer music*. 2009.

R. A. Kendall and E. C. Carterette. The communication of musical expression. *Music perception*, 8(2):129–163, 1990.

C. Kiefer. Musical instrument mapping design with echo state networks. In *Proc. Int. Conf. on New Interfaces for Musical Expression*, London, UK, 2014.

K. N. King. *C programming: a modern approach*. WW Norton & Co., Inc., 1996.

L. E. Kinsler, A. R. Frey, A. B. Coppens, and J. V. Sanders. *Fundamentals of acoustics*. John wiley & sons, 2000.

R. Kirkbride. Foxdot: Live coding with python and supercollider. In *Proceedings of the International Conference on Live Interfaces*, 2016.

R. Kirkbride. Troop: A collaborative tool for live coding. In *Proceedings of the 14th Sound and Music Computing Conference*, pages 104–9, 2017.

R. P. Kirkbride. *Collaborative interfaces for ensemble live coding performance*. PhD thesis, University of Leeds, 2020.

S. Knotts. Live coding and failure. *The Aesthetics of Imperfection in Music and the Arts: Spontaneity, Flaws and the Unfinished*, page 189, 2020.

T. Kvifte. What is a musical instrument. *Svensk tidskrift för musikforskning*, 1 (2008):45–56, 2008.

Q. Lan and A. R. Jensenius. Browser-based collaborative live coding with glicol: A graph-oriented live coding language written in rust. 2021.

N. Laurent and K. Mens. Parsing expression grammars made practical. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 167–172, 2015.

M. Lee, A. Freed, and D. Wessel. Real-time neural network processing of gestural and acoustic signals. In *Proc. of the Int. Computer Music Conf.*, pages 277–277, Montreal, Quebec, Canada, 1991.

M. Leman, P.-J. Maes, L. Nijs, and E. Van Dyck. What is embodied music cognition? In *Springer handbook of systematic musicology*, pages 747–760. Springer, 2018.

S. Letz, Y. Orlarey, and D. Fober. Compiling faust audio dsp code to webassembly. 2017.

A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, pages 21–26, 2015.

G. Loy. Musicians make a standard: The MIDI phenomenon. *Computer Music Journal*, 9(4):8–26, 1985.

X. Lv, F. He, W. Cai, and Y. Cheng. A string-wise crdt algorithm for smart and large-scale collaborative editing systems. *Advanced Engineering Informatics*, 33:397–409, 2017. ISSN 1474-0346. DOI: https://doi.org/10.1016/j.aei.2016.10.005. URL https://www.sciencedirect.com/science/article/pii/S1474034616301811.

T. Magnusson. Of epistemic tools: Musical instruments as cognitive extensions. *Organised Sound*, 14(2):168–176, 2009.

T. Magnusson. ixi lang: a supercollider parasite for live coding. In *Proceedings of International Computer Music Conference 2011*, pages 503–506. Michigan Publishing, 2011.

T. Magnusson. Code scores in live coding practice. In *TENOR 2015: International Conference on Technologies for Music Notation and Representation*, volume 1, pages 134–139. Institut de Recherche en Musicologie, 2015.

T. Magnusson. *Sonic writing: technologies of material, symbolic, and signal inscriptions*. Bloomsbury Academic, 2019.

M. Mahon. Music, power, and practice. *Ethnomusicology*, 58(2):327–333, 2014.

Y. Mann. Interactive music with tone. js. In *Proceedings of the 1st annual Web Audio Conference*. Citeseer, 2015.

C. P. Martin and J. Torresen. An interactive musical prediction system with mixture density recurrent neural networks. *arXiv preprint arXiv:1904.05009*, 2019.

C. P. Martin, K. O. Ellefsen, and J. Torresen. Deep predictive models in interactive music. *arXiv preprint arXiv:1801.10492*, 2018a.

C. P. Martin, A. R. Jensenius, and J. Torresen. Composing an ensemble standstill work for myo and bela. In *Proc. Int. Conf. on New Interfaces for Musical Expression*, Blacksburg, VA, 2018b.

C. P. Martin, A. R. Jensenius, and J. Torresen. Composing an ensemble standstill work for myo and bela. *arXiv preprint arXiv:2012.02404*, 2020.

J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

S. McCoid, J. Freeman, B. Magerko, C. Michaud, T. Jenkins, T. Mcklin, and H. Kan. Earsketch: An integrated approach to teaching introductory computer music. *Organised Sound*, 18(2):146–160, 2013.

J. McCormack, T. Gifford, P. Hutchings, M. T. Llano Rodriguez, M. Yee-King, and M. d'Inverno. In a silent way: Communication between ai and improvising musicians beyond sound. In *Proc. of the Conf. on Human Factors in Computing Systems*, page 38, Glasgow, UK, 2019. ACM.

S. McGuire and R. Pritts. *Audio sampling: a practical guide*. Routledge, 2013.

A. McLean. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 63–70. ACM, 2014.

A. McLean and R. T. Dean. Musical algorithms as tools, languages, and partners. *The Oxford Handbook of Algorithmic Music*, page 1, 2018.

A. McPherson. Bela: An embedded platform for low-latency feedback control of sound. *The Journal of the Acoustical Society of America*, 141(5):3618–3618, 2017.

A. McPherson and K. Tahıroğlu. Idiomatic patterns and aesthetic influence in computer music languages. *Organised Sound*, 25(1):53–63, 2020.

S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio. Samplernn: An unconditional end-to-end neural audio generation model. *arXiv preprint arXiv:1612.07837*, 2016.

E. Miranda. *Composing music with computers*. Focal Press, 2001.

E. R. Miranda and M. M. Wanderley. *New digital musical instruments: control and interaction beyond the keyboard*, volume 21. AR Editions, Inc., 2006.

C. Mitcham. *Thinking through technology: The path between engineering and philosophy*. University of Chicago Press, 1994.

M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

F. R. Moore. The dysfunctions of MIDI. *Computer music journal*, 12(1):19–28, 1988.

J. A. Moorer. Music and computer composition. *Communications of the ACM*, 15(2):104–113, 1972.

G. Mori. Analysing live coding with ethnographical approach: A new perspective. In *Proceedings of the first International Conference on Live Coding (ICLC), University of Leeds, UK, July*, pages 13–15, 2015.

T. Mudd. Material-oriented musical interactions. In *New Directions in Music and Human-Computer Interaction*, pages 123–133. Springer, 2019.

S. Nagarakatte, M. M. Martin, and S. Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

A. Ng. why ai is the new electricity. *Nikkei Asian Review Online*, 27, 2016.

D. Norman. *The design of everyday things: Revised and expanded edition*. Basic books, 2013.

S. J. Norman. Senses of liveness for digital times. IETM Amsterdam, IETM Amsterdam Plenary Meeting, 2016.

D. Ogborn. Network music and the algorithmic ensemble. In *The Oxford Handbook of Algorithmic Music*. 2018.

D. Ogborn, J. Beverley, L. N. del Angel, E. Tsabary, and A. McLean. Estuary: Browser-based collaborative projectional live coding of musical patterns. In *International Conference on Live Coding (ICLC) 2017*, 2017.

Y. Orlarey, D. Fober, and S. Letz. Faust: an efficient functional approach to dsp programming, 2009.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, Long Beach, CA, 2017.

M. Pearce and M. Rohrmeier. Musical syntax ii: empirical perspectives. In *Springer handbook of systematic musicology*, pages 487–505. Springer, 2018.

A. Phinyomark, E. Campbell, and E. Scheme. Surface electromyography (emg) signal processing, classification, and practical considerations. In *Biomedical Signal Processing*. Springer, 2020.

W. C. Pirkle. *Designing Audio Effect Plugins in C++: For AAX, AU, and VST3 with DSP Theory*. Routledge, 2019.

M. S. Puckette et al. Pure data. In *ICMC*, 1997.

J. D. Reiss and A. McPherson. *Audio effects: theory, implementation and application*. CRC Press, 2014.

D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

C. Roads. Artificial intelligence and music. *Computer Music Journal*, 4(2):13–25, 1980.

C. Roads. Research in music and artificial intelligence. *ACM Computing Surveys (CSUR)*, 17(2):163–190, 1985.

C. Roads, J. Strawn, et al. *The computer music tutorial*. MIT press, 1996.

C. Roberts and J. Kuchera-Morin. Gibber: Live coding audio in the browser. In *ICMC*, 2012.

C. Roberts, G. Wakefield, M. Wright, and J. Kuchera-Morin. Designing musical instruments for the browser. *Computer Music Journal*, 39(1):27–40, 2015a.

C. Roberts, K. Yerkes, D. Bazo, M. Wright, and J. Kuchera-Morin. Sharing time and code in a browser-based live coding environment. In *Proceedings of the First International Conference on Live Coding*, pages 179–185, 2015b.

F. Roche, T. Hueber, S. Limier, and L. Girin. Autoencoders for music sound modeling: a comparison of linear, shallow, deep, recurrent and variational models. *arXiv preprint arXiv:1806.04096*, 2018.

G. F. C. Rogers. *The nature of engineering: a philosophy of technology*. Macmillan International Higher Education, 1983.

J. Rohrhuber. Network music., 2012.

M. Rohrmeier and M. Pearce. Musical syntax i: Theoretical perspectives. In *Springer handbook of systematic musicology*, pages 473–486. Springer, 2018.

R. Rowe. Split levels: Symbolic to sub-symbolic interactive music systems. *Contemporary Music Review*, 28(1):31–42, 2009.

S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited„ 2016.

J. C. Schacher, C. Miyama, and D. Bisig. Gestural electronic music using machine learning as generative device. In *Proc. Int. Conf. on New Interfaces for Musical Expression*, Baton Rouge, LA, 2015.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

S. Shirinivas, S. Vetrivel, and N. Elango. Applications of graph theory in computer science an overview. *International journal of engineering science and technology*, 2(9):4610–4621, 2010.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

D. M. Simmonds. The programming paradigm evolution. *Computer*, 45(06): 93–95, 2012.

C. Small. *Musicking: The meanings of performing and listening*. Wesleyan University Press, 1998.

B. D. Smith and G. E. Garnett. Unsupervised play: Machine learning toolkit for max. In *Proc. Int. Conf. on New Interfaces for Musical Expression*, Ann Arbor, MI, 2012.

B. Smus. *Web Audio API: Advanced Sound for Games and Interactive Apps.* " O'Reilly Media, Inc.", 2013.

J. Snyder and D. Ryan. The birl: An electronic wind instrument based on an artificial neural network parameter mapping structure. In *Proc. Int. Conf. on New Interfaces for Musical Expression*, London, UK, 2014.

K. Stetz. Slang: An audio programming language built in js. https://github.com/kylestetz/slang, 2018.

B. Stroustrup. *A Tour of C++.* Addison-Wesley Professional, 2018.

B. L. Sturm, O. Ben-Tal, U. Monaghan, N. Collins, D. Herremans, E. Chew, G. Hadjeres, E. Deruty, and F. Pachet. Machine learning research that matters for music creation: A case study. *Journal of New Music Research*, 48(1):36–55, 2019.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

D. E. Tabachnick. Techne, technology, and tragedy. *Techné: Research in Philosophy and Technology*, 7(3):90–111, 2004.

K. Tatar and P. Pasquier. Musical agents: A typology and state of the art towards musical metacreation. *Journal of New Music Research*, 48(1):56–105, 2019.

L. S. Theremin and O. Petrishev. The design of a musical instrument based on cathode relays. *Leonardo Music Journal*, 6(1):49–50, 1996.

V. Valimaki, J. D. Parker, L. Savioja, J. O. Smith, and J. S. Abel. Fifty years of artificial reverberation. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(5):1421–1448, 2012.

A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR abs/1609.03499*, 2016.

A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

C. van Mazijk. Heidegger and husserl on the technological-scientific worldview. *Human Studies*, 42(4):519–541, 2019.

E. Varèse and C. Wen-Chung. The liberation of sound. *Perspectives of new music*, 5(1):11–19, 1966.

E. Visser. Webdsl: A case study in domain-specific language engineering. In *International summer school on generative and transformational techniques in software engineering*, pages 291–373. Springer, 2007.

S. Waisbord. The 5ws and 1h of digital journalism. *Digital Journalism*, 7(3): 351–358, 2019.

G. Wang. A history of programming and music. *The Cambridge Companion to Electronic Music*, pages 55–71, 2007.

G. Wang. *The ChucK Audio Programming Language: An Strongly-timed and On-the-fly Environ/mentality*. PhD thesis, Princeton University, 2008.

G. Wang, P. R. Cook, and S. Salazar. Chuck: A strongly timed computer music language. *Computer Music Journal*, 39(4):10–29, 2015.

M. R. Ward. *Electrical engineering science*. McGraw-Hill, New York, NY, 1971.

A. P. Wierzbicki. *Philosophy Versus History of Technology*, pages 243–267. Springer International Publishing, Cham, 2015. ISBN 978-3-319-09033-7. DOI: 10.1007/978-3-319-09033-7_13. URL https://doi.org/10.1007/978-3-319-09033-7_13.

M. Wright. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.

A. Xambó, P. Shah, G. Roma, J. Freeman, and B. Magerko. Turn-taking and chatting in collaborative music live coding. In *Proceedings of the 12th International Audio Mostly Conference on Augmented and Participatory Sound and Music Experiences*, page 24. ACM, 2017.

I. Xenakis. *Formalized music: thought and mathematics in composition*. Number 6. Pendragon Press, 1992.

S. Yi, V. Lazzarini, and E. Costello. Webassembly audioworklet csound. In *4th Web Audio Conference, TU Berlin*, 2018.

Papers

Paper I

RaveForce: A Deep Reinforcement Learning Environment for Music Generation

Qichao Lan, Jim Torresen, Alexander Refsum Jensenius

Published in *Proceedings of the SMC Conferences. Society for Sound and Music Computing*, 2019, pp. 217–222.

RaveForce: A Deep Reinforcement Learning Environment for Music Generation

Qichao Lan
RITMO
Department of Musicology
University of Oslo
qichao.lan@imv.uio.no

Jim Tørresen
RITMO
Department of Informatics
University of Oslo
jimtoer@ifi.uio.no

Alexander Refsum Jensenius
RITMO
Department of Musicology
University of Oslo
a.r.jensenius@imv.uio.no

ABSTRACT

RaveForce is a programming framework designed for a computational music generation method that involves audio sample level evaluation in symbolic music representation generation. It comprises a Python module and a SuperCollider quark. When connected with deep learning frameworks in Python, RaveForce can send the symbolic music representation generated by the neural network as Open Sound Control messages to the SuperCollider for non-real-time synthesis. SuperCollider can convert the symbolic representation into an audio file which will be sent back to the Python as the input of the neural network. With this iterative training, the neural network can be improved with deep reinforcement learning algorithms, taking the quantitative evaluation of the audio file as the reward. In this paper, we find that the proposed method can be used to search new synthesis parameters for a specific timbre of an electronic music note or loop.

1. INTRODUCTION

In a computational music generation task, what is essentially generated? This question leads to a debate on either to generate music in symbolic music representation, e.g. MIDI (Music Instrument Digital Interface) or to generate the audio waveform directly. Symbolic music representations can generally reflect the idiosyncrasy of a music piece, but they can hardly trace detailed music information, such as micro-tonal tunings, timbre nuances and micro-timing. Signal-based music representations are better at preserving micro-level details that are not captured well by the symbolic representations. Thus signal-based workflows—including raw audio generation—may be a solution for computational music generation. However, since raw audio generation requires much more computational resources than symbolic representation methods, there are still some difficulties for this method to generate long multi-track music pieces [1]. Furthermore, without a symbolic representation, these methods can be too sophisticated to explain from a music-theoretical perspective. Hence, our motivation is to find a balance between these two forms of music representation in computational music generation.

Our research question is: how can an A.I system be trained to consider the music sound while generating symbolic music representation? Technically speaking, we hope that the neural network in an A.I system can not only generate symbolic sequences but also convert the symbolic representation into an audio waveform that can be evaluated. To do so, we need to use non-real-time synthesis for the transformation from symbolic music representation to an audio file which will become the input of the neural network, and the output will be accordingly the next symbolic representation. Compared with pure symbolic generation, this method also outputs the corresponding audio waveform, which may broaden the application fields. Besides, different from raw audio generation, we fix the transforming function for the neural network, which may make the computational resource focus more on the target music information than on the function estimation.

In this paper, we will explain the proposed method and provide a programming implementation as well as two simplified music tasks as examples. We start with the background of deep learning music generation in Section 2, demonstrating the relationship between the data type and the neural network architecture. In Section 3, we present our method to improve the symbolic representation and the reason why we choose to use deep reinforcement learning. Section 4 introduces the implementation details of our deep reinforcement learning environment with an emphasis on how we optimise it for a musical context. Section 5 describes the reward function design in customised tasks and explains the evaluation from running time and music quality perspective. In Section 6, we summarise the innovations and limitations of our method as well as our future directions.

2. BACKGROUND

Computational music generation has for a long time been an intriguing topic for musicologists and computer scientists [2]. Of current algorithmic methods, deep learning seems to be particularly relevant for music generation tasks [3]. Deep learning is a method that learns from data representations, so in terms of music generation, it is essential to study the background of how the music representation influences the learning process and result.

2.1 Symbolic vs signal-based representations

Music can typically be represented as either signals (audio) or symbols (score representations). Popular symbolic representation methods include MIDI, musicXML, MEI, and others [4]. Among them, MIDI is one of the most popular data formats being used in deep learning music generation tasks. In some particular styles of music, and particularly the ones based on traditional music notation, MIDI data can be an efficient representation. One example is the piano score generation in the DeepBach project [5]. Another example is that of machine-assisted composition applications, in which MIDI allows for editable features [6]. However, as mentioned in the introduction, there are also many cases in which symbolic representations are inadequate in capturing the richness and nuances of the music in question.

One way to address limitations of symbolic representations is the use of sample-level music generation, as demonstrated in WaveNet [7] and WaveRNN [8]. However, although some progress has been made, the raw audio generation requires a lot of computational resources, and it is too complicated to explain how these samples get organised from a musicology perspective.

The data format can also influence the design of the neural network. In symbolic representations, supervised learning can be found in many applications [9]. For raw audio signals, unsupervised learning techniques such as autoencoder and generative adversarial network (GAN) are frequently adopted [10, 11].

2.2 Reinforcement learning

Reinforcement learning is different from supervised or unsupervised learning techniques in that its updating strategy relies on the interaction between an agent and the environment rather than the function gradient. In a given period—that is, an *episode* in reinforcement learning—the agent will try to maximise the reward it can get. The reward is calculated in each episode, and it is used to update the parameters of the agents [12].

The connection between reinforcement learning and music generation goes back to the use of Markov models in algorithmic composition. As one of the pioneers in automated music generation, in the piece called *Analogique A*, Iannis Xenakis uses Markov models for the order of musical sections [13]. The use of Markov models in composition reveals its connection with reinforcement learning as the action of the agent only depends on the current state. However, in previous research on reinforcement learning in computational music generation [14], the reward function calculation is not based on the sample-level evaluation.

Recently, deep learning technology has brought new possibilities to reinforcement learning as it allows the agents to examine higher-level information. In deep reinforcement learning, the agent can be represented by a neural network, which makes it capable of evaluating the raw audio signal and then output the decision. Deep reinforcement learning has been a success during the past few years since it shows that a virtual agent can surpass human beings in several tasks, e.g. Atari games [15]. After that, there appear more and more algorithms such as Proximal Policy Optimization (PPO) [16]. For testing these algorithms, there are many simulation environments, e.g. the OpenAI Gym [1]. For music, deep reinforcement learning has been used for the score following [17]. However, there is still no environment designed for music generation.

3. DESIGN CONSIDERATION

Though symbolic representations have shown some limitations, generating music at the audio sample level can be computationally expensive. Therefore, we propose to generate the symbolic representation first, and then use these representations to synthesise audio for evaluation.

3.1 From symbolic notation to audio

Our first step is to choose a proper method to convert a symbolic representation to an audio file. Three options are considered:

1. to send the generated sequence to an instrument and record the sound for evaluation.
2. to use other general-purpose programming languages such as C++ for the sound synthesis.
3. to use music programming languages like Max/MSP, Pure Data, Csound and SuperCollider for non-real-time synthesis.

We exclude the first option because it would be too time-consuming, considering there would be a considerable number of iterations in the deep learning training process. The second option is the most efficient in synthesis speed, but it lacks the extensibility from a music perspective as users have to be familiar with the C-style programming languages. The third option best balances the efficiency and usability as music programming languages have already been ubiquitous in the electronic music field [18].

However, both the second and the third option are faced with the same challenge—the *gradient*. In supervised learning, we need to know all the functions and their gradient. After comparing the output of the neural network and the training data, we should fine-tune the parameter of the neural network to minimise the loss with the help of these gradients [19]. In our proposed method, since we involve the non-real-time synthesis, back-propagation cannot be done in this context as the functions used for transforming symbolic representation to audio files are unknown.

3.2 Addressing the gradient problem with deep reinforcement learning

Deep reinforcement learning can solve the gradient problem mentioned above as it relies only on the interaction reward rather than the gradient. Though we cannot get the gradient from the symbolic-to-audio transforming function, We can quantitatively evaluate the synthesised audio to get a reward. Concretely, we train a neural network to output

[1] `https://gym.openai.com`

a sequence of symbolic music notations (such as the parameters for a synthesiser) and send the information to an audio programming language for non-real-time synthesis. Then, we compare the synthesised audio file with the target file, or we can use a neural network to grade the audio file directly. When an action brings a positive reward, the probability of the action should increase, and vice versa.

There are several important concepts in deep reinforcement learning that need to be defined in the music context (see Fig. 1):

1 *Step* refers to the process of executing what has been decided to do in the next 16 th note or rest.

2 *Episode* refers to a series of continuous interactions before the *done* attribute turns to *true*, e.g. the end of a game. In a musical context, we use a *total-step* attribute to decide the length of an episode. Thus, it can vary from one single note to a note sequence.

3 *Observation-space* refers to the current state. In our musical context, we set the currently synthesised audio file to be the observation-space. In other words, the agent should be "aware" of the previous state (synthesised audio) and take the next step accordingly.

4 *Action-space* refers to the set of action choices for the agent. In a musical context, the action-space can be discrete (e.g. a note pitch) or continuous (e.g. the amplitude).

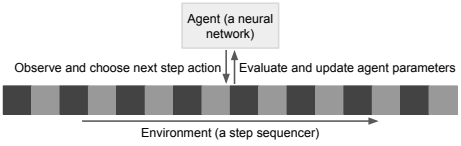


Figure 1. RaveForce architecure: in each note (step), the agent (neural network) will choose an action according to its observation on the current state (currently synthesised audio).

4. IMPLEMENTATION AND OPTIMISATION

As is discussed above, the key to our proposed method is to have an environment that can handle the non-real-time synthesis and evaluate the result. In our implementation of RaveForce [2], we follow the OpenAI Gym interfaces in our Python module, and in SuperCollider, we create a quark to execute the non-real-time audio synthesis. In order to connect with deep learning frameworks, some optimisation is necessary for the observation space.

4.1 The idea from a live coding session

To implement the environment, we refer to a live coding session [20]. In many live coding sessions, SuperCollider [3] has been used as the audio engine as it tracks the time and beat accurately [21]. SuperCollider employs a client-server architecture that contains two parts: the *scsynth* (SuperCollider Synthesiser) and the *sclang* (SuperCollider Language). The *sclang* will be combined in real-time to a simplified version of Open Sound Control (OSC) messages [22] and sent to the *scsynth* to control the sound. This architecture allows the *scsynth* server to run alone, while *sclang* can be replaced by other domain-specific languages (DSLs) like TidalCycles [4]. In short, in a live coding session, the live coders use DSLs as a client to control the real-time sound synthesis in the SuperCollider server. For our need, instead of using SuperCollider to output real-time audio signals, we use it for non-real-time audio synthesis.

As for the client, we choose to write it in Python because several deep learning frameworks (such as PyTorch [5]) have been implemented in Python, and the Python module Gym is one of the most important benchmarks for deep reinforcement learning. By designing the client part in Python, we can follow the Gym interface and connect with a deep learning framework, while we move the interaction part (the audio synthesis) to the SuperCollider server. With the help of Open Sound Control messages, we link the neural network training with the audio synthesis (see Fig. 2).

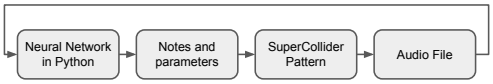


Figure 2. Python-SuperCollider communication: a neural network (agent) is trained in Python; it sends symbolic music representations(e.g. notes and synthesiser parameters) as Open Sound Control messages to the SuperCollider pattern; then the pattern will be synthesised to an audio file in non-real-time and sent back to Python as the input of the neural network, forming an iteration.

4.2 Code implementation

The pseudo-code of the implementation is as follows:

1 Use *make* function in the client to create the required environment, which will send a message to the server, asking the server to load related music patterns, synthesise an empty file and return the address of the file to the client side. On receiving the returning message, the client should read the action space and the observation space.

2 Send the *reset* message to the server side. Empty the observation space if it is not.

3 According to the observation space, decide what action to take. Send the *step* message to the server side with chosen actions in each step. The server will do non-real-time synthesis in each step according to the given

[2] https://github.com/chaosprint/RaveForce
[3] https://supercollider.github.io
[4] https://tidalcycles.org
[5] https://pytorch.org

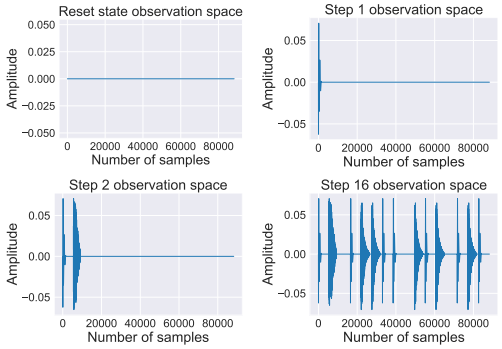


Figure 3. The observation space in each state has the same length. For instance, the step 1 only has the audio information for the first 16th note, but it is padded to have the same length as the reset state (about 90000 samples).

action message. Also, the server should return the client with the synthesised file address.

4 The client should use the address to load the currently synthesised sound file and set it as the observation space. Calculate the reward by comparing the generated audio file with the target audio file.

5 Send the reward back to the client for updating the neural network.

6 Repeat from Step 3 until the limit of episode length is reached

4.3 Optimisation

In the implementation, a unique strategy is designed for the observation space. As neural networks typically require a fixed length input, the observation space needs to be padded to have the same length in every step. Hence, in the initialisation stage, we require SuperCollider to generate an empty full-step (16-step by default) long audio file corresponding to the beats per minute (BPM) parameter. The length of this empty file will be set as the *total-length* attribute. In the following steps, though the actual output of the audio file varies in length, it will be padded with zeros to match the total-length attribute. With this strategy, the observation spaces in each step can share the same length (see Fig. 3).

5. TASK DESIGN AND EVALUATION

After implementing the environment, it is necessary to examine what kind of tasks it can handle and evaluate how the environment performs with the given task.

5.1 Challenges with the reward function design

The reward function in reinforcement learning measures how well the agent chooses the action in the current step. Its design is challenging for music generation, especially in those tasks whose evaluation criteria are subjective. It can be feasible to evaluate the similarity between the generated music piece and the songs in a music corpus. At the same time, pursuing similarity in music can lead to plagiarism, which is an essential issue to address [23].

Currently, we provide four criteria for evaluation: (1) mean square error (MSE) of all the samples; (2) MSE of the Mel-frequency cepstral coefficients (MFCCs); (2) MSE of the short-time Fourier transform (STFT) coefficients, both real and imaginary parts; (4) MSE of the constant-Q transform (CQT) coefficients, both real and imaginary parts. These four criteria are used to measure the similarities between two audio files. Also, as the whole programming framework is customisable, it can be connected with other criteria, e.g. a well trained neural network that can grade a music file.

5.2 The example tasks

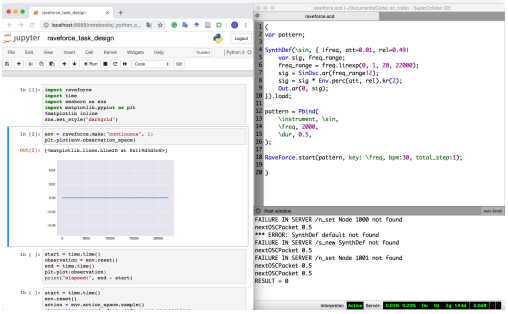


Figure 4. RaveForce workflow: first run SuperCollider code, and then open Python IDE (e.g. Jupyter Notebook) to train the agent.

In RaveForce, the music task should be defined by the user (see Fig. 4). We provide two music examples to explain the environment better.

5.2.1 Drum loop imitation

The example task *drum-loop* uses music samples from three drum components (kick drum, snare drum, and hi-hat) to imitate the target drum loop as much as possible. The action space in the example is a discrete set that contains all eight possible combinations in each note from which the agent should choose one action, and a reward will be calculated according to the choice (see Fig. 5).

Different from some other reinforcement tasks, the reward in this task is precisely the state value function. If we use Deep Q-learning (DQN) for this task, the Q function in each step can be calculated as follows:

$$Q^{\pi}(a|s) = V(s_{t+1}) - V(s) \tag{1}$$

Also, as a specific drum combination only has a fixed reward, we can use the traditional dynamic programming method to find the best drum pattern in this case.

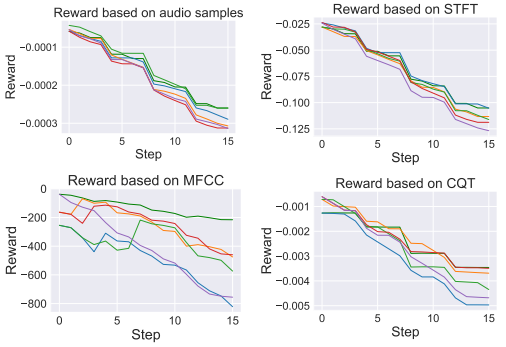


Figure 5. Drum loop combination reward with different criteria. The green line represents the reward of the optimal drum combination which is closest to the target drum loop while the rest are random combination rewards. The MFCC criterion tends to outperform others in this task.

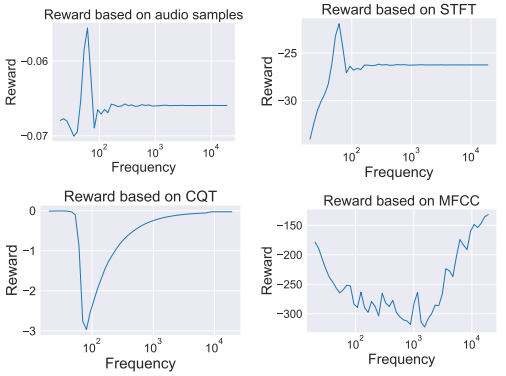


Figure 6. Different criteria for kick drum simulation task. MFCC and CQT tend to show poor performance in this task.

5.2.2 Kick drum optimal frequency search

In this example, we aim to use a sine wave oscillator, controlled by an amplitude envelope to simulate a kick drum audio sample. To make it easier for visualisation, we fix the envelop shape and make the frequency of the sine wave oscillator the only controllable parameter. The relationship between the frequency and the reward is shown in Fig. 6. The *total-step* attribute in SuperCollider can be set to one, which makes the pattern become a single note. In each iteration, the parameter updating of the whole loop is done for this single note. Also, the example can be extended to more parameters and more steps.

With the frequency-reward distribution, we can use the neural network to search for the optimal frequency. First, we train a critic-network which takes the frequency as input and predicts the reward. When connected with the critic-network, an actor-network can be trained until it converges to the optimal frequency.

5.3 Evaluation

We will evaluate the environment from two angles: (1) whether the environment is fast enough for the training; (2) if the symbolic-to-audio conversion can help the music generation.

As a support to our method, the programming framework implementation is the focal point of this paper. In previous sections, we have introduced our environment design and the optimisation we have made, which makes it feasible to use audio evaluation methods for symbolic generation within an acceptable running time. To illustrate, we provide the running time of a 16-step task in one episode (see Fig. 7), which is calculated with the *drum-loop* task mentioned above.

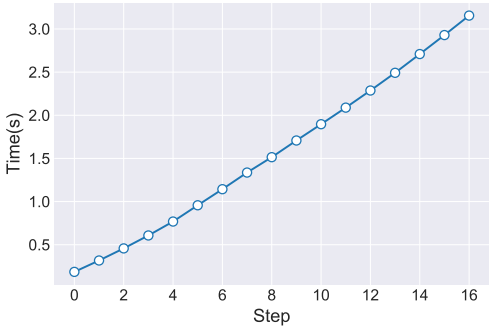


Figure 7. The running time of RaveForce example task *drum-loop*. Step 0 refers to the reset state and some time will be spent for calculating the total-length. All 16 steps will take around 3.2 seconds on an Apple MacBook Pro 13-inch (Mid 2017, i5, without Touch Bar).

Regarding the quality of the music, there are still some uncertainties, for the generated music quality may change with different algorithms, tasks and music genres. Currently, limited by computational resources, we focus mainly on the programming framework implementation, and only pay particular attention to electronic music loop or note.

Also, it is arguable that the predefinition of synthesiser architecture can be a limitation of music complexity. However, this trade-off is significant to our proposed method. With a fixed transforming function, for example, the neural network will no longer need to organise all the audio samples to form an audio waveform which is aurally similar to an FM synthesis tone. Instead, the computational resources can be used to focus on optimising the parameters of a predefined FM synthesiser. This trade-off may even bring new possibilities in music creation because mismatching the target tone with a random synthesiser architecture can potentially generate a tone which is similar but slightly different from the target.

6. CONCLUSION

In this project, we propose a new music generation design that employs deep reinforcement learning, and we have im-

plemented an environment for testing the design. It follows the OpenAI Gym interfaces but moves the interaction to SuperCollider. It turns out that the SuperCollider is fast enough in non-real-time audio synthesis, which makes the reward calculation and the neural network training feasible. Meanwhile, there are some uncertainties if this method can improve the music generation, which should be tested with different tasks, algorithms and music genres. It can be one of our future directions. Nevertheless, the whole implementation produces an environment for researches to explore new algorithms for music generation tasks, e.g. music sequence generation or timbre parameter searching. It provides a new perspective to music generation, especially for those tasks in which users can find a determined reward function.

Acknowledgments

This work was partially supported by the Research Council of Norway through its Centres of Excellence scheme, project number 262762 and by NordForsks Nordic University Hub Nordic Sound and Music Computing Network NordicSMC, project number 86892.

7. REFERENCES

[1] S. Dieleman, A. van den Oord, and K. Simonyan, "The challenge of realistic music generation: modelling raw audio at scale," in *Advances in Neural Information Processing Systems*, 2018, pp. 8000–8010.

[2] D. Herremans, C.-H. Chuan, and E. Chew, "A functional taxonomy of music generation systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, p. 69, 2017.

[3] J.-P. Briot, G. Hadjeres, and F. Pachet, "Deep learning techniques for music generation-a survey," *arXiv preprint arXiv:1709.01620*, 2017.

[4] I. Fujinaga, A. Hankinson, and L. Pugin, "Automatic score extraction with optical music recognition (omr)," in *Springer Handbook of Systematic Musicology*. Springer, 2018, pp. 299–311.

[5] G. Hadjeres, F. Pachet, and F. Nielsen, "Deepbach: a steerable model for bach chorales generation," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1362–1371.

[6] I. Simon and S. Oore, "Performance rnn: Generating music with expressive timing and dynamics," 2017.

[7] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR abs/1609.03499*, 2016.

[8] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. v. d. Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient neural audio synthesis," *arXiv preprint arXiv:1802.08435*, 2018.

[9] B. L. Sturm, O. Ben-Tal, U. Monaghan, N. Collins, D. Herremans, E. Chew, G. Hadjeres, E. Deruty, and F. Pachet, "Machine learning research that matters for music creation: A case study," *Journal of New Music Research*, vol. 48, no. 1, pp. 36–55, 2019.

[10] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "Samplernn: An unconditional end-to-end neural audio generation model," *arXiv preprint arXiv:1612.07837*, 2016.

[11] C. Donahue, J. McAuley, and M. Puckette, "Synthesizing audio with generative adversarial networks," *arXiv preprint arXiv:1802.04208*, 2018.

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[13] I. Xenakis, *Formalized music: thought and mathematics in composition*. Pendragon Press, 1992, no. 6.

[14] N. Collins, "Reinforcement learning for live musical agents." in *ICMC*, 2008.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[17] M. Dorfer, F. Henkel, and G. Widmer, "Learning to listen, read, and follow: Score following as a reinforcement learning game," *arXiv preprint arXiv:1807.06391*, 2018.

[18] G. Wang, "A history of programming and music," *The Cambridge Companion to Electronic Music*, pp. 55–71, 2007.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[20] A. McLean and R. T. Dean, "Musical algorithms as tools, languages, and partners," *The Oxford Handbook of Algorithmic Music*, p. 1, 2018.

[21] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[22] M. Wright, "Open sound control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[23] A. Papadopoulos, P. Roy, and F. Pachet, "Avoiding plagiarism in markov sequence generation." in *AAAI*, 2014, pp. 2731–2737.

Paper II

QuaverSeries: A Live Coding Environment for Music Performance Using Web Technologies

II

Qichao Lan, Alexander Refsum Jensenius

Published in *Proceedings of the International Web Audio Conference (WAC)*, NTNU, 2019, pp. 41–46.

QuaverSeries: A Live Coding Environment for Music Performance Using Web Technologies

Qichao Lan
RITMO
Department of Musicology
University of Oslo
qichao.lan@imv.uio.no

Alexander Refsum Jensenius
RITMO
Department of Musicology
University of Oslo
a.r.jensenius@imv.uio.no

ABSTRACT

QuaverSeries consists of a domain-specific language and a single-page web application for collaborative live coding in music performances. Its domain-specific language borrows principles from both programming and digital interface design in its syntax rules, and hence adopts the paradigm of functional programming. The collaborative environment features the concept of 'virtual rooms', in which performers can collaborate from different locations, and the audience can watch the collaboration at the same time. Not only is the code synchronised among all the performers and online audience connected to the server, but the code executing command is also broadcast. This communication strategy, achieved by the integration of the language design and the environment design, provides a new form of interaction for web-based live coding performances.

1. INTRODUCTION

Live coding, when used in a musical context, refers to a form of performance in which the performers produce music by writing program code rather than playing physical instruments [4]. During the past decade, dozens of live coding languages have emerged.[1] These languages run in various environments, such as the desktop, the browser, and embedded systems (Raspberry Pi, BeagleBone, etc.). The number of programming languages developed for live coding can, in some ways, indicate that performers want to develop their subjective language syntaxes tailored to their musical expressions. Some examples of such new syntaxes are TidalCycles [18], ixi lang[11], Lich.js [6], and Mercury [17].

There have been some discussions about how live coding languages relate to musical instruments [3], but relatively little attention has been devoted to analysing how it is possible to 'transduce' electronic instrument knowledge to the syntax design itself. That is, what types of symbols should be used for what musical purposes, how should different elements be connected, and so on. Instead, the syntax in most live coding languages is mainly borrowed from other programming languages. For example, the use of parentheses is ubiquitous in programming languages, and it is adopted in almost every live coding language. That is the case even though live coding without the parentheses is (more) readable for humans [12].

Inheriting standard programming syntax may create difficulties for non-programmers who want to get started with live coding. We have therefore been exploring how it is possible to design a live coding syntax based on the design principles of digital musical interface design. This includes elements such as audio effect chains, sequencers, patches, and so on. The aim has been to only create a rule by 1) borrowing from digital musicians' familiarity with the digital interfaces mentioned above, or 2) reusing the syntax from existing programming languages to help the parser to work. The design principle is also aligned with the concept of *ergomimesis*, namely the 'application of work processes from one domain to another' [13]. The goal of this principle is to lower the learning curve of our language syntax, especially for non-programmers.

The second aim of our current exploration is to develop a live coding language that is usable for a larger group of people. This can be seen as part of the trend of 'musical democratisation' [8]. Our experience with running workshops for larger groups of university students or pupils in schools is that software that needs to be installed locally makes it much more difficult to get started making music quickly. We, therefore, see browser-based interfaces as the best solution for minimal setup time.

Finally, as part of our interest in exploring the blurring of roles between performers and perceivers, we have also looked at how it is possible to include the 'audience' in live coding. An online deployment makes it possible to not only share the code among performers, but it also makes it possible for the audience to easily join into the online live coding performance. This requires a delicate organisation of a stack of web technologies.

In the current paper, our main research question is:

- How can web technologies influence the music interaction between performers and the audience?

From this two sub-questions emerge:

- How can we design a live coding environment that makes the audience part of the performance? How should the environment be modified to meet this requirement?

[1] See, for example, the TOPLAP overview here: https://github.com/toplap/awesome-livecoding

Web Audio Conference WAC-2019, December 4–6, 2019, Trondheim, Norway.

- How can we use knowledge from digital musical instrument design when developing the syntax of a live coding language? What are the trade-offs that we have to make to achieve this goal?

The paper starts with a background section in which we introduce some related work. Section 3 presents the new domain-specific language, elaborating on how the parser is designed and its semantics. In Section 4, we describe the interface design and explain the communication strategies based on the Firebase real-time database. Finally, in Section 5, we discuss how the environment fits the language design, and how it experimentally changes the relation between performers and audience.

2. BACKGROUND

Many existing live coding environments are installed locally and use the SuperCollider music programming language as the sound engine [15]. With the advent of the Web Audio API, there has been a shift towards developing live coding environments with web technologies. In the following, we will reflect on these two approaches to live coding.

2.1 Live coding with SuperCollider

SuperCollider consists of a programming language called *sclang* and an integrated development environment (IDE). In the IDE, users can boot an audio server (*scsynth*) in the background, and write the code following the syntax of *sclang*. The code, when executed, will be compiled into Open Sound Control (OSC) messages, and sent to the *scsynth* server to control the music sequence. One typical workflow in SuperCollider is to define the synthesiser architecture with the keyword `SynthDef`, and then play the *Synth* in a SuperCollider music sequence (*Pattern*).

Several live coders have chosen to design their syntaxes on top of SuperCollider. For instance, TidalCycles (Tidal) is a domain-specific language written in the Haskell programming language [18]. During live coding, the Tidal code will be interpreted and sent as OSC messages to control the sound engine called *SuperDirt* running in SuperCollider. FoxDot follows a similar architecture but uses Python as the host programming language [9]. Additionally, Troop is a collaborative environment developed for both Tidal and FoxDot, which allows users at the same network to co-edit and share the code on the screen [10].

An inconvenience with the above-mentioned environments, relying on one or more programming languages in addition to SuperCollider, is that it requires several steps of installation. A more user-friendly solution, then, is Sonic Pi, which is also built on SuperCollider audio engine, but offers a single, complete installation package [1]. Even though several OSes are supported, it does not currently run on desktop Linux or Chrome OS. In our experience, this makes it less ideal for schools. And for quick introductory workshops to live coding, we find that having to rely on software installs is less than ideal. For such situations, a web-based solution is more feasible and scalable.

2.2 Web-based live coding

Web-based or browser-based live coding environments only require an up-to-date browser to get started with live coding. With the rapid progress of the Web Audio API, the sound synthesis possibilities and timing capabilities for browser-based live coding have matured quickly. Two good examples of this are the JavaScript-based Gibber environment [20], and the Lisp-style language Slang.js [21]. Although the latter currently does not support collaboration, its parser, written in Ohm.js, provides a valuable example for our development. Another inspiration for us is from EarSketch [16], a music producing environment mainly designed for normal programming education, and its use of the Firebase real-time database pointed us in the direction of a collaborative live coding solution [23].

Some other web-based environments serve as interfaces for other languages. Estuary is a system built for live coding with Tidal in browsers [19]. It has several unique features: collaboration in four different text fields, the support for both SuperCollider and the Web Audio API, and so on. Estuary makes it possible to live code together from different locations, and has been shown to work reliably in cross-continental live coding.

As can be summarised from the brief review of existing live coding environments, the programming languages, syntaxes, and interfaces are diversified. In our exploration, we have been borrowing parts from many of these when designing our syntax and environment.

3. LANGUAGE DESIGN

The syntax design of QuaverSeries is based on a functional programming paradigm.[2] The following sections will describe its syntax and how Ohm.js and Tone.js have been used to implement the parsing and semantics.

3.1 Note representation

The note representation is probably the element that is varied the most among live coding languages. QuaverSeries is based on the idea of a music sequencer. Our prototype syntax looks like this:

```
60  _62  63_64_65_  66_67_68_69
```

The *sequence* has only three elements: numbers, underscores and blank spaces. The numbers refer to MIDI notes, with 60 being 'middle C'. A blank space indicates a separation of individual *notes*, while an underscore denotes a musical *rest*.

A *sequence* will always occupy the duration of a whole note, and all the *notes* will be divided equally. To illustrate, the one-line *sequence* above will be divided into four *notes*: `60`, `_62`, `63_64_65_`, and `66_67_68_69_`, with each of them occupying a quarter note length. Each *note* can be further (equally) divided by the total number of MIDI notes and rests. On the example above, `_62` means that an eighth MIDI note 62 will be played on the off-beat, after an eighth rest. Likewise, `63_64_65_` means eighth note triplets.

As can be seen from the examples above, we create the syntax rule by referring to the musical sequencer, and add extra programmability to the syntax using the dividing algorithm invented in TidalCycles [18]. One direct influence here is that we form a left-to-right typing flow. For the sake of consistency, this flow is kept in other parts of the syntax design as well. Hence, there is no pairing symbol such as parentheses and quotation marks in the syntax.

The sequence can then be connected to a sound generating module using the double greater-than sign (`»`) which is

[2]https://github.com/chaosprint/QuaverSeries

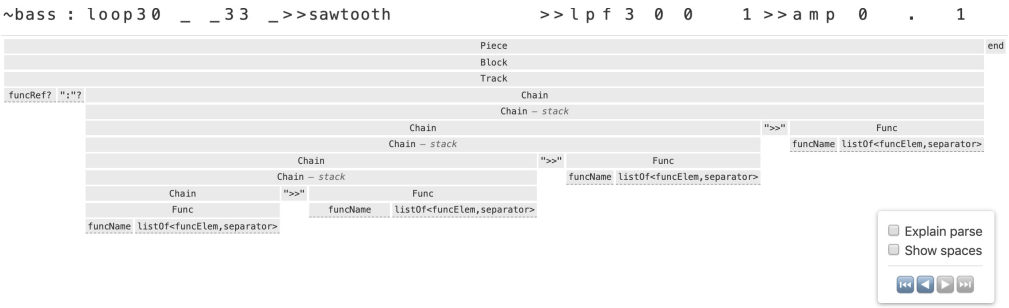


Figure 1: An example of an Ohm.js parsing tree. The hierarchy from top to bottom mainly includes: *Piece*, *Block*, *Chain*, *Func*, etc. In this example, the one-line code makes a *Piece*. This *Piece* contains one *Block*, and the *Block* is a *Track* rather than a *comment*. This *Track* can be further divided into a function reference name (*funcRef*) and a function chain (*Chain*). The function chain will be parsed using left recursion, and the semantics definition is written in JavaScript using Tone.js.

prevalent in programming languages, e.g. C++. In QuaverSeries, it indicates a signal chain flow, from left to right:

```
loop 20 20 20 20 >> membrane >> amp 0.8
```

The one-line code above will create a 'flat four' kick drum sequence using the oscillator function (`membrane`), and with an amplitude of 0.8 (`amp`). This syntax style naturally leads to the choice of using a functional programming paradigm.

3.2 Functional programming

Functional programming languages are prevalent in musical applications, such as the above mentioned Tidal. Overtone is another functional live coding language, following the syntax of Clojure, a dialect of Lisp language [2]. In a functional programming language, everything is a function in the mathematical sense. For example, for the function `y = f(x)`, `x` refers to the *independent variable*, `y` refers to the *dependent variable*, and `f` refers to the transformation. In a similar manner, to express an electronic music signal flow, we can, for example, write the pseudo code:

```
(lpf (square 440 1) 1000 1)
```

Here the `square` function, followed by `440` and `1` means a square wave oscillator function with two parameters as independent variables (inputs): frequency and amplitude. In this example, `lfp` is the abbreviation of *low-pass filter*, and refers to another function that receives three parameters: the audio signal for filtering, the cut-off frequency, and the Q-value. A pair of parentheses is here used to wrap the function that is passed to the next function as its input. This is typical in functional programming languages such as Lisp.

The same synthesis architecture can be rewritten in an object-oriented programming style as:

```
osc = Square(440, 0.8)
osc.connect(LPF(30, 1500))
```

Here `Square` refers to a *class* with a *constructor*. When the constructor is called, an *instance* will be created, and we can save it as in a variable called `osc`. The instance can call its *methods* predefined in the class.

Both functional and object-oriented programming paradigms have both pros and cons. But since we have chosen to start our syntax with a sequencer as the main defining element, we have found it most practical to use functional programming in the syntax design.

In the one-line functional programming code mentioned above, `loop`, followed by a sequence of MIDI numbers is the first function. When followed by `membrane`, the function on the left should become the frequency parameter of `membrane`, with an implicit conversion from MIDI notes to frequency. The `amp` is a function that sends the audio signal generated by the function chain to the audio interface, with a sound level scaling of 0.8. The equivalent Lisp-style would be:

```
(amp (membrane (loop 20 20 20 20) 0.8))
```

3.3 Parsing

The parser in QuaverSeries is built from scratch with the help of Ohm.js. This requires to first program in its domain-specific language, describing how the parser should act (see Appendix B). The parser will generate a parsing tree, identifying the structure of the code (see Figure 1). An example of the QuaverSeries syntax may help to explain how it works:

```
bpm 120

~bass: loop 30 _ _33 _
>> sawtooth >> adsr 0.04 0.3 0 _
>> lpf ~cutoff_freq 1
>> amp 0.1

~cutoff_freq: lfo 8 300 3000
```

The whole *Piece* in this example can be divided into different *Block(s)*, with each *Block* containing at least one function separated by an empty line. The first line is a function for setting the tempo of the piece (120 beats per minute). All the function names are typed in lower case, with an optional underscore in between.

Each function is typically followed by the function elements (*funcElem*). For example, the `adsr` function has four parameters, i.e. the *attack*, *decay*, *sustain* and *release* of an audio envelop. The usage of the underscore is flexible. Apart from its usage in the note representation (to denote a rest), an underscore can also be used as a Python-style placeholder

to keep the default value of a parameter. For instance, the `adsr` function has a value of zero for the *sustain* parameter, which means that there is no need to write the *release* value. Hence, we can use an underscore to represent the release.

The second block demonstrates a concept called *reference*. With a tilde-prefix (∼), a function name becomes a reference that can link two signal chains with one signal chain modulating a parameter of the other. In the example above, the cut-off frequency of the low-pass filter is modulated by a low-frequency oscillator (`lfo`). Hence, to keep the consistency, it is suggested users add a reference at the beginning of every function chain.

3.4 Semantics

The semantics part of QuaverSeries defines how the code should be executed after being parsed. In Ohm.js, the parsing and semantics definitions are separated. Thus after the parser reads through the code and identifies several valid functions, Ohm.js needs further instructions on how to deal with these functions. For instance, when the parser detects a number, the parser will return the number as a string character. It is therefore necessary to write the semantic action as a JavaScript function that converts the string to a float in JavaScript, so that it can be used for numerical operations.

The semantics definition of QuaverSeries is written with Tone.js, a JavaScript audio and sound synthesis library based on the Web Audio API [14]. Currently, the functions are categorised into three parts: *control*, *effect* and *synth*. In the semantics definition, each function is organised into different tracks. Each track has its attributes, including *note*, *synth*, and *effect*.

Once a `run` message is received, the parser will read through the whole page, and convert every function to Tone.js code based on the semantics definition. For instance, when the `loop` function is detected, a Tone.js *Sequence* instance will be created. Likewise, if a `synth` function is identified, a Tone.js *Synth* instance will be created. If audio effects are found, the relevant Tone.js effect instances can be created. Finally when the `amp` is detected, the `connect` method of the *Synth* instance will be called to connect all the effects, with the amplifier (`Tone.Master`) at the end of the effect list.

As a summary, when the `run` command is given, the parser will read through the whole page and identify the functions. Next, semantics action will be executed by constructing Tone.js instances and calling their methods. The `update` command also reads the whole page, and updates each node that is playing, although it will first be effective at the beginning of the next bar.

4. ENVIRONMENT DESIGN

Collaboration has been an important motivation when developing the Quaverseries live coding environment.[3] The aim is to create a web application that live coders can use to collaborate in different *virtual rooms*, and where the audience can go to a particular room to watch an ongoing performance, albeit with a different level of access (see Figure 2).

4.1 Collaboration support

[3]https://quaverseries.web.app

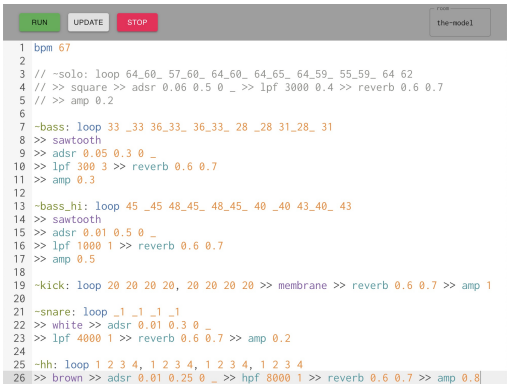


Figure 2: The QuaverSeries interface prototype. The syntax highlight has been implemented as a Ace editor theme. The buttons (`Run`, `Update` and `Stop`) are mapped to the keyboard shortcuts `Command/CTRL + Enter`, `Shift + Enter`, and `Command/CTRL + Period(.)`, respectively. The keyboard shortcut `Command/CTRL + Slash(/)` is for commenting out lines of code, which can be useful for muting a track during the performance.

Tools and algorithms such as Firebase and Operational Transformation have made the implementation of real-time code sharing much more approachable than it was only some years ago [5]. Firepad is an open-source tool that mainly uses Firebase realtime database and Operational Transformation algorithm. Thus, it provides a solution for synchronising code and sharing the cursor position between clients. In QuaverSeries, Firepad is used to share the code, while a customised strategy is designed to broadcast the related `run` and `update` commands to every client connecting to the database, including both the performers' and the audience members' clients. In this way, a live coder can control the sound running in all the browser clients. This is a similar strategy to what can be found in the Hydra synth, an environment developed for sharing visuals in the browser [7].

In the server database, two entries are storing the states of the `run` and `update`. Hence, once a user sends the `run` command by clicking the button or using the keyboard shortcut, the value of the entry `run` in the database will be set to the Boolean value `true`. As each client connecting to the database has its monitoring function for the value, once the Boolean value `true` is detected, each client will execute a relevant handling function. This function will do two things: 1) execute the code in the editor, 2) set the `run` entry back to the Boolean value `false`. Here is an example to illustrate this in pseudo code:

```
# the server
if Server get "run":
    send "run" to every client

# the client
if Client get "run":
    execute Music Code

# the interface
```

```
if Button "run" is pressed:
    sent "run" to Server
```

The principle of `update` is almost the same as `run`. The only difference is that `update` is used to renew the piece while the music is already on, that is, to calculate the current time and schedule what to play from the beginning of the next bar.

How does the system work in terms of stability? Fortunately, the transmission of only code between clients makes it possible to run the system over connections with very limited bandwidth. Furthermore, since the system is based on looped sequences, and an updating strategy per bar, allows for a considerable network delay without necessarily influencing the final musical result. This system design can, of course, be problematic if an `update` message is sent at the end of a bar. Still, the worst-case scenario is a one-bar offset among different locations. In our real-world testing so far, however, this has not been a problem in practice.

4.2 Live coding democratisation

Musical democratisation—in the sense of making music available to larger audiences—has been a growing trend ever since the invention of the phonograph [22]. Up until now, live coding has been an activity practised by relatively few. This is not only because it has been technically difficult to get started, but also because the community has been comparably small, and access to venues has been limited. Web-based live coding may help to address both of these problems, at the same time making it easier to provide access for online performances to get started.

QuaverSeries is a novel music streaming solution in that it does not stream audio or video but rather focuses on *code streaming*. Thus, instead of watching the audio/video of a performer's screen, the audience can enter a virtual room, watch new code appear on the screen, and have the musical sound rendered locally in the user's own browser. The audience can unidirectionally receive the `run` and `update` message from the server. This makes it possible to stop the rendering of music in the local browser at any time without influencing any other instances running on the machines of other performers or audience members.

The main difference between the performer and audience modes is that in the latter the code is not editable. Technologically speaking, though, every audience has the complete instrument locally, with the performers triggering the code. Thus every audience member could be seen as a collaborator and parttaker in the musicking.

5. CONCLUSION

In this paper we have presented the three parts of QuaverSeries: 1) a new domain-specific language, 2) an interface to edit and run the code, and 3) a new way of collaborating using 'virtual rooms.'

The environment draws extensively on new web technologies, utilising the power of local rendering in the user's browser thanks to the advance of the Web Audio API. This makes it possible to easily and quickly share live coding with lots of people, hence allowing the audience to become 'active participants' in a live coding session. Sharing only code between users makes it possible to create a collaborative environment with low network bandwidth. To minimise the risk of delay in the network, we have employed two strategies: 1) transmitting only code, and 2) calculating the updating based on musical bars. The trade-off of this approach, however, is that it also limits the musical possibilities of the system.

At the moment, the system is based on looped sequences only, resulting in beat-based music. Still, we have found that it is possible to create fairly complex musical results by adding multiple layers in the code. In future development, our first priority is to explore how it is possible for the audience to participate more in online performances. One approach is to build a chatting system in which the audience can write their own code, and propose this code to the performers. It would be up to the performer whether to accept the proposed code or not, similar to the way a *fork* works in the git version control system (such as used on GitHub).

QuaverSeries is currently at a prototype stage. It is fully functional, and we have explored it in a number of jam sessions. The aim now is to test it more in different performance contexts. We also plan to perform a more systematic user study, to understand more about how it works for beginning live coders.

6. ACKNOWLEDGMENTS

This work was partially supported by the Research Council of Norway through its Centres of Excellence scheme, project number 262762 and by NordForsk's Nordic University Hub Nordic Sound and Music Computing Network NordicSMC, project number 86892.

7. REFERENCES

[1] S. Aaron. Sonic pi–performance in education, technology and art. *International Journal of Performance Arts and Digital Media*, 12(2):171–178, 2016.

[2] S. Aaron and A. F. Blackwell. From sonic pi to overtone: creative musical experiences with domain-specific and functional languages. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 35–46. ACM, 2013.

[3] A. F. Blackwell and N. Collins. The programming language as a musical instrument. In *PPIG*, page 11, 2005.

[4] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised sound*, 8(3):321–330, 2003.

[5] C. A. Ellis and C. Sun. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. Citeseer, 1998.

[6] T. Hoogland. Mercury: a live coding environment focussed on quick expression for composing, performing and communicating. 2019.

[7] O. Jack. Livecoding networked visuals in the browser. https://github.com/ojack/hydra, 2019.

[8] D. Kim-Boyle. Network musics: Play, engagement and the democratization of performance. *Contemporary Music Review*, 28(4-5):363–375, 2009.

[9] R. Kirkbride. Foxdot: Live coding with python and supercollider. In *Proceedings of the International Conference on Live Interfaces*, 2016.

[10] R. Kirkbride. Troop: A collaborative tool for live coding. In *Proceedings of the 14th Sound and Music Computing Conference*, pages 104–9, 2017.

[11] T. Magnusson. ixi lang: a supercollider parasite for live coding. In *Proceedings of International Computer Music Conference 2011*, pages 503–506. Michigan Publishing, 2011.

[12] T. Magnusson. Code scores in live coding practice. In *TENOR 2015: International Conference on Technologies for Music Notation and Representation*, volume 1, pages 134–139. Institut de Recherche en Musicologie, 2015.

[13] T. Magnusson. *Sonic writing: technologies of material, symbolic, and signal inscriptions*. Bloomsbury Academic, 2019.

[14] Y. Mann. Interactive music with tone. js. In *Proceedings of the 1st annual Web Audio Conference*. Citeseer, 2015.

[15] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[16] S. McCoid, J. Freeman, B. Magerko, C. Michaud, T. Jenkins, T. Mcklin, and H. Kan. Earsketch: An integrated approach to teaching introductory computer music. *Organised Sound*, 18(2):146–160, 2013.

[17] C. McKinney. Quick live coding collaboration in the web browser. In *NIME*, pages 379–382, 2014.

[18] A. McLean. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 63–70. ACM, 2014.

[19] D. Ogborn, J. Beverley, L. N. del Angel, E. Tsabary, and A. McLean. Estuary: Browser-based collaborative projectional live coding of musical patterns. In *International Conference on Live Coding (ICLC) 2017*, 2017.

[20] C. Roberts, G. Wakefield, M. Wright, and J. Kuchera-Morin. Designing musical instruments for the browser. *Computer Music Journal*, 39(1):27–40, 2015.

[21] K. Stetz. Slang: An audio programming language built in js. https://github.com/kylestetz/slang, 2018.

[22] T. D. Taylor. The commodification of music at the dawn of the era of "mechanical music". *Ethnomusicology*, 51(2):281–305, 2007.

[23] A. Xambó, P. Shah, G. Roma, J. Freeman, and B. Magerko. Turn-taking and chatting in collaborative music live coding. In *Proceedings of the 12th International Audio Mostly Conference on Augmented and Participatory Sound and Music Experiences*, page 24. ACM, 2017.

APPENDIX

A. TECHNICAL STACK

The following table lists the main dependencies of QuaverSeries, and their current version number.

Dependencies	Version
React.js	16.2.8
Ohm.js	0.15.0
Tone.js	13.4.9
Material-ui	1.0.0
Firebase	7.0.0
Firepad	1.5.3
Ace	1.4.6

B. PARSER

This is how the parser of QuaverSeries is currently set up.

```
Quaver {

    Piece = Piece #"\\n"? #"\\n"? #"\\n"?
        Block --stack
    | Block

    Block = comment | Track

    comment = "//" c+

    c = ~"\\n" any

    Track = funcRef? ":"? Chain

    Chain = Chain ">>" Func -- stack
    | Func

    Func =  funcName listOf<funcElem,
        separator>

    funcElem = para | funcRef

    para = para subPara -- combine
    | subPara

    subPara = number | "_"

    number = "-"? digit* "." digit+ --
        fullFloat
    | "-"? digit "." -- dot
        | "-"? digit+ -- int

    funcRef = "~" validName

    funcName = validName

    validName = listOf<letter+, "_">

    separator = ","? space
}
```

Paper III

Towards Playing in the "Air": Modeling Motion-Sound Energy Relationships in Electric Guitar Performance Using Deep Neural Networks

Cagri Erdem, Qichao Lan, Julian Fuhrer, Charles Patrick Martin, Jim Torresen, Alexander Refsum Jensenius

Published in *Proceedings of the 17th Sound and Music Computing Conference*, 2020, pp. 177–184.

III

Towards Playing in the 'Air': Modeling Motion–Sound Energy Relationships in Electric Guitar Performance Using Deep Neural Networks

Çağrı Erdem
RITMO, Department of Musicology
University of Oslo
cagri.erdem@imv.uio.no

Qichao Lan
RITMO, Department of Musicology
University of Oslo
qichao.lan@imv.uio.no

Julian Fuhrer
RITMO, Department of Informatics
University of Oslo
julianpf@ifi.uio.no

Charles Martin
Research School of Computer Science
The Australian National University
charles.martin@anu.edu.au

Jim Tørresen
RITMO, Department of Informatics
University of Oslo
jimtoer@ifi.uio.no

Alexander Refsum Jensenius
RITMO, Department of Musicology
University of Oslo
a.r.jensenius@imv.uio.no

ABSTRACT

cagr

1. INTRODUCTION

Playing 'in the air' can be seen as a way of music appreciation [1], and also has potential for music expression [2]. But when you play an 'air instrument'—for example, the 'air guitar'—what are you actually playing? What kind of sound is supposed to be produced, and which strategies can be used in the design of such mappings? Our approach is based on the idea of letting the action–sound *couplings* found in acoustic instruments guide the design of action–sound *mappings* in a digital musical instrument [3]. The aim is not to recreate the action–sound couplings of (electro)acoustic guitar performance directly, but rather let them inspire the mappings in a new 'air instrument'.

There are several examples of air guitar instruments based on fairly coarse body movement, such as, the Virtual Air Guitar [4] and the Virtual Slide Guitar [5]. There are also more recent examples of using deep learning and computer vision to map between fingers and tones, for example, the 'deep air guitar' framework [6]. Such a camera-based approach is less useful in a performance scenario, since it is so dependent on the placement of the camera.

An alternative to using cameras to look at hands or fingers, is to use muscle information as the input of an air instrument. The muscle signals on the forearms are closely related to the finger movement, and the muscle signals can be measured by technologies such as electromyography (EMG) [7]. This approach is promising, and affordable muscle-sensing devices (such as the Myo) have been widely used in digital musical instrument designs [8]. Working with the muscle signals is not trivial, however, and often results in arbitrary mappings between action (captured as muscle signals) and the generated sound. In this paper, we therefore ask the question:

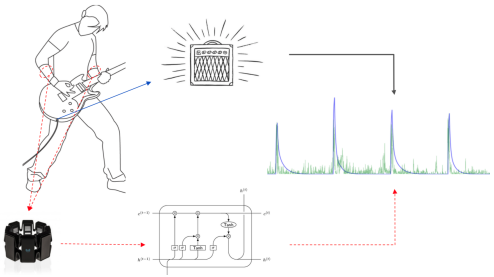


Figure 1. Overview of the data collection used in the study.

- How can we model relationships between action and sound in guitar playing, using muscle-sensing as the input?

This question has been broken down to two main challenges that will be presented in the following: (1) building a dataset that can be used for machine learning, and (2) developing a model based on the dataset. We first introduce the background of action–sound analysis and the application of machine learning in music performance. Then we elaborate on the data collection and the tools used for recording the dataset. Finally, we describe the model architecture and discuss the results.

2. BACKGROUND

2.1 Music-related body motion

Imagine a guitarist playing a song. For each chord to be strummed, the guitarist would lift a limb upwards, and move it back downwards to hit the strings. This process relies on *motion* and *force*. The former is defined as the continuous displacement of a limb or an object in space over time, while the latter refers to the push or pull experiences during the interaction. Force can set an object into motion, and motion can lead to the experience of force. While these can be objectively measured using a range of sensing devices (see, for example, [9] for an overview of different methods for sensing music-related body motion), we reserve the term *action* to what can be described as the

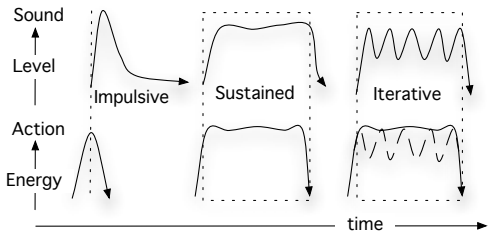


Figure 2. Illustration of the three basic action–sound types: impulsive, sustained, and iterative (from [14]).

goal-oriented *chunking* of such continuous physical phenomena, what Godøy and Leman refers to as 'cognitive units' [10].

2.2 Sound-producing actions

There are several types and categories of music-related body motion [11], but in this context we will primarily focus on the *sound-producing actions* that are responsible for note production. These can be further divided into *excitation* actions, such as the right hand that excites the strings, and *modification* actions, such as the left hand modifying the pitch. The excitation can be further divided into three main categories [12] (as sketched in Figure 2):

- *Impulsive*: fast attack, discontinuous energy transfer
- *Sustained*: gradual onset, continuous energy transfer
- *Iterative*: series of discontinuous energy transfer

Musical performances typically combine all these types in expressive ways. Drawing on such a conceptual apparatus, we can however assume the continuous music-related body motion/force as a series of goal points, which, when temporally close enough, can overlap and become *coarticulated* [13]. In other words, we can think of an entire instrumental performance in terms of such coarticulated combinations of the three aforementioned motion types.

2.3 Action–sound couplings and mappings

The relationships between action and sound in acoustic instruments are dictated by the laws of physics, and can be thought of as *action–sound couplings* [14]. However, digital musical instruments (DMIs) are based on the creation of *action–sound mappings*, in which the relationships between the physical energy of the input action may not necessarily correspond to that of the output sound. The creation of meaningful action–sound mappings in digital musical instruments is therefore critical for how they are perceived [15], and has been a central topic in the field of new interfaces for musical expression (NIME) over the last decades [16].

2.4 Machine learning in mapping design

Machine learning has been a part of NIME design since the early 1990s [17]. Well-known examples include the *Wekinator* [18], Gesture Follower [19], ml.* library [20], Gesture Recognition Toolkit (GRT) [21], Gesture Variation Follower (GVF) [22], and ml.lib [23]. These (and other) tools allow for using machine learning algorithms through either a graphical user interface (GUI), or, in the form of external libraries for audio programming platforms, such as Max/MSP and Pure Data. A number of new musical interfaces have employed such systems, such as Snyder's *The Birl* [24], Kiefer's use of Echo State Networks (ESNs) [25], and Schacher and colleagues' *Double Vortex* [26].

In recent years there has been an increasing interest in applications of deep neural networks (DNNs) for symbolic music generation or audio modelling. There are fewer musical examples of physical interaction (see, for example, [27] for an overview of deep predictive models in interactive music). A recent interactive music framework for deep learning is *IMPS*, which uses a mixture density network (MDN) over LSTM layers, and provides a low-entry-fee for musical exploration of DNNs [28]. Within instrument design, Gregorio's intelligent mapping structure [29], and McCormack et al's human–machine collaborative improvisation system [30] are some of the recent works.

2.5 Conceptual Idea

The central idea of this project is to investigate the action–sound couplings found in electric guitar performance, and use these for the creation of action–sound mappings in interfaces that do not rely on a physical controller. This can be thought of as the creation of technologies that allow for sonic interaction in the 'air' [2]. Previous research on the topic has primarily focused on capturing 'overt' motion, using optical or inertial motion tracking devices. The challenge is, then, how to exert effort while the haptic feedback of a physical interface is not available. To tackle this issue, we explore how the 'covert' muscle signals related to physical motion can be used for such interaction, in which the authors have been working on artistic-scientific projects in the recent years ([31, 32]). Thus, we focus on electromyographic (EMG) signals that represent the electrical activity of muscles [33].

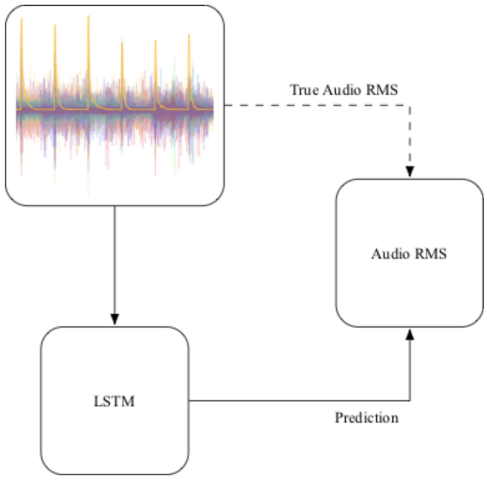


Figure 3. Sketch of the model: Raw EMG data and audio RMS are input to the neural network. The network (LSTM architecture) then outputs a predicted audio RMS.

The idea is to create a model of relationships between extracted muscle activity and sound features. The model is trained on raw EMG signals and the RMS of the resultant sound. Finally, the system is tested with the EMG input from freely improvised recordings.

3. DATA COLLECTION

3.1 Participants

A total of 36 music students and semi-professional musicians took part in the study, three of which were excluded due to incomplete data. Thus, our dataset consists of data from 33 participants (32 male, 1 female, mean age and standard deviation is 27 ± 7 years). All the participants had some formal training in playing the guitar, ranging from private lessons to university-level education. The recruitment was done through an online form published on the web site of the University of Oslo, which was announced in various communication channels targeting music students. Participation was rewarded with a gift card (valued approx. €30). The study obtained ethical approval from the Norwegian Centre for Research Data (NSD), with project number 872789.

3.2 Apparatus

Recordings took place in the fourMs motion capture lab at the University of Oslo. We recorded the audio at 16-bit 48 kHz using an Universal Audio Apollo Twin audio interface. All participants used the same performance setup: A Sadowsky Semihollow guitar with 11-49 gauge roundwound strings, a 1.5mm Jim Dunlop Tortex plectrum, a Roland AC-40 acoustic guitar amplifier (clean tone with all-flat equalizer settings) connected into the audio interface through the line output. The sound level was set to be comfortably loud for the participant.

We recorded the participants' muscle activity as surface EMG with two systems: consumer-grade Myo armbands and medical-grade Delsys Trigno. The former has a sample rate of 200 Hz, while the latter has a sample rate of 2000 Hz. Overt body motion was captured with a twelve-camera Qualisys Oqus infrared, optical motion capture system at a frame rate of 200 Hz. This system tracked the three-dimensional positions of reflective markers attached to each participant's upper-body and instrument. A trigger unit was used to synchronise the Qualisys and Delsys Trigno systems. We have also developed our own software for recording data from the Myo armband in synchrony with the audio (see Section 3.4). Regular video was recorded with a Canon XF105, synchronised with the Qualisys motion capture system.

For the current paper, only EMG data from the Myo will be considered, since the aim is to use the trained model in performance. Data from the Delsys system, as well as the motion capture and video recordings, have been used for reference only.

3.3 Procedure

The participants were recorded individually and were asked to perform warm-up, four specific performance tasks, and a final improvisation:

0. A warm-up improvisation with metronome at 70 bpm
1. Task 1
   (a) Softly played *impulsive* notes
   (b) Strongly played *impulsive* notes
2. Task 2
   (a) Softly played *iterative* 16^{th} notes
   (b) Strongly played *iterative* 16^{th} notes
3. Task 3
   (a) Softly played hammer-ons and pull-offs
   (b) Strongly played hammer-ons and pull-offs
4. Task 4
   (a) Softly played *sustained* semi-tone bending
      i. 'As fast as possible'
      ii. 'As slow as possible'
   (b) Strongly played *sustained* semi-tone bending
      i. 'As fast as possible'
      ii. 'As slow as possible'
5. A free improvisation (the tone features and the use of metronome are at the participant's discretion)

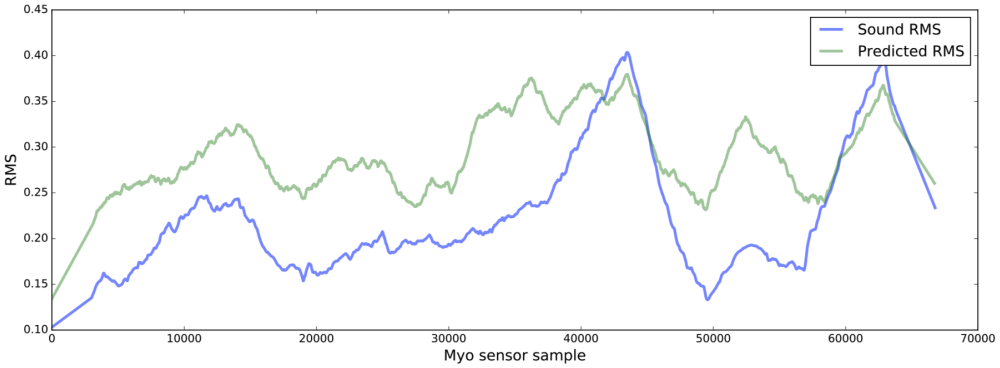


Figure 4. The RMS of the recorded sound and the model prediction. Both curves are processed with a Savitzky-Golay filter to reflect the general shape of the RMS comparison.

Figure 5. A participant during the recording session. Motion capture cameras can be seen hanging in the ceiling rig behind, and on stands in front of, the performer. The monitor with instructions can be seen below the front left motion capture camera.

Figure 6. Placement of the EMG sensors on the arms of the guitarists. Two delsys EMG sensors were placed on each side of the arm, right below the Myo armbands.

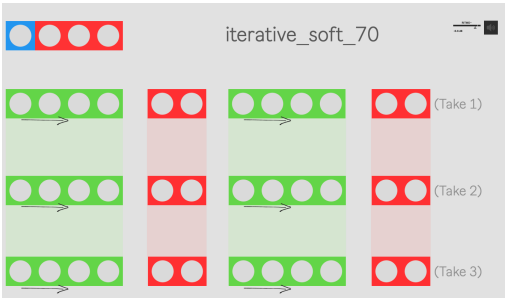


Figure 7. A screenshot of the 'prompter' that the participants would see on the screen in front of them during the experiment.

All the given tasks (1–5) focused on the notes B3 and C4 on the 4th (D) string played by index and middle fingers. Each task was recorded as a fixed-form track of duration 2'16", where participants were instructed to play for 4 bars, rest for 2 bars, and repeat the same pattern for 5 more times. All tasks are prompted through a Max/MSP patch on a screen (Figure 7), which allowed for a consistent and efficient experiment process.

3.4 Data Acquisition

We built a custom Python interface to record synchronised sensor data and audio. This was using our previously developed *myo-to-osc* bridge [34], which implements low-latency support for multiple Myo armbands connected via individual Bluetooth Low Energy (BLE) adapters. This is necessary to overcome possible bandwidth limitations. The latency can also be documented and eliminated after the recording. [1]

The data acquisition interface contains three parts: (1) data collection from the two Myo armbands, (2) generation of a metronome sound for the performers, and (3) au-

[1] https://github.com/chaosprint/dual-myo-recorder

dio recording using *PyAudio*. Audio and metronome timeline information was captured alongside the EMG data to simplify the segmentation and organisation of the training dataset.

3.5 Post-processing the data

To prepare data for our model, we first aligned EMG and audio arrays based on the recorded metronome timeline, then we applied interpolation on the EMG data and calculated the root mean square (RMS) from the audio signal.

3.5.1 Interpolation of the EMG data

The data recorded from Myo armbands needs to be pre-processed before it can be used for further analysis. This is to compensate for noise and possible data loss during recording. Here we solved this by performing a linear interpolation on the data. Since the data was recorded at a frequency of 200 Hz, the data loss is usually not more than a few samples. Thus, this additional step to account for the lost data should not create much of an error.

3.5.2 Root Mean Square of the audio signal

The root mean square (RMS) was calculated to reduce the dimension of the discrete signals and to characterise the signal. The RMS of a discrete signal $x = (x_1, x_2, \ldots, x_n)^\intercal$ with n components is defined by

$$\mathrm{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + \cdots + x_n^2}{n}}. \quad (1)$$

Even though it is a simple measure, the RMS can be argued to have both physical and perceptual significance. Its physical significance is related to the proportionality to the effective power of the signal. On average, one could argue that RMS is also correlated to perceptual loudness. The brain can judge whether a signal is loud, soft or in between, but it cannot infer where a periodic signal is peaking or is at a zero-crossing [35, 36]. Thus, for our purposes, RMS is a better feature than simply taking just the peak value within a given time interval.

4. DEVELOPING A MODEL

The aim of the model is to map the EMG data (raw muscle signals) to the RMS of the instrument's audio signal. Concretely, the input to the neural network is every 50 samples of the EMG recorded from all 16 channels of the two Myo devices (e.g. sample N 0 to 49, sample N 1 to 50, etc.). As we use the data from both hands, and each Myo has 8 analogue channels, there are 16 channels for each sample. The output of the neural network is the predicted sound RMS energy on the guitar.

The Long Short-Term Memory (LSTM) recurrent neural network (RNN) model was built in PyTorch [37], a popular model for time-series prediction. [2] As depicted in Figure 3, the LSTM network receives the raw EMG data and audio RMS, which were aligned during the pre-processing, and produces a predicted audio RMS. The training loss function was defined as

$$\begin{aligned}\mathcal{L}(x_{\mathrm{RMS}}, \hat{x}_{\mathrm{RMS}}) &= \frac{1}{n}\|x - \hat{x}_{\mathrm{RMS}}\|_2^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}(x_{\mathrm{RMS,i}} - \hat{x}_{\mathrm{RMS,i}})^2\,,\end{aligned} \quad (2)$$

where x_{RMS} are the recorded values, and $\hat{x}_{\mathrm{RMS}}$ are the values to be predicted. The sliding window has size n. The predicted RMS is computed according to Equation 1.

4.1 Training

A relatively small RNN was used for the training, consisting of five hidden layers and with 32 LSTM units in each layer. The window size of the input is 50, which is in line with the size of the input layer that is 50. For training, we used the data (excluding the improvisations) of 15 subjects out of 20 and validated it on the remaining subjects. We chose a batch size of 100 for determining the gradient of the cost function. Typically, at the first 5 epochs, the loss drops quickly and becomes stable after 10 epochs, which takes around 3 hours. Overall, we managed to finalize the training within the 12-hour limit of *Google Colab*'s graphics processing unit (GPU) resources.

4.2 Training result

The model is generally capable of predicting RMS. This can be seen in the figures of the recorded versus predicted RMS of the tasks of playing impulsive notes (Figure 9) and iterative 16th notes (Figure 10). For the latter, the model can generate a similar consecutive energy shape as a series of attacks.

We were also positively surprised to see that the model could predict the general trend of the sound energy in free improvisation tasks (Figure 4). This is the task that is most relevant for our ultimate goal of creating an 'air instrument' to perform with. So we are particularly pleased that the model can, indeed, account for this, at least on a level of the sound envelope.

An interesting result can be seen for the prediction of the bending task (Figure 11). Although we describe three distinct motion types in Section 2.2 (impulsive, sustained, iterative), regular performance on the guitar does not afford sustained motion during the excitation phase (it could be done with a bow on the strings, but not with a plectrum). In other words, one can hit on a string either once (impulsive), or as a series of impulses (iterative). However, sustained motion is available for the modification action, such as, bending the string with a finger on the left hand. Therefore in the prediction, we observe a longer decay when compared to a impulsive, single attack of the right arm. We think that this is an interesting in-between result, which can be further interpreted as an augmentation of the guitar for creative purposes. In other words, this also shows that the model is promising for expanding the player's control space.

[2] https://github.com/cerdemo/air_model

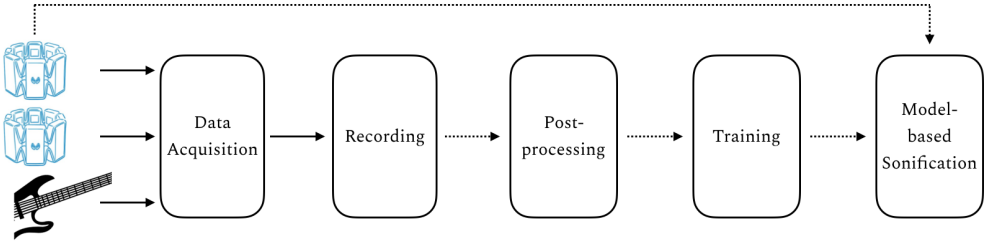


Figure 8. Simplified signal flow diagram of the system.

4.3 Testing the model

The predicted features were tested using a preliminary sonification strategy. Here the trained model was fed with 16 channels of raw EMG test data to generate predicted RMS values. These values were then imported (as CSV files) into a Max/MSP patch that runs through the values at the same rate as the Myo armband (200 Hz). The sonification is built around a simple Karplus-Strong algorithm programmed in the Gen environment within Max/MSP, where the RMS value is mapped to the *decay* and *damping* parameters of the physical model. This effectively 'shapes' the white noise to simulate a (guitar-like) plucked string sound. For further sound design purposes, we use a sine-wave-based low frequency oscillator (LFO), and a fair amount of pitch shifting. This creates a lower octave of the sound that 'feels' similar to sub-frequencies of naturally resonating bodies of acoustic instruments and speakers.

The demonstration video[3] is structured as an alternation between the originally recorded sounds, and an offline sonification that relies on the predicted RMS values mapped to the temporal envelope of the sound synthesis. The onsets are extracted from the predicted values within the Python script, and stored in the CSV file along with the RMS values. In the video, it is easily noticeable that when playing a series of fast attacks during the *iterative* task, onsets of the 'air instrument' often lose the action–sound synchrony. This reveals an important weakness of the strategy. As such, it motivated us towards modelling the entire spectrum of the recorded audio, which will allow more reliable onset detection algorithms based on the spectral flux.

5. CONCLUSIONS

The paper has presented a method for building a neural-network model based on recordings of action–sound couplings from (electro)acoustic guitar performance. We show that the model can predict the overall trend of the sound energy (measured as RMS) of a freely improvised performance, solely based on a training dataset of particular motion types.

As part of the data collection, we had to develop a solution for low-latency recording of multiple Myo armbands, synchronised with audio and metronome. We also developed tools for post-processing the data including an interpolation algorithm to compensate the sample loss happened in Bluetooth transmission. This framework can be applied to the analysis and modelling of action–sound relationships in playing a variety of acoustic and digital instruments. As such, we will openly share our dataset and tools in service of further scientific and artistic studies.

Although no systematic evaluation has taken place, our sonification experiment shows that the trained model can be used reliably to control the 'feel' of an 'air instrument', using only muscle sensing as input. As such, we believe that creating models trained on recorded action–sound couplings from acoustic instruments is a promising strategy for the design of action–sound mappings in DMIs.

Of course, the prediction of a single temporal feature is insufficient for capturing the complexity of musical sound. The next step is therefore to expand the model with spectral, temporal and spatial features. This will allow for a wider and more flexible sound palette in real-time musical settings. Furthermore, how to use the space, how to structure the time, and how to interact with the audience and/or ensemble members while playing muscle-based 'air' instruments, introduce a number of conceptual, practical, and technological challenges. Thus, the relationship between different design considerations and the spatiotemporality of the performance will be explored. Future work should also focus on conducting a thorough user study of the model's use in real-time. We will also conduct a series of analysis on the 'muscle–sound' relationships, in order to improve the model and diversify its potential output. Finally, we also see the potential for conducting other types of analyses on the gathered dataset. It would be particularly interesting to perform a more systematic analysis of the different types of action–sound couplings, and how they were captured by the different recording devices (EMG, video, motion capture, sound). One can also envisage between-participant comparisons, to reveal individual differences. With an interdisciplinary approach that draws on sound theory and embodied music cognition, we can design more 'economical' deep learning models for music interaction. The results of such analyses could also prove valuable when improving the modelling framework and further sonification strategies.

Acknowledgments

This work was partially supported by the Research Council of Norway (# 262762) and NordForsk (# 86892).

[3] Video is available at `http://bit.ly/air_guitar_smc`

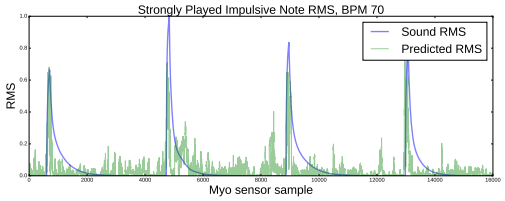


Figure 9. The RMS of the recorded sound and the model prediction for the impulsive note playing task.

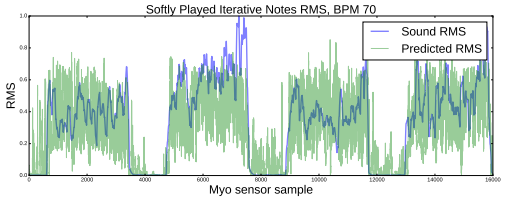


Figure 10. The RMS of the recorded sound and the model prediction for the iterative notes playing task.

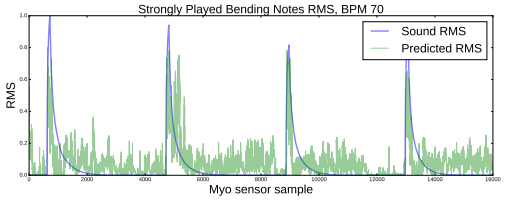


Figure 11. The RMS of the recorded sound and the model prediction for the bending (*sustained*) note playing task.

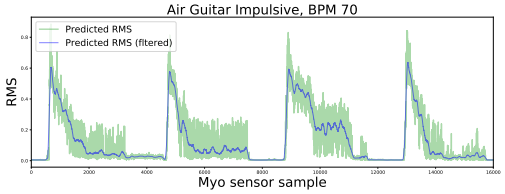


Figure 12. The predicted sound RMS of impulsive playing in the 'air' (as demonstrated in the video excerpts).

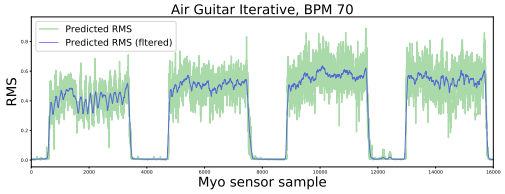


Figure 13. The predicted sound RMS of iterative playing in the 'air' (as demonstrated in the video excerpts).

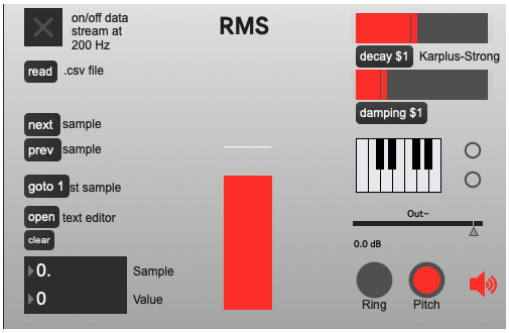


Figure 14. The user interface of the sonification patch.

6. REFERENCES

[1] R. I. Godøy, E. Haga, and A. R. Jensenius, "Playing "air instruments": Mimicry of sound-producing gestures by novices and experts," in *Gesture in Human-Computer Interaction and Simulation*, S. Gibet, N. Courty, and J.-F. Kamp, Eds. Berlin, Heidelberg: Springer, 2006, pp. 256–267.

[2] A. R. Jensenius, "Sonic microinteraction in "'the air"'," in *The Routledge Companion to Embodied Music Interaction*, M. Lesaffre, P.-J. Maes, and M. Leman, Eds. New York, NY: Routledge, 2017, ch. 46, pp. 431–439.

[3] ——, "An Action-Sound Approach to Teaching Interactive Music," *Organised Sound*, vol. 18, no. 2, pp. 178–189, 2013.

[4] M. Karjalainen, T. Mäki-Patola, A. Kanerva, and A. Huovilainen, "Virtual air guitar," *Journal of the Audio Engineering Society*, vol. 54, no. 10, pp. 964–980, 2006.

[5] J. Pakarinen, T. Puputti, and V. Välimäki, "Virtual slide guitar," *Computer Music Journal*, vol. 32, no. 3, pp. 42–54, 2008.

[6] Google, *Teachable Machine*. https://teachablemachine.withgoogle.com/, 2020.

[7] G. KaMen and E. Kinesiology, "Research methods in biomechanics," *Champaign, IL, Human Kinetics Publ*, 2004.

[8] A. Tanaka, "Embodied musical interaction," in *New Directions in Music and Human-Computer Interaction*. Springer, 2019, pp. 135–154.

[9] A. R. Jensenius, "Methods for studying music-related body motion," in *Springer Handbook of Systematic Musicology*. Springer, 2018, pp. 805–818.

[10] R. I. Godøy and M. Leman, *Musical gestures: Sound, movement, and meaning*. Routledge, 2010.

[11] A. R. Jensenius and M. M. Wanderley, "Musical gestures: Concepts and methods in research," in *Musical Gestures*. Abingdon, UK: Routledge, 2010, pp. 24–47.

[12] R. I. Godøy, "Gestural-sonorous objects: embodied extensions of schaeffer's conceptual apparatus," *Organised sound*, vol. 11, no. 2, pp. 149–157, 2006.

[13] ——, "Understanding coarticulation in musical experience," in *International Symposium on Computer Music Multidisciplinary Research*. Marseille, France: Springer, 2013, pp. 535–547.

[14] A. Jensenius, "Action – sound - developing methods and tools to study music-related body movement," Ph.D. dissertation, University of Oslo, 01 2007.

[15] A. Hunt and M. M. Wanderley, "Mapping performer parameters to synthesis engines," *Organised sound*, vol. 7, no. 2, pp. 97–108, 2002.

[16] A. R. Jensenius and M. J. Lyons, *A NIME Reader: Fifteen Years of New Interfaces for Musical Expression*. New York, NY: Springer, 2017, vol. 3.

[17] M. Lee, A. Freed, and D. Wessel, "Real-time neural network processing of gestural and acoustic signals," in *Proc. of the Int. Computer Music Conf.*, Montreal, Quebec, Canada, 1991, pp. 277–277.

[18] R. A. Fiebrink, *Real-time human interaction with supervised learning algorithms for music composition and performance*. Princeton, NJ: Citeseer, 2011.

[19] F. Bevilacqua, B. Zamborlin, A. Sypniewski, N. Schnell, F. Guédy, and N. Rasamimanana, "Continuous realtime gesture following and recognition," in *International gesture workshop*, Bielefeld, Germany, 2009, pp. 73–84.

[20] B. D. Smith and G. E. Garnett, "Unsupervised play: Machine learning toolkit for max." in *Proc. Int. Conf. on New Interfaces for Musical Expression*, Ann Arbor, MI, 2012.

[21] N. Gillian and J. A. Paradiso, "The gesture recognition toolkit," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3483–3487, 2014.

[22] B. Caramiaux, N. Montecchio, A. Tanaka, and F. Bevilacqua, "Adaptive gesture recognition with variation estimation for interactive systems," *ACM Transactions on Interactive Intelligent Systems (TiiS)*, vol. 4, no. 4, p. 18, 2015.

[23] J. Bullock and A. Momeni, "Ml. lib: robust, cross-platform, open-source machine learning for max and pure data." in *Proc. Int. Conf. on New Interfaces for Musical Expression*, Baton Rouge, LA, 2015.

[24] J. Snyder and D. Ryan, "The birl: An electronic wind instrument based on an artificial neural network parameter mapping structure." in *Proc. Int. Conf. on New Interfaces for Musical Expression*, London, UK, 2014.

[25] C. Kiefer, "Musical instrument mapping design with echo state networks," in *Proc. Int. Conf. on New Interfaces for Musical Expression*, London, UK, 2014.

[26] J. C. Schacher, C. Miyama, and D. Bisig, "Gestural electronic music using machine learning as generative device," in *Proc. Int. Conf. on New Interfaces for Musical Expression*, Baton Rouge, LA, 2015.

[27] C. P. Martin, K. O. Ellefsen, and J. Torresen, "Deep predictive models in interactive music," *arXiv preprint arXiv:1801.10492*, 2018.

[28] C. P. Martin and J. Torresen, "An interactive musical prediction system with mixture density recurrent neural networks," *arXiv preprint arXiv:1904.05009*, 2019.

[29] J. Gregorio and Y. Kim, "Augmenting parametric synthesis with learned timbral controllers," in *Proc. Int. Conf. on New Interfaces for Musical Expression*, Porto Alegre, Brazil, 2019.

[30] J. McCormack, T. Gifford, P. Hutchings, M. T. Llano Rodriguez, M. Yee-King, and M. d'Inverno, "In a silent way: Communication between ai and improvising musicians beyond sound," in *Proc. of the Conf. on Human Factors in Computing Systems*. Glasgow, UK: ACM, 2019, p. 38.

[31] C. Erdem, A. Camci, and A. Forbes, "Biostomp: a biocontrol system for embodied performance using mechanomyography." in *Proc. Int. Conf. on New Interfaces for Musical Expression*, Copenhagen, Denmark, 2017.

[32] C. Erdem, K. H. Schia, and A. R. Jensenius, "Vrengt: A shared body-machine instrument for music-dance performance," in *Proc. Int. Conf. on New Interfaces for Musical Expression*, Porto Alegre, Brazil, 2019.

[33] A. Phinyomark, E. Campbell, and E. Scheme, "Surface electromyography (emg) signal processing, classification, and practical considerations," in *Biomedical Signal Processing*. Springer, 2020.

[34] C. P. Martin, A. R. Jensenius, and J. Torresen, "Composing an ensemble standstill work for myo and bela," in *Proc. Int. Conf. on New Interfaces for Musical Expression*, Blacksburg, VA, 2018.

[35] L. L. Beranek and T. J. Mellow, "Chapter 1 - introduction and terminology," in *Acoustics: Sound Fields and Transducers*, L. L. Beranek and T. J. Mellow, Eds. Academic Press, 2012, pp. 1 – 19.

[36] M. R. Ward, *Electrical engineering science*. New York, NY: McGraw-Hill, 1971.

[37] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS Autodiff Workshop*, Long Beach, CA, 2017.

Paper IV

Exploring relationships between effort, motion, and sound in new musical instruments

Cagri, Erdem, Qichao Lan, Alexander Refsum Jensenius

Published in *Human Technology*, November 2020, Volume 16, Issue 3, pp. 314–347.

IV

HUMAN TECHNOLOGY

ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 16(3), November 2020, 310–347

EXPLORING RELATIONSHIPS BETWEEN EFFORT, MOTION, AND SOUND IN NEW MUSICAL INSTRUMENTS

Çağrı Erdem
RITMO Centre for Interdisciplinary Studies in Rhythm, Time and Motion
University of Oslo
Norway

Qichao Lan
RITMO Centre for Interdisciplinary Studies in Rhythm, Time and Motion
University of Oslo
Norway

Alexander Refsum Jensenius
RITMO Centre for Interdisciplinary Studies in Rhythm, Time and Motion
University of Oslo
Norway

Abstract: *We investigated how the action–sound relationships found in electric guitar performance can be used in the design of new instruments. Thirty-one trained guitarists performed a set of basic sound-producing actions (impulsive, sustained, and iterative) and free improvisations on an electric guitar. We performed a statistical analysis of the muscle activation data (EMG) and audio recordings from the experiment. Then we trained a long short-term memory network with nine different configurations to map EMG signal to sound. We found that the preliminary models were able to predict audio energy features of free improvisations on the guitar, based on the dataset of raw EMG from the basic sound-producing actions. The results provide evidence of similarities between body motion and sound in music performance, compatible with embodied music cognition theories. They also show the potential of using machine learning on recorded performance data in the design of new musical instruments.*

Keywords: *EMG, music, machine learning, musical instrument, motion, effort, guitar, embodied.*

DOI: https://doi.org/10.17011/ht/urn.202011256767

INTRODUCTION

What are the relationships between action and sound in instrumental performance, and how can such relationships be used to create new instrumental paradigms? These two questions inspired the experiments presented in this paper. Our research is based upon two basic premises: It is possible to find relationships between the continuous, temporal shape of an action and its resultant sound and that embodied knowledge of an existing instrument can be translated into a new performative context with different instrument. Thus, we are interested in exploring whether it is possible to create mappings in new instruments based on measured actions on and sounds from an existing instrument. It is common to create such action–sound mappings based on overt motion features. However, in our study, we were interested primarily in exploring whether covert muscle signals can be used for new musical instruments.

Embodied Knowledge

The body's role in the experience of sound and music is central to the embodied music cognition paradigm (Leman, 2008). Several studies have explored the embodiment of musical experiences by investigating how musicians and nonmusicians transduce what they perceive as musical features into body motion. Sound-tracing is one such experimental paradigm that has been used to study how people spontaneously follow salient features in music (Kelkar, 2019; Kozak, Nymoen, & Godøy, 2012; Nymoen, Caramiaux, Kozak, & Torresen, 2011). Sound mimicry is a similar approach, based on examining how sound-producing actions can be imitated "in the air," that is, without a physical interface (Godøy, 2006; Godøy, Haga, & Jensenius, 2005; Valles, Martínez, Ordás, & Pissinis, 2018). Several other studies have aimed at identifying musical mapping strategies, drawing on concepts of embodied music cognition as a starting point (e.g., Caramiaux, Bevilacqua, Zamborlin, & Schnell, 2009; Françoise, 2015; Maes, Leman, Lesaffre, Demey, & Moelants, 2010; Tanaka, Donato, Zbyszynski, & Roks, 2019; Visi, Coorevits, Schramm, & Miranda, 2017).

In this study, we took bodily imitation as the starting point for the creation of action–sound mappings. The idea was to transfer the acquired skills of playing traditional instruments to a new context. Here the term traditional refers to the recognizability of performance skills, what Smalley (1997) explained as an intuitive knowledge of action–sound causalities in traditional sound-making. The idea was to exploit such proprioceptive relationships between musician and instrument (Paine, 2009). The premise is that skill can be understood as embodied knowledge (Ingold, 2000) that leads to lower information processing at a cognitive level (Dreyfus, 2001). It also builds upon the idea that spectators can perceive and recognize skill as an embodied phenomenon (Fyans & Gurevich, 2011).

One outcome of this research was aimed at developing solutions for creating musical instruments that can be performed in the air. However, it should be clear from the start that we are not interested in making "air" versions of the guitar or any other physical instrument. Rather, our attention is devoted to reusing the embodied knowledge of one type of instrumental performance in new ways (Magnusson, 2019). The lack of a haptic and tactile experience creates a significantly different experience when playing a physical instrument as compared to a touchless air instrument. According to the "gestural agency" concept of Mendoza Garay & Thompson (2017), the instrument is as much an agent in the musical transaction as the performer:

They influence each other within a musical ecosystem. In this system, the agents' communication is multimodal. Therefore, the act of instrument playing accommodates not only the auditory, tactile, and haptic channels but also the visual, kinetic, proprioceptive, or any other kind of interactions that have a musical influence. The human agent becomes the participant that is expected to adapt; thus, any change in the environment can be seen as a creative challenge.

From Body Motion to Musical Actions

Gesture is employed frequently in the literature on music-related body motion (Cadoz & Wanderley, 2000; Gritten & King, 2011; Hatten, 2006). We understand gesture as related to the meaning-bearing aspects of performance actions. In this project, we focus not on such meaning-bearing aspects and thus will not use that term in the following discussion. Instead, we will use *motion* to describe the continuous displacement of objects in space and time, and *force* to explain what sets these objects into motion. Both motion and force are physical phenomena that can be captured and studied using various devices (see Jensenius, 2018a, for an overview of various methods for sensing music-related body motion). Hitting a guitar string is an example of what we call motion, which can be studied through motion capture data of the arm's continuous position. Muscle tension is an example of the force involved in the sound production and can be studied through electromyography (EMG).

Motion and force describe the kinematic and kinetic aspects of performance, respectively. These relate to—but are not the same as—the experienced action within a performance (Jensenius, Wanderley, Godøy, & Leman, 2010). Thus, in our research, we use *action* to describe a cognitive phenomenon that can be understood as goal-directed units of motion and/or force (Godøy, 2017). Many actions are based on visible motion, but an action also can be based solely on force. For example, some electroacoustic musical instruments are built with force-sensitive resistors that can be pressed by the performer, even without any visible motion. Hence the player's action can change drastically over time even with no or only little observable body motion.

Music-related body motion comes in various types (see Jensenius et al., 2010, for an overview). Here we primarily focus on the *sound-producing actions*. These can be subdivided into *excitation* actions, such as the right hand that excites the strings on a guitar, and *modification* actions, such as the left hand modifying the pitch. The excitation action can be divided further into the three main categories proposed by Schaeffer (2017), as sketched in Figure 1: *impulsive*, *sustained*, and *iterative*. An impulsive excitation is characterized by a fast attack and discontinuous energy transfer, while a sustained excitation has a gradual onset and continuous energy transfer. An iterative excitation is based on a series of discontinuous energy transfers.

Action–Sound Coupling and Mappings

Sound production on a traditional instrument is bound by the physical constraints of the instrument and the capabilities of human body. For example, although both are plucked instruments, a banjo, and an oud have different damping characters due to the resonant features of the instruments' bodies. The physical properties of the instruments also define their unique timbre and how they are played. Additionally, the human body has its expressive limitations. These limitations can be in

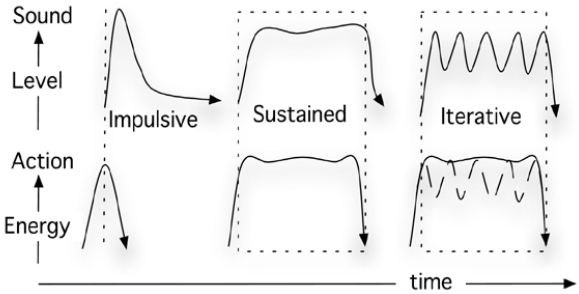


Figure 1. Illustration of the three, basic action–sound types: impulsive, sustained, and iterative (Jensenius, 2007; Used with permission).

the form of what Godøy (2018) suggested as "effort constraints," meaning "limits to endurance," which necessitate an optimization of muscle contractions (i.e., to prevent injuries). He described these limitations as also leading to "coarticulation," which results from multiple individual actions merging into larger units. All these levels of constraints are part of the transformation of biomechanical energy to sound features. We think that during the transformations in *action–sound couplings* (Jensenius, 2007), the relationships between actions and sounds are dictated by the laws of physics.

When playing a traditional instrument, one must exercise muscular exertion to abide by the instrument's physical boundaries. In the case of the guitar, this prevents the player from breaking a string due to excessive effort or not producing sound due to the lack of energy input (Tanaka, 2015a). After centuries of design, the construction of traditional instruments is no longer open to much interpretation, except for using some extended playing techniques or additional equipment. To the contrary, electroacoustic musical instruments are based on the creation of *action–sound mappings*. Here the constraints of hardware and/or software elements often are open to interpretation. In other words, the relationships between biomechanical input and the resultant sound are designed and may not correspond to each other. However, the creation of meaningful action–sound mappings is critical for how an instrument's playing and its sound are perceived (Hunt & Wanderley, 2002; Van Nort, Wanderley, & Depalle, 2014). This is often discussed as the "mapping problem" (Maes et al., 2010), which has been a central research topic in the field of new interfaces for musical expression over the last decades (Jensenius & Lyons, 2017).

New Musical Interactions

The number of artists and researchers interested in using the human body as part of their musical instrument has been growing over the last decades. Such interests often lead to the use of gestural controllers, which are types of wearable sensors or camera-based devices that allow for touchless performance, that is, a type of performance not based on touch of physical objects. As such, these instruments allow for sonic interaction in the air (Jensenius, 2017). Examples of such instruments are the Virtual Air Guitar (Karjalainen, Mäki-Patola, Kanerva, & Huovilainen, 2006), the Virtual Slide Guitar (Pakarinen, Puputti, & Välimäki, 2008), and Google's Teachable Machine, which lets users mimic guitar-playing in front of a web camera (Google, 2020).

The above-mentioned examples focus mainly on creating an air guitar. However, this is not the focus of our current research; rather, we seek to explore new ways of performing in the air. Although motion-based tracking often is employed for air instruments, we are interested specifically in measuring muscle tension through electromyography (EMG). When worn on the forearm, EMG sensors can provide muscle activation information related to the motion of hand and fingers (Kamen, 2013). EMG goes beyond measuring limb positions and provides information of the muscle articulation throughout the preparation for and execution of an action (Tanaka, 2019). The use of muscle activation data in musical performance was pioneered by Knapp & Lusted (1990) and has been practiced extensively by Tanaka (1993, 2015b). Mechanomyograms (MMGs), as a signal for muscle-based performance (Donnarumma, 2015), also have been studied.

Performing in the air introduces several conceptual and practical challenges. For example, when does a sound-producing action begin and end when no physical instrument defines the performance space? How can one handle the use of physical effort as part of that action without being restricted to a physical instrument? To address such problems, we drew on what Tanaka (2015a) suggested as an embodied interaction strategy: He replaced constraints, such as those experienced while playing a traditional instrument, with "restraints," that is, the "internalization of effort" (p. 299). Such restraints can help define a set of affordances that can replace the physical constraints found in a traditional instrument.

Even though we are interested in creating new instrument concepts, this may not necessarily require developing an entirely new action–sound repertoire. Michel Waisvisz, the creator of *The Hands* (Waisvisz, 1985), focused on maintaining the action–sound mappings of his instrument. This helped him develop and maintain a skill set over time. We propose a design strategy based on what Magnusson (2019) referred to as an "ergomimetic" structure. Here *ergon* stands for work memory and *mimesis* for imitation. Such an ergomimetic structure may help in reusing well-known interactions of a performer in a new performative context. Of course, such an approach raises some questions. For example, what types of errors and surprises emerge when a physical pipeline is replaced by software? We aim through our research to contribute to better understanding how a musician's physical skills could transfer to new air instruments.

Machine Learning

Machine learning is a set of artificial intelligence techniques for tackling tasks that are too difficult to solve through explicit programming; it is based on finding patterns in a given set of examples (Fiebrink & Caramiaux, 2016). Deep learning is a subset of machine learning, where artificial neural networks allow computers to understand complex phenomena by building a hierarchy of concepts out of simpler ones (Goodfellow, Bengio, & Courville, 2016). Machine learning has been an important component in the design of and performance with new interfaces for musical expression since the early 1990s (Lee, Freed, & Wessel, 1991). Several easy-to-use tools have been developed over the years for artists and musicians (see, e.g., Caramiaux, Montecchio, Tanaka, & Bevilacqua, 2015; Fiebrink, 2011; Martin & Torresen, 2019), and many new instruments have explored the creative potential of artificial intelligence in music and performance (Caramiaux & Donnarumma, 2020; Kiefer, 2014; Næss, 2019; Schacher, Miyama & Bisig, 2015; Tahiroğlu, Kastemaa & Koli, 2020). However, unlike the applications for generating music in the form of musical instrument digital interface (MIDI)

data (Briot, Hadjeres, & Pachet, 2020) or generating music in the wave-form domain (Purwins et al., 2019), the use of deep learning techniques for interactive music is rather rare. We see that deep learning can be particularly useful when dealing with complex muscle signals.

Research Questions

The brief theoretical discussion above has shown that a number of questions remain open regarding how musical sound is performed and perceived and how it is possible to create new empirically based sound-making strategies. Thus, in the current two-experiment study, we were interested particularly in

1. What types of muscle signals are found in electric guitar performance and how do these signals relate to the resultant sound?
2. How can we use deep learning to predict sound based on raw electromyograms?

We begin by explaining the methodological framework that has been developed for the first empirical study, followed by a presentation and discussion of the results. We then reuse some of the data from the first experiment to pursue a preliminary predictive model for action–sound mappings. We conclude with a general discussion of the findings of these two experiments.

EXPERIMENT 1: MUSCLE–SOUND RELATIONSHIPS

Methods

Research Design

This aspect of our research is based on the outcomes of an experiment with electric guitar players. Each of the guitarists performed, while wearing various sensors, a set of basic sound-producing actions as well as free improvisations. To collect the data these actions produced, we built a multimodal dataset of EMG and motion capture data; additionally, video and sound recordings of each performer were made. For this paper, we focus only on a statistical analysis of the EMG data and sound recordings from this first experiment, with a particular emphasis on similarity measures. Prior to conducting the research, we obtained ethical approval from the Norwegian Center for Research Data (NSD), Project Number 872789.

Participants

Thirty-six music students and semiprofessional musicians took part in the study. Five of the datasets turned out to be incomplete and these were excluded from further analysis. Thus, the final dataset consisted of 31 participants (30 male, 1 female, M_{age} = 27 years, SD = 7), all right-handed. All the participants had some formal training in playing the electric guitar, ranging from private lessons to university level education. The recruitment was conducted through an online invitation published on a specified web site of the University of Oslo, Norway, and announced in various communication channels targeting music students. Participation was rewarded with a gift card (valued at approximately €30).

Data Collection

The participants' muscle activity was recorded as surface EMG with two systems: consumer-grade Myo armbands and a medical-grade Delsys Trigno system. The former has a sample rate of 200 Hz, while the latter has a sample rate of 2000 Hz. Overt body motion was captured with a 12-camera Qualisys Oqus infrared optical motion capture system at a frame rate of 200 Hz. This system tracked the three-dimensional positions of reflective markers attached to each participant's upper body and the instrument. A trigger unit was used to synchronize the Qualisys and Delsys Trigno systems. Additionally, we developed a custom-built software solution to capture data from the Myo armbands in synchrony with the audio. Regular video was recorded with a Canon XF105 camera, which was synchronized with the Qualisys motion capture system. Figure 2 demonstrates the two major means for gathering data: the motion-capture configuration and the EMG system.

Procedure

Each participant was recorded individually. One recording session took 90-105 minutes. First, the participants received a brief explanation about the experiment, before they signed the consent form. Following the recording session, they completed a short survey regarding their musical background, their use of musical equipment, and their thoughts on new instruments and interactive music systems.

The participants were instructed to stand at the same marked spot in the laboratory. We asked them to perform tasks based on well-known electric guitar techniques. The hammer-on and pull-off are similar techniques that allow the performer to play multiple notes connected in a legato manner (tied together). In both techniques, the left-hand fingers hit multiple notes with a single excitation action. Hammer-on refers to bringing down another finger with sufficient force to hit a

(a) (b)

Figure 2. (a) A participant during the recording session. Motion capture cameras are visible hanging in the ceiling rig behind and on stands in front of the performer. The monitor with instructions for the performer can be seen below the front left motion capture camera. **(b)** The protocol used for placement of the EMG electrodes: Two Delsys EMG sensors were placed on each side of the arm corresponding to the extensor carpi radialis longus and flexor carpi radialis muscles, just below the Myo armbands.

neighboring note on the fretboard. Pull-off refers to moving the finger from one fret to another to modify the pitch. Bending is achieved by a finger pulling or pushing the string across the fretboard to smoothly increase the pitch. The given tasks were as follows:

- A warm-up improvisation with metronome at 70 bpm
- Task 1
  - Softly played impulsive notes B and C in 3rd and 4th octaves, respectively
  - The same task, played strongly
- Task 2
  - Softly played iterative notes
    - Single pitch (B3)
    - Double pitches (B3–C4)
  - The same task, played strongly
- Task 3
  - Softly played legato
  - The same task, played strongly
- Task 4
  - Softly played bending (semi-tone)
  - The same task, played strongly
- A free improvisation (the tone features and the use of metronome are at the participant's discretion)

We based the tasks on performing guitar-like versions of each of the three action–sound types. Tasks 1 and 4, for instance, lie somewhere in between classes considering that the right hand excites the string in an impulsive manner while the left hand keeps sustaining the tone as much as the construction of the instrument allows. In Task 2, participants were asked to alternate between single and double pitches in different takes. Finally, Task 3 presents a hybrid of the impulsive and sustained types. All given tasks focused on the notes B3 and C4 on the D string, played by index and middle fingers.

Each task was recorded as a fixed-form track, 2 min 16 s in duration, along with a metronome click at 70 BPM. The participants were instructed to play for 4 bars, rest for 2 bars, play the variation for 4 bars, rest another 2 bars and repeat this same 12-bar pattern two more times. See Table 1 for a detailed list of finger and style variations. To help the participants perform the tasks correctly, they were standing in front of a custom-built prompter screen. On the screen, they could follow animated circles, which signified the beat and the bar they were supposed to be at with respect to the predefined form of the given task. This allowed for a more comfortable and efficient experiment process. For the pilot study, we used a text-based prompting. However, this increased the cognitive load of the participants. Thus, for the full experiment we implemented a simple geometry-based design.

Table 1. Detailed Fingerings and Playing Styles Instructed to Participants for Particular Tasks.

	Takes 1-3-5	Takes 2-4-6
Impulsive	Index	Middle
Iterative	Index	Index–middle
Bending	Middle, as fast as possible	Middle, as slow as possible
Legato	Index–middle, hammer-on	Middle–index, pull-off

Note. Fingering and playing styles were organized based on the odd- and even-numbered takes to have a systematic approach to labeling different action features recorded within a single track. This approach facilitated the groupings of segmented individual takes during the preprocessing step.

Data Acquisition

Figure 3 shows the recording setup, which was based on two separate personal computers running the data collection software. In the first one, we used an external trigger to send the start pulse to the Qualisys motion capture system, which allowed an in-sync recording of the motion capture cameras, the Delsys Trigno EMG sensors, and the Canon video camera. The second computer recorded signals from the Myo armbands and the audio as line input from the guitar amplifier. This was accomplished using a custom-built Python program to record synchronized sensor data and audio. The Myo armbands were interfaced through improving the myo-to-osc framework for the Bluetooth API (Martin, Jensenius, & Torresen, 2018). To overcome possible bandwidth limitations, we implemented low-latency support for the multiple Myo armbands connected to the computer via individual Bluetooth Low Energy adapters. PyAudio was used for the audio recording (Pham, 2006). The Python interface ran as four simultaneous processes: data acquisition from each armband, the metronome, and the audio recording.

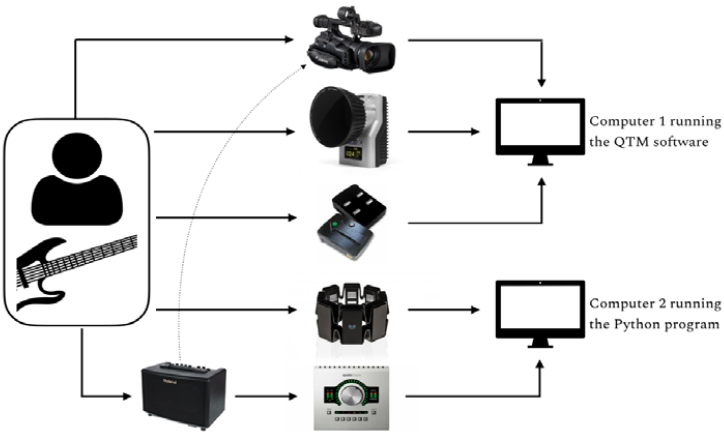


Figure 3. A simplified signal flow diagram of the experimental setup. Representative pictures of the equipment used, from top to bottom: Canon video camera, Qualisys Oqus infrared camera, Delsys Trigno electrodes, Myo armband, and Roland guitar amplifier, and Universal Audio Apollo Twin sound card.

Preprocessing

Preprocessing of our data for further analysis and modeling purposes was handled separately for the data from the Delsys and Myo systems. The medical-grade Delsys system provided high-quality data suitable for analytical purposes, while the Myo is a consumer-grade product that works well for interactive applications (see Pizzolato et al., 2017, for a comparison of various EMG acquisition setups). For the Delsys data, preprocessing included filtering, segmentation, and feature extraction methods. For the Myo data, we worked on interpolation and alignment of the raw data instead.

Synchronization

We synchronized the recorded data and audio through a custom-built metronome script within our Python program. This script recorded the timestamps of the metronome clicks together with the start point of the audio recording in a CSV file. This strategy helped in two ways. First, we could calculate lags at less than 0.1s among the various recording channels. As a result, we could align all the data types, based on their start points, to the metronome timeline. The synchronization strategy also helped in conforming the Qualysis data captured on Computer 1 with the line-audio recordings on Computer 2. Computer 1 ran the Qualisys software, which also recorded a standard video file synchronized with embedded audio.

We first extracted the audio stream from the video recording, and then decomposed the signal into its percussive and harmonic components. Applying an onset detection algorithm on the percussive component made it possible to obtain a timeline of metronome clicks from the ambient audio recording. This allowed us to measure the clicks and compare them to the logged timestamps of the original metronome clicks from Computer 2. Because the Delsys data shared the same timestamps with those of the metronome onsets, and the line audio recording shared the same timestamps with those of the metronome logs, we were able to align all the recorded data and media.

EMG Signal

Drawing on the method proposed by De Luca, Gilmore, Kuznetsov, & Roy (2010), we recorded the raw EMG data at 2000 Hz using the Delsys Trigno system, which were first run through a high-pass filter with a cutoff frequency of 20 Hz, and a low-pass filter with a cut-off of 200 Hz. Both filters were fourth-order Butterworth type (Selesnick & Burrus, 1998). Next, we segmented the synchronized and normalized EMG data into 5-beat sequences (1 bar created from the last beat of the previous bar in the timeline). This was to capture also muscle activation preceding the sound-producing action. The muscle activation necessarily precedes the motion of the hand and the audio onset.

Each task was recorded as a single track that contained six takes (see Table 1). Then, we selected one segment from each of them following this protocol:

1. Takes that featured the index finger on B3 were chosen from the impulsive and iterative tasks. In addition to an effort for narrowing the scope by focusing on the index finger for the impulsive task, we were interested in exploring how two motion types combine in the iterative task.

2. Takes that were played "as slow as possible" were chosen from the bending task. Slow bending (over a period of approximately a bar) is fairly similar to the sustained motion type. The guitar does not actually afford sustained performance in the same way as, for example, a violin does. However, the more the bending is prolonged, the more the damping is shortened. This results in two almost opposing input and output amplitude envelopes. The sustaining muscle amplitude envelope has an increased tension. The sound energy, on the contrary, decays quicker than that of an impulsive attack.
3. Takes that featured the hammer-on technique were chosen from the legato task. We observed that a majority of the participants was more comfortable with the hammer-on technique than a pull-off. This was also something we observed in the recorded data. In addition, hammer-on can be seen as a variation of the impulsive tasks played with both fingers.

Finally, each segment was divided into four EMG channels (i.e., the extensor and flexor muscles of each forearm). This resulted in 992 segments (31 participants, 8 tasks, 4 channels) of EMG data. Each segment had a duration of 4.29 s.

For the feature extraction, we were interested primarily in the amplitude envelopes. This was extracted as the root mean square (RMS) of the continuous signal. The moving RMS of a discrete signal is defined by St-Amant, Rancourt, & Clancy (1996) as

$$\hat{x}_1(t) = \left[\frac{1}{N}\sum_{i=t-N+1}^{t} m^2(i)\right]^{1/2}$$

where $\hat{x}$ is the EMG amplitude estimate at sample t, using a smoothing window length of N. The recommended window length for calculating the RMS of an EMG signal is 120–300 ms (Burden, Lewis, & Willcox, 2014). After several trials, we noticed that shorter window lengths better covered the peaks of fast attacks. Thus, we used a 50 ms sliding window with 12.5 ms (25%) overlaps.

Muscle onsets were calculated using the Teager-Kaiser Energy (TKE) operation to improve the accuracy of the detection (Li, Zhou, & Aruin, 2007). The TKE operation is defined in the time domain as

$$y(n) = x^2(n) - x(n-1)x(n+1)$$

Audio Signal

The sound analysis was based primarily on the RMS envelopes. Additionally, we computed the spectral centroid (SC) of the sound, as it has been shown to correlate with the perception of brightness in sound (Schubert, Wolfe, & Tarnopolsky, 2004), that is, how the spectral content is distributed between high and low frequencies. The RMS signal is particularly relevant in that our primary interest in this study is in the amplitude envelope of the sound. RMS correlates with perceptual loudness; people can judge whether a signal is loud, soft, or in between but cannot infer where a periodic signal is peaking or is at a zero-crossing (Beranek & Mellow, 2012; Ward, 1971). Thus, for our purposes, RMS served as an appropriate feature, providing more information than simply identifying the peak value within a given time interval.

Analysis

Our analysis focused on exploring similarities between the amplitude envelopes of the EMG signals and the sound. We achieved this by comparing the beginning and the end of the body–sound interactions identified when playing the electric guitar. Muscle activation was observable at the beginning, followed by motion, and then the resulting sound. We conducted the entire analysis through in a custom-built toolbox programmed in Python.

EMG Analysis

The initial component of the EMG analysis focused on exploring the similarities between the RMS of each of the four channels (two per arm) and the sound RMS for each of the participants. We used a Pearson's product–moment correlation, Spearman's rank correlation, and analysis of variance.

Also known as linear correlation coefficient (LCC), Pearson's product–moment correlation is a parametric correlation of the degree to which the change in one variable is linearly associated with a change in another continuous variable. In its equation form, LCC is commonly abbreviated as r while, in our case, x and y represent EMG and audio signals, respectively,

$$r = \frac{\sum(x - \overline{x})(y - \overline{y})}{\sqrt{\sum(x - \overline{x})^2 \sum(y - \overline{y})^2}}$$

where $LCC > 0$ denotes a positive correlation while the opposite ($LCC < 0$) refers to an inverse correlation. The LCC approaches 0 when the correlation weakens. To our knowledge, this measure has not been used to compare audio and EMG signals.

A common assumption of the Pearson's correlation is that the continuous variables follow a bivariate normal distribution. In other cases, where the data is not normally distributed and the relationship of two variables rather seems nonlinear, the Spearman's rank correlation (SCC) is suggested to measure the monotonic relationship (Schober, Boer, & Schwarte, 2018). SCC is fairly similar to LCC, but it calculates the ranks of the pair of values. It is abbreviated as r_s (or ρ) in its mathematical representation where D is the difference between ranks and n denotes the number of data pairs:

$$r_s = 1 - \frac{6\sum D^2}{n(n^2 - 1)}$$

A positive r_s denotes a covariance toward the same direction, whereas a negative r_s refers to fully opposite directions. It is a correlation measure that is commonly used in validating EMG data (Fuentes del Toro et al., 2019; Nojima, Watanabe, Saito, Tanabe, & Kanazawa, 2018).

A third approach was to calculate the pairwise *t* tests and one-way analysis of variance (ANOVA) to explore the variances of correlation values across participants and different dynamics. Here, we tested the assumptions of normality and homogeneity of variances of the independent samples in the dataset using the Shapiro-Wilk and Levene tests (Virtanen et al., 2020), respectively.

In addition to the above-mentioned analysis strategies, we explored other representations of the EMG signals. Inspired by Santello, Flanders, & Soechting (2002) and González Sánchez, Dahl, Hatfield, & Godøy (2019), we applied the time-varying Principal Component Analysis

(PCA) to merge all four channels and investigate prominent features across all participants. The input matrix for the PCA is defined as $A \in \mathbb{R}^{m \times n}$ where m is the number of participants and n denotes the number of EMG channels. For each of the 8 tasks, in which half employed soft dynamics and the other half strong dynamics, we obtained two principal components (PCs), which represented a combination of both excitation and modulation actions on the guitar, as shown by the following equation,

$$EMG_{\mathrm{m}} = \mathrm{meanEMG}_{\mathrm{m}} + PC1 \times EMG1_{\mathrm{m}} + \cdots + PCn \times EMGn_{\mathrm{m}}$$

Additionally, we applied Singular Spectrum Analysis (SSA) to principal components of EMG for further signal–noise separation. SSA is a technique of time series analysis used for decomposing the original series by means of a sliding window into a sum of small number of interpretable components, such as slowly varying trend, oscillatory (periodic) components, and structureless noise (Golyandina & Zhigljavsky, 2013). The algorithm for SSA is similar to that of PCA in multivariate data. In contrast to the PCA, which is applied to a matrix, SSA provides a representation of the given time series in terms of a matrix made of the time series (Alexandrov, 2009). In this way, we applied SSA on the EMG principal components and extracted the trend, which is a smooth additive component that contains information about the time series' global change (Alexandrov, Bianconcini, Dagum, Maass, & McElroy, 2012). This procedure allowed us to obtain better visualizations of the nonlinearity of relationships between EMG and audio waveforms.

It should be noted that researchers in the literature have suggested a variety of specialized methods for choosing the SSA window length (L). Knowing that it is highly difficult to define a universal method to find an optimal L value for an arbitrary time series and that the practitioners should therefore investigate this issue with care, Khan & Poskitt (2011) suggested a rule as $L = (\log N)^c$ with $c \in (1.5, 3.0)$ for assigning a window length that will yield near optimal performance. Starting from there, as the RMS segments of our interest were at a fixed length of $N = 344$, we empirically chose $c = 2.5$, which yielded $L = 10$.

Video Analysis

We used the Musical Gestures Toolbox (Jensenius, 2018b) to extract the sparse optical flow from the video recordings, with the goal of identifying to what extent participants moved unintentionally. This information allowed us to make comparisons with other data at hand and open a better understanding of unexpected muscle activations.

Sound Analysis

Our aim in the sound analysis was to quantify how the different dynamics influenced the overall brightness of the sound. To this end, we averaged the SC across all participants. Note that the sound data in this study is presented in approximately 4.29 s chunks. However, we also investigated chunks of a shorter duration in order to explore whether dynamic fluctuations of particularly the iterative task had an effect on the mean brightness. Moreover, considering the damping character of the guitar, which is relatively short in duration, we explored how decay times influenced the overall brightness value.

Results

The 36 participants completed 360 tasks in total. However, we excluded five datasets due to incomplete data. After also excluding the improvisations—which were intended to be used in the modeling experiment detailed below—we analyzed 248 tasks from 31 participants. An overview of how muscle activation patterns transform to sound features in each task is illustrated in Figure 4.

LCC and SCC

The correlation coefficients among participants were computed using the LCC and SCC measures. Table 2 shows positive correlation, negative correlation, mean, and standard deviation for each factor. Figures 5 and 6 show the distribution of LCC and SCC correlations.

The analysis shows to what extent the muscle activation underlying the sound-producing motion and the resultant sound on the same musical instrument can have similar amplitude envelopes. This is supported by the ANOVA results. The correlation of muscle–sound amplitude envelopes—whether positive, negative, or close to 0—does not exhibit a noteworthy variance between participants. That is, the ANOVAs for EMG–sound similarities across participants (for all EMG channels and tasks) are as follows: LCC, $F(30,961) = 1.6$, $p = 0.02$, and SCC, $F(30,961) = 1.59$, $p = 0.02$.

The comparisons of the correlation values between left and right hands supports the functional distinction between the right and left actions (see Table 3). Another clear distinction was revealed when we compared to what extent the EMG and sound envelopes correlated with respect to soft and strong dynamics (see Table 4). When the participants played strongly, the muscle and resultant sound amplitude envelopes correlated better.

PCA and SSA

Figure 7 shows the waveforms of the two principal components of the combined EMG channels across all participants for impulsive, iterative, bending, and legato tasks, separately for soft and strong dynamics. Each panel shows the activation patterns for the characteristics of these tasks.

The trends of the same principal component waveforms via signal–noise separation were extracted using SSA ($L = 10$) and have been plotted against the averaged sound RMS on the horizontal axis in Figure 8. Here we can observe the varying level of nonlinearities of the muscle–sound relationship for the tasks played at different dynamic levels.

Spectral Centroid

Figure 9 shows the distribution of the SC of the sound across all participants for each soft and strong task, separately. Although stronger dynamics show a clear strength in the upper end of the sound spectrum, the distribution among particular tasks varied depending on the chosen timescale. As such, SC values of all tasks with soft dynamics ($M = 299.03$, $SD = 124.24$), compared to the SC values of tasks with strong dynamics ($M = 585.93$, $SD = 141.22$), demonstrated significantly lower mass of the spectrum, $t(246) = 16.98$, $p < .001$

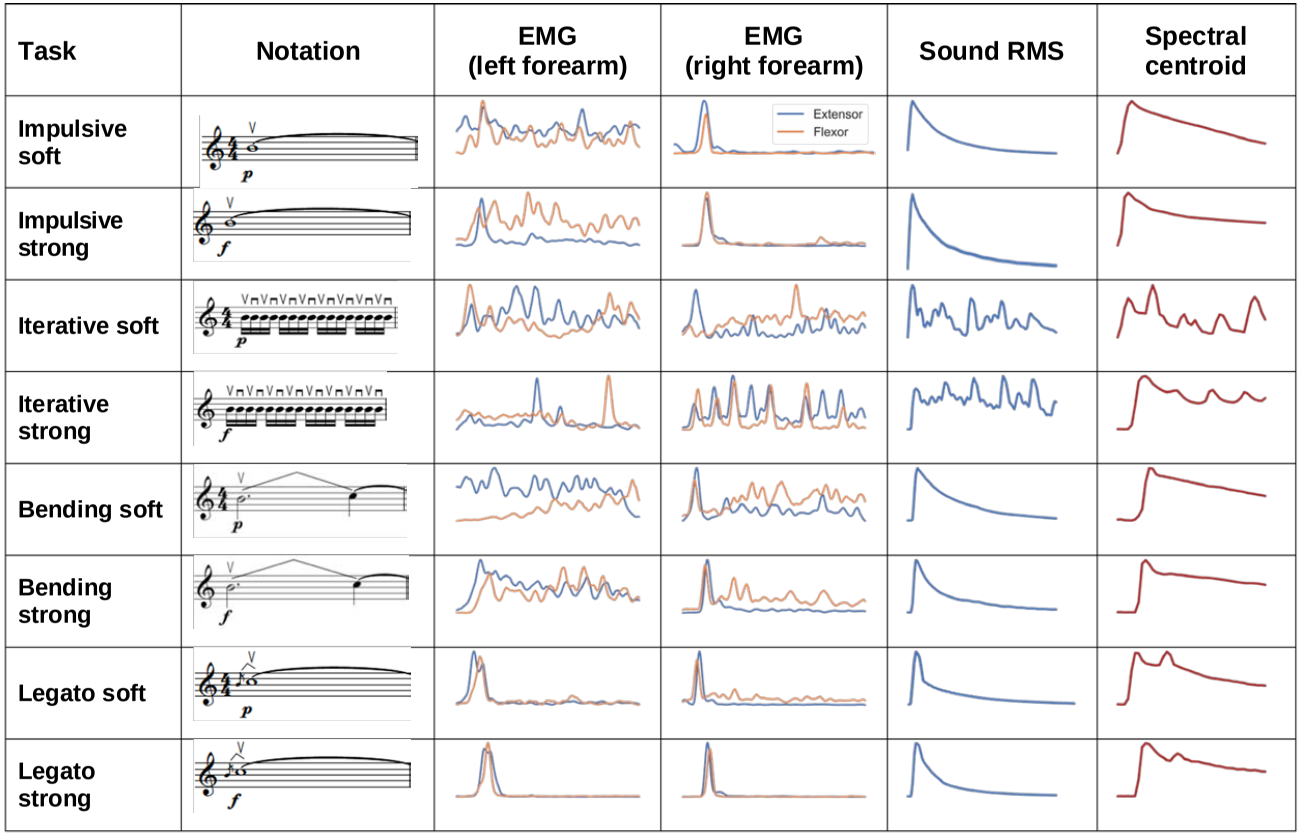


Figure 4. An overview of how notated music transforms into an audio waveform when playing the electric guitar. Trends of signals were extracted using Singular Spectrum Analysis (SSA) with a window length $L = 10$.

Table 2. Correlation Coefficients for Each Factor (LCC and SCC): The Positive, Negative, Mean and Standard Deviation of Correlation Coefficients.

			Impulsive soft	Impulsive strong	Iterative soft	Iterative strong	Bending soft	Bending strong	Legato soft	Legato strong
LCC	r	Extensor (right)	0.66	0.59	0.64	0.68	0.60	0.73	0.46	0.53
		Flexor (right)	0.65	0.54	0.51	0.86	0.65	0.69	0.42	0.55
		Extensor (left)	0.72	0.62	0.74	0.64	0.63	0.76	0.44	0.60
		Flexor (left)	0.55	0.55	0.65	0.65	0.48	0.63	0.51	0.48
	$-r$	Extensor (right)	–0.24	–0.03	–0.24	–0.24	–0.12	–0.10	–0.38	–0.24
		Flexor (right)	–0.34	–0.25	–0.10	–0.07	–0.34	–0.10	–0.33	–0.32
		Extensor (left)	–0.66	–0.61	–0.35	–0.35	–0.51	–0.66	–0.35	–0.33
		Flexor (left)	–0.62	–0.62	–0.53	–0.51	–0.54	–0.46	–0.30	–0.53
	μ	Extensor (right)	0.17	0.24	0.28	0.33	0.26	0.28	0.00	0.09
		Flexor (right)	0.13	0.23	0.22	0.33	0.21	0.27	0.02	0.03
		Extensor (left)	–0.23	–0.08	0.21	0.25	0.18	0.22	–0.02	0.01
		Flexor (left)	–0.34	–0.24	0.20	0.21	0.03	0.15	–0.01	–0.02
	σ	Extensor (right)	0.23	0.14	0.17	0.18	0.18	0.19	0.15	0.20
		Flexor (right)	0.25	0.17	0.17	0.19	0.21	0.17	0.13	0.18
		Extensor (left)	0.35	0.36	0.26	0.23	0.27	0.24	0.16	0.16
		Flexor (left)	0.28	0.25	0.28	0.20	0.14	0.22	0.14	0.12

(continued)

Table 2. Correlation Coefficients for Each Factor (LCC and SCC): The Positive, Negative, Mean and Standard Deviation of Correlation Coefficients. (continued)

			Impulsive soft	Impulsive strong	Iterative soft	Iterative strong	Bending soft	Bending strong	Legato soft	Legato strong
SCC	r_s	Extensor (right)	0.66	0.71	0.68	0.71	0.58	0.78	0.55	0.61
		Flexor (right)	0.49	0.71	0.58	0.74	0.66	0.74	0.27	0.66
		Extensor (left)	0.65	0.84	0.77	0.81	0.81	0.84	0.66	0.42
		Flexor (left)	0.70	0.70	0.69	0.63	0.43	0.70	0.43	0.34
	$-r_s$	Extensor (right)	–0.45	–0.15	–0.25	–0.30	–0.14	–0.17	–0.42	–0.33
		Flexor (right)	–0.41	–0.43	–0.18	–0.04	–0.41	–0.19	–0.19	–0.42
		Extensor (left)	–0.85	–0.89	–0.56	–0.56	–0.61	–0.85	–0.32	–0.61
		Flexor (left)	–0.77	–0.78	–0.50	–0.50	–0.62	–0.78	–0.55	–0.61
	μ	Extensor (right)	0.08	0.27	0.25	0.41	0.27	0.35	–0.01	0.10
		Flexor (right)	0.07	0.26	0.17	0.38	0.18	0.37	0.01	0.02
		Extensor (left)	–0.27	–0.08	0.27	0.35	0.19	0.25	0.00	0.00
		Flexor (left)	–0.38	–0.26	0.21	0.29	0.04	0.17	0.00	0.00
	σ	Extensor (right)	0.22	0.19	0.20	0.23	0.15	0.25	0.14	0.25
		Flexor (right)	0.24	0.21	0.19	0.19	0.18	0.25	0.12	0.20
		Extensor (left)	0.40	0.46	0.31	0.23	0.30	0.24	0.14	0.14
		Flexor (left)	0.31	0.31	0.31	0.23	0.16	0.26	0.13	0.10

Note. The zeros in the table represent rounded values that were smaller than three decimal places, thus a “close-to-zero” correlation.

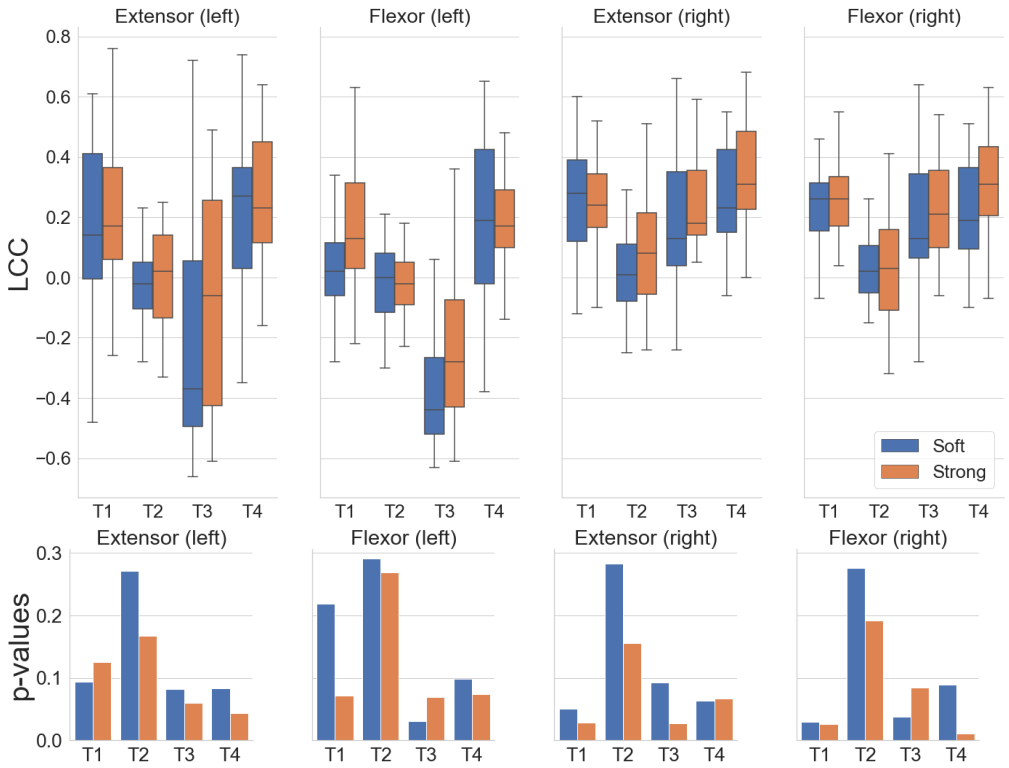


Figure 5. Pearson's product–moment correlations between EMG and Sound RMS envelopes. LCC > 0 denotes a positive correlation while LCC < 0 refers to the negative. The box plots show the interquartile ranges of correlation distribution per task, separately for soft and strong dynamics. The bar plots below show the distribution of *p*-values showing the significance of the correlations. T1, T2, T3 and T4 refer to impulsive, iterative, bending and legato tasks, respectively.

Table 3. Pairwise *t* tests Demonstrating How Modification (Left Forearm) and Excitation (Right Forearm) Actions Have Distinct EMG–Sound Amplitude Envelopes.

	Modification action	**Excitation action**	**Variance**
LCC	$M = 0.03$, $SD = 0.30$	$M = 0.19$, $SD = 0.21$	$t(495) = 11.41$, $p < .001$
SCC	$M = 0.05$, $SD = 0.34$	$M = 0.20$, $SD = 0.24$	$t(495) = 9.04$, $p < .001$

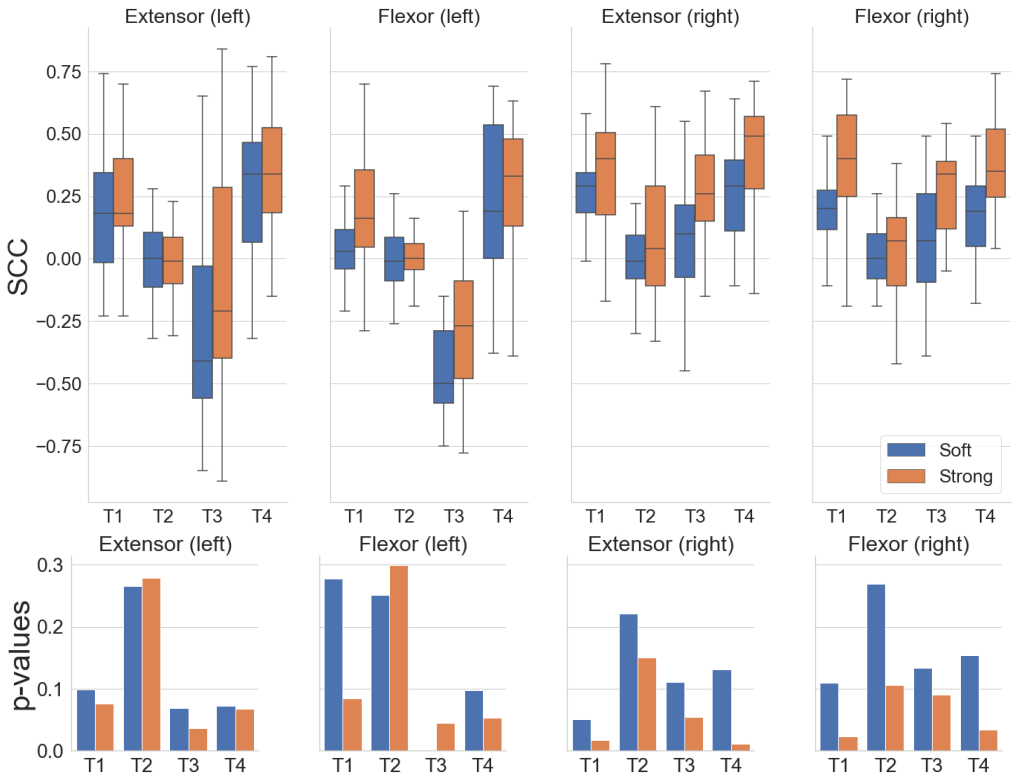


Figure 6. Spearman's rank correlations between EMG and Sound RMS amplitude envelopes. SCC > 0 denotes a covariance in the same direction while SCC < 0 refers to the opposite direction. The box plots show the interquartile ranges of correlation distribution per task, separately for soft and strong dynamics. The bar plots below show the distribution of *p*-values showing the significance of the correlations. T1, T2, T3 and T4 refer to impulsive, iterative, bending and legato tasks, respectively.

Table 4. Means, Standard Deviations and *t*-scores for LCC and SCC Metrics.

	Soft	Strong	Variance
LCC	$M = 0.08$, $SD = 0.27$	$M = 0.14$, $SD = 0.26$	$t(495) = 5.41$, $p < .001$
SCC	$M = 0.07$, $SD = 0.29$	$M = 0.18$, $SD = 0.31$	$t(495) = 8.33$, $p < .001$

Note. Pairwise t-tests show EMG–sound amplitude envelopes correlations between soft and strong dynamics.

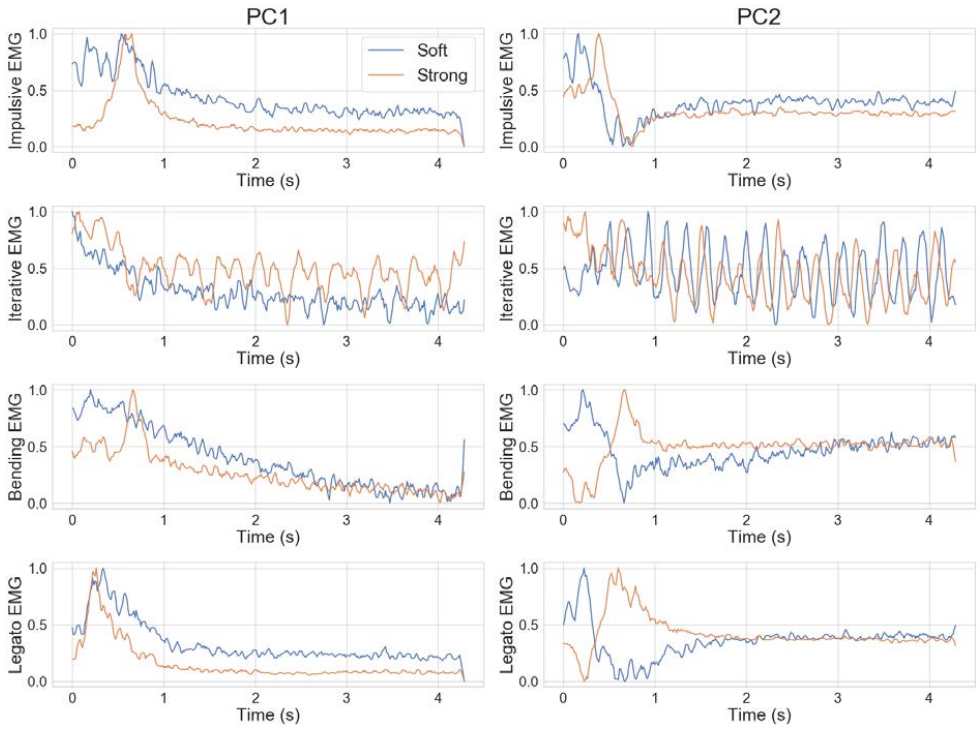


Figure 7. Two principal components (PC1 and PC2) of the combined left and right forearm EMG data of all participants rescaled to (0,…,1) (See the text for more information about the PCA analysis).

Discussion

The analyses showed that sound production on musical instruments is a phenomenon that involves many physical and physiological processes. For example, Figure 10 shows the activation patterns of the extensor and flexor muscles during down- and up-stroking using a plectrum. This figure illustrates only two muscles groups from the right forearm. However, a musical note often is produced as a more complex combination of both arms, as shown in Figure 4.

Similarity Between EMG and Sound Shapes

Our experiment results show that the relations between the muscle energy envelope and the envelope of the resultant sound have similarities between participants. The results show a significant variance when comparing attacks with soft and strong dynamics using pairwise *t*-tests (Table 4). As shown in Figures 5 and 6, the correlation values are higher, and the directionality is more apparent when the same task is played with strong dynamics. This may be due to two factors. First, greater energy input results in larger sound amplitude, which is less biased to base noises, such as the inherent postural instability of the human body.

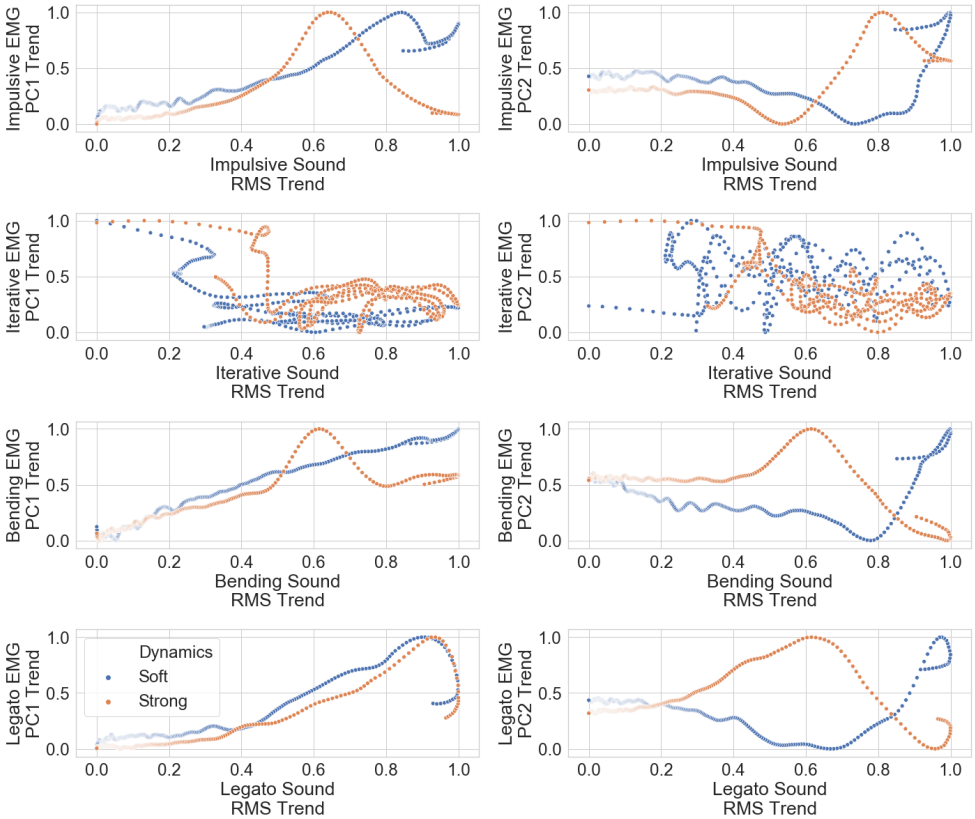


Figure 8. Decomposed principal components (PC1 and PC2) against resultant Sound RMS of all participants (SSA window length *L = 10*). The plots show to what extend the EMG and resultant sound RMS envelopes have a linear relationship at every time step.

Second, we know that expert players tend to use less tension in the forearm muscles (Winges, Furuya, Faber, & Flanders, 2013). Most of our participants can be considered semiprofessionals and thus may have felt less comfortable with stronger dynamics. As a result, they may have employed forearm muscles more explicitly. Unfortunately, we do not have data to check this hypothesis.

The results in Table 3 are in line with the conceptual distinction provided in our Introduction. The excitation action, which typically is performed by the right arm for right-handed players, determines the main characteristics of the resultant sound amplitude envelope. The difference between the activation patterns of both forearms is also observable in Figure 4. The impulsive tasks noted on the top two rows, for example, show the right forearm muscles have envelopes similar to that of the resultant sound while the activation patterns from the left forearm seem to resemble a continuous sound envelope, somewhat between the sustained and iterative types. This is due mainly to a continuous effort exerted by the left forearm over the period of the given task, which is different from the right forearm that excites the string once,

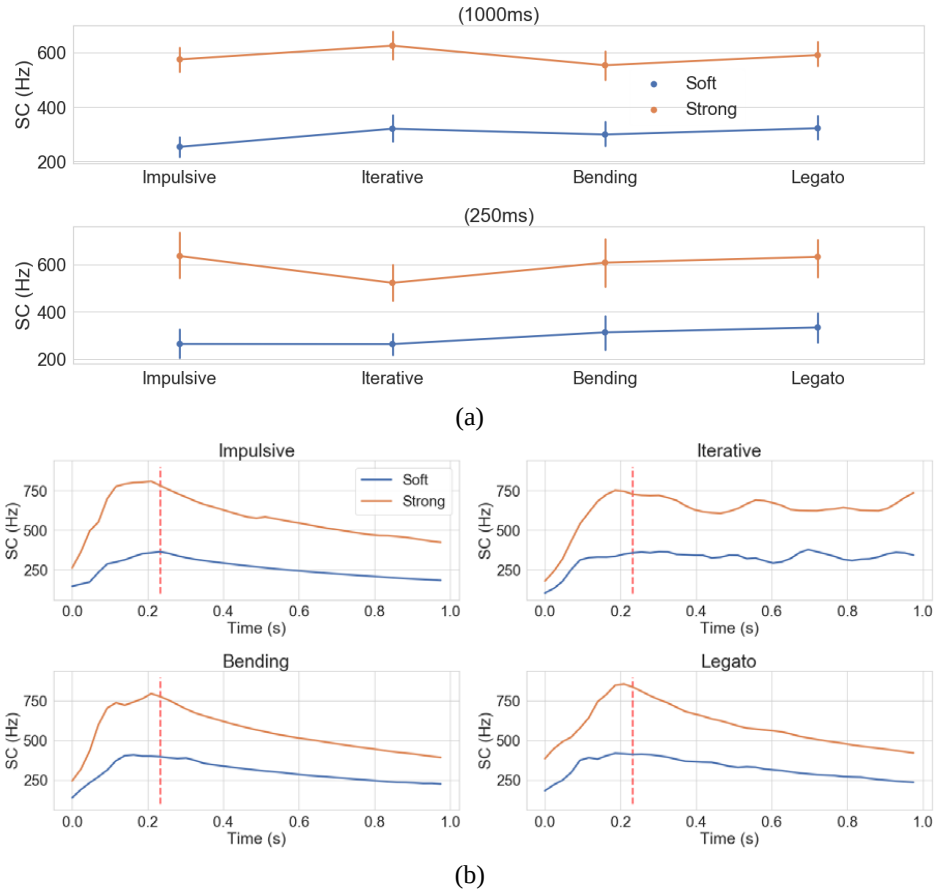


Figure 9. Spectral centroid (SC) of the resultant sound (a) SC distribution between soft and strong dynamics in chunks of 1000 ms and 250 ms duration. (b**)** SC envelopes averaged across all participants. The red vertical lines on the left sides of the plots show the cut point of 250 ms. Note that the segments are 1 s long, which is different than 4 s segments that we initially used. Doing so removed most of the decay that contributes to mean SC.

exerting effort for just a short period. During continuous exertion, we see that bioelectric muscle signals do not exhibit a smooth trend yielding a nearly iterative shape.

Furthermore, any additional ancillary motion, such as moving parts of the body to the beat, or a further modification motion, such as a vibrato to add expression to the sustaining tone, also can be considered as possible artifacts contributing to the envelope of muscular activation. When inspecting the individual participants' video recordings, we noticed that such spontaneous motions are fairly common. Figure 11 provides an example of this. We extracted the sparse optical flow by tracking certain points on a close-up video recording of a participant playing the impulsive task. The participant's ancillary motion is observable in the position of the guitar in relation to the camera and captured possibly by the EMG sensors on the left forearm.

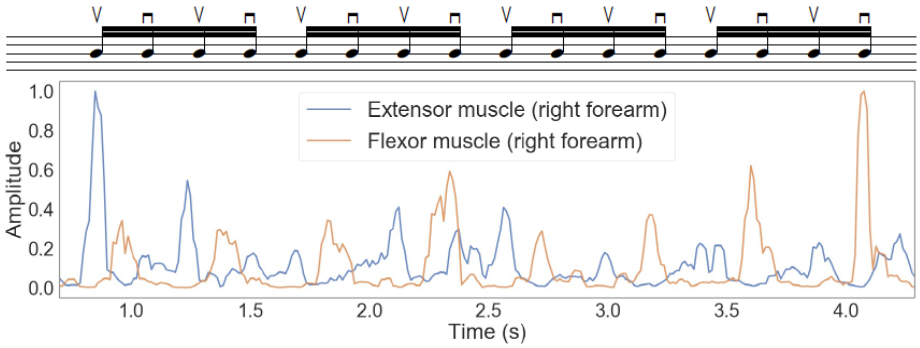


Figure 10. EMG amplitude of the excitation motion during iterative task demonstrating distinct activation of extensor and flexor muscles for down and up strokes, respectively, during a series of 16th notes.

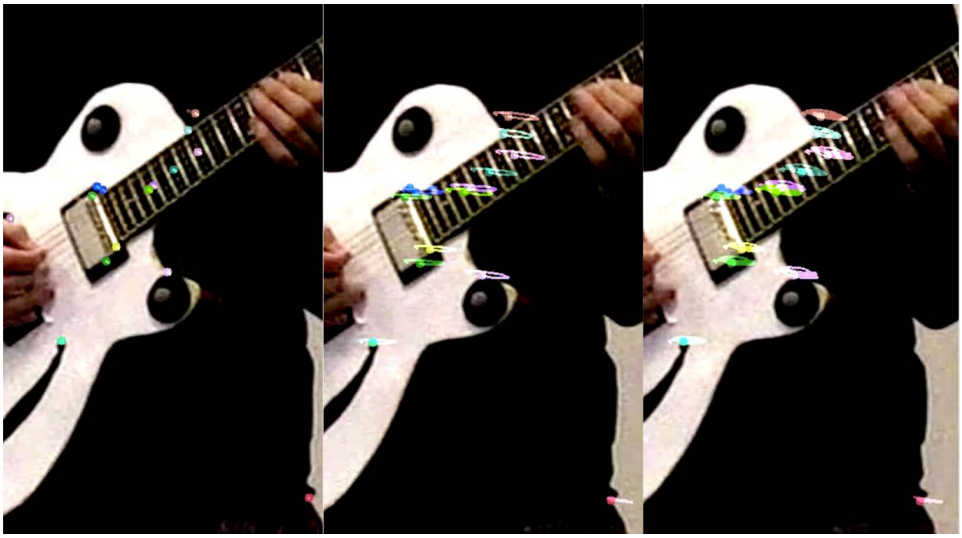


Figure 11. The sparse optical flow shows the trajectory of multiple points on a close-up video segment while a participant is performing an impulsive task. Three subsequent screenshots demonstrate the ancillary motion reflected on the guitar over the period of 1 bar (~3.43 s). The multicolored points on the left picture yield certain patterns in their trajectories reflecting participant movement patterns in the center and right pictures.

We suggest that such ancillary motion influences more directly the ongoing muscle activation as compared to right forearm muscles, which were resting at that moment.

When comparing left and right forearm muscle activation patterns, the negative directionality is noteworthy. This is particularly clear during the bending tasks (see Figures 5 and 6), a playing technique in which the right arm excitation is equivalent to the impulsive task. The left arm modifies the pitch and has a sustained envelope. This is unique to the guitar, as this instrument

does not afford sustained sound as do the bowed strings instruments. We should also mention that both the exerted effort and the resultant damping character of the sound would be different if other equipment were used, such as a harder wood and/or pickups with stronger magnets in instrument design, high-gain amplifiers, electronic effects units, or any other room acoustics resulting in greater feedback.

Another interesting observation when comparing data from the left and right forearms is the similarity between positive correlation values of the Impulsive and Legato. This could result from coarticulation. In this task, the left hand executes two consecutive (impulsive) attacks. These are quite different from the impulsive task, however. Because the two consecutive attacks are close temporally, they merge to form one large, coarticulated shape.

Finally, the iterative tasks showed the most idiosyncratic patterns and the least shape similarity. We observed that playing consecutive notes as a series of relatively fast attacks was the most challenging task for many of our participants. Depending on the level of expertise, each participant demonstrated signs of slogging to some extent, which arguably resulted in unique timing characteristics. Effort constraints may be a relevant topic here: Although some players are able to optimize their muscle contractions, others can exert more or less than optimal effort. In addition to the participants' level of expertise, the iterative task may have led to muscle fatigue. None of the participants mentioned this condition, but the possibility deserves further exploration in the context of musical performance.

Exploring Dimensions

The main objective of this investigation was to explore the quantifiable similarities of the amplitude envelopes of sound-producing actions on the electric guitar. In the first part of our analysis, we explored such relationships between two muscle groups against the resultant sound amplitude envelopes from each participant. In the second, we focused on a combination of results from all muscles on both forearms across all participants. We performed PCA on concatenated EMG channels, aiming to render additional observations and visual perspectives. In this part of the analysis, then, we aimed at exploring the signal PCs that can reflect a combination of simultaneous processes. Our interpretation of the PCA is that although PC1 reflected the overall dissipating aspect of the excitation motion, PC2 revealed the variation in the energy input of the modulation motion. This is the case even though we did not specify the decomposition to be separate.

From these observations, we can group all types of EMG patterns under two conceptual categories: (a) impulsive, where a single impulse or a series of impulses is applied, and (b) sustained, denoting a constant muscle energy. The experimental approach of decomposing the PCs using SSA (Figure 8) provided alternative perspectives for exploring the nonlinearities of the relationships. Whereas series of impulses yielded fewer regular patterns, sustaining energy showed clearer similarities. These findings are in line with the results presented in the previous subsection.

The Resultant Sound

Figure 9a demonstrates how SC was distributed across various tasks and dynamics. The main observation here was that stronger dynamics led to a brighter sound. We also should note that plucked strings have what may be called incidental nonlinearities that can have effects, depending on the intensity of excitation (Fletcher, 1999). Moreover, we used 1000 ms and 250 ms segments

in these two subplots, respectively. These durations were different from the approximately 4.29 s segments we relied on in our analysis. This shift was intended to remove the tail of the waveform during the decay, which affects the mean brightness value. So, our results support previous work suggesting that timescales shorter than 500 ms reflect most of the timbral features that happen during the attack phase of the excitation (Godøy, 2018).

Figure 9a shows how Iterative had a brighter character than the others when the averaged segments are a longer duration (1000 ms). However, Iterative's mean SC decreased when shorter segments (250 ms) were used for comparison. This indicated a timbral difference between the impulsive and iterative tasks. That is, the impulsive tasks tended to demonstrate a single peak in the exerted energy, reflecting in a brighter sound. The series of attacks of the latter, however, showed more fluctuating energy. This also revealed that during those series, the energy that was transduced into the attacks also made the SC change dynamically. As such, the plots of the averaged SC shaped over time (Figure 9b).

EXPERIMENT 2: A PRELIMINARY PREDICTIVE MODEL

Following the empirical exploration of how biomechanical energy transforms into sound, we used these transformations as part of a machine learning framework based on a long short-term memory recurrent neural network for action–sound mappings. We engaged an interdisciplinary approach that draws on a combination of sound theory and embodied music cognition. Our starting point involved an idea of developing a model that is trained solely on fundamental sound-producing action types. The aim this component of our research was to predict the sound amplitude envelopes of a freely improvised performance. We see this as a preliminary step toward designing an entirely new instrument concept.

Conceptual Design

Our motivating concept was to develop a model that allows for coadaptation, meaning the system not only learns from the user but the user adapts to the behavior of the system (Tanaka & Donnarumma, 2018). Knowing that EMG is a stochastic and nonstationary signal (Phinyomark, Campbell, & Scheme, 2019), even simple trigger actions are quite complex in nature. Although it may seem handy to use well-known machine learning methods, such as classification for triggering sounds or regression to map continuous motion signal (Caramiaux & Tanaka, 2013), we are interested in developing beyond a one-directional control. This vision is conceptually different from, for example, using machine learning for EMG-based control aimed at prosthetic research (Jaramillo-Yánez, Benalcázar, & Mena-Maldonado, 2020).

We also were intrigued with another design concept: predictive modeling. Following various control structures that we had explored in previous work (Erdem, Camci, & Forbes, 2017; Erdem & Jensenius, 2020; Erdem, Schia, & Jensenius, 2019), we were interested more with the ways of how the system can behave differently from interactive music systems that react primarily to the user (Rowe, 1992). Drawing on the work of Martin, Glette, Nygaard, & Torresen (2020), we began exploring the potential of artificial intelligence tools generally, and predictive models in particular, that facilitate not only the input–output mapping of complex signals in new instruments but also enable self-awareness.

Methods

Data Preparation

Our modeling process relied heavily on data from Myo armbands, as they are a cheaper and more portable solution than the Delsys Trigno system. As described in detail in the Methods section of Experiment 1, we synchronized the EMG data and audio arrays based on the recorded metronome timeline. The primary difference in our analysis procedure in this experiment was that we kept all data for modeling. That is, the data were not segmented nor did we eliminate the material collected in-between tasks, when the participants were waiting for the next instruction. This latter set of material made it possible to have the model learn to distinguish between rest and motion states.

We applied linear interpolation to the EMG data and calculated the RMS from the audio signal. The data preparation process resulted in eight segments per participant of EMG and audio data as training examples. The preliminary architecture focused on mapping the raw EMG data to the RMS envelope of the sound as the target.

Predictive Model

We used nine model configurations based on a long short-term memory (LSTM) recurrent neural network (RNN) architecture. Drawing on previous research that suggested 32 or 64 LSTM units in each layer as the most appropriate for integrating the model into an interactive music system (Martin & Torresen, 2019), we wanted to test different configurations. Thus, we used models with one, two, and five hidden layers and each containing 16, 32, and 64 units. Each model was trained on sequences that were 50 data points. This window size refers to 250 ms at Myo armband's 200 Hz sample rate.

Following the LSTM layer(s), a fully connected layer passes a single data point into the activation layer, using a rectified linear activation (ReLU) function. From there, a final layer returns the mean value of the input tensor in order to map an EMG window to one data point of the sound RMS, a many-to-one sequence modeling problem. In short, an array of raw EMG signal with a dimensionality of (50,16) was fed into the network as sliding windows (e.g., sample N_0 to N_{49}, sample N_1 to N_{50}, etc.) to predict a single value of sound RMS at a time step (see Figure 12 for a simplified diagram). The training loss function was defined as

$$\mathcal{L}(x_{\mathrm{RMS}}, \hat{x}_{\mathrm{RMS}}) \quad = \frac{1}{n}\sum_{i=1}^{n}(x_{\mathrm{RMS,i}} - \hat{x}_{\mathrm{RMS,i}})^2 ,$$

where x_{RMS} are the recorded values, $\hat{x}_{\mathrm{RMS}}$ are the values to be predicted, and the sliding window has size n.

Training

The dataset was limited to 160 training examples from 20 participants in which 40 examples were used for validation. We conducted the training using the Adam optimizer (Kingma & Ba, 2014) with a batch size of 100. As we executed multiple trainings to test various configurations, we limited the trainings to 20 epochs. The duration of trainings varied from 4 to 10 hours, depending

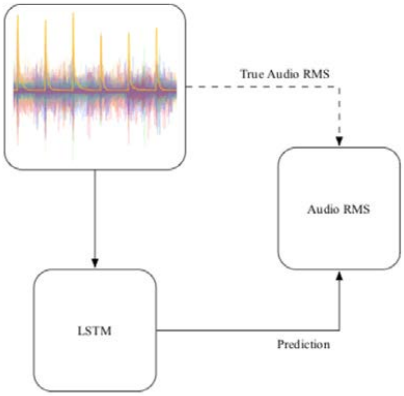


Figure 12. Sketch of the training model: A 16-channel Raw EMG as the source and sound RMS as the target data are passed into an LSTM cell, which then outputs a prediction.

on the quantity of trainable parameters in relation to the number of hidden layers and units. Even though we report here the final results from training locally on a single Nvidia GeForce GTX 1080Ti graphics processing unit (GPU), we also ran the trainings on Google's browser-based coding notebook, *Colaboratory;* we did not observe any remarkable difference in the training duration.

Results

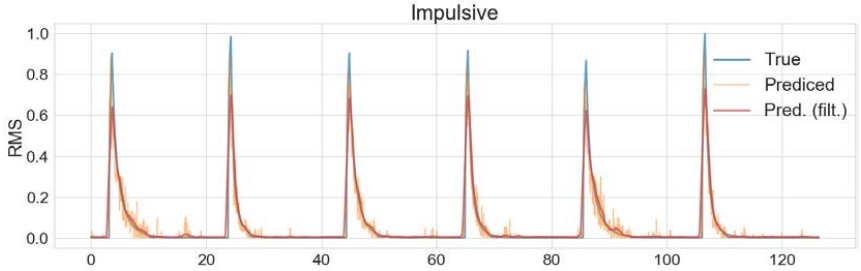
All model configurations were generally capable of predicting the sound RMS (see Figure 13). The model with two hidden layers and 64 units had the best results, which can be seen in the figures of recorded versus predicted RMS of the impulsive (Figure 13a) and iterative tasks (Figure 13b). For the latter, the model could generate similar consecutive envelopes resembling a series of attacks.

One goal in developing this preliminary model was to test the performance of the LSTM based on a limited dataset. In this case, the limitation refers to the type of dataset rather than its size. We were encouraged to see that the model could predict the general trend of the sound energy when tested using the free improvisation dataset (Figure 14).

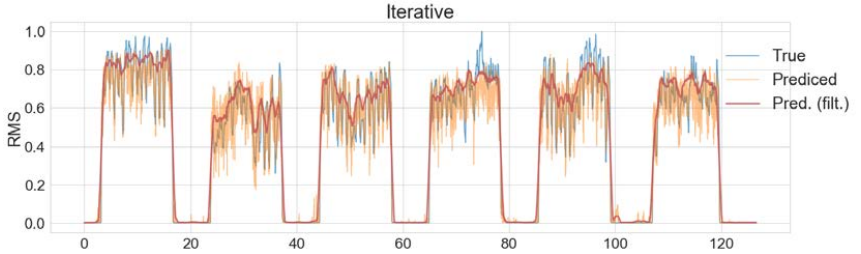
The prediction of the bending task brought an interesting result (Figure 13c). Normal guitar performance does not afford sustained excitation action, although it can be accomplished with a bow on the strings, as Led Zeppelin's guitarist, Jimmy Page, popularized in the late 1960s. However, apart from using extended playing techniques—such as pressing on the strings with the hands or using additional equipment, such as a bow, vibrato arm, or electronic effects processing units—a player can only hit on a string once (impulsive) or as a series of impulses (iterative). Thus, sustained motion is available only for the modification action, such as bending the string with a finger on the left hand.

In the prediction, however, we observed a longer decay as compared to an impulsive, single attack of the right arm. This interesting in-between result suggests a means for augmenting the guitar for creative purposes.

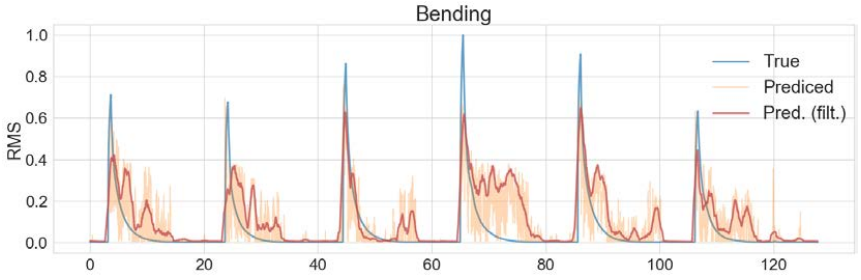
We also tested various model sizes using Euclidean distance measure (EDM), which is a common method for measuring the distance between objects. EDM is calculated as the root of square



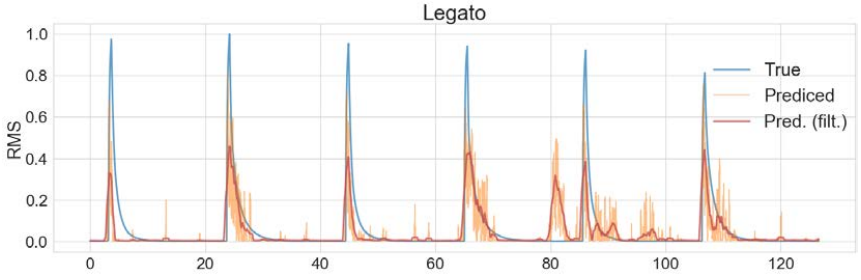
(a) The RMS of the recorded sound and the model prediction for the impulsive task.



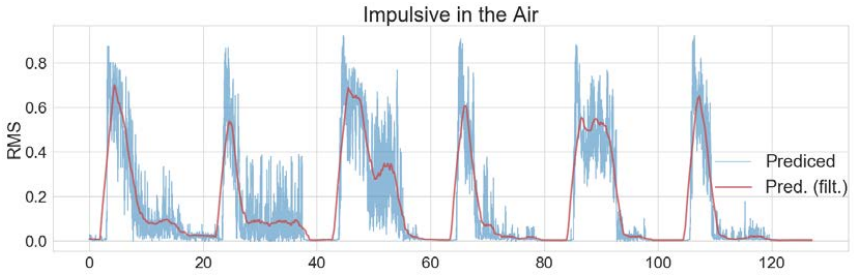
(b) The RMS of the recorded sound and the model prediction for the iterative task.



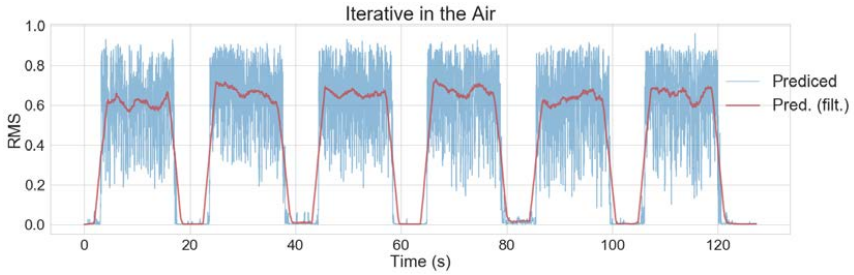
(c) RMS of the recorded sound and the model prediction for the bending task.



(d) RMS of the recorded sound and the model prediction for the legato task.



(e) The predicted sound RMS of impulsive playing in the air.



(f) The predicted sound RMS of iterative playing in the air.

Figure 13. The performance of the model with two hidden layers and 64 units in given tasks. Plots a through d show the true sound RMS and predicted RMS envelopes. Because we recorded impulsive and iterative tasks performed in the air as test data for further exploration, plots e and f show only the predicted sound RMS envelope based on the EMG data of an air performance. The time axis is shared across all plots and predicted curves are processed with a Savitzky-Golay filter (Savitzky & Golay, 1964) to reflect the general shape and facilitate the visual inspection.

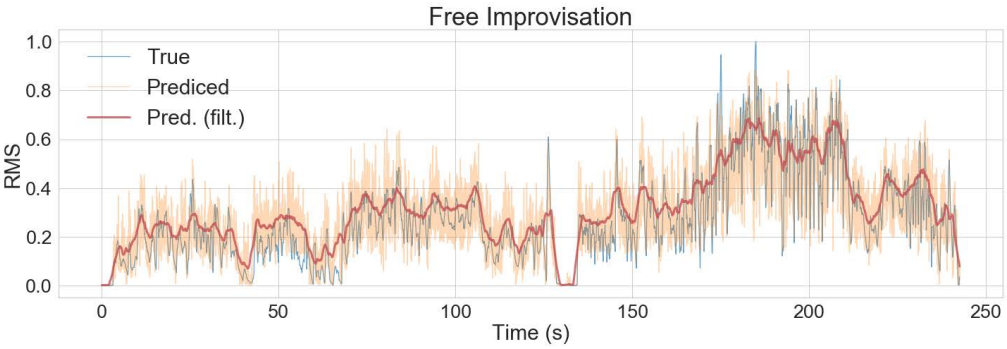


Figure 14. The RMS of the recorded sound and the model prediction of a free improvisation task. Predicted curves are filtered to reflect the general shape and facilitate the visual inspection.

differences between coordinates of two objects (Kang, Cheng, Lai, Shiu, & Kuo, 1996). Given the normalized true and predicted sound RMS vectors $\vec{p}, \vec{s} \in \mathbb{R}^n$, we can find the distances in Euclidean n-space as $\sqrt{(p_1 - s_1)^2 + (p_2 - s_2)^2 \ldots (p_n - s_n)^2}$. The distances between the true RMS and predicted RMS envelopes of the nine models of different configurations were calculated using the free improvisation recordings from 20 participants, of which given tasks were used as training data. This provided us with a statistical measure for evaluating the performance of different model configurations for mapping 16-channel raw EMG data to sound RMS envelope. Figure 15 provides the distribution of distances together with the latency of single-threaded prediction processes on the central processing unit (CPU) of a MacBook Pro 2018. According to results, we observed a trend that the model performance increases along with additional LSTM layers and units; unfortunately, however, the model's performance decreases when the model becomes too large. The prediction time also increases drastically with additional parameters. However, models with a single hidden layer have the least latency even while having a fairly large margin of error. Thus, according to the results, a two-layer stacked LSTM with 32 or 64 units can be seen as a "sweet spot" configuration.

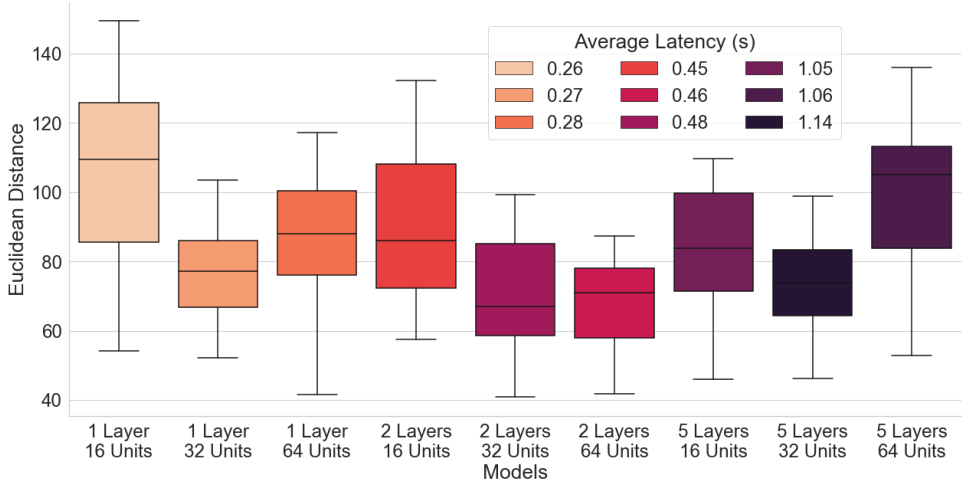


Figure 15. Euclidean distances between true RMS envelope of the free improvisation task and its corresponding prediction of RMS envelope based on nine model configurations. The boxes display the interquartile ranges while the central lines show the median. The whiskers show the minimum and maximum values of the distribution.

Discussion

The implemented model can predict the overall trend of the sound energy of a freely improvised performance based solely on a training dataset of particular action types. As shown in Figure 13, some similarities are evident between the EMG signal and the sawtooth-like patterns of the predicted waveforms. We think this is acceptable, as these fluctuating patterns can be filtered easily and used as an amplitude parameter in the sound synthesis. However, considering that

the prediction of a single temporal feature is insufficient for capturing the complexity of musical sound, these patterns might cause problems. These predictions also may lead to unpredictable sound features that could be aesthetically pleasing in an improved model.

Drawing on the results from the tests between different model configurations, we see that, as the model size increases, the distance between the true RMS and predicted RMS generally decreases, but the similarity tends to increase. However, larger model sizes also result in a larger latency, which can cause problems in real-time performance situations. We believe that although a lower similarity can be utilized creatively, higher similarity with a larger latency is much less usable.

Another step in the future development of the system will be to conduct a thorough user study to test the framework. It will be particularly interesting to explore how possible it is to obtain near-optimal latency using the trained model and, moreover, how to use the latency creatively. Also relevant is the exploration of how motion data from an inertial measurement unit can add to the information provided by the EMG data. At its core, the question remains how the spatiotemporality of the performance can be further explored and evaluated.

GENERAL DISCUSSION AND CONCLUSIONS

The main research question that inspired the first experiment of the study regarded the relationships between action and sound in instrumental performance. To answer that, we performed statistical analyses on the data from an experiment in which 31 electric guitarists performed a set of basic sound-producing actions: impulsive, sustained, and iterative. The results showed clear action–sound correspondences, compatible with theories of embodied music cognition. These correspondences' statistical levels varied, depending on the given task. The relatively less-challenging tasks, such as impulsive, yielded higher correlation values. Conversely, we observed how participants' varying level of motor control resulted in unique EMG and audio wave-forms for the iterative tasks, which involved performing a series of impulsive sound-producing actions merged into a single shape. Here, the way participants used rhythms and structured the musical time had a determinant role in the coarticulated muscle activations. Thus, we can argue that complex rhythms yield unique bodily patterns.

An important limitation of Experiment 1 was the gender imbalance. Unfortunately, only one female joined the study. The participants were recruited via local communication channels; thus the range of participants was limited to whoever volunteered. Another limitation was the experimental setup in a controlled laboratory environment, which may have felt unnatural to many participants. The same could be said about the very constrained tasks, which restricted the participants' musical expression. For example, the use of physical effort is most likely quite different than in a live music-making situation. Also, we provided the participants with the instrument, which may have influenced the results. Musicians typically develop bodily habits based on particular instruments—including the string gauge and plectrum. Thus, unfamiliarity with the electric guitar used in this study could have affected the relationships between EMG and audio signals. Furthermore, the analyses clearly showed that these relationships contain nonlinear components, so we could question the reliability of using linear methods. Still, we believe that the use of such methods can provide an example for future work. The results were satisfactory

for such an exploratory study, but the choice of statistical methods for correlating bodily signals with sound features remains an open question.

The second research question involved how such relationships between action and sound can be used to create new instrumental paradigms. Relying on the notion of imitating existing interactions in new instruments, we aimed in our second experiment at modeling the action–sound relationships found in playing the guitar. We explored some aspects of this question through a series of analyses in the first experiment. However, we were more focused in Experiment 2, employing our multimodal dataset to train LSTM networks of different configurations. Our results showed that the preliminary models could predict audio energy features of free improvisations on the guitar, relying on an EMG dataset of three distinct motion types. These results satisfied our expectations concerning the size and type of the training dataset. Considering the nonlinear components found in the analysis of the relationships between the EMG and sound RMS envelopes (see Figure 8), the satisfactory outcome of our model corresponded to the known ability of neural networks that, in theory, any continuous function can be approximated by computing the gradient through a neural network. This is achieved by breaking down a complex function into several step-functions computed by the network's hidden neurons. How good the approximation is often depends on the depth or number of layers in the network and the width or number of neurons of each layer (Goodfellow et al., 2016).

A caveat of our research in our second experimental setup is that even the smallest model configuration achieved a much higher latency (see Figure 15 for the results of our analysis on different model configurations) than acceptable ranges (20–30 ms) for real-time audio applications (Lago & Kon, 2004). Although it is possible to reduce the latency using elaborated programming structures, a single predicted feature would still be limited. Moreover, a similar output can be achieved using traditional signal processing methods. Thus, a next step in our research will include expanding the model with spectral, temporal, and spatial features from both motion and audio data. It would also be relevant to explore the potential of what such a deep learning-based framework can afford for musical performance and creativity in a new instrumental concept.

In the future, we will continue to build on this two-fold strategy of combining empirical data collection and machine learning-based modeling. We intend to explore deep learning features for myoelectric control that can be applied to extracting discriminative representations of coarticulated sound-producing actions. We remain interested especially in exploring the creative potential of such models: How can artificial intelligence generally—and deep neural networks particularly—be used to explore the aesthetics of, and embodied interaction with, the transformations of biomechanical waveforms into sound? To answer such a question, we will emphasize exploring the conceptual and practical challenges of space and time, particularly when using the human body as part of the musical instrument. By conducting more user studies, we expect to provide valuable information about conceptual approaches of translating embodied knowledge of actions into the use of new musical instruments.

IMPLICATIONS FOR RESEARCH

The studies presented in this paper are situated within the interdisciplinary research field of music technology (see Serra, 2005). This field involves both practitioners and researchers working with both artistic and scientific methods. Both groups will benefit from the knowledge gained from our

empirical studies of basic sound-producing actions and the artificial intelligence methods developed for modeling relationships between muscle energy and audio energy. More broadly, the outcomes of applying multimodal machine learning for creative purposes opens new research activities. These contributions include a new multimodal dataset, the development of custom software tools, statistical analyses between action and sound, and an evaluation of various machine learning configurations. Furthermore, the study provides additional support for previous research on action–sound relationships and embodied music cognition. Our emphasis on EMG irregularities as a control signal suggests an alternative perspective for music technology research on performing arts and human-computer interaction. These irregularities and imperfections open for new creative possibilities.

REFERENCES

Alexandrov, T. (2009). *A method of trend extraction using singular spectrum analysis.* Retrieved from https://arxiv.org/abs/0804.3367

Alexandrov, T., Bianconcini, S., Dagum, E. B., Maass, P., & McElroy, T. S. (2012). A review of some modern approaches to the problem of trend extraction. *Econometric Reviews, 31*(6), 593–624. https://doi.org/10.1080/07474938.2011.608032

Beranek, L. L., & Mellow, T. J. (2012). Chapter 1: Introduction and terminology. In L. L. Beranek & T. J. Mellow (Eds.), *Acoustics: Sound fields and transducers* (p. 1–19). Cambridge, MA, USA: Academic Press. https://doi.org/10.1016/B978-0-12-391421-7.00001-4

Briot, J.-P., Hadjeres, G., & Pachet, F.-D. (2020). *Deep learning techniques for music generation*. Cham, Switzerland: Springer. https://doi.org/10.1007/978-3-319-70163-9

Burden, A. M., Lewis, S. E., & Willcox, E. (2014). The effect of manipulating root mean square window length and overlap on reliability, inter-individual variability, statistical significance and clinical relevance of electromyograms. *Manual Therapy, 19*(6), 595–601. https://doi.org/10.1016/j.math.2014.06.003

Cadoz, C., & Wanderley, M. M. (2000). Gesture-music. In M. M. Wanderly & M. Battier (Eds.), *Trends in gestural control of music* (Vol. 12, pp. 71–94). Paris, France: IRCAM. Retrieved from https://hal.archives-ouvertes.fr/hal-01105543

Caramiaux, B., Bevilacqua, F., Zamborlin, B., & Schnell, N. (2009). Mimicking sound with gesture as interaction paradigm (Technical Report). Paris, France: IRCAM. Retrieved from http://articles.ircam.fr/textes/Caramiaux10d/index.pdf

Caramiaux, B., & Donnarumma, M. (2020). *Artificial intelligence in music and performance: A subjective art-research inquiry*. Retrieved from https://arxiv.org/abs/2007.15843

Caramiaux, B., Montecchio, N., Tanaka, A., & Bevilacqua, F. (2015). Adaptive gesture recognition with variation estimation for interactive systems. *ACM Transactions on Interactive Intelligent Systems, 4*(4), 18–52. https://doi.org/10.1145/2643204

Caramiaux, B., & Tanaka, A. (2013). Machine learning of musical gestures: Principles and review. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 513–518). Daejeon, Republic of Korea: Zenodo. https://doi.org/10.5281/zenodo.1178490

De Luca, C. J., Gilmore, L. D., Kuznetsov, M., & Roy, S. H. (2010). Filtering the surface EMG signal: Movement artifact and baseline noise contamination. *Journal of Biomechanics, 43*(8), 1573–1579. https://doi.org/10.1016/j.jbiomech.2010.01.027

Donnarumma, M. (2015). Ominous: Playfulness and emergence in a performance for biophysical music. *Body, Space & Technology, 14*, unpaginated. http://doi.org/10.16995/bst.30

Dreyfus, H. L. (2001). Phenomenological description versus rational reconstruction. *Revue Internationale de Philosophie, 216*(2), 181–196. https://doi.org/10.3917/rip.216.0181

Erdem, C., Camci, A., & Forbes, A. (2017). Biostomp: A biocontrol system for embodied performance using mechanomyography. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 65–70). Copenhagen, Denmark: Zenodo. http://doi.org/10.5281/zenodo.1176175

Erdem, C., & Jensenius, A. R. (2020). RAW: Exploring control structures for muscle-based interaction in collective improvisation. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 477–482). Birmingham, UK: Birmingham City University.

Erdem, C., Schia, K. H., & Jensenius, A. R. (2019). Vrengt: A shared body–machine instrument for music–dance performance. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 186–191). Porto Alegre, Brazil: Zenodo. http://doi.org/10.5281/zenodo.3672918

Fiebrink, R. A. (2011). *Real-time human interaction with supervised learning algorithms for music composition and performance* (Doctoral dissertation, Princeton University). Retrieved from https://www.cs.princeton.edu/research/techreps/TR-891-10

Fiebrink, R. A., & Caramiaux, B. (2016). *The machine learning algorithm as creative musical tool*. Retrieved from https://arxiv.org/abs/1611.00379

Fletcher, N. H. (1999). The nonlinear physics of musical instruments. *Reports on Progress in Physics, 62*(5), 723–764. http://doi.org/10.1088/0034-4885/62/5/202

Françoise, J. (2015). *Motion-sound mapping by demonstration* (Doctoral dissertation, Université Pierre et Marrie Curie). Retrieved from https://www.julesfrancoise.com/documents/JulesFRANCOISE_phdthesis.pdf

Fuentes del Toro, S., Wei, Y., Olmeda, E., Ren, L., Guowu, W., & Díaz, V. (2019). Validation of a low-cost electromyography (EMG) system via a commercial and accurate EMG device: Pilot study. *Sensors, 19*(23), 1–16. https://doi.org/10.3390/s19235214

Fyans, A. C., & Gurevich, M. (2011). Perceptions of skill in performances with acoustic and electronic instruments. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 495–498). Oslo, Norway: Zenodo. http://doi.org/10.5281/zenodo.1178019

Godøy, R. I. (2006). Gestural-sonorous objects: Embodied extensions of Schaeffer's conceptual apparatus. *Organised Sound, 11*(2), 149–157. https://doi.org/10.1017/S1355771806001439

Godøy, R. I. (2017). Key-postures, trajectories and sonic shapes. In D. Leech-Wilkinson & H. M. Prior (Eds.), *Music and Shape* (pp. 4–29). New York, NY, USA: Oxford University Press. https://doi.org/10.1093/oso/9780199351411.003.0002

Godøy, R. I. (2018). Sonic object cognition. In R. Bader (Eds.), *Springer handbook of systematic musicology* (pp. 761–777). Berlin, Germany: Springer. https://doi.org/10.1007/978-3-662-55004-5

Godøy, R. I., Haga, E., & Jensenius, A. R. (2005). Playing "air instruments": Mimicry of sound-producing gestures by novices and experts. In S. Gibet, N. Courty, & J. F. Kamp (Eds.), *International gesture workshop: Gesture in human–computer interaction* (pp. 256–267). Berlin, Germany: Springer. http://dx.doi.org/10.1007/11678816_29

Golyandina, N., & Zhigljavsky, A. (2013). *Singular spectrum analysis for time series*. Berlin, Germany: Springer. http://dx.doi.org/10.1007/978-3-642-34913-3

González Sánchez, V. E., Dahl, S., Hatfield, J. L., & Godøy, R. I. (2019). Characterizing movement fluency in musical performance: Toward a generic measure for technology enhanced learning. *Frontiers in Psychology, 10*, 1–11. https://dx.doi.org/10.3389/fpsyg.2019.00084

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge, MA, USA: The MIT Press. Retrieved from https://www.deeplearningbook.org/

Google. (2020). *Teachable machine*. https://teachablemachine.withgoogle.com/

Gritten, A., & King, E. (Eds.). (2011). *New perspectives on music and gesture*. London, UK: Routledge. https://doi.org/10.4324/9781315598048

Hatten, R. S. (2006). A theory of musical gesture and its application to Beethoven and Schubert. In A. Gritten & E. King (Eds.), *Music and gesture* (pp. 1–23). London, UK: Routledge. https://doi.org/10.4324/9781315091006

Hunt, A., & Wanderley, M. M. (2002). Mapping performer parameters to synthesis engines. *Organised Sound*, *7*(2), 97–108. https://doi.org/10.1017/S1355771802002030

Ingold, T. (2000). *The perception of the environment*. London, UK: Routledge.¨ https://doi.org/10.4324/9780203466025

Jaramillo-Yánez, A., Benalcázar, M. E., & Mena-Maldonado, E. (2020). Real-time hand gesture recognition using surface electromyography and machine learning: A systematic literature review. *Sensors*, *20*(9), Art. 2467. https://doi.org/10.3390/s20092467

Jensenius, A. R. (2007). *ACTION–sound: Developing methods and tools to study music-related body movement* (Doctoral dissertation, University of Oslo). Retrieved from http://urn.nb.no/URN:NBN:no-18922

Jensenius, A. R. (2017). Sonic microinteraction in "the air." In M. Lesaffre, P.-J. Maes, & M. Leman (Eds.), *The Routledge companion to embodied music interaction* (pp. 431–439). New York, NY, USA: Routledge. https://doi.org/10.4324/9781315621364

Jensenius, A. R. (2018a). Methods for studying music-related body motion. In R. Bader (Ed.), *Springer handbook of systematic musicology* (pp. 805–818). Berlin, Germany: Springer. https://doi.org/10.1007/978-3-662-55004-5

Jensenius, A. R. (2018b). *The musical gestures toolbox for matlab*. In the *Late-Breaking/Demo Session Abstracts for the 2018 International Society for Music Information Retrieval Conference*. Retrieved from http://www.arj.no/wp-content/2018/09/Jensenius-ISMIR2018.pdf

Jensenius, A. R., & Lyons, M. J. (2017). *A nime reader: Fifteen years of new interfaces for musical expression*. Cham, Switzerland: Springer. https://doi.org/10.1007/978-3-319-47214-0

Jensenius, A. R., Wanderley, M. M., Godøy, R. I., & Leman, M. (2010). Musical gestures: Concepts and methods in research. In R. I. Godøy & M. Leman (Eds.), *Musical gestures: Sound, movement, and meaning* (pp. 12–35). New York, NY, USA: Routledge. https://doi.org/10.4324/9780203863411

Kamen, G. (2013). Electromyographic kinesiology. In D. G. E. Robertson, G. E. Caldwell, J. Hamill, G. Kamen, & S. N. Whittlesey (Eds.), *Research methods in biomechanics* (2nd ed., pp. 179–202). North Yorkshire, UK: Human Kinetics, Inc.

Kang, W.-J., Cheng, C.-K., Lai, J.-S., Shiu, J.-R., & Kuo, T.-S. (1996). A comparative analysis of various EMG pattern recognition methods. *Medical Engineering & Physics*, *18*(5), 390–395. https://doi.org/10.1016/1350-4533(95)00065-8

Karjalainen, M., Mäki-Patola, T., Kanerva, A., & Huovilainen, A. (2006). Virtual air guitar. *Journal of the Audio Engineering Society*, *54*, 964–980. Retrieved from http://users.spa.aalto.fi/mak/PUB/AES12860.pdf

Kelkar, T. (2019). *Computational analysis of melodic contour and body movement* (Doctoral dissertation, University of Oslo). Retrieved from http://urn.nb.no/URN:NBN:no-74166

Khan, M. A. R., & Poskitt, D. S. (2011, November). *Window length selection and signal-noise separation and reconstruction in singular spectrum analysis* (Working Paper 23/11). Retrieved from the Monash University Department of Econometrics and Business Statistics website: http://business.monash.edu/econometrics-and-business-statistics/research/publications/ebs/wp23-11.pdf

Kiefer, C. (2014). Musical instrument mapping design with echo state networks. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 293–298). London, UK: Zenodo. http://doi.org/10.5281/zenodo.1178829

Kingma, D. P., & Ba, J. (2014). *Adam: A method for stochastic optimization*. Retrieved from https://arxiv.org/pdf/1412.6980.pdf

Knapp, R. B., & Lusted, H. S. (1990). A bioelectric controller for computer music applications. *Computer Music Journal*, *14*(1), 42–47. https://doi.org/10.2307/3680115

Kozak M., Nymoen K., & Godøy R. I. (2012). Effects of spectral features of sound on gesture type and timing. In E. Efthimiou, G. Kouroupetroglou, & S. E. Fotinea (Eds.), *Gesture and sign language in human–computer interaction and embodied communication* (pp. 69–80). *Lecture Notes in Computer Science*, Vol, 7206. Berlin, Germany: Springer. o

Lago, N. P., & Kon, F. (2004). The quest for low latency. In *Proceedings of the International Computer Music Conference* (pp. 33–36). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.1143

Lee, M., Freed, A., & Wessel, D. (1991). Real-time neural network processing of gestural and acoustic signals. In *Proceedings of the International Computer Music Conference* (pp. 277–281). Retrieved from http://hdl.handle.net/2027/spo.bbp2372.1991.064

Leman, M. (2008). *Embodied music cognition and mediation technology*. Cambridge, MA, USA: The MIT Press. https://doi.org/10.7551/mitpress/7476.001.0001

Li, X., Zhou, P., & Aruin, A. S. (2007). Teager–kaiser energy operation of surface emg improves muscle activity onset detection. *Annals of Biomedical Engineering*, *35*(9), 1532–1538. https://doi.org/10.1007/s10439-007-9320-z

Maes, P.-J., Leman, M., Lesaffre, M., Demey, M., & Moelants, D. (2010). From expressive gesture to sound. *Journal on Multimodal User Interfaces*, *3*(1-2), 67–78. https://doi.org/10.1007/s12193-009-0027-3

Magnusson, T. (2019). *Sonic writing: Technologies of material, symbolic, and signal inscriptions*. London, UK: Bloomsbury Academic. https://doi.org/10.1080/14794713.2020.1765577

Martin, C. P., Glette, K., Nygaard, T. F., & Torresen, J. (2020). Understanding musical predictions with an embodied interface for musical machine learning. *Frontiers in Artificial Intelligence*, *3*, Art. 6. https://doi.org/10.3389/frai.2020.00006

Martin, C. P., Jensenius, A. R., & Torresen, J. (2018). Composing an ensemble standstill work for myo and bela. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 196–197). Blacksburg, VA, USA: Zenodo. http://doi.org/10.5281/zenodo.1302543

Martin, C. P., & Torresen, J. (2019). An interactive musical prediction system with mixture density recurrent neural networks. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 260–265). Porto Alegre, Brazil: Zenodo. http://doi.org/10.5281/zenodo.3672952

Mendoza Garay, J. I., & Thompson, M. (2017). Gestural agency in human-machine musical interaction. In M. Lesaffre, P.-J. Maes, & M. Leman (Eds.), *The Routledge companion to embodied music interaction* (pp. 431–439). New York, NY, USA: Routledge. https://doi.org/10.4324/9781315621364

Nojima, I., Watanabe, T., Saito, K., Tanabe, S., & Kanazawa, H. (2018). Modulation of EMG-EMG coherence in a choice stepping task. *Frontiers in Human Neuroscience*, *12*, Art. 50. https://doi.org/10.3389/fnhum.2018.00050

Nymoen, K., Caramiaux, B., Kozak, M., & Torresen, J. (2011). Analyzing sound tracings: A multimodal approach to music information retrieval. In *Proceedings of the International ACM Workshop on Music Information Retrieval with User-centered and Multimodal Strategies*, (pp. 39–44). New York, NY, USA: ACM. https://doi.org/10.1145/2072529.2072541

Næss, T. R. (2019). *A physical intelligent instrument using recurrent neural networks* (Master's thesis, University of Oslo). Retrieved from http://urn.nb.no/URN:NBN:no-73901

Paine, G. (2009). Towards unified design guidelines for new interfaces for musical expression. *Organised Sound*, *14*(2), 142–155. https://doi.org/10.1017/S1355771809000259

Pakarinen, J., Puputti, T., & Välimäki, V. (2008). Virtual slide guitar. *Computer Music Journal*, *32*(3), 42–54. https://doi.org/10.1162/comj.2008.32.3.42

Pham, H. (2006). *Pyaudio: Portaudio v19 python bindings*. Retrieved from https://people.csail.mit.edu/hubert/pyaudio

Phinyomark, A., Campbell, E., & Scheme, E. (2019). Surface electromyography (EMG) signal processing, classification, and practical considerations. In G. Naik (Ed.), *Biomedical signal processing* (pp. 3–29). Singapore: Springer. https://doi.org/10.1007/978-981-13-9097-5_1

Pizzolato, S., Tagliapietra, L., Cognolato, M., Reggiani, M., Müller, H., & Atzori, M. (2017). Comparison of six electromyography acquisition setups on hand movement classification tasks. *PloS One*, *12*(10), e0186132. https://doi.org/10.1371/journal.pone.0186132

Purwins, H., Li, B., Virtanen, T., Schlüter, J., Chang, S.-Y., & Sainath, T. (2019). Deep learning for audio signal processing. *IEEE Journal of Selected Topics in Signal Processing*, *13*(2), 206–219. https://doi.org/10.1109/JSTSP.2019.2908700

Rowe, R. (1992). *Interactive music systems: Machine listening and composing*. Cambridge, MA, USA: The MIT Press. Retrieved from https://wp.nyu.edu/robert_rowe/text/interactive-music-systems-1993/

Santello, M., Flanders, M., & Soechting, J. F. (2002). Patterns of hand motion during grasping and the influence of sensory guidance. *Journal of Neuroscience*, *22*(4), 1426–1435. https://doi.org/10.1523/JNEUROSCI.22-04-01426.2002

Schacher, J. C., Miyama, C., & Bisig, D. (2015). Gestural electronic music using machine learning as generative device. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 347–350). Baton Rouge, Louisiana, USA: Zenodo. http://doi.org/10.5281/zenodo.1179172

Schaeffer, P. (2017). *Treatise on musical objects* (C. North & J. Dack, Trans.). Oakland, CA, USA: University of California Press. https://doi.org/10.1525/9780520967465-001 (Original work published in 1967)

Schober, P., Boer, C., & Schwarte, L. A. (2018). Correlation coefficients: Appropriate use and interpretation. *Anesthesia & Analgesia*, *126*(5), 1763–1768. https://doi.org/10.1213/ANE.0000000000002864

Schubert, E., Wolfe, J., & Tarnopolsky, A. (2004). Spectral centroid and timbre in complex, multiple instrumental textures. In *Proceedings of the International Conference on Music Perception and Cognition* (pp. 654–657). Retrieved from http://newt.phys.unsw.edu.au/~jw/reprints/SchWolTarICMPC8.pdf

Selesnick, I. W., & Burrus, C. S. (1998). Generalized digital Butterworth filter design. In *IEEE Transactions on Signal Processing*, *46*(6), 1688–1694. https://doi.org/10.1109/78.678493

Serra, X. (2005). Towards a roadmap for the research in music technology. In *Proceedings of the International Computer Music Conference.* Retrieved from https://repositori.upf.edu/handle/10230/34486?locale-attribute=en

Smalley, D. (1997). Spectromorphology: Explaining sound-shapes. *Organised Sound*, *2*(2), 107–126. https://doi.org/10.1017/S1355771897009059

St-Amant, Y., Rancourt, D., & Clancy, E. A. (1996). Effect of smoothing window length on rms emg amplitude estimates. In *Proceedings of the IEEE Annual Northeast Bioengineering Conference* (pp. 93–94). New Brunswick, NJ, USA: IEEE. https://doi.org/10.1109/NEBC.1996.503233

Tahiroğlu, K., Kastemaa, M., & Koli, O. (2020). Al-terity: Non-rigid musical instrument with artificial intelligence applied to real-time audio synthesis. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 331–336). Birmingham, UK: Birmingham City University.

Tanaka, A. (1993). Musical technical issues in using interactive instrument technology with application. In *Proceedings of the International Computer Music Conference* (pp. 124–126). Tokyo, Japan: ICMC. Retrieved from http://hdl.handle.net/2027/spo.bbp2372.1993.023

Tanaka, A. (2015a). Intention, effort, and restraint: The EMG in musical performance. *Leonardo Music Journal*, *48*(3), 298–299. https://doi.org/10.1162/LEON_a_01018

Tanaka, A. (2015b). *Myogram* [Music composition and performance]. Retrieved from https://youtu.be/G6H1J2k--5I

Tanaka, A. (2019). Embodied musical interaction. In S. Holland, T. Mudd, K. Wilkie-McKenna, A. McPherson, & M. Wanderley (Eds.), *New directions in music and human−computer interaction* (pp. 135–154). Cham, Switzerland: Springer. https://doi.org/10.1007/978-3-319-92069-6

Tanaka, A., Donato, B. D., Zbyszynski, M., & Roks, G. (2019). Designing gestures for continuous sonic interaction. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 180–185). Porto Alegre, Brazil: Zenodo. http://doi.org/10.5281/zenodo.3672916

Tanaka, A., & Donnarumma, M. (2018). The body as musical instrument. In Y. Kim & S. L. Gilman (Eds.), *The Oxford handbook of music and the body*. Oxford, UK: Oxford University Press. https://doi.org/10.1093/oxfordhb/9780190636234.013.2

Valles, M. L., Martínez, I. C., Ordás, M. A., & Pissinis, J. F. (2018). Correspondence between the body modality of music students during the listening to a melodic fragment and its subsequent sung interpretation [Abstract]. In *Proceedings of th 15th International Conference on Music Perception and Cognition & the*

10th Triennial Conference of the European Society for the Cognitive Sciences of Music (p. 308). Retrieved from http://sedici.unlp.edu.ar/handle/10915/70462

Van Nort, D., Wanderley, M. M., & Depalle, P. (2014). Mapping control structures for sound synthesis: Functional and topological perspectives. *Computer Music Journal*, *38*(3), 6–22. https://doi.org/10.1162/COMJ_a_00253

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J. Polat, I., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Visi, F., Coorevits, E., Schramm, R., & Miranda, E. R. (2017). Musical instruments, body movement, space, and motion data: Music as an emergent multimodal choreography. *Human Technology*, *13*(1), 58–81. https://doi.org/10.17011/ht/urn.201705272518

Waisvisz, M. (1985). The hands, a set of remote midi-controllers. In *Proceedings of the International Computer Music Conference* (pp. 313–319). Burnaby, BC, Canada: ICMC. Retrieved from http://hdl.handle.net/2027/spo.bbp2372.1985.049

Ward, M. R. (1971). *Electrical engineering science*. New York, NY, USA: McGraw-Hill.

Winges, S. A., Furuya, S., Faber, N. J., & Flanders, M. (2013). Patterns of muscle activity for digital coarticulation. *Journal of Neurophysiology*, *110*(1), 230–242. https://doi.org/10.1152/jn.00973.2012

Authors' Note

The authors thank the participating musicians, as well as Victor Evaristo González Sánchez and Julian Führer, for their contributions during the data collection and modeling processes. This work was supported in part by the Research Council of Norway (Project 262762) and NordForsk (Project 86892).

All correspondence should be addressed to
Çağrı Erdem
RITMO Centre for Interdisciplinary Studies in Rhythm, Time and Motion
Department of Musicology
University of Oslo
Postboks 1133 Blindern 0318 Oslo, Norway
cagri.erdem@imv.uio.no

Human Technology
ISSN 1795-6889
www.humantechnology.jyu.fi

Paper V

Glicol: A Graph-oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet

Qichao Lan, Alexander Refsum Jensenius

Published in *Proceedings of the International Web Audio Conference (WAC)*, 2021

Glicol: A Graph-oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet

Qichao Lan
RITMO
Department of Musicology
University of Oslo
qichao.lan@imv.uio.no

Alexander Refsum Jensenius
RITMO
Department of Musicology
University of Oslo
a.r.jensenius@imv.uio.no

ABSTRACT

This paper introduces the new music live coding language Glicol (graph-oriented live coding language) and its web-based run-time environment. As the name suggests, this language is designed to represent directed acyclic graphs (DAG), using a syntax optimised for live music performances. The audio engine and the language interpreter are both developed with the Rust programming language. With the help of WebAssembly and AudioWorklet, this language can run in web browsers. It also supports co-performance with the support for collaborative editing. Taking advantages of the Rust programming language design, the run-time environment is both safe and efficient. Documentation and error handling messages can be accessed in the web browser. All in all, we see Glicol as an efficient and future-oriented language for collaborative text-based musicking.

1. INTRODUCTION

When used in music contexts, the term *live coding* refers to musical performances during which performers write computer programs in real-time to make music [5]. So far, dozens of live coding languages have been developed[1]. Among all these languages, SuperCollider [14] and its relevant clients have been ubiquitous in the live coding community. One reason for its popularity is the *client-server architecture*, which means that the audio engine works as a server and communicates with the language side via Open Sound Control messages [7]. Users can replace the language as long as the communication to the audio server follows the specified protocol. Such an architecture inspired several early live coding languages, such as TidalCycles [15], ixilang [12], and Sonic Pi [1].

The early live coding languages were based on OS-specific applications. In recent years, browser-based environments have become increasingly popular, of which Gibber.js [17] may be the most well-known example. Such systems can more easily be used on many different platforms, but have so far been less powerful than its OS-specific siblings. Our previous live coding environment, QuaverSeries [9], is also browser-based. Its audio engine relies on Tone.js [13], a JavaScript library built on top of the Web Audio API. Based on our performance experience and user feedback, we see that the syntax of QuaverSeries is relatively intuitive to use. We have also found its browser-based collaborative environment beneficial for online collaboration. Still, there are three main limitations of QuaverSeries:

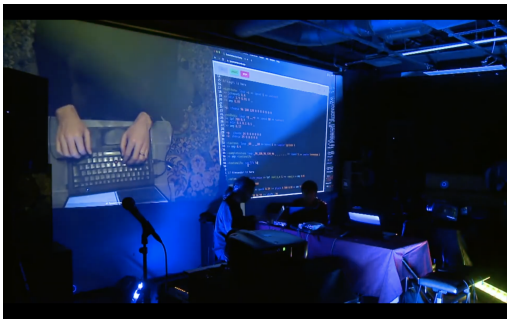


Figure 1: The authors engaged in a live coding performance during the Web Audio Conference 2019 in Trondheim. Here Glicol's precursor QuaverSeries was used. Glicol inherits the syntax style and the collaborative environment of QuaverSeries, while most of the back-end has been redeveloped.

Error handling and tracing JavaScript uses a try–catch approach for error handling. However, once an error occurs, the audio application can be broken and cause the music to stop if the error, such as a memory error, is not handled correctly.

Music information retrieval The ability to analyse the performed music is an intriguing topic in live coding [20]. This has many potential applications, for example, collaborating with virtual agents [21]. In QuaverSeries, we have seen that it is challenging to build a non-real-time function to take the code as input and immediately output the audio float array.

Portability As QuaverSeries is developed with Ohm.js and Tone.js, it is hard to utilise its full potential on different types of hardware, such as Bela[16].

[1]See, for example, the TOPLAP overview here: https://github.com/toplap/awesome-livecoding

Web Audio Conference WAC-2021, July 5–7, 2021, Barcelona, Spain.

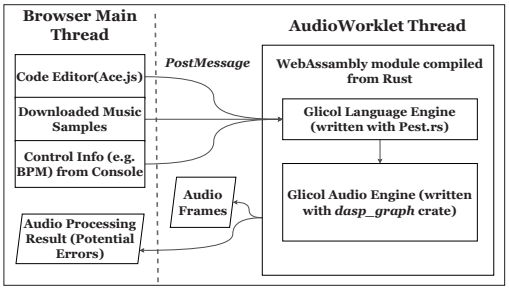


Figure 2: An overview of the Glicol architecture. Both the language and the audio engine are written in Rust and compiled to a WebAssembly module that runs in an AudioWorklet thread.

These reasons have motivated the development of Glicol,[2] the successor of QuaverSeries. It has a customised audio engine that can work seamlessly with the language parser. To do so, we decided to write the parser and the audio engine in the Rust programming language. Previously C++ has dominated audio engine development, but Rust is becoming a good alternative for its safety and efficiency. Also, Rust has a particular error handling method that is very suitable for audio applications. Last but not least, either using C++ or Rust, we can export the audio engine as a WebAssembly [8] module and use it in the AudioWorklet [4] thread in the browser (see Figure 2). This combination of WebAssembly and AudioWorklet has also been used in Csound [22], Faust [11], and Sema [3].

This paper will describe the design (Section 2) and implementation (Section 3, 4, 5) of Glicol, followed by a general discussion (Section 6) and conclusion.

2. DESIGN CONSIDERATIONS

As mentioned in the introduction, Glicol aims to keep the syntax simplicity of QuaverSeries. However, the development of the new audio engine necessitated some changes in the language.

2.1 Paradigm: graph-oriented

Rust is a multi-paradigm programming language, which means that it supports several popular programming paradigms, including functional programming (FP) or object-oriented programming (OOP), although Rust does not support the concept of *inheritance* in OOP. Instead, it takes a concept called *trait* method functions, which means that different structures (*struct*) can share similar *traits*.

In QuaverSeries, we choose the functional programming paradigm, in order to leverage the simplicity and the flexibility of function composition. Though this goal is satisfied by wrapping Tone.js *objects* in different functions, there are some potential issues. First, the users cannot know the exact data structure of the function input and output intuitively. Second, in each function, mutating variables/state/data outside of its scope, e.g. those related to Tone.js, may bring some instabilities such as data conflict. This is also called *impure functions* in functional programming [19].

[2]https://github.com/chaosprint/glicol

```
1 ~lead: play >> sin_osc 440.0 >> amp ~mod
2
3 ~mod: sin_lfo 1.5 0.1 0.5
```

Figure 3: QuaverSeries amplitude modulation syntax. We wrapped Tone.js *instances* in each function. There are some hacks here: all the functions before the amp are simply used to organise required information, and finally, in the amp function, we call the .play() method of a Tone.js object to make sound.

Rust solves this issue by requiring programmers to ensuring the *ownership* of each data, i.e. which variable each data belongs to, but this makes it hard to capture some global states when building the audio engine in a functional programming paradigm.

Instead, we find that the graph-oriented paradigm may be more suitable for developing a new music language in Rust. The term *graph* means the abstract collection of a series of *nodes* connected by *edges* [18]. In audio programming, the concepts of *graph* and *nodes* are ubiquitous, such as in the Web Audio API and SuperCollider. In Rust Audio community, the concept of graph has also been widely adopted ,e.g. the `FunDSP` project [3] and the `dasp_graph` [4] library. They both take advantages of the *trait* feature in Rust by offering a template for implementing the `Node` *trait* for different structures.

2.2 Syntax and data structure

```
1 lead: sin 440.0 >> mul ~mod
2
3 ~mod: sin 1.5 >> mul 0.2 >> add 0.3
```

Figure 4: Glicol syntax for amplitude modulation.

Though the concept of graphs and nodes is widely used in audio programming and development, few languages adopt a graph-oriented programming paradigm in the syntax design. In Glicol syntax design, the goal is to use a minimal but reader-friendly grammar to represent an audio graph. A *node* is represented by using the specific keywords such as the `sin`, `mul` and `add` in the example shown in Figure 4, following by its required parameters. A *chain* can be created by connecting nodes in series with the double greater-than sign (>>), and a *reference* can be used to denote this chain of signal flow. In the example, both `lead` and `~mod` are *references*. Using the *reference* as the parameter of a node means that this parameter is controlled by another chain of signal, which is also called *side-chain* in signal processing. Note that only the *reference* that comes without a tilde (~) will be sent to the audio interface. This is the syntax for separating control signals and audio signals, although they both run at audio rate.

3. IMPLEMENTATION IN RUST

To implement the syntax in Rust, we need to build different node structures, together with a language engine that can convert the code text to an audio graph. Also, we need

[3]https://github.com/SamiPerttu/fundsp
[4]https://docs.rs/dasp_graph/0.11.0/dasp_graph/

```
pattern: speed 2.0 >> seq 60 _50 ~a _~a
>> sp \bd

~a: choose 60 50 0 0 0
```

Figure 5: Sample playback in Glicol. The notes in the *seq* node can be controlled by a *choose* node, whose parameter number determines the probability of random selection.

to consider how we can dynamically manage these nodes in a graph during a live coding session.

3.1 Implementing the node trait

As is mentioned in Section 2.1, we use the `dasp_graph` library to develop different node structures. Concretely, we implement a *trait* called `Node` for all these node structures such as `SinOsc`, etc. With this *trait*, these node structures are all embedded with a method called `process`. This method takes an input *buffer* array and output a *buffer* array, and within its definition, we write the DSP code to determine how the output *buffers* should be calculated from the input *buffers*.

So far, we have developed 30 node structures, including oscillators, filters, math operators, etc. From a signal processing perspective, connecting nodes can be relatively straight-forward for sound synthesis and audio effects chaining. But when it comes to the `sp` (sample playback) node, there are some exceptions to consider. In Glicol, once the `sp` node receives a non-zero value from its input—which should be placed at the first position of the incoming block signal array—it will schedule a sample playback inside the node. The playback rate is determined by the trigger's value, which will consequently alter the playing pitch of the audio sample. For instance, the value 1.0 triggers the default playback rate of the audio sample. A trigger value of 2.0 will play the sample one octave higher (see Figure 5).

Thus, many nodes can be used to trigger audio sample playback. For instance, the `imp` node, i.e. 'impulsive node,' and can send out impulse signal that triggers a sample playback periodically. The node `seq` takes a sequence of MIDI note values or underscores as parameters. Notes are represented by integers while underscores denote rests (silence). The parser will divide one bar into equal length based on the spaces. The default bar duration is 2 seconds, equivalent to 120 beats per minute with a time signature of 4/4. Then, each segment can be further divided into smaller, equidistant sub-segments based on the number of MIDI notes and rests.

3.2 Converting code text to an audio graph

To convert code string to an audio graph, the first thing is to build a parser to process the code. In Glicol, we choose Pest.rs as the parsing tool [5]. It allows us to define the language rules in PEGs (parsing grammar expressions) paradigm [10]. Next, we can call its API to parse the code to node information such as the *reference* of a *chain* of nodes, name of a single node and its parameters.

To maintain the lazy evaluation manner, i.e writing first and defining later, we parse the code first, but keep the node information in a *Vector* structure (re-sizable arrays in Rust) chain by chain, and save these *Vectors* in a *HashMap*

[5]https://pest.rs/

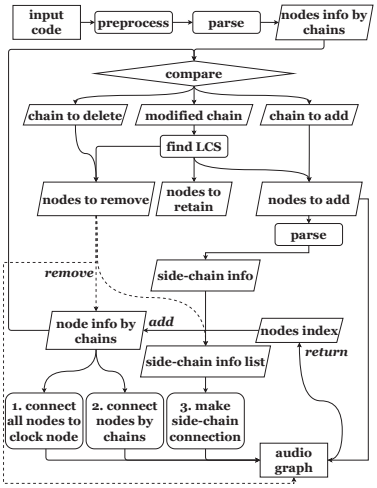


Figure 6: The process of converting code string into an audio graph. We come up with a dynamic node management strategy with the LCS algorithm.

structure (a dictionary-style data structure in Rust). Also, when parsing each node, the side-chain information, i.e. a pair of node index and *reference* name, is stored in another vector called `sidechain_list`. It is only after all nodes are parsed and the relevant information is stored that the edges are handled.

To ensure the synchronisation, a `clock` node is connected to all the user-created nodes. The `clock` node is invisible to the users but plays a vital role in avoiding over-processing for some nodes used as *references* in more than one places. When the `process` method is called within each node, the internal clock of that node will be compared with the input from the clock node. If it is already processed once, the node should yield a stored output buffer rather than calculate a new one.

3.3 Dynamic node management

In live coding, it is necessary to update the audio graph in real-time. In Glicol, we have chosen a WYSIWYG (what you see is what you get) approach. This means that every time the user runs the code, the new audio graph is completely dependant on all the lines of code. However, it would be a huge waste to the performance efficiency and would be prone to audio clicks if we reset the entire audio graph every time the update is scheduled.

As is shown in Figure 6, the key of the solution is to manage the nodes dynamically using the classic longest common sequence (LCS) algorithm [2]. First, we parse the new code and process it chain by chain. When dealing with a chain, we compare it with the node by chain *HashMap* stored previously. The result comparing has three possible outcomes. In the first case, the chain shows in the previous code, but not in the new one. Then we will remove all the nodes in this chain from the graph and the side-chain information

list. The second case is that this chain is a completely new one, so we can simply add all the node information to the graph. The last case is that this chain shows previously, but modified in the new input code. For this situation, we use the LCS algorithm to find out the nodes to add and nodes to remove, and keep most of the nodes untouched in the graph.

Taking advantage of this dynamic node management algorithm, we further add a strategy to optimise the audio performance. In the pre-processing stage, the parameter of all the `mul` nodes, as well as oscillator nodes such as `sin` (sine wave oscillator) will be replaced by a control signal that contains a single `const` node.

Listing 1: Code pre-processing. Provide the 'sin' node with a constant frequency control. Replace the 'mul' node parameter with reference.

```
// (user input) aa: sin 440 >> mul 0.1

// what is actually sent to the parser:
aa: const 440 >> sin 1 >> mul ~aa_mul_a

~aa_mul_a: const 0.1
```

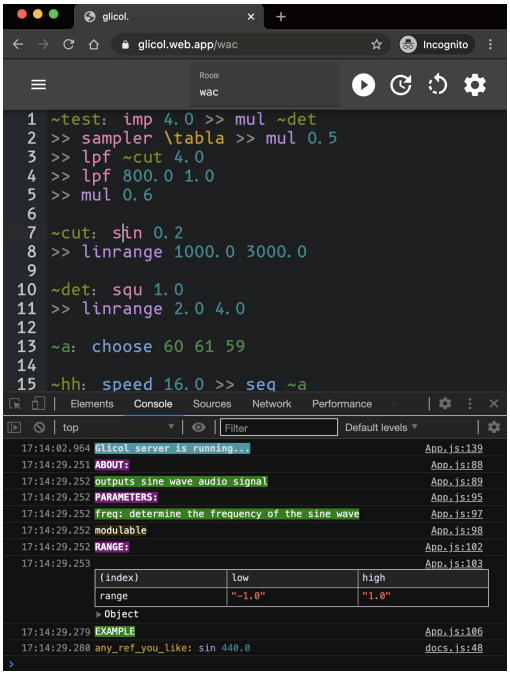


Figure 7: The interface of Glicol. The editor is with syntax highlights implemented with regular expressions. Help files can be accessed from the browser console.

In this way, when the user changes the parameter of these nodes, it is the `const` node that will be removed from the graph and a new `const` node will be added. Then, within the `mul` or `sin` node, the previous audio buffer can be retained and a smooth transit can be thus created.

4. A BROWSER-BASED IDE

The language and audio engine we build in Rust is compiled to a WebAssembly module to run in the browser-based IDE (integrated development environment). To improve the user experience of the environment, we have implemented syntax highlighting in the code editor. We have also built in the documentation in the browser and added support for collaborative coding similar to QuaverSeries.

4.1 Communication between JavaScript, AudioWorklet and WebAssembly

As Figure 7 shows, the browser-based code editor is developed with `Ace.js` [6]. Also, it is implemented with syntax highlighting written in regular expressions. When users click the `run` icon in Glicol, the code string will first be encoded into UTF-8 format. Then it will be posted to the AudioWorklet thread as an *unsized 8-bit array* with a label `run`, using the `postMessage()` method in AudioWorklet.

Once the AudioWorklet thread gets the message, it will call a function `'alloc_u8'` exported from the WebAssembly module. This function will create memory space for the code string. On the JavaScript side, we use the `array.set()` method to write the UTF-8 array to the allocated memory location. Then we pass the pointer (to the memory location) and the size to the `run` function exported from Rust/WebAssembly, which will read the code string and do the parsing.

Similarly, we can pass the audio samples to the WebAssembly/Rust module as array messages. First, users need to switch on the `use sample` option in the settings to fetch the audio samples from the Internet. These downloaded audio samples will be stored temporarily as JavaScript *float 32 arrays*. Then, similar to how we pass the UTF-8 array to AudioWorklet, these sample arrays can also be sent to the Rust/WebAssembly engine by `postMessage()` method, although different from sending the code strings, sending the audio samples requires the WebAssembly/Rust side to allocate memory locations for both the sample names and the sample data. The audio samples will be stored in a *HashMap*, which will later be passed to the `sp` node mentioned before.

To get audio from WebAssembly/Rust, we use the `process` method exported from the WebAssembly. It takes the audio input and output pointer and writes the memory area using the returned audio stream data inside the WebAssembly.

Users can also use the browser console for communication to the WebAssembly module. Adding samples can be done with the `addSamples(name, url)` function export to the browser window. Beats per minutes can be set with the `bpm()` function. And the amplitude of each audio node chain can be set with the `trackAmp()` function.

4.2 Documentation

Our preliminary user testing quickly uncovered the need to add easy access to documentation of the code. Our solution is to use the browser console for documentation. When a user opens the Glicol interface, all available nodes will be printed in the console. Similarly, when users load samples,

[6] https://ace.c9.io/

all available samples will be shown. Users can also choose to select a keyword and get help using the keyboard shortcut command-slash (Figure 7).

4.3 Collaborative coding

Based on our positive experiences with collaborative coding in QuaverSeries, we have also added this feature to Glicol. This has been implemented with Firepad [7]. Users can choose to enter the same Glicol 'room', a space where they can start collaborating immediately. Each user can decide when to run or update the code on their machines.

5. PERFORMANCE AND RELIABILITY

We have not yet done an extensive evaluation of the system, but have done some measurements of the audio performance and some preliminary user studies.

5.1 Real-time audio performance

In real-time audio processing, the audio is processed in blocks. In Web Audio API, the block size is 128. This means that our WebAssembly module needs to yield 256 audio samples (128 stereo frames) in each block time, i.e. the real-time budget. The load index in real-time audio is calculated as:

$$load[t] = \frac{render\,time[t]}{realtime\,budget[t]} \quad (1)$$

and should not exceed one. Figure 8 shows the result of a performance test in Glicol, where we simulate a real-world live coding performance and use the tracing tool in Chrome to monitor the performance of AudioWorklet. It shows that the real-time audio performance is stable, efficient, and below the danger zone.

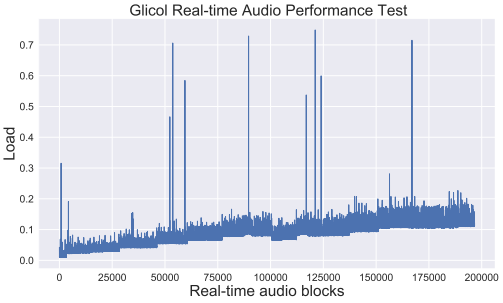


Figure 8: Glicol audio engine performance during a performance test with Chrome tracing tool.

Another way to test the audio performance of Glicol is by monitoring the load value reported by the Web Audio tool in the Google Chrome console. From the videos of Glicol recordings in this YouTube playlist [8], we can see that that the audio load of Glicol is typically stable and well below the limit.

[7]https://firepad.io
[8]https://bit.ly/3nUMYDH

5.2 Error handling

Handling code errors is imperative for the success of a live coding performance. A single character typo can cause the whole program to err and the music to stop. Fortunately, Rust offers a robust error handling solution with `Option` or `Result`. Each time the user makes an update, the engine will try to parse the code and make the graph. If there is an error, the result will be an Error type. Should this happen, the engine will use code from the last successful update and continue to play music. Information about the error's type and position will then be passed back to the AudioWorklet thread along with the ongoing audio array. A message will also be printed in the console as is shown in Figure 9.

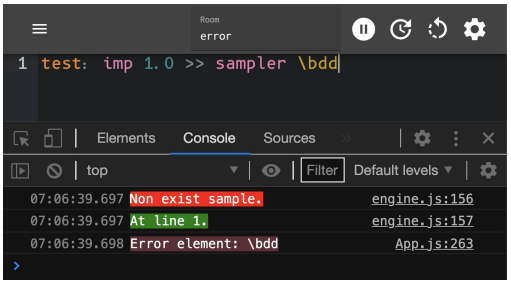


Figure 9: Error messages are dumping to the console. With hints on where the error comes and the error element.

6. DISCUSSION

Our experience with the development of Glicol is that Rust is a powerful language for audio-based live coding applications. It also provides new ways of handling errors, making the environment safe to run and efficient to program. Glicol is currently stable and deployed on a website[9] with documentation and tutorials available. In the following, we will reflect on how Glicol compares to some other live coding languages.

6.1 Glicol as an audio server

One of Glicol's main contributions is its graph-oriented syntax, together with the audio engine implemented seamlessly. With the help of Rust's famous zero-cost abstraction [10], we believe that this design pattern can bring a minimal performance loss for live coding language design. In view of that, Glicol's language can also be viewed as an intermediate language that connects the user's intention with the audio engine. Therefore, a potential use case is that users can create customised DSLs and then interpret them into Glicol syntax and send to the Glicol audio engine, similar to SuperCollider and its relevant clients. One difference from SuperCollider is that we choose the WYSIWYG design model and come up with a dynamic node management method to satisfy its use in live performances, but in SuperCollider and its clients, performers should typically determine which paragraph(s) of code to execute.

[9]https://glicol.web.app
[10]https://boats.gitlab.io/blog/post/zero-cost-abstractions/

6.2 Potential in music education

Although Glicol is designed for live performances, its graph-oriented nature also makes it suitable for sound and audio effect design. For example, the `plate` node in Glicol that wraps the Dattorro's reverb effect [6] is written with the Glicol syntax in Rust [11]. Thus, we believe it has great potential in music education, and this will be studied further in more extensive user studies.

6.3 Limitations and future work

Even though Glicol is currently fully functional in its current state, it has several limitations. For example, it has limited input capabilities. We are planning inputs for MIDI, and it would also be useful to receive OSC messages.

As for future work, the main priority is to organise a formal user study. Here the idea is to test Glicol on people with different levels of musical and live coding experience. The user study will also guide the development of tutorials, possibly targeted at different user groups.

In parallel to the user study, we will extend the APIs, so that the audio engine can be used as a standalone library. Another possibility is to develop more APIs for users to create their own customised language based on Glicol. When a stable version is finished, we also aim to run the language on microcontrollers, such as Bela. Also, we plan to port it to Python to leverage various types of machine learning libraries. The idea is to explore how it is possible to train virtual agents for different types of collaborative performance.

7. ACKNOWLEDGEMENTS

This work was partially supported by the Research Council of Norway through its Centres of Excellence scheme, project number 262762 and by NordForsk's Nordic University Hub Nordic Sound and Music Computing Network NordicSMC, project number 86892.

8. REFERENCES

[1] S. Aaron and A. F. Blackwell. From sonic pi to overtone: creative musical experiences with domain-specific and functional languages. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 35–46. ACM, 2013.

[2] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48. IEEE, 2000.

[3] F. Bernardo, C. Kiefer, and T. Magnusson. Designing for a pluralist and user-friendly live code language ecosystem with sema. In *International Conference on Live Coding*.

[4] H. Choi. Audioworklet: the future of web audio. In *ICMC*, 2018.

[5] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised sound*, 8(3):321–330, 2003.

[6] J. Dattorro. Effect design, part 1: Reverberator and other filters. *Journal of the Audio Engineering Society*, 45(9):660–684, 1997.

[11] https://bit.ly/2WQht1C

[7] A. Freed. Open sound control: A new protocol for communicating with sound synthesizers. In *International Computer Music Conference (ICMC)*, 1997.

[8] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[9] Q. Lan and A. R. Jensenius. Quaverseries: A live coding environment for music performance using web technologies. In *Proceedings of the International Web Audio Conference (WAC)*, pages 41–46. NTNU, 2019.

[10] N. Laurent and K. Mens. Parsing expression grammars made practical. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 167–172, 2015.

[11] S. Letz, Y. Orlarey, and D. Fober. Compiling faust audio dsp code to webassembly. 2017.

[12] T. Magnusson. ixi lang: a supercollider parasite for live coding. In *Proceedings of International Computer Music Conference 2011*, pages 503–506. Michigan Publishing, 2011.

[13] Y. Mann. Interactive music with tone. js. In *Proceedings of the 1st annual Web Audio Conference*. Citeseer, 2015.

[14] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[15] A. McLean. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 63–70. ACM, 2014.

[16] A. McPherson. Bela: An embedded platform for low-latency feedback control of sound. *The Journal of the Acoustical Society of America*, 141(5):3618–3618, 2017.

[17] C. Roberts and J. Kuchera-Morin. Gibber: Live coding audio in the browser. In *ICMC*, 2012.

[18] S. Shirinivas, S. Vetrivel, and N. Elango. Applications of graph theory in computer science an overview. *International journal of engineering science and technology*, 2(9):4610–4621, 2010.

[19] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.

[20] A. Xambó, A. Lerch, and J. Freeman. Music information retrieval in live coding: A theoretical framework. *Computer Music Journal*, 42(4):9–25, 2019.

[21] A. Xambó, G. Roma, P. Shah, J. Freeman, and B. Magerko. 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*, 2017.

[22] S. Yi, V. Lazzarini, and E. Costello. Webassembly audioworklet csound. In *4th Web Audio Conference, TU Berlin*, 2018.

Appendices

Appendix A

Appendix

Thesis repository

This thesis contains four projects and three performances. The projects are all open-sourced. The link to the projects source code can be found in this repository and it will be updated in case of any changes:

https://github.com/chaosprint/phd

DOIs

For citation and archiving purpose, the source code, dataset, performance videos and tools used in the performance/data collection all come with DOIs.

Source code

RaveForce: https://doi.org/10.5281/zenodo.6539900
Air Guitar: https://doi.org/10.5281/zenodo.6529455
QuaverSeries: https://doi.org/10.5281/zenodo.6527560
Glicol: https://doi.org/10.5281/zenodo.6539797

Performance

Oslo World Music Festival 2019: https://doi.org/10.5281/zenodo.6539865
WAC 2019: https://doi.org/10.5281/zenodo.6539871
WAC 2021: https://doi.org/10.5281/zenodo.6539831

Datasets and tools

The Myo recorder: https://doi.org/10.5281/zenodo.6539906
Air Guitar datasets: https://doi.org/10.5281/zenodo.6470236
The SuperCollider interface for the Oslo World Performance:
https://doi.org/10.5281/zenodo.6539759