The Artists who Say Ni!: Incorporating the Python programming language into creative coding for the realisation of musical works

Alexandros Drymonitis

A thesis submitted in partial fulfilment of the requirements of
Birmingham City University
for the degree of Doctor of Philosophy

December 2021

The Faculty of Arts, Design and Media, Royal Birmingham Conservatoire,
Birmingham City University

Abstract

Even though Python is a very popular programming language with a wide range of applications, in the domain of music, specifically electronic music, it is much less used than other languages and programming environments that have been built explicitly for musical creation, such as SuperCollider, Pure Data, Csound, Max, and Chuck. Since 2010 a Python module for DSP called Pyo has been available. This module consists of a complete set of DSP algorithms, Unit Generators, filters, effects, and other tools for the creation of electronic music and sound, yet its community is rather limited. Being part of Python, this module can be combined with a big variety of native and external Python modules for musical or extra-musical tasks, facilitating the realisation of interdisciplinary artworks focusing on music and sound. Starting a creative journey with this module, I was led to more Pythonic techniques for tasks other than music, like mining tweets from Twitter or creating code poetry, which I incorporated into my musical activity. This practice-based research explores the field of the creation of musical works based on Python by focusing on three works. The first one is a live coding poetry opera where the libretto is written in Python. The second one is a live algorithmic composition for an acoustic ensemble based on input from Twitter. The last work is a combination of live coding with live patching on a hardware modular synthesiser system. The main objective of this thesis is to determine the creative potential of Python in music and mixed media art by posing questions that are answered through these works. By doing this, this research aims to provide a conceptual framework for artistic creation that can function as inspiration to other musicians and artists. The title of this thesis is based on one of the most popular lines of the Monty Python comedy troupe, “the Knights who say Ni!”, since the initial developer of the Python programming language, Guido van Rossum, gave this name to this language inspired by Monty Python.

Keywords: Music composition, electronic music, Python, computer programming, live coding, live patching, code poetry, data mining

Acknowledgements

I would like to thank my supervisor Dr James Dooley for believing in my work and opening the door to PhD research. My supervisor Dr Simon Hall for always being friendly and supportive. My supervisor Prof Lamberto Coccioli for his perspective on my work and for his problem solving approach at a crucial time during my research. All three supervisors have provided excellent guidance and have shown patience throughout my PhD. They have helped me develop my research and academic skills.

I would like to thank Carrie Churnside for providing answers to most of my administrative questions, and for always forwarding my queries to the appropriate staff.

The Birmingham City University Collections team has been very helpful with various financial matters.

I would like to thank Dr John Richards for being my supervisor at the first stages of my research.

I would like to thank Tim Chilina from the openFrameworks forum, for helping out and providing functional code for a certain task.

Above all, I am grateful to my wife Katia, for supporting me in many different ways throughout my research.

Related Publication

The thesis may include text from the following publications.

1. Alexandros Drymonitis and Nicoleta Chatzopoulou. (2021). Data Mining / Live Scoring – A Live Performance of a Computer-Aided Composition Based on Twitter. Proceedings of the 2nd Joint Conference on AI Music Creativity, 10. https://doi.org/10.5281/zenodo.5137948

2. Alexandros Drymonitis and Nicoleta Chatzopoulou. (2021). Data Mining / Live Scoring – A Live Acoustic Algorithmic Composition Based on Twitter. Proceedings of the 26th Annual Conference on Auditory Display (ICAD 2021).

3. Alexandros Drymonitis. 2021. "Live Coding on a Modular Synthesizer." In *Proceedings of the Sixth International Conference on Live Coding, ICLC 2021*. Valdivia, Chile.

Table of Contents

Table of Figures

Chapter One

Introduction

This practice-based research is an exploration of the possibilities afforded by the Python programming language[1] in the field of music creation, both electronic and acoustic. Even though Python is a very popular language, its use in the field of music creation is limited. Personally, I started learning computer programming for music with Max[2] and Pure Data (Pd) (Puckette, 1997), two of the most popular visual programming environments designed for multimedia artistic applications, and Python occurred at a later stage out of a wider interest in programming. Willing to combine this language with my main focus, music, I presented my first experiments in small local festivals in Athens, around 2015. These first experiments grew into bigger projects which became the foundation for a number of works presented at major stages in Athens. Through the course of the four years of my PhD research I was led through a creative journey that resulted in the realisation of three music works and a significant, stand-alone technological artefact, all centred around Python, all three of which are presented, discussed and analysed in this thesis. These works are a live coding poetry opera called *Echo and Narcissus*, where the libretto was written in Python, a live algorithmic composition for acoustic ensemble based on Twitter called *Data Mining / Live Scoring* and a hardware module called Code, that enables live coding on a modular synthesizer system I have developed, called 3dPdModular[3]. These Python-themed works were created within a personal musical aesthetic and idiom, which I will outline in section 1.2, to offer their context and help framing my research.

1 https://www.python.org/
2 https://cycling74.com/products/max/
3 https://3dpdmodular.cc/

1.1. Layout of the Thesis

The portfolio of this thesis consists of three pieces with a total duration of approximately 2 hours and 5 minutes. This thesis accompanies and focuses on these three works, analysing them from various points of view. The focus of the research is on the artistic outcome of the artefacts, their achievements in terms of innovation in the arts, and their conceptual context. This thesis aims to highlight these aspects through critical analysis and comparison between the works of the portfolio and related works. The layout of the commentary is as follows. Within the introductory chapter, Section 1.2 onwards provides an aesthetic context based on my activity from an early stage up to the point of the beginning of my research. Section 1.3 offers an analysis of various software concepts. Section 1.4 presents the research questions and section 1.5 the methodology. The second chapter focuses on the opera *Echo and Narcissus*; the third chapter focuses on the algorithmic composition *Data Mining / Live Scoring*; and the fourth chapter focuses on the live coding / live patching performance enabled by the Code module. Chapter five concludes the thesis.

1.2. Personal Aesthetics and Computer Programming

My Bachelor’s and Master’s degree studies were centred on the classical guitar. During my Master’s studies, I focused on contemporary music, either by performing compositions by established contemporary composers, compositions from the 1960s onward, or by collaborating with students from the Composition department to provide advice concerning extended techniques, so that they could compose new pieces for the guitar. This pursuit of new pieces for the guitar was extended after my studies. Some composers whose pieces I performed included Helmut Oehring, Claude Vivier, Elliott Carter, James Dillon, and Giacinto Scelsi, with *Sequenza XI* for guitar by Luciano Berio (McKenzie, 1991) being the most technically demanding piece that I have performed[4]. Fausto Romitelli's *Trash TV Trance* for electric guitar, which I performed in 2014[5] is also a piece that has influenced me aesthetically. This period of guitar performance – both classical and electric – played a crucial role in the development of my musical identity.

Looking for rare contemporary guitar pieces proved to be satisfactory for a certain amount of time. Nevertheless, no matter how personal a performers' playing can be, interpreting music composed by others can sometimes leave a gap in the creative process of music making. In this sense, I reached a point where I needed a way to express myself at a more personal level. This is when I started to make a shift towards improvisation. This shift started while I was in the last year of my Master's studies. It was initiated with compositions by Christian Wolff with an open form such as *For 1, 2, or 3 people* (Gardner, 1971). From structured, through semi-structured improvisation, my practice developed into completely free improvisation influenced by local fellow musicians, but also artists with a wider reach, such as Evan Parker, John Zorn, and George E. Lewis.

Another period that ran in parallel and was aesthetically very influential to me also took place during my studies. During the first two years of my studies in the Guitar department at the Conservatorium van Amsterdam, I attended composition classes as a secondary subject, and the year after that, I attended the first year of the Bachelor's program in the Composition department. Here I was introduced to 20th century modernist works by composers such as Iannis Xenakis, Karlheinz Stockhausen, Brian Ferneyhough, and others, and their composition techniques. Even though I did not continue as a

4 https://vimeo.com/433336227
5 https://www.youtube.com/watch?v=DkgXeLafJu8

composer of that type, the stimuli from that period affected my phraseology as an improviser and my overall conception in music creation.

While the period of contemporary music for classical guitar or other instruments was coming to an end for me, another period was beginning, that of electroacoustic and electronic music, and especially noise. In the academic years 2006/2007 and 2009/2010 when I was residing in Amsterdam, I was frequently attending the DNK concert series[6], which used to take place weekly during 2006/2007 and every few weeks during 2009/2010. These concerts hosted new and experimental music, oftentimes electronic and noise sets. This is how I was introduced to the noise genre which greatly affected me from an aesthetic point of view. Upon returning to my home town, Athens, I joined the artist group Medea Electronique[7] as a member of the now-dissolved Instant Synthesis Ensemble (I.S.E.), a free-improvisation electroacoustic ensemble that was initiated by Medea in 2010. With I.S.E. I took part in a period of two to three years of intensive rehearsing and several concerts. I also started attending concerts at the Knot Gallery[8] in Athens which, much like the DNK series, hosted performances of experimental music with a strong focus on electronic music and noise.

From 2012 to 2018 I was the main organiser of Medea's electronic music festival *Electric Nights*[9]. This festival started as a twenty-seven-hour long electronic music marathon and evolved into a two-evening festival with concerts and workshops. During the six editions of the festival with which I engaged, we hosted over one hundred artists from four continents showcasing a wide range of electronic music idioms.

All the concerts and festivals mentioned above hosted mainly artists from underground scenes. The nature of this kind of scene meant that the careers of many

6 http://www.dnk-amsterdam.com/index.html
7 https://medeaelectronique.com/
8 http://knotarts.blogspot.com/
9 http://www.medeaelectronique.com/electricnights/

artists were quite transient and short-lived. Often, I would encounter an artist only once without being able to further follow their work. Still, this extended period of exposure to electronic music and noise left an indelible mark on me which is reflected in my personal aesthetic, musical voice, and working practices. For example, the textures found in the *Echo and Narcissus* opera as well as the performance with the Code module for the 3dPdModular system have their roots in a developing process that started early on during the period of attending the DNK concert series.

The period of electronic music led me to programming. Having started with Max and Pd, I was introduced to textual programming through the C programming language (Ritchie et al., 1981), Arduino (Drymonitis, 2015), and Python (Lutz, 2010). I used C for writing external objects for Pd, the Arduino programming environment for controlling Arduino boards, and Python for more generic programming. Python is a programming language with a very wide range of applications and a significant community of programmers. Its simple syntax makes it the language of choice for beginner programmers, while at the same time it is being used at a very high professional level (for example, the code base of the popular social platform Instagram is written in Python[10]). It provides a high level of code abstraction, making it very flexible and powerful for many different tasks. Yet, this very popular language is rarely used in the field of music. Music analysis is a field in which Python might be used more frequently with the use of modules like Music21[11], but for electronic music production, it is a rarity.

When I started experimenting with computer music programming with textual coding, instead of choosing SuperCollider (McCartney, 1996), Chuck (Wang and Cook, 2003), or Csound (Lazzarini et al., 2016) which are the most popular textual programming languages for computer music, I preferred to use Python combined with the Pyo module, a

10 http://euccas.github.io/blog/20170616/how-instagram-moved-to-python-3.html
11 https://web.mit.edu/music21/

Digital Signal Processing (DSP)[12] module for Python written in C (Bélanger, 2016). The introduction of this module provided a gateway to Python programmers who wanted to experiment with electronic sound and music. The reason I chose to experiment with this setup was that I would get to write code in a very popular language that reached far outside the realm of music. By writing Python code for electronic music, I started incorporating various Python modules or techniques, that were not specific to music or sound in general. These included mining tweets from Twitter or utilising Python's simple syntax to translate poems to the code poetry paradigm, resulting in the creation of the works of this research.

Before moving on to the next section and the research questions, it is worth briefly clarifying some basic concepts around computer programming and coding. General-purpose programming languages are computer languages with a broad spectrum of applications that lack features that are specialised to a specific domain – such as audio programming. Domain-specific languages are the opposite, they are specialised to a certain domain and offer a framework for programming applications within that domain. For example, audio programming languages offer a set of sound-related features, such as audio processing effects and sound synthesis. Visual programming languages, like Pd, provide a graphic interface representing the flow of data within a program, in contrast to textual programming languages, like SuperCollider, where programs are written as text. Finally, live coding is a practice where a coder writes computer code during a performance and this code is being executed in real-time. By convention the coder's screen is being projected so the audience can see what is being typed.

12 DSP refers to the processing of various types of signals, either digitised analogue or purely digital signals, for a variety of tasks.

1.3. Software Concepts

In this section, I will review a number of software platforms for musical creativity that utilise Python in some way, and reflect to why these were inappropriate in the creation of the works of this research. I will also briefly analyse the Pyo Python module for DSP. Finally I will reflect on live coding, the various languages used for it, and argue how Python can be a strong candidate for this application.

The main reason for the absence of Python in music is that it is a rather slow language when programs that are written in it are executed, whereas digital audio is a demanding task with strict temporal requirements, something that can be effectively handled by languages like C or C++ (De Pra and Fontana, 2020, p. 2). Most of the popular computer music programming environments have been built with these requirements in mind, thus their audio engines are built either with C or C++. Pd and Max objects are written in C, while SuperCollider and Chuck classes are written in C++, and whilst it is possible to write Pd and Max objects in other languages, the default is C. The scipy.signal[13] Python module provides an extensive list of classes for DSP, but its focus is oriented towards scientific analysis of digital and digitised[14] signals not necessarily relating to audio or music. The Pyo module is written in C and therefore has the capability of being very efficient in terms of the demands of a DSP engine. It includes an extensive set of classes for various DSP processes, focused on music. Its infrastructure loosely relates to the SuperCollider and Chuck paradigms, whose audio engines run on C++ and their control interfaces run on a different language, as this module has a C backend and its interface is pure Python. The introduction of this module brings Python, a mature language

13 https://docs.scipy.org/doc/scipy/reference/signal.html
14 Here digital refers to signals concerning communication between digital devices (clock and data signals), and digitised refers to analogue signals converted to digital so they can be further processed in a computer.

with a high volume of extensions, which runs on many different types of hardware and can be easily embedded to other languages, to the field of computer music.

1.3.1. Paradigms of Python in Music Software

Even though Python is rarely used in music, one can find examples of music software that utilises it, usually in combination with other languages. Since Python alone is not very efficient for real-time audio, most cases of musical software using this language depend on other software that runs in parallel and specialises in sound synthesis. A brief description of some projects using Python that are contextually relevant to my work follows.

1.3.1.1. FoxDot

FoxDot is a Python module that communicates with SuperCollider to create sounds (Kirkbride, 2021). The user must launch SuperCollider and run a command in that environment before launching FoxDot through Python. Once FoxDot launches, an editor opens up and the user can write code and execute it in real-time. This module provides a set of predefined instruments which the user can utilise with minimal coding. Such a feature has the benefit of fast prototyping, which is usually desirable by composers as they can focus on the creative aspect of their work instead of developing their programming skills, but it also bears the limitation of the tools available by the specific library. If the user wants to create a custom sound, they must write code in SuperCollider and bind it to FoxDot. Even within FoxDot, the user must get accustomed to a certain syntax for certain tasks. For example, to create a rhythmical sequence, a syntax similar to the following must be used:

```
d1 >> play(“(x[--])xo{-[--][-x]”)
```

This is already a very specific syntax that is not self-explanatory and requires the user to get acquainted with it to a certain extent. The user of FoxDot must become familiar with the syntax of this software, apart from that of Python, and perhaps another programming language, SuperCollider.

There are various reasons I did not choose this software for any of the three works of this research. The main reason I did not choose FoxDot for the *Echo and Narcissus* opera was the fact that I wanted to maintain code integrity and not spread this project among more than one programming language. For this work, I would have had to write SuperCollider code to create the various sounds I used, as FoxDot is a library focused on melodic and rhythmic live coding, which is not related to my aesthetics.

For the Code module using Python with Pyo was my main objective as this is my main textual language of choice for computer music. The 3dPdModular system is programmed in Pd and any sound generating engine should run within this environment. Pyo comes with a Pd external called *pyo*~ which embeds Python with the Pyo module into Pd. This makes using Pyo with the Code module a much more streamlined process than trying to port FoxDot into Pd.

Data Mining / Live Scoring does not aim at creating electronic sounds, so FoxDot would not serve any useful purpose in that sense. For this work I needed to use Python for its more generic capabilities – such as Machine Learning and text file manipulation – so there was no specific need to include any software based on SuperCollider.

1.3.1.2. py/pyext

py/pyext is an external object for Pd and Max which integrates Python into these visual programming languages[15]. Unlike most Pd objects which import other programming languages in the control domain only, this object supports operations on audio signals. Its main disadvantage is that it is currently built against Python2.7 which is now declared as End of Life (EOL) software, with Python3 being the actively developed branch. This object could have been a strong candidate for *Data Mining / Live Scoring*, especially if it was combined with the *notes*[16] external object for Pd which uses Lilypond (Nienhuys and Nieuwenhuizen, 2003) to create music scores. The reason for not using *py/pyext* was that I intended to make future versions of this work, therefore I aimed at using software that would not become obsolete in the near future. For *Echo and Narcissus* and the Code module it was not possible to use this external object as it does not allow live coding in any way.

1.3.1.3. SuperCollider Through Python

Currently there is a Python module under development, called sc3, that introduces the SuperCollider library to Python, by using the latter to control the scsynth[17] (Samaruga, Silvani and Saladino, 2021). This module is very promising, but it is still in an alpha stage, with some features of the SuperCollider environment still missing, and the documentation still under construction. Besides that, even though this module bears SuperCollider's elegant SynthDefs[18] and other features, its syntax is not as simple as Pyo's.

15 https://grrrr.org/research/software/py/
16 nyu-waverlylabs.org/notes/
17 SuperCollider is split between the scsynth (standing for SuperCollider Synth) and sclang (SuperCollider Language). The scsynth is the audio engine of the SuperCollider environment, and the sclang is its programming language.
18 SynthDef stands for Synth Definition, where a chunk of code provides a complex synthesiser structure.

1.3.1.4. Other Software Based on Python

There is software based on Python that whilst contextually or historically relevant, is either obsolete or not entirely fitting to the needs of the three works of this research. In the interest of completeness these are mentioned to add to the software context of this research.

Pyata[19] is a Python module that enables the user to control Pd patches in Python This approach might introduce some advantages compared to programming in Pd alone, as it provides much greater flexibility for live patching since this can be done with computer functions and loops and not necessarily manually. Nevertheless, the user must be accustomed to programming in Pd, as Pyata is only responsible for creating objects and making connections between them in the Pd environment. This software should be considered obsolete since at the time or writing the last update was made in 2010.

JythonMusic is a software environment based on Python and jMusic, a Java library to support music composition (Manaris et al., 2018). jMusic relies mainly on the MIDI protocol and focuses on creating and manipulating music structures as MIDI data. JythonMusic provides its own Integrated Development Environment (IDE)[20] where the user can write Python code to call the various JythonMusic commands, but also use it via a terminal window. Its audio capabilities are rather limited since it does not provide Unit Generators (audio wave generators)[21], filters, or other DSP processes.

19 https://code.google.com/archive/p/pyata/
20 An IDE is an editor that provides a set of tools specific to the language it is used for, like text highlighting, code snippet insertion, keyboard shortcuts, etc.
21 A Unit Generator is a mechanism that generates audible sound waves through mathematical functions.

1.3.2. Live Coding

According to zmölnig and Eckel, live coding started to flourish around the end of the 1990s with one of the first known live coding sessions dating back to 1985, by Ron Kuivila (zmölnig and Eckel, 2007, p. 295). This practice is still under active development, with new languages and programming environments being created especially for live coding. In this section I mention some of the most popular languages for live coding as well as derivatives of those, together with arguments about how Python can be a strong candidate for this practice.

1.3.2.1. The Textual Programming Languages

Based on the existing literature on live coding, which tends to foreground SuperCollider as a lingua franca, with Chuck as the second most popular tool, one can assume that these are the two most popular languages for practitioners working in this context. While these two languages provide a very flexible and powerful framework for live coding (Wang and Cook, 2003, p. 6; Rohrhuber and de Campo, 2011), in my own research, I wanted to use a general-purpose language instead of a domain-specific one, in order to exploit the advantages allowed by its generic capabilities.

Apart from FoxDot, there are further derivatives of SuperCollider. Ixi lang is a SuperCollider parasite for live coding (Magnusson, 2011). It aims to enable the live coder to “create a tune with rhythm and melody within a few seconds from the performance starting” (Magnusson, 2011, p. 1). This idea became apparent to the developer of the language by attending live coding sessions where the coder would take minutes to produce very simple elements. Sonic Pi (Aaron and Blackwell, 2013, p. 38) is another language that utilises the SuperCollider audio engine. This system was built with teaching in mind. Its main aim is to teach coding to students by live coding music to stimulate them and maintain their engagement (Aaron and Blackwell, 2013, p. 35). Tidal Cycles is another language which uses SuperCollider for creating sound (McLean, 2021), aiming at creating rhythmic sequences and patterns. All these languages aim at creating rhythmic and melodic structures and are mainly used for loop-based music. Breaking out of their framework includes writing SuperCollider code and bind it to these languages. This breaks language integrity, as with the case of FoxDot, and adds another level of complexity. For my work, since Python already had a module for DSP focused on music, and due to the various extra-musical elements that are very well supported by the generic nature of this language, paired my familiarity with it, it made sense to develop work purely within Python.

1.3.2.2. The Visual Programming Languages

Besides the textual domain, there are two dominant languages that are rooted in the visual domain: Max and Pd. These programming environments allow for live coding though with a different paradigm to that of textual coding due to the dataflow nature of these environments. In this case live *patching* is a better nomenclature than live coding since programming with these visual environments is called patching. The user is called to connect objects which are represented by small boxes with the name of the object typed in them. Lines connecting objects represent a control or audio signal connection. Figure 1 shows a simple Pd patch to generate a 440Hz sine wave at 1/5 of its amplitude. Textual code is often seen as daunting by non-programmers, so a visual language is often the language of choice of beginners in computer music programming. Whilst pervasive, neither of these languages was appropriate for my works whether they utilise live coding or not. The works of this research either treat code as narrative text, or are based on a concept that focuses on textual coding, or require functionalities that are not music-based, thus not included in a computer music programming language like Pd or Max.

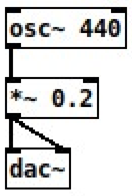


Figure 1: A simple Pd patch.

1.3.2.3. Python for Live Coding

There is very little use of Python alone in music programming (De Pra and Fontana, 2020), and even less, if any at all, in a live coding context. Languages that are popular within the live coding community, usually provide their own IDE including a range of functionalities that facilitate writing code within their domain. Python provides an interpreter – usually inside a terminal window – where the programmer can write Python code which is executed line by line, much like the live coding practice. The Python interpreter is usually limited in its editing capabilities, as the user can edit only single previously written lines, and not bigger chunks of code, plus it lacks features like text highlighting and code completion. A Python interpreter though, can be embedded in a text editor, such as with the Hydrogen package for the Atom editor[22]. The programmer can take advantage of the tools of these editors in a Python live coding session. The combination of these tools – the Pyo module, the Python interpreter, and text editors with an embedded Python interpreter – can provide an IDE for Python that can be a strong candidate for live coding, standing

22 https://atom.io/packages/hydrogen

next to languages that are much more popular and have been built explicitly for this artistic practice.

1.4. Research Questions

The three works that constitute my practice-based research and are analysed in this dissertation, aim to answer the following questions:

- What advantages can a general-purpose programming language like Python have over domain-specific languages in audio programming, such as SuperCollider, especially in a live coding context?
- How can creative coding in a general-purpose programming language enrich an acoustic musical work by applying concepts outside the field of music and how can it help in the realisation of such works?
- What creative possibilities can a language with a simple syntax have over languages with a more esoteric syntax?

These questions percolate through the works presented.

1.5. Methodology

The methodology applied throughout this research is based on an iterative design cycle where observed outcomes of one stage inform its successive stage of investigation. Through this process, a system for each work was developed. Being a practice-based research, the production of the three works of the portfolio was the initial objective. The artistic outcome of these works served to evaluate the systems developed by testing various elements of these works, like structure, sonic textures, and conceptual framework, against hypotheses posed during development. This thesis serves to illuminate these works rather than the works serve to illuminate the thesis (Candy and Edmonds, 2018, p. 65).

The objective of each work was the starting point for defining the methods used to realise each idea. In the case of *Echo and Narcissus*, incorporating the code poetry and live coding practices was the main objective from the very early stages of the conception of the work. This restriction applied a framework to the development of the technological architecture of the play, that of textual coding, and eventually the use of Python. As this work was produced by Medea, and was not a strictly personal work, the aesthetics of the group applied. From an artistic point of view, the approach by Medea is based on novelty, inter-disciplinary practice, and compound aesthetics. The objective of the group is to provide large-scale works with these characteristics made apparent. The artistic elements of a work though are of the highest priority, with technology being given focus but introduced to the work as seamlessly as possible. These aesthetic principles provided a methodological framework for the development of the opera, where the code poetry and live coding practices had to integrate into the aesthetics of the play without becoming the central focus. Even though these elements defined the character of this work and were the main focus in communicating and advertising the performance, we aimed at producing a play that would still be identified as opera by the audience. Thus, weight was given to the lyrical singing. In doing so, the technological elements – live coding and code poetry – were made apparent as if they were being projected through a semi-transparent veil, still visible, but the foreground – the opera singers – was the main focus of the audience. As opera is an inter-disciplinary form of art, there were many different elements that were developed, each by a different member of Medea, or a different sub-group. This is were the iterative design cycle mentioned in the beginning of this section was applied. These elements – the code poetry libretto, the electronic music, the singing, and the visuals – were being developed either in parallel or in an overlapping fashion. The design cycle approach served to partly develop one element, test against other elements, and revisit, until a satisfactory level was reached.

Data Mining / Live Scoring is a more conceptual work than the opera, with the music being the core element, but combined with other, extra-musical, elements. Before anything else, I knew I wanted to download tweets and analyse them in a way that would somehow connect them to the resulting music. Thus, the music would both be the core of the work, and it would also project these tweets by augmenting them in a theatrical or cinematic fashion. This idea resulted in applying sentiment analysis on the text, the result of which would define various rules for the created music. As my works are usually inter-disciplinary, my method dictates that each discipline is being realised by a competent person. Even though I compose music myself, for reasons that are analysed in the third chapter of this thesis, I collaborated with a composer who wrote a music library that was used for the algorithmic composition of this work, and I undertook the development of the algorithm and all of the computer programming. From an early stage of development, the resulting compositions were performed in rehearsals by the ensemble the work was made for. Based on intuition both by the composer and me, but also on the various rules that emerged during the development of the algorithm, we would change certain elements of the music library or the algorithm that were not satisfactory, we would further develop others that were, or introduce new ones to extend the variety of the work.

The Code module was developed as an extension to the 3dPdModular system. This fact alone bore restrictions in the possibilities of the software used. The main objective was to have the ability to write Python code live and create or process audio signals, combined with the tangibility of the modular synthesiser. Before starting to develop the music performance, I had to find a way to integrate Python and the Pyo module into Pd. The performance I realised with the Code module was based on the concept of utilising extra-musical elements as a core part of the work, to project the flexibility of the generic nature of Python, in combination with its DSP capabilities. In this case, geolocation data from Google served as the main characteristic of the performance. I iterated through possible ways of utilising these data in a musical context, that would also project their conceptual use. In need to bind geolocation to sound, I developed the concept of this performance further and decided to make a critique on localism, projected through sound that was accompanied by the projection of the code, therefore the projection of the geolocation data and their cities. Besides the conceptual and artistic character of the work, I tried to highlight the inspiration one can get from using Python. All the code written for this performance, either musical or extra-musical, was done in this language only. By providing a multitude of modules for many different tasks, Python can prove to be very inspiring for the creation of multi-media and conceptual artworks.

Chapter Two

Echo and Narcissus, Code Poetry in Opera

Musical theatre today must be conscious of its genetic inheritance. It must not forget that an opera house remains a museum, sacrosanct and indestructible, and that conditions are ripe for constructing a better and different future, for forming a more aware and interesting audience, an audience of real listeners and not consumers of music.

Luciano Berio, 1999

Echo and Narcissus is a mixed media opera for two singers, live electronics, and live visuals staged by the artist group Medea Electronique. The story is based on Ovid's 'Echo and Narcissus' from his poem *Metamorphoses*. In this story, Echo is a Nymph who is cursed to only repeat the words of others and to not be able to speak her own. Narcissus is a beautiful youth who she sees in the forest when he goes hunting. Narcissus sees his own reflection on the surface of a lake when he tries to drink water and falls in love with it, finally drowning while trying to kiss his own reflection. Following from the example of Luciano Berio, Medea Electronique aimed to create a performance with a new point of view on opera, introducing new elements to most aspects of the work. The source story follows a classical operatic approach, where an old myth is used as the theme for the work. This is arguably one of the few aspects of *Echo and Narcissus* that does not introduce something new to the opera, but the specific myth served Medea well, as it is the first myth to fuse image and sound, mirroring the audio-visual nature of the group.

The key aspect of *Echo and Narcissus* is that it is based on the practice of live coding and code poetry. The libretto was written in the Python programming language instead of English or another spoken language. It is intended to be typed live during the performance where it serves three purposes. First, it expresses the story as it unfolds in this opera. Second, it triggers all sound processes of the opera, whether processing the voices of the singers or creating electronic sounds through Unit Generators. Third, it functions as a kind of conductor, giving pace to the performers, as they have to wait for the next libretto line to be typed before they are able to sing it. The novelty of this opera is based not only on the complete absence of an orchestra and the only acoustic sounds being the voices of the two singers, but also on the actual libretto itself, the way it was written and its functionality. My role in this project was to adapt the libretto to the Python language, and create the processing algorithms of the singers' voices as well as the electronic sounds that accompanied them. During the development of this work, I collaborated closely with Medea members Angeliki Poulou, the dramaturgist of the opera, and Manolis Manousakis, the music director of this production. It was premiered on the 19th of April 2018, at the Onassis Cultural Centre[23].

2.1. Related Works

According to Jessop et al., the opera is a new model of performance, compared to music, dance, and theatre, which is "still developing and still free for experimentation and exploration" (Jessop, Torpey and Bloomberg, 2011, p. 349). Nevertheless, globally, most opera houses are mainly focused on hosting productions of opera works from the classical and romantic repertoire, even though these productions often include contemporary and complex technological systems. The opera repertoire is centred around compositions from

23 https://www.onassis.org/video/echo-and-narcissus-by-%CE%BCedea-electronique

the 19th and early 20th century, with most contemporary or modern contributions still following the well-established paradigm of the symphonic orchestra, occasionally combined with live electronics – e.g. Luciano Berio and Karlheinz Stockhausen. In this section I will mention some operatic works that bear similarities to or had an impact on *Echo and Narcissus* in some way, though most of these examples fall in the well-established paradigm mentioned above.

2.1.1. Extending the vocal aspect

Carl Unander-Scharin is a Swedish composer who integrates technology in various ways in his work. Two works of his are worth mentioning that have had an impact on *Echo and Narcissus*, his opera *The Elephant Man* (Unander-Scharin, Höök and Elblaus, 2013) and his project *The Vocal Chorder* (Unander-Scharin et al., 2014). The former is an opera based on the true story of Joseph Merrick (b. 1862 – d. 1890), the man also known as the "Elephant Man" due to his visible physical deformities. In this work, Unander-Scharin collaborated with Ludvig Elblaus and Kjetil Falkenberg Hansen from the Royal Institute of Technology of Stockholm to "develop a gesture controlled signal processing device for stage use" (Elblaus, Falkenberg and Unander-Scharin, 2011, p. 2). This device, called *The Throat III*, aimed to augment the signing of Håkan Starkenberg based on notes from the autobiography of Frederick Treves regarding Merrick, where he states that "his face was incapable of expression", and "his attitude [was] that of one whose mind was void of all emotions" (Elblaus, Falkenberg and Unander-Scharin, 2011, p. 2). Bearing this information in mind, the team developed a device that reflected on these aspects of the personality of Merrick, focusing on his very limited ability of speech. *Echo and Narcissus* follows a similar path, as the processes of the voices of the two singers for Acts I and II are created in a way that gives identity to certain aspects of the personalities of the two characters of the opera, and aim to project them through sound. Narcissus' processed voice gives the impression of an imposing and confident character, and Echo's voice resembles natural echo in a more abstract way.

The Vocal Chorder is a technological work by Unander-Scharin. It is a large interactive instrument to create accompaniment to solo opera singers. The user can control it by using steel wires that produce voltage changes which are turned to MIDI messages and controlled in a Max patch. This is a rather large-scale construction that should somehow be integrated to the scenic design, were it to be used in a live opera performance. The initiative of developing this device is based on the evolution of the opera singing through the centuries where composers were stretching the boundaries of the human voice. Its aim is to provide the singers with the control of their own accompaniment, therefore pushing the boundaries of operatic singing. Act III of *Echo and Narcissus* is based on a process controlled by the voice of the baritone that creates a choir-like accompaniment for the singer based on his voice. Unlike the *Vocal Chorder*, the only control parameter for this process is the pitch of the voice without any hardware needed to control it.

2.1.2. Sound Spatialisation and Dramaturgical Augmentation in the Opera

In 1996 James Oliverio produced a mixed media opera called *StarChild*. In this opera an eight speaker surround sound system was used to project various electronic sounds, part of which was a sonification of data of the comet Shoemaker-Levy 9's collision with Jupiter (Oliverio and Pair, 1996, p. 203). The surround sound system was intended to give to the audience the sensation that it was surrounded by characters conversing telepathically across the vast distances of interstellar space (Oliverio and Pair, 1996, p. 202).

K… is an opera by Philippe Manoury for fourteen singers, a large orchestra, a forty-voice virtual choir, and electronic sounds, as well as sound spatialisation, based on *The Trial* by Franz Kafka (Ramstrum and Lemouton, 2003). It was premiered in 2001 in Paris. The electronics included pre-recorded samples, live samples, synthesized sounds, spatialised events, plus various effects that transform sounds. The spatialisation techniques served to give an extra dimension. For example, in scene 11, spatialisation is used to give the sense of a cathedral (Ramstrum and Lemouton, 2003, p. 143).

Echo and Narcissus bears similarities to both operas as it deployed sound spatialisation in Acts I and II, to provide a sense of localisation. In both Acts, delayed copies of the voice of Echo are diffused in the surround sound system of the venue, giving the impression of the singer being in an imaginary landscape where real echo occurs as her voice is reflected on various surfaces. In Act II, raindrop-like sounds are also diffused in the surround sound system. Combined with the video projection of this Act, the sound gives the audience the impression of being inside a rainy forest, surrounded by trees.

The electronic sounds in Manoury's opera were also used to augment or express feelings created by the operatic singers. In this context, the opera's prologue is an instrumental scene where the electronics depict K's turmoil (Ramstrum and Lemouton, 2003, p. 139). Act IV of *Echo and Narcissus* uses live electronics to express the mourning of Echo for the loss of Narcissus. As the Act progresses, the voice of the singer becomes incomprehensible and the sonic element alone aims to express her bitter feelings.

2.1.3. Mixed Media Operas

A contemporary artist who has created several mixed media operas is Michel van der Aa. His operatic works combine orchestral music with lyrical singing and fixed media, be it soundtrack or film. In the context of *Echo and Narcissus* his chamber opera *Blank Out* where he used 3D film, is worth mentioning (Michel van der Aa, 2016). In this opera the video projections integrate into the stage scenery to augment the on-stage activity. Singers are being both projected life-size and appearing on stage at the same time, thus creating the illusion of the same person appearing multiple times. In Act III of *Echo and Narcissus*, the baritone creates a choir with his own voice, through the live electronics. Hearing the voice of the singer multiple times simultaneously, can be thought of as a sonic equivalent to van der Aa's 3D film.

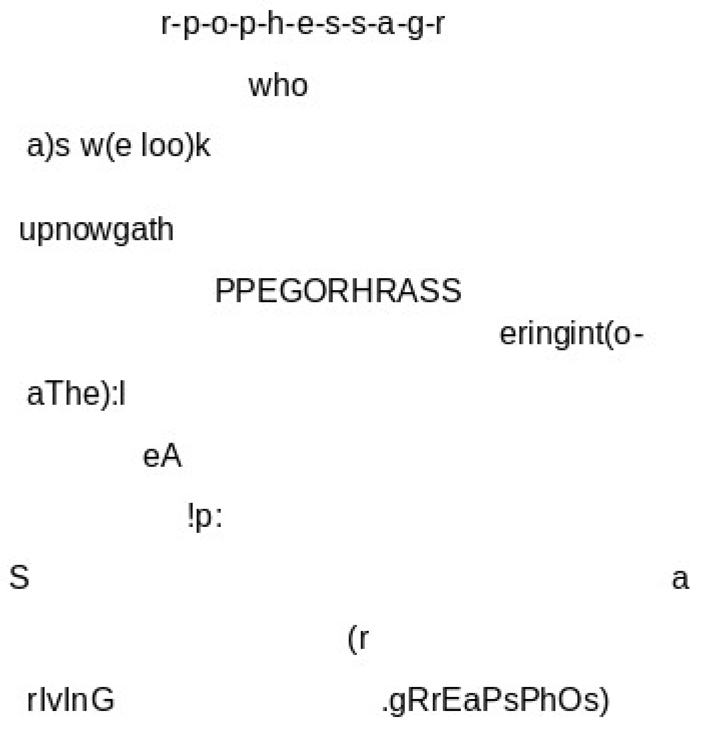
2.2. Code Poetry

As with all forms of art, poetry evolves, and code poetry is a way to incorporate computer technology to this form of art. Despite being an art form with a wide range of styles and techniques, a rule that unifies its various approaches is that the source code has to compile[24] without any errors, thus resulting in a functioning program (Hopkins 1992, p. 391). One form of code poetry is to write a poem in a programming language instead of a spoken one. For example, a poet can choose to write a poem in C++ instead of English. Another form is to write an algorithm that will result in a poem in a spoken language, like English. A third form is to write a poem in a programming language that, when compiled, will result in a different poem with a similar or completely different meaning than that of the source code. This section outlines a few examples of code poems.

24 To compile code means to create an executable program from the source code.

2.2.1. A Predecessor to Code Poetry

A popular poem among the code poetry community is *r-p-o-p-h-e-s-s-a-g-r* by e. e. cummings, illustrated in Figure 2. This poem invites the reader to jump back and forth in order to assemble words correctly so that the poem makes sense. This jumping of the reader’s eye gives a sense of the chaotic motion of a jumping grasshopper (Alvarez, 2017, p. 40). This poem precedes computer coding as we know it. It was written using a typewriter but it is considered a predecessor of code poetry mainly due to is unconventional use of punctuation.



```
                  r-p-o-p-h-e-s-s-a-g-r
                        who
 a)s w(e loo)k
 upnowgath
                     PPEGORHRASS
                                        eringint(o-
 aThe):l
          eA
             !p:
S                                               a
                             (r
 rIvInG                         .gRrEaPsPhOs)
```

Figure 2: The r-p-o-p-h-e-s-s-a-g-r by e. e. cummings (1935).

2.2.2. Early Code Poems

Early examples of poetry that utilises computer commands are found in poems written by members of the Oulipo group in 1960s, where commands like BEGIN, For, Do, Else, etc. Were being used. (Forero 2021, p. 263). Code poetry can also be found in the beginning of 1990s, where Perl seemed to be the language of choice of code poets for a number of reasons. As this language permits a somewhat free use of white spaces and new lines, plus it will not complain when a big subset of its keywords are being abused (Hopkins 1992, p. 392), many code poets of that time preferred to use Perl to write their code poems. A paper from Sharon Hopkins from 1992 demonstrates a few different Perl poems. Figure 2 shows a haiku written by Larry Wall, where Hopkins states that it is the first Perl poem to have ever been written. This poem is meant to be read out loud, so a few conventions concerning pronunciation stand. For example, “STDOUT” must be read “Standard Out” to fill in the necessary amount of syllables of haiku.

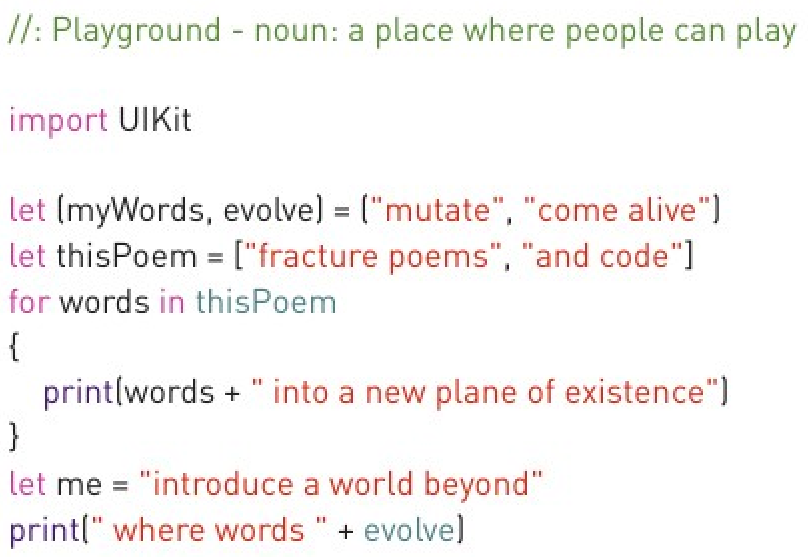
```
print STDOUT q
Just another Perl hacker,
unless $spring
```

Figure 3: "Just Another Perl Hacker" by Larry Wall (1990).

2.2.3. Poems That Mutate When the Source Code is Compiled

One form of code poetry is to write a poem in a programming language and then compile it to generate a different poem, thus ending up with two poems. One example is shown in

Figure 4 which is a poem by Christian Iñigo D.L. Alvarez (Alvarez, 2017). The result of the compiled poem is shown in Figure 5.

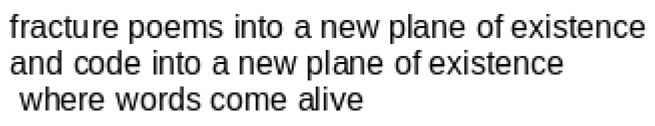


```
//: Playground - noun: a place where people can play

import UIKit

let (myWords, evolve) = ("mutate", "come alive")
let thisPoem = ["fracture poems", "and code"]
for words in thisPoem
{
   print(words + " into a new plane of existence")
}
let me = "introduce a world beyond"
print(" where words " + evolve)
```

Figure 4: A code poem by Christian Iñigo D.L. Alvarez.



```
fracture poems into a new plane of existence
and code into a new plane of existence
 where words come alive
```

Figure 5: A compiled poem by Christian Iñigo D.L. Alvarez.

Here we have one poem written in the source code and a completely different poem resulting when we compile the code. One can note that the poet in this case makes extensive use of strings[25] which facilitate the writing of a poem. This is because in programming strings are used to display human readable text. In this sense one can write proper English words or even sentences and include them in their code. Another way to

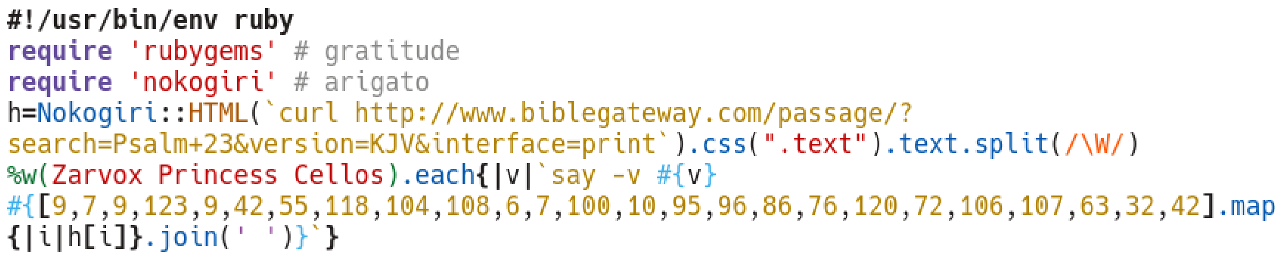
25 A string in computer programming is a sequence of characters, e.g. “this is a string”.

integrate proper English to a code poem is through comments[26] as Alvarez does with a poem written in HTML which mutates when the file is opened in a web browser (Alvarez, 2017, p. 35).

Echo and Narcissus is a live coding poetry opera which follows the live coding paradigm as much as possible. In this sense, writing a libretto that differs from its source code to the compiled version did not make much sense, as the audience should see the screen of the programmer, hence the source code. For this reason we did not chose this style of code poetry.

2.2.4. Poems Read by the Computer

In Stanford's Code Poetry Slam, the winning poem was written by Leslie Wu, called *Say 23*. In this poem Wu "typed 16 lines of computer code that were projected onto a screen while she simultaneously recited the code aloud. She then stopped speaking and ran the script, which prompted the computer program to read a stream of words from Psalm 23 out loud three times, each one in a different pre-recorded-computer voice"[27]. The script is shown in Figure 6.



```
#!/usr/bin/env ruby
require 'rubygems' # gratitude
require 'nokogiri' # arigato
h=Nokogiri::HTML(`curl http://www.biblegateway.com/passage/?
search=Psalm+23&version=KJV&interface=print`).css(".text").text.split(/\W/)
%w(Zarvox Princess Cellos).each{|v|`say -v #{v}
#{[9,7,9,123,9,42,55,118,104,108,6,7,100,10,95,96,86,76,120,72,106,107,63,32,42].map
{|i|h[i]}.join(' ')}`}
```

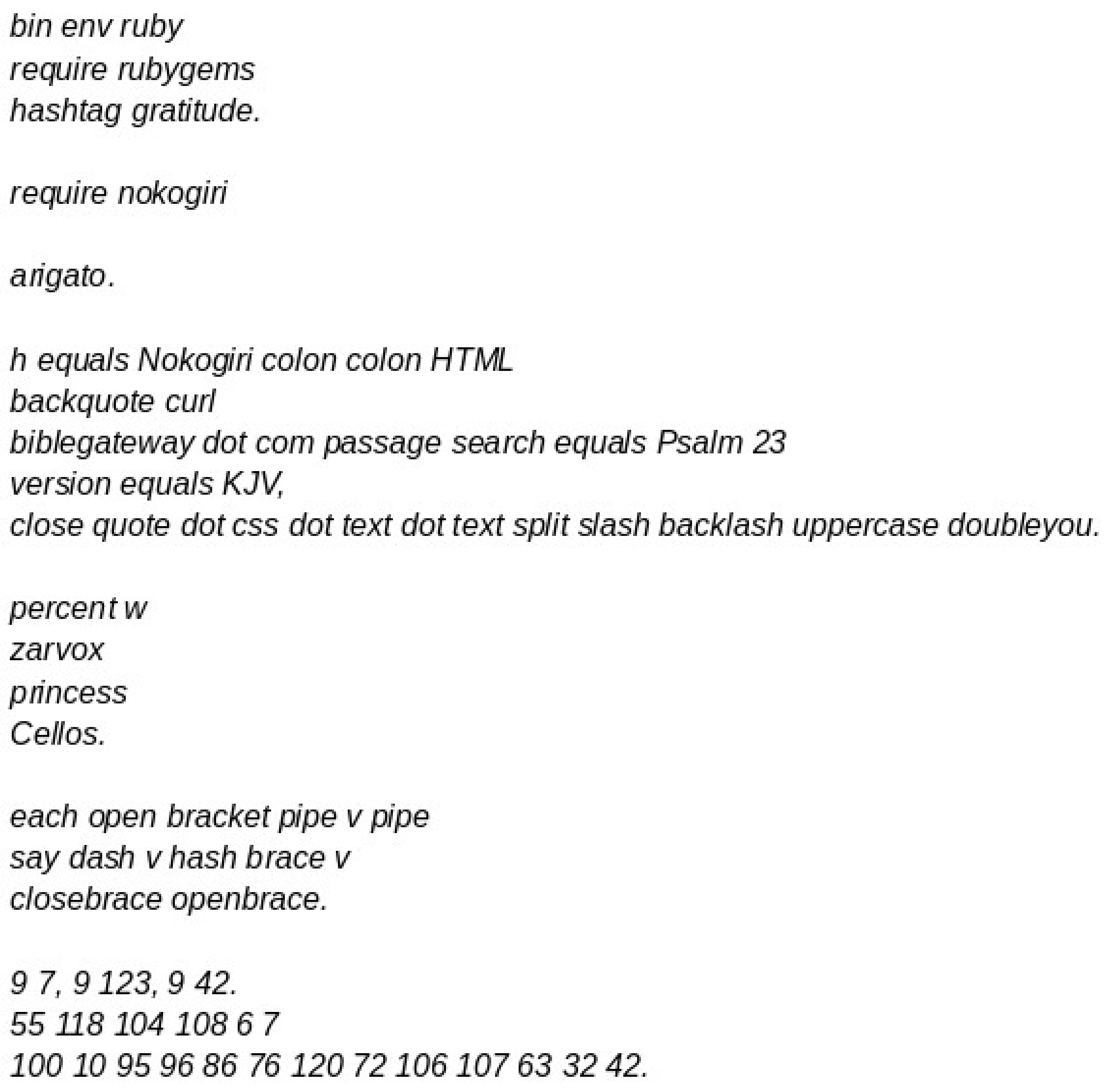
Figure 6: The poem Say 23 by Leslie Wu.

26 Comments are ignored by compilers and are used by programmers to explain chunks of code.
27 https://news.stanford.edu/news/2013/december/code-poetry-slam-122013.html

This poem is supposed to be read aloud, and this can be done in a few different ways, depending on how the reader will pronounce certain punctuation marks. One reading suggestion by the author is shown in Figure 7.

This approach is similar to the first two poems presented in this section, in that the source code and an actual resulting poem from the compiled version are two completely different poems. It is very different though in the use of natural language when recited. It pronounces every punctuation mark, something that is usually omitted in code poetry recitation.

Writing a libretto in a programming language already seemed to us very advanced for the field of operatic works. We wanted to write computer code that resembled a natural language, yet retained its own idiomatic syntax. In this respect, having to pronounce punctuation marks and other symbols was not our goal.



bin env ruby
require rubygems
hashtag gratitude.

require nokogiri

arigato.

h equals Nokogiri colon colon HTML
backquote curl
biblegateway dot com passage search equals Psalm 23
version equals KJV,
close quote dot css dot text dot text split slash backlash uppercase doubleyou.

percent w
zarvox
princess
Cellos.

each open bracket pipe v pipe
say dash v hash brace v
closebrace openbrace.

9 7, 9 123, 9 42.
55 118 104 108 6 7
100 10 95 96 86 76 120 72 106 107 63 32 42.

Figure 7: The reading suggestion of Say 23 by Leslie Wu.

2.2.5. A Collection of Code Poems

./code –poetry is a project by Daniel Holden and Chris Kerr which presents twelve code poems (Holden and Kerr, 2016). Each of these poems is written in a different programming language and when the poems are compiled and run, a visual representation of the poems is also presented. The languages used in these poems are Haskell, Ruby, Go, Piet,

Matlab, Python, Node.js, C, C++, Racket, C#, and Brainfuck. When one opens the pages of this project's website which contain the poems, the screen is split in two, with the right side projecting the source code of the poem, and the left side projecting its visual representation.

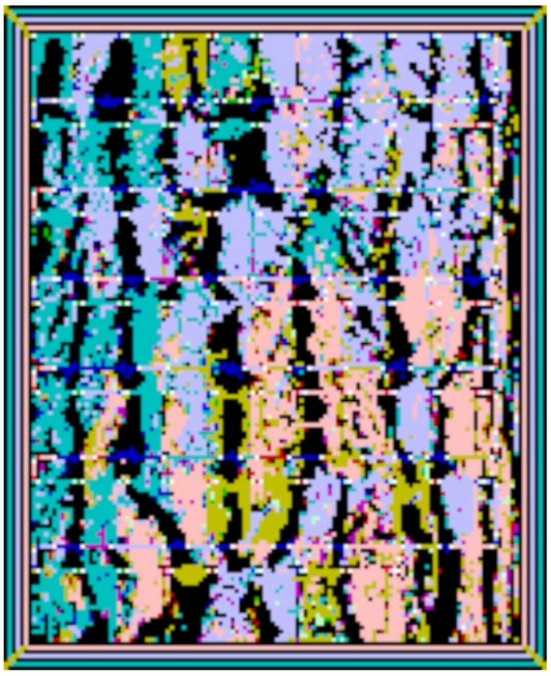


Figure 8: A Piet programming language program found in the ./code --poetry project.

Providing an analysis of each of the twelve poems found in this project is beyond the scope of this section. Nevertheless I will provide some information on one of them. Perhaps, the most noticeable programming language used in this project is Piet, which "is a programming language in which programs look like abstract paintings" (Morgan-Mar, 2018). In the page of the code of the poem written in Piet in this project, the right side of the screen, instead of projecting textual code, projects the image of Figure 8. Here, along with the resulting poem, a visual artefact occurs as well. The use of this programming language in code poetry demonstrates the variety of approaches to this artistic practice.

2.3. The Libretto: Its Conception and How It Was Written

Medea Electronique is an interdisciplinary art group. When we were asked to write a proposal for an opera, the myth of Echo and Narcissus was proposed by Poulou. The reasons for this choice are explained in the beginning of this chapter. While developing the proposal, the intention was to create a contemporary opera which would be attuned to our time, concerning both the aesthetics and the technology used. With the aesthetic outcome being the most important, we had to find a way to integrate technology into the opera as seamlessly as possible. Since computer programming is my main artistic tool, I proposed to the rest of the Medea members to adopt the code poetry paradigm and write the whole libretto in the Python programming language. As with most operas, the libretto would be displayed during performance for the audience to read as surtitles. This fact already borne resemblance to the practice of live coding, which led us to decide to combine code poetry with live coding and make the libretto in such a way that it would also serve as the audio engine of the entire opera, replacing the orchestra altogether. In this section I will analyse the libretto and the way it was written.

Python was the language of choice for writing the libretto in the code poetry paradigm. The original lyrics of the libretto were written by Poulou and I undertook the translation and adaptation to Python. All three, Poulou, Manousakis and myself, co-curated the entire process of the libretto, each at a different stage. The main reasons for choosing this language were the simplicity of its syntax, the use of English words instead of various punctuation symbols, the lack of having to declare the type of each variable in a program –

for example, whether a variable is a whole number or a decimal, known as integers and floats respectively – and the approach to the way various code structures are written. These are analysed in greater detail from subsection 2.3.3. onward. This was mainly a pragmatic choice, as, artistically, what was important to us was that we adopt the code poetry paradigm. We also wanted to write a libretto with computer code syntax, that would make sense to non-computer-literate people. In this context, Python was not only pragmatic, but also artistically appealing.

2.3.1. Other Candidate Languages

There are other languages with similar characteristics, with absence of variable type declaration and use of English words, such as Lua and Ruby. The main issue with these languages is integrating audio. LuaAV (Wakefield, Smith and Roberts, 2010) is a library for Lua to facilitate the creation of audio-visual works with this language. This library though seems to be rather outdated as it has not been updated since 2012. In the case of Ruby there is no updated stable library for audio with ruby-audio[28] appearing as one such library which has not been updated since 2008.

Another issue with these two languages is the keyword *end*, which has to be placed at the end of chunks of code with certain functionality, like loops, if-else structures, and others. In case one such chunk of code is included in another such chunk, then this keyword has to be written twice. This might seem as a small detail, but when writing a libretto of a duration of one hour, having to repeat this word very frequently can become problematic from the perspective of narrative, as this is likely to lead to a certain aesthetic with a centripetal force around this keyword. We wanted to free our aesthetic from pragmatic obstacles as much as possible, so we did not take these two languages into

28 https://github.com/fugalh/ruby-audio

account for the libretto. Lastly, Ruby lacks a simple-to-configure interpreter which would facilitate writing code live.

2.3.1.1. Natural Programming Languages

Following the code poetry paradigm, one could argue that using a natural programming language would be desirable, as these languages aim to import the expressiveness of a natural language (e.g. English) to a programming language (Pulido-Prieto and Juárez-Martínez, 2017). Aiming to create an idiomatic libretto, using a language like Python seemed a better choice, as it can be used in a way that brings it close to English, still it maintains its own syntax which is very different from the English one. This way we – Medea Electronique – incorporated Python's idiomatic syntax into the aesthetics of the libretto, a fact that gave a distinct character to the overall opera.

2.3.2. Example of Python Classes Written for the Libretto

Since the libretto of the opera combines code poetry with live coding, everything that was being typed during performance had to be projected for the audience to see. This condition ruled out most of the approaches to code poetry, rendering the most suitable the one where the actual code is the poem. Being a hybrid project – live coding and code poetry –, the actual libretto did not use the language intact, but a number of classes where written in order for the libretto to be functional. For example, the first line of the libretto reads:

```
I = Narcissus()
```

This line means nothing to the computer unless a class called `Narcissus` is defined in a file where the computer is directed to look for. The simplest definition of such a class, provided here as an example, is the following:

```python
class Narcissus():
     def __init__(self):
     pass
```

The entire `Narcissus` class consists of 126 lines of code and it is included in the appendix to this thesis. This class, as many other classes, contained both methods for triggering audio events, as well as passive methods that functioned only as libretto, not needing to trigger any sound. The declaration of this class was hidden from the audience and was never typed during performance. In this sense, the resulting libretto is not the *actual* source code, since there is a substantial body of code written beforehand behind the scenes that is not shown to the audience.

The whole libretto was written in Python, without using comments, and occasionally using single word strings. The requirement of having to define a set of classes in files not presented in the resulting text can be thought of as another approach to code poetry.

2.3.3. English Words Instead of Punctuation Marks

Python uses very few punctuation marks and mostly English words, even in places where other programming languages use punctuation marks. For example, the Boolean AND, in C, C++, and other languages is expressed with &&. In Python the same expression is written with the English word `and`. The same applies to expressions of other Boolean operators like OR, and NOT, where other programming languages use `||` and `!` and

Python uses the equivalent English words. Here is one example from the libretto of the opera, from Act III where Narcissus talks to his reflection on the lake:

```
You.yourself(‘ ‘ in ‘water ‘)
I.myself(‘ ‘ not in ‘water’)
```

Let me first explain the functionality of the word `in`. This is a Python keyword that tests if a value is present in a sequence, known as a membership test operation. For example the following line will return `True`:

```
‘r’ in ‘Narcissus’
```

because the substring `‘r’` is a member of the string `‘Narcissus’`. The two lines of the libretto presented here as an example consist of the classes `You` and `I`, which have the methods `yourself` and `myself` respectively. These methods take a Boolean argument, either `True` or `False`. The contents of the first parenthesis result in `True`, as the string `‘ ‘` is inside the string `‘water ‘` (note the white space). The contents of the second parenthesis also result in `True`, as `‘ ‘` is not inside the string `‘water’` (now the string has no white space), but the way it is written, using the keyword `not`, `False` is turned to `True`. This way we get that “You are in the water” as the argument passed to `yourself` is `True`, while “I am not in the water”, as the argument passed to `myself` is also `True`. The result of the code when run gives another dimension to the libretto as it emphasises certain lines by providing a `True` or `False` statement at the end, although this was evident to computer-literate audience members only. In this respect, the libretto bears similarities to code poems that mutate when compiled.

A similar example is the following two lines which occurs in Act II where Narcissus meets Echo:

```
interested = False
I.am(not interested)
```

In these two lines Narcissus states that he is not interested in her by setting the value `False` to `interested` and then calling `I.am()` with a `True` statement, which is the result of `not interested`, which is `not False`, so it is `True`.

2.3.4. Lack of Variable Type Declaration

In Python a variable needs not be defined as far as its type is concerned. This means that one can define a Boolean, integer, float, string, or other type, without needing to type any keyword before the name of the variable, like `bool`, `int`, `float`, `string`, etc. In Act I, Narcissus introduces Echo to the audience by saying:

```
She = Echo()
```

This line creates an object of the class `Echo`, named `She`. Later on in this Act, Narcissus says:

```
naive = True
thoughtless = True
She is naive and thoughtless
```

The three lines above need only the object `She` apart from native Python elements. A variable named `naive` is set to `True`, without needing to declare its type, in this case, a Boolean. Another variable named `thoughtless` is also set to `True`. Finally the last line can be compiled without producing any errors. Without needing to type the keyword `bool` at the beginning of the first two lines, we could write these lines in plain English, with only two equals signs as punctuation marks that are not part of a similar sentence in English. A deficiency of the last line is that it results to `False`, since `naive and thoughtless` results to `True`, but the `Echo` class (which `She` is an object of) is not a Boolean.

2.3.5. Loops and Statements in Python

Another example of Python being more English-like is the **`for`** loop. This loop, in C, is written as follows:

```
for (i = 0; i < a_limit; i++) {}
```

`i` is a variable used by convention in such loops. `a_limit` is supposed to be a variable which will hold the number of iterations of the loop. In Python a similar loop is expressed as follows:

```
for i in a_list;
```

`a_list` is supposed to be a Python list, for example [1, 2, 3, 4]. The functionality of the keyword **`in`** changes in the case of a **for** loop. Here it is used to iterate through a sequence. If we take the example of `a_list` being [1, 2, 3, 4], then **`in`** will iterate through the items of this list and assign each one at every iteration to the variable `i`. Such syntax, combined with various classes being defined beforehand, enabled us to write phrases like the following:

```
for i in I.myself(‘’):
      I.belong()
```

The lines above are part of the first Act where Narcissus expresses his self-sufficiency. With these lines we were able to express the narcissistic nature of the character, as he refers to himself three times in this short sentence. The object `I` of the class `Narcissus`, along with its internal methods `myself` and `belong`, is the only non-native Python element. In the source code of the opera, `Narcissus.myself(‘’)` returns an empty string, so the **`for`** loop above will actually not run as it tries to iterate over a string with no elements, so `I.belong()` is never called. The code though does not produce any errors.

The `with` statement in Python also served the libretto well as it was used in phrases similar to the following which enabled us to write the phrase where Narcissus says that he talks persistently with himself. This is in the third Act where he sees his reflection on the surface of a lake:

```
persistently = True
I.talk(persistently)
with I: I.myself()
```

2.3.6. How the Sound Processes Were Triggered

As mentioned in this section, there was a substantial body of code written prior to the performance that was imported to the main Python instance at the beginning of the performance. This body of code includes a number of Python classes with keyworded names, such as *Narcissus, Echo, Nymphs,* etc. These classes include various methods, some of which are passive, functioning solely as libretto lines, and some are active, calling certain functions that either trigger sound processes, or change parameters of these processes. For example, the *Narcissus* class calls a number of functions upon creation of an object of this class, like `createLowNonHarm()`, `lowNonHarmPlay()`, `createNarcFreqShift()`, `narcFreqShiftPlay()` – see the definition of the `Narcissus()` class in the *libretto_classes.py* file of the *Echo and Narcissus* Appendix, specifically the `__init__()` function in line 131. These functions, defined in the *functions.py* file – see Appendix – create sound processes that are defined in the *audio_classes.py* file – see Appendix – and add them to a virtual mixer that sends the audio to the PA. This way, when the following libretto line is typed and executed:

```
I = Narcissus()
```

all these functions are called and the generative sounds, as well as the processing of the voice of the baritone of the beginning of the first Act are triggered.

2.4. An Analysis of the Opera’s Electronic Music

In this section I will reflect on the electronic music of the opera – I will only refer to the singers concerning the processing of their voices – based on various concepts of the opera that were realised through the electronic music. A common concept for the greatest part of *Echo and Narcissus* is to give the impression of a timeless and spaceless place. We achieved this by not providing any visual reference to any specific time or space, but also through certain sonic gestures that were being introduced in various Acts of the opera. Namely these gestures are continuous sweeping drones of multiple oscillators detuned from a centre frequency. Two distinct such drones are among the main sonic elements of the opera, one paralleled with the coder presented as the poet of the opera, and the other paralleled with Narcissus himself. These sonic gestures do not provide a clear melody or rhythm – although the frequency sweep has a steady pulse – and seem to fit the timeless / spaceless concept well. One more sound that reinforced this concept is a bell-like sound that is triggered with every key stroke at the introductory Act of the opera – the Ode. This sound alternates between fast-attack / slow-decay and slow-attack / fast-decay, with the former giving the impression of a sound playing in forward direction, while the latter simulates backward playback. This forward / backward alternation aims to create a confused sense of time.

Narcissus introduces himself in Act I, right after the Ode. His very first line is “`I = Narcissus()`” and it is sung in an imposing manner. This condition is augmented by the processing of the voice of the baritone which is fed to a frequency shifting effect. This effect shifts frequencies linearly distorting the harmonic relationship between overtones, resulting in a non-harmonic sound. The effect mainly shifts the baritone's voice to lower harmonics, creating a bass and distorted voice. This effect forms Narcissus' sonic character for the first two Acts. In the first Act we are also introduced to Echo and her sonic character. Echo was cursed to be unable to speak words of her own. She could only repeat the last words of her interlocutor, thus becoming a real echo. In order to simulate natural echo but also give it a more personal characteristic, I created a delay effect that alternated between forward and backward playback of the original signal. The soprano impersonating Echo is deliberately singing incomprehensible words and the alternation between forward and backward delay augments this non-sense speaking / singing.

Sound spatialisation is an important aspect of the character of Echo as the intention was to partly simulate real echo. The delayed copies of the voice of the soprano are being diffused randomly to the speakers of the surround sound system of the main stage on the Onassis Cultural Centre. This spatialisation gives to the audience the impression of them witnessing the real echo of the soprano's voice reflecting on various surfaces of an imaginary landscape.

Another sonic element that aims at localisation – thus breaking the spaceless condition prevailing for most of the opera – is a raindrop-like sound introduced in Act II. This sound is also diffused in the surround system of the venue. It aims to support the scene where Narcissus goes hunting into the forest. The video projection in this Act shows a foggy forest scenery and the raindrop sounds augment the impression of moisture and aim to give the audience the impression of being surrounded by trees in a forest. A bandpass filtered noise generator accompanies this sound where the cut-off frequency of the filter sweeps slowly back and forth, as with most sonic elements of the opera. The Q[29]

29 The Q refers to the Q factor, or quality factor, in a filter, a factor related to the bandwidth, where the higher the Q, the narrower the bandwidth of a filter, and vice versa.

of the filter is quite high so we can hear a distinct pitch, but at the same time there is also an underlying broader noisy spectrum. This combination of sounds serves as an abstract representation of wind howling through the forest's trees.

Act III celebrates the voice extending paradigm. In this Act Narcissus sees his reflection on the surface of the lake where he went to drink water while hunting. Being enchanted by his own reflection he falls in love and converses with it until he tries to kiss his reflection and eventually drowns. During this conversation with himself, the voice of the baritone is being time-stretched whenever he changes pitch, and the time-stretched voice is being heard together with the unprocessed voice of the singer. This processed sound lasts eight seconds, so if the singer changes pitch multiple times within eight seconds, more time-stretched copies of his voice are sounding simultaneously. This way Narcissus creates a choir with his voice alone, an effect that reflects his narcissistic nature. In this Act some generative sounds from previous Acts are heard at certain points, but these are subtle and minimal. The main sound for the whole Act is the voice of the baritone with its time-stretched process.

After Narcissus drowns in the lake, Echo mourns for his loss in Act IV which is a solo for the soprano. In this Act, Echo does not speak incomprehensibly but speaks the words of the Nymphs, who up to that moment were presented in the videos of the Intermedia, in between the Acts. Their words are judgmental for both Echo and Narcissus, so Echo's mourning cannot be expressed with words. There is no delay in this process as the focus has shifted away from her curse and the echo effect. A series of sine wave oscillators that resonate in a metallic, gong-like sound, based on the frequency of the singer's voice are heard together with the unprocessed signal. A white noise burst is heard when the singer crosses an amplitude threshold. As the Act progresses, the effect of Echo's voice is distorted more and more, becoming incomprehensible, and the white noise bursts are filtered through a lowpass filter whose cut-off frequency drops lower each time, giving more space to the main, now very distorted, processed voice of the singer. The constant crescendo of this act also serves to express the growing grief of Echo.

2.5. Conclusions

Echo and Narcissus is a mixed media opera attuned to the technology and aesthetics of its time. It utilised the practice of live coding which, at the time of writing, is a well-established practice in the digital arts. It also integrated the less common practice of code poetry with several new techniques advancing new ways of writing poems with computer code. In the course of nine months, a one-hour long opera was prepared with many different elements, including singing, choreography, live electronics, code poetry, live visuals, as well as fixed media. Time constraints were strict and did now allow for experimentation once the system for the code poetry libretto was functional, so an issue that arose during development, namely the deterministic nature of the libretto, was not addressed.

Due to this issue, no variation of the text was possible as it would have produced an error in Python's interpreter. In previous attempts in live coding poetry[30] I used methods with variable arguments in order to introduce variations in the resulting poem. I could then call the same method of a given class with a variety of arguments. Another possible solution to this issue could have been sentiment analysis, but that would have required strings with whole sentences, which was not desirable.

Despite of the issues above, *Echo and Narcissus* resulted in a four-act opera that was presented on April 19th 2018 at the main stage of the Onassis Cultural Centre in

30 https://vimeo.com/187892132

Athens, one of the most important artistic venues in Greece. The opera was successful in that it was well received both by the audience and the producers.

2.5.1. Contribution to Knowledge

The contribution to knowledge of *Echo and Narcissus* lies mainly in the fact that this work combines code poetry with live coding in a novel way. Up to this point, to the best of my knowledge, there have been no similar examples, where the narrative of a libretto is realised through computer code. *Echo and Narcissus* provides a concrete example of how the two practices of code poetry and live coding can be used in an operatic context. In this sense, artists and researchers working in the medium can indeed study the source code of the opera from a technical point of view, thus laying the ground for further experimentation and different approaches to these two practices.

In addition to promoting code poetry and live coding within the opera, *Echo and Narcissus* uses extensively the Pyo module for DSP in Python. This module remains rather obscure with only a few literature refrences and limited use in the field of computer music. The source code of the opera shows how several available classes of the Pyo module can be used for a variety of DSP algorithms, from Unit Generators to filters and effects.

Chapter Three

Data Mining / Live Scoring, A Live Algorithmic Composition Based on Twitter

Data Mining / Live Scoring is a live algorithmic composition for acoustic ensemble that combines Data Sonification with audience input. Data Sonification occurs by using the Twitter social platform to retrieve tweets that are transformed into music scores during the performance. By using Twitter, the audience can – and is encouraged to – participate by writing tweets during the performance. This project aimed to create a composition that can be perceived as a soundtrack for Twitter. Loosely related to soundtrack for film, which augments the experience of the spectator (Thompson, Russo and Sinclair, 1994), the resulting music aims to add an emotional dimension to the tweets. Based on sentiment analysis of each tweet, the algorithm produces musical sequences that reflect this sentiment, based on how positive or negative a tweet is.

This work includes a number of music creation techniques and approaches, including audience input, Data Sonification, input from Twitter, sonification from text analysis, Live Scoring, and Computer Aided Composition. Even though a substantial body of work exists in each of these fields, what is unique with *Data Mining / Live Scoring* is their combination and the way these techniques were applied. It was conceived by myself, commissioned by the Onassis Cultural Centre, and was created in collaboration with the composer Nicoleta Chatzopoulou[31] for the ARTéfacts Ensemble[32]. It was presented at the Onassis Cultural Centre, in Athens, on the 3rd and 4th of April, 2019[33].

31 https://www.nicoletachatzopoulou.com/
32 http://artefactsensemble.gr/
33 https://www.onassis.org/whats-on/data-mining-live-scoring

The concept of *Data Mining / Live Scoring* was born by the desire to create a work for acoustic ensemble that would include audience input and at the same time sonify data. Before I started any of the realisation processes, I had to first clarify how the audience would participate, what data would be sonified, and in what way. According to Hödl et al., audience appreciation of their participation in a concert, in cases where this is achieved through a controller type interface, varies greatly (Hödl, Kayali and Fitzpatrick, 2012). On the other hand, Mikalauskas et al. provide parts of interviews of improvising actors, where one reads that in improvised theatre “the audience loves nothing more than to hear themselves, centre stage” (Mikalauskas et al., 2018, p. 662). Personal experience from a *Boom Chicago* show in Amsterdam[34], an improvised comedy show, falls in line with Mikalauskas et al., where most of the audience members kept on shouting suggestions for the development of a sketch when they were prompted to. Based on the above literature and my personal experience, I was convinced that the participation of the audience would augment their engagement in the performance.

Besides engagement, audience participation provides a random factor to a performance (Mikalauskas et al., 2018), something that I find attractive and I often pursue in my own work. In addition to the participation of people present, one can include input from other sources that are not directly related to the actual performance. Such inclusions can augment the randomness of the input. Bearing all of the above in mind, *Data Mining / Live Scoring* used the Twitter streaming Application Programming Interface (API) in order to retrieve tweets during the performance which served as the sole input for the creation of the scores for the musicians of the acoustic ensemble. Twitter can provide a massive database with an immense volume of information, both from users who are part of the audience, and users who are not physically present and are not aware of the performance

34 https://boomchicago.nl/

or that their tweets are being analysed in this context. Something to consider is that audience participation can be a delicate matter, depending on whether this is done in an anonymous or an identified manner. Identified participation requires the consent of the participants, as it is very likely to make the participant visible or audible to the rest of the audience. Writing tweets allows involvement only through the personal decision of each participant, but without exposing them, as during the performance, the tweets were being projected without displaying any information about the user – name, location, etc. It was up to the participants whether they exposed personal information or not. Finally, using Twitter for audience participation implied the use of the smartphone as the interface. This served my purposes well as I did not have to worry about supplying any sort of dedicated interface – DIY or commercial.

This work is split among three main strands, programming, composing, and performing. Even though I compose electronic music myself, I felt the need to collaborate with a composer for two reasons. First, I do not consider myself a composer of acoustic music for ensemble, and I strongly believed that a composer as a co-creator, with deeper experience in acoustic music composition for ensemble would be a better fit for carrying out the acoustic compositional process of this work. The second reason was that I wanted to focus on the programming aspect, as this work was very demanding in this respect. If I were to undertake the compositional part as well, I believe that I would not have been able to complete any of the two parts at a satisfactory level. Nevertheless, the collaboration between Chatzopoulou and me was very close throughout the entire development process of the work. The work by Chatzopoulou – analysed in greater detail in the “Music Library” section of this chapter – was led by her aesthetics and intuition. Still, it was developed based on the various programming concepts of this work, giving prominence to the overarching idea, that of creating music to augment the overall emotion of the tweets.

3.1. Audience Input and Data Sonification in Related Works

In the context of incorporating audience input through the phone – either landline or mobile –, one can find early examples, including *Public Supply* by Max Neuhaus (Neuhaus, 2000) and *Dialtones* by Golan Levin (Hödl et al., 2017, p. 28). In *Public Supply*, which was performed in 1966, people could call at the radio station where this work was broadcasted, and participate. *Dialtones* was performed in 2001 and it used mobile phone ring tones from the audience as the only sound element of the performance. In more recent works, the phone – in this case, the smartphone – is still the most common approach to audience participation (Gimenes, Largeron and Miranda, 2016; Zhang, Wu and Barthet, 2016; Egozy and Lee, 2018; Xambó and Roma, 2020). Other methods incorporate EEG brain wave recorders (Eaton, Jin and Miranda, 2014) or other DIY devices like armbands (Turchet and Barthet, 2018). Among these projects, Egozy et al., Turchet et al., Eaton et al., and Gimenes et al. combine electronic elements produced by the audience with acoustic instruments. In this context, they bear similarities to *Data Mining / Live Scoring*, although the latter includes acoustic instruments only, without any electronics. A project that includes audience input and acoustic instruments only is “Tin Men and the Telephone” (Tin Men and the Telephone, 2020). This is an improvisational band, stylistically closer to jazz. Through a bespoke mobile app, the audience can influence the performance live. There are three more works that are relevant here and are mentioned in the “Live Scoring Works” section of this chapter. One of them includes audience input as well. The drawing lines between the various fields – audience input, Data Sonification, Live Scoring – are somewhat blurred, and for this reason I chose to analyse these additional works in the “Live Scoring Works” section.

Data Sonification is a practice with a very wide range of techniques and processes for mapping data to sounds. The first experiments took place in the 50s and 60s. Alvin

Lucier created one of the first works which are considered to sonify data (Straebel and Thoben, 2014). *Music for Solo Performer* was performed in 1965 and it utilised the brain waves of the performer to control a number of percussive instruments. *Pythoprakta* by Iannis Xenakis was premiered in 1957. It used the motion of Brownian particles in order to define a sequence of glissandi for string instruments (McKell, 2016, p. 524). In contemporary music and sound art, many different types of data sonification can be found. Some examples include sonification of soil elements (Suchánek, 2020), georeferenced data (Park et al., 2010), earthquake data (Lindborg, 2017), web tracking (Hutchins et al., 2014), the movement of fish (Mercer-Taylor and Altosaar, 2015), Nuclear Magnetic Resonance data (Morawitz, 2016), or data from ongoing experiments at CERN (Cherston et al., 2016). Along with the diversity of the source of the data, the sonification techniques used also vary, as the relationship between the data and the resulting sound is highly subjective (Schoon and Dombois, 2009). For this reason the works mentioned here serve to give a more generic context on Data Sonification, as they are not related to *Data Mining / Live Scoring* in a straightforward manner.

3.2. Twitter Input in Related Works

Twitter has been used in various music and sound projects, also with different approaches to which data are used, how they are analysed, and how this analysis is mapped to sound. Most of these projects use the actual text of the tweets to associate them to the resulting music/sound, but other projects utilise different types of information that can be derived from Twitter.

3.2.1. Projects Utilising Tweets

Examples of projects whose sonification is based on the text of the tweets include *TweetDreams* (Dahl, Herrera and Wilkerson, 2011), *Tweet Harp* (Endo, Moriyama and Kuhara, 2012), *MMODM* (Tome et al., 2015), and *Affective States* (Ash, 2012). From all these examples, the only project that incorporates sentiment analysis of the tweets is *Affective States* by Ash. This project analysed the sentiment of tweets on trending musicians on Twitter and used the outcomes of this analysis to control an additive synthesis program, where sentiment controlled the frequencies and the number of harmonics. The sentiment analysis of this project is based on “a 2D space with quality running along the X-axis […] and perceived energy along the Y-axis” (Ash, 2012, p. 258), in contrast to the analysis applied in *Data Mining / Live Scoring* which is a linear analysis based on sentiment valence of words and phrases.

MMODM is another project that applies text analysis, but in a completely different manner. *MMODM* stands for Massively Multiplayer Online Drum Machine. Users can tweet using the *#mmodm* hashtag, but they must include a specific syntax in their tweet, and place that within square brackets for the algorithm to trace it. An example is the line below:

riff on this [a-a-a-abc-cc] with me on #mmodm

The letters inside the square brackets symbolise MIDI instruments and the hyphens symbolise rests. The association between text and sound is very direct and clear, but this project removes the actual meaning of the natural language that is used in the Twitter social platform, transforming it into a controller for a drum machine.

Tweet Harp is a sound installation with an Arduino-based laser harp[35] where visitors can trigger the recitation of tweets with Text-To-Speech synthesis. This project does not

35 The laser harp is an electronic instrument with laser beams where sound is produced when the performer blocks them.

employ any kind of text analysis. Another project that does not analyse text is *TweetDreams*, a live performance where audience members are prompted to tweet based on certain keywords. The authors found sentiment analysis interesting, but quite challenging, and chose a different technique (Dahl, Herrera and Wilkerson, 2011, p. 273). Tweets were grouped according to similarity, but the authors provided no details on how similarity is interpreted. Similar tweets were grouped in tree structures and matched to randomly generated melodies and mutations of those melodies for the branches of the trees. However, there is no clear association between the text and the resulting sound. One characteristic of this project is that tweets with a local search term used to distinguish tweets by audience members from tweets by users who were not present were given prominence (Dahl, Herrera and Wilkerson, 2011, p. 273). In *Data Mining / Live Scoring* no such prominence was given.

3.2.2. Projects Using Tweet Data Other Than Text

Apart from the text, tweets contain other types of user data. These data can include geolocation, number of followers and any profile information that users have given their consent to make visible. This fact has been exploited by some musical/sound projects. Two examples are *I Hear NY4D* (Boren et al., 2014) and *Tweetscapes* (Hermann et al., 2012). *I Hear NY4D* is an auditory display platform which uses ambisonic recordings[36] from New York City to create an augmented reality environment based on the user's location. This project does use detection of keywords, but it does not apply any further text analysis. Its approach attempts to associate tweets with sound based on frequency of keywords or frequency of tweets sent from within a given radius of a focal point. Quoting

36 Ambisonic recordings are full-sphere surround sound recordings that cover sound sources from above and below, in addition to the horizontal plane.

the author, “a user may be interested in hearing a representation of the rate of Tweets coming from or mentioning Central Park” (Boren et al., 2014, p. 3). The relationship between sound and text is relevant in the context of field recording and textual mentions of the location of a recording. As no further text analysis of the tweets is made, this relation is likely to be taken out of context.

Tweetscapes is a radio broadcasting project that incorporates granular synthesis along with other techniques processing various audio samples. This project too uses geolocation data around a focal point, but it also uses other information, like the number of followers of the user who tweets and the longitude of the user’s position. Like most projects mentioned in this section, no association between the meaning of the text of the tweet and the resulting sound is made. Nevertheless, this project distinguishes different categories of tweets, based on whether a tweet is a reply to another tweet or not, whether a hashtag is used or not, and tweets around the most trending topics (Hermann et al., 2012, p. 114). These categories are mapped to different groups of synthesis techniques, thus establishing a connection between the category of the tweet and the resulting sound. *Data Mining / Live Scoring* uses trending hashtags too, but the actual mapping depends on the sentiment analysis of the tweets and not the hashtags.

3.2.3. Projects Applying Text Analysis Outside of Twitter

Alt et al. realised a project that does not use Twitter but does employ text analysis to generate audio (Alt et al., 2010). This project aims to create notification sounds for text messaging that denote the intention of the message to the receiver. The realised algorithm analyses text by trying to detect keywords, punctuation marks, and emoticons for each different sentence of the message. The resulting audio is based on major and minor pentatonic scales that create a sequence of melodies for all the sentences of the message. This project is similar to *Data Mining / Live Scoring* in that it tries to express the mood of a message through musical phrases. It is different in that it does not aim to create complete music compositions. Moreover, it is confined to only two generic types of music scales, without aiming to convey a certain compositional style. It has an advantage over *Data Mining / Live Scoring* in that it also takes emoticons and punctuation marks into account in order to analyse the sentiment of the text.

3.3. Generative Compositions

Numerous works apply various techniques to create compositions that are generated algorithmically. *sound/tracks* is a real-time synaesthetic sonification system based on the view from the window of a train (Knees et al., 2008). This is achieved by analysing the images of a camera and splitting it horizontally to four regions, mapped to four octaves of the piano. The pixels of the images are analysed vertically, and for each horizontal zone, the colour of the analysis is mapped to notes according to the synaesthetic scale by Alexander Scriabin. Koutsomichalis and Gambäck produced a sound installation that created audio mashups and synthetic soundscapes with audio downloaded from the Internet (Koutsomichalis et al., 2018). Their system performed onset analysis and spectral feature extraction on the downloaded audio, together with temporal scheduling and spatialising.

AutoFoley is a project for the generation of Foley audio tracks (Ghose, 2020). This system analyses the image and the sound of movie clips and generates audio that matches the respective scene, like the gallop of a horse or the strokes on a computer keyboard. Another generative system for audio is *Music Paste* (Lin et al., 2009). This system chooses music clips from an audio collection and creates transition segments to compensate for tempo or loudness changes, or other parameters, to create a seamless concatenation of them.

3.4. Live Scoring

Since the initial concept of *Data Mining / Live Scoring* was to create music for an acoustic ensemble, Live Scoring was a vital technique. In this section I will provide a critical overview of several software packages for this purpose. A music score can have many forms, from graphic to verbal, and can even be in the form of a database for digital instruments (Puckette, 2004). *Data Mining / Live Scoring* focuses on the creation of music scores in the Western notation format, so my review will be limited to software that utilises this notation.

3.4.1. Live Scoring Software

There are various tools for creating scores in real-time. These include INScore (Fober, Orlarey and Letz, 2012), the family of the bach objects for the Max environment (Agostini and Ghisi, 2013), Maxscore - another Max external object (Hajdu and Didkovsky, 2012) which has evolved to a full Max and Ableton Live package[37] -, and Open Music (OM) (Bresson, 2014). Offline – non real-time – software include the Guido Notation Format (Hoos et al., 1998), the Guido Engine, based on the Guido Notation Format (Daudin et al., 2009), and Lilypond. The offline software are mentioned due to their textual nature that makes it possible to write scripts for the automatic realisation of scores via such a system.

37 http://www.computermusicnotation.com/#Maxscore

A clear distinction between real-time and offline scoring must be made. A real-time scoring system is capable of rendering a score in no – perceived – time[38]. The graphic rendering happens within a programming environment as in the case of the bach family of Max objects or OM, which means that no universal file type, like PDF or PNG, is exported during the process. A characteristic of these systems is that an already existing graphic representation of a score can change by inputting new data to the graphic objects that render the score – or by sending Open Sound Control (OSC)[39] messages in the case of INScore –, whereas in the case of offline systems, every change to a score results in exporting a new file. Another advantage of some real-time systems is interactivity with existing scores. This interactivity can be the highlighting of a note with colour or other techniques for augmenting notation.

Gemnotes is another Live Scoring system (Kelly, 2011) in the form of a set of objects and abstractions for Pd Gem. This is rather CPU intensive and seems outdated with the last update having taken place in 2012. The *notes* external object for Pd creates scores using Lilypond and it is very flexible in the variety of notations it can produce. By default *notes* produces PDF files and opens them as soon as they are created with the system's default PDF reader, or it just creates the Lilypond file without rendering the score. However, none of these scenarios fit the needs of *Data Mining / Live Scoring*, as we will see further below in the sections "The Algorithm" and "The Score Rendering System", so this Pd object was not an option.

38 Computational processes obviously take time, but from the perspective of human perception they are instantaneous.
39 http://opensoundcontrol.org/

3.5. Computer Aided Composition

Data Mining / Live Scoring comes under the definition of Computer Aided Composition (CAC) as it incorporates audience input together with Data Sonification. For this reason, in this section I will provide context on CAC in general, and on CAC software.

3.5.1. Definition of CAC

Similar to Live Scoring, CAC techniques can be roughly categorised in two sections, offline and real-time. Offline is used when the desirable outcome is a score to be performed at a different time than when it was created, or a score to be further edited in a dedicated notation software. This can include acoustic pieces which need to be practiced before they are performed for an audience. In this case the composer uses various methods where usually a computer program is fed a set of rules and the outcome results in a set of instructions, audio or MIDI data, a partial or full score, and so on. Real-time CAC, on the other hand, is mainly used in electronic music. This is due to the nature of real-time which may translate into a very high rate and density of information for a performer to sight-read (each element must become audible as soon as it is composed). Real-time CAC can be possibly used in combination with human performers, or with a computer that simulates acoustic sounds via MIDI instruments. A self-playing piano like the Yamaha Disklavier could also be used with a real-time CAC, with no human performer involved.

The field of CAC can overlap with Live Scoring, as in the case of a CAC that produces a score close to real-time, but gives enough time to the performers to sight-read it. Bearing in mind the fact that a score-reading performer reads from a few notes to one or two bars ahead of the bar he/she is currently playing, the system should render chunks of at least a few bars of music at a time. Since a real-time CAC renders music elements –

notes, rest, etc. – one by one, this approach cannot be entirely real-time. On the other hand, if such a composition is to be played live while it is being created, there are strict constraints that make this CAC not an offline one. *Data Mining / Live Scoring* is such a CAC.

3.5.2. CAC software

CAC and Live Scoring software usually coincide. The Max external library bach and OM are both Live Scoring and CAC software. The bach library can take advantage of Max's generative capabilities, which when combined with the various bach objects can provide a CAC environment. OM on the other hand was initially built as a CAC software and its true Live Scoring capabilities were added later on (Bresson, 2014).

An older example of this type of approach in a system is the UPIC[40] machine, developed by Iannis Xenakis in the 1970s (Marino, Serra and Raczinski, 1993). The UPIC was a combination of hardware and software where the composer would use a tablet connected to a computer on which he/she could "draw" sound and hear the results that were created by the software. Even though the original system is of its time, it is still considered innovative with software like UPISketch (Bourotte and Kanach, 2019), IanniX (Coduys and Ferry, 2004), and Synthésie[41] being derived from or inspired by it.

A CAC system with a more narrowed focus is materialssoundmusic, which is based on Data Sonification (Buongiorno Nardelli, 2015). This system is based on Python, Max and Ableton Live and is designed to sonify data from the AFLOWLIB scientific repository[42]. This system bears similarities to *Data Mining / Live Scoring* as it combines CAC with Data

40 UPIC stands for Unité Polyagogique Informatique CEMAMu (Centre d'Etudes de Mathematique et Automatique Musicales)
41 https://synesthesie.buzzinglight.com/
42 http://aflowlib.org/

Sonification, but also due to the use of Python, though in a completely different context. It does not provide a score system but outputs MIDI note on and note off messages. It provides freedom to the user as to the scales or note patterns the data will be mapped to. The MIDI messages can then be used in any desired way.

3.6. Live Scoring and CAC Works

Live Scoring and CAC tend to inter-relate, especially in the case of Live Scoring works. For this reason, works from either category are merged in this section. The works noted here aim to provide context on Live Scoring and CAC from an artistic point of view.

3.6.1. CAC Works

The first score recognised as being composed with a computer is the *Illiac Suite* by L. M. Hiller and L. M. Isaacson (Funk, 2018). The two composers programmed the Illiac computer to produce pitches from random integers based on rules and operations they set themselves. This piece was premiered in 1956. ST/10-1, 080262 by Iannis Xenakis is a composition for acoustic ensemble realised with calculations made in the IBM-7090 computer that was premiered in 1962 (Xenakis, 1972). Other early works date back to the early 80s, such as *Protocol* (Ames, 1981), *Amazonia* and *Aus den Tiefen den Rheines* (Mikulska, 1981). These works are offline CACs and therefore bear limited direct resemblance to *Data Mining / Live Scoring*, but they do all share with it a rule based system for score creation, which is at the heart of any CAC.

3.6.2. Live Scoring Works

Live Scoring is from its nature a CAC derivative as it is the algorithm which generates the musical score, hence the actual composition. For this reason the Live Scoring works are mentioned in the same section with the CAC works. *Comprovisador* (Louzeiro, 2018) is a Live Scoring / CAC work for soloist and ensemble where the system creates a score by “listening” to a soloist and decoding his/her playing. The resulting score is displayed and animated for an ensemble which sight-reads it in order to accompany the soloist. This is very similar to *Data Mining / Live Scoring* since it creates scores for an ensemble in real-time, but the input to the composing algorithm is very different.

Flood Tide by John Eacott sonifies tidal flow (Eacott, 2011) by using a sensor which reads the speed of tidal water. The sensor is sampled every five seconds and its output is fed to a series of processes, each contributing to a different compositional aspect. This work aimed at long performances, up to six hours, with musicians changing shifts in order to rest. The resulting composition is rather static as the sensor values apparently change slowly, and they are sampled at a low frequency.

No Clergy is a Live Scoring composition by Kevin C. Baird utilising Python and Lilypond incorporating audience input (Baird, 2005). The audience influences the evolution of the piece while it is being performed. This happens by typing values for note duration, dynamics, etc. in a Common Gateway Interface (CGI) form on a web browser (Baird, 2005, p. 241). This work has many similarities to *Data Mining / Live Scoring*, especially concerning the software used. The difference is that in *No Clergy* the audience has much more direct control on the resulting composition as audience members input values which are explicitly mapped to discrete musical elements known to them. In *Data Mining / Live Scoring* the audience is not aware of how their tweets precisely affected the music.

Livecell is an interactive generative composition and real-time scoring application that utilises Cellular Automata (CA) that are used for musification and the creation of music scores (Ash and Stavropoulos, 2011). The user interface provides a way to create or destroy cells, supervise the shape and direction of the CA evolution, and provide information concerning pitch set classes and CA rule sets that determine the behaviour of the cells. The Live Scoring part is realised with the MaxScore object for the Max environment. This application is designed for a string quartet.

3.7. Sonification Technique

In this section I will provide an analysis of the way the tweets were sonified. As mentioned in paragraph 3.2.1., very few projects using Twitter apply sentiment analysis. This technique though, can provide valuable information on the texts. For example, sentiment analysis is widely used on product reviews in large volumes where the opinions of customers matter (Liu and Zhang, 2012). The aim of *Data Mining / Live Scoring* was to create music that would relate to the text that was used as its source. Having this goal, sentiment analysis was regarded as the best approach to map tweets to the resulting music.

3.7.1. Sentiment Analysis

There are two main techniques applied in sentiment analysis. One is based on lexicons with various keywords and expressions often used in the context of the text to be analysed. This is the semantic approach. The other technique incorporates Machine Learning with various supervised learning algorithms, including Support Vector Machines

(SVM), Naive-Bayes, and K-Nearest Neighbor (KNN), which is the learning-based approach (Poonguzhali et al., 2018). For *Data Mining / Live Scoring* the semantic approach was chosen. The lexicon that was used is an Opposing Polarities Phrases (OPP) lexicon which consists of 602 single words and 576 phrases (Kiritchenko and Mohammad, 2016). It provides a valence value between -1 for 100% negative to +1 for 100% positive words or phrases derived from Twitter. The following line is an example of this lexicon:

```
seriously great	1	R+A	17
```

"R+A" is the Part-Of-Speech (POS) pattern, and 17 is the frequency of the term. The only value used was the second field – in the example above, number 1 – which is the valence of the text – in this case "seriously great". Every word of every tweet downloaded during the performance was tested against this lexicon. When a phrase or a single word of a tweet matched an entry in the lexicon, the valence value was accumulated to a variable that held the overall valence for a tweet. Once the whole tweet had been analysed the accumulated value was normalised to a range between -1 and 1.

The algorithm would download up to thirty tweets in order to create a block of music. The valence range was mapped to five sentiments, very negative – negative – neutral – positive – very positive. The accumulated valence of the downloaded tweets was fed to an SVM classifier, along with the density of the tweets – density was determined according to the number of tweets compared to the time it took to download them, thus expressing the traffic on Twitter in a given time window. The classifier had been trained with sets of valence and density, each mapped to one of nine predefined music form structures. Each time the classifier was fed a new set it would predict the next form. These forms contained information about harmony, orchestration, melody, rhythm, dynamics, and techniques and were composed by Chatzopoulou. Her intention was to write these form structures in an intuitive way based on her aesthetics, but in such a way that reflected the sentiment they were mapped to.

3.7.2. Prosody Analysis

Another element that was analysed, that seems to not have been applied to another music project, was the prosody of the tweets. This was done with the Prosodic Python module[43]. This module applies prosody analysis on text. Since prosody "explores the impact of the way an utterance is said – its tune, emphasis, and rhythm" (Carlson, 2009, p. 1188), some ambiguity should be expected when analysing written text. However, the developers of the module state that it produces a convincing percentage of successful analysis when compared to analysis of text entries by literate humans.

This analysis occasionally served to determine the rhythmic structure. Whether the prosody was used for the rhythm or not, was hard-coded into the form structures mentioned in the previous section. In case the prosody was not used, pre-defined rhythmic tree structures were used that will be further described in the next section. When the prosody determined the rhythm, the number of strong syllables was used to determine the number of rhythmic elements for a phrase, be it a note or a rest. These phrases were predefined and stored in a table indexed according to their number of elements. The algorithm would randomly choose a phrase with so many elements as the strong syllables of a tweet. The idea behind this technique was to provide a musical way to recite the tweets along with the music. The tweets were being projected during the performance by a patch made in Isadora[44] by ARTéfacts member Louizos Alsanidis, but there was no

43 https://pypi.org/project/prosodic/
44 https://troikatronix.com/

recitation of them. Nevertheless, this approach was used to give variety to the resulting composition and strengthened the connection between the tweets and the music.

3.8. The Music Library

The form structures mentioned in the “Sentiment Analysis” section were part of a music library written for this project by Chatzopoulou. This library was the only source of musical elements for the algorithm to create the scores. The nine distinct form structures contained hard-coded information about rhythm, tempo, orchestration, dynamics, and various techniques for each instrument. The library also contained five different melodic scales, one for each subdivision of the valence range. These scales were independent of the form structures and were determined by the overall sentiment of the tweets alone.

3.8.1. The Melodic Scales

The scales were formed in tree structures with one root with two branches, where each branch had another two branches, resulting in three levels in total. An example is shown in Figure 9.

```
      B
     / \
    C#  D
   / \  / \
  F# A  E# G#
```

Figure 9: An example of a scale tree.

The scales were separated in three categories, “solo”, “accompaniment”, and “mixed”. One scale for each category was written for each sentiment subdivision. The solo category was more agile in pitch movement, the mixed category was less agile, and the accompaniment category was characterised by relative stillness in pitch variety. Once a scale was chosen based on the sentiment analysis of the tweets, the algorithm would do a random walk on the tree of the scale. Whether the scale was from the solo, mixed, or accompaniment category was determined by the form structure.

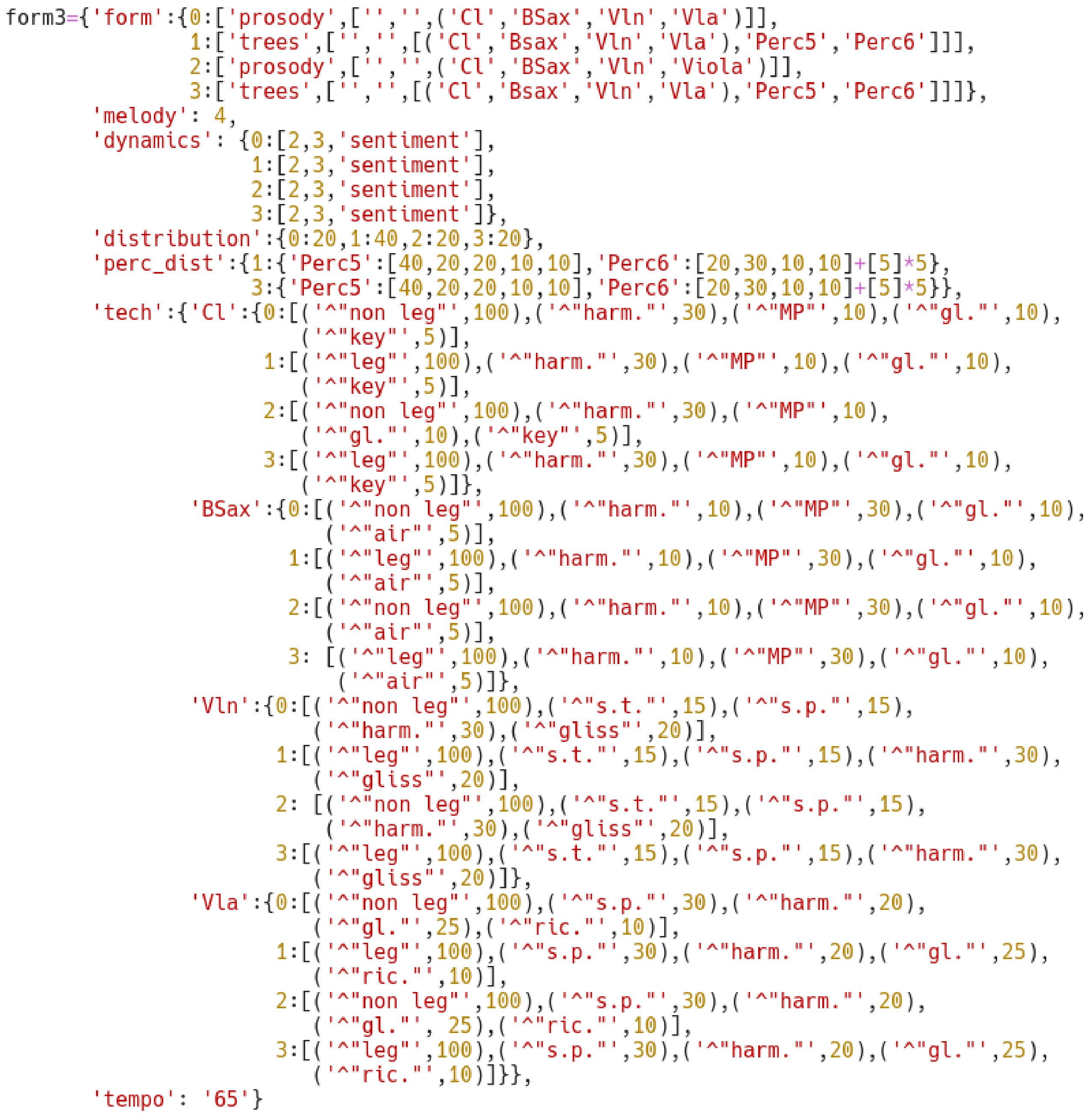
3.8.2. The Rhythms

The rhythm phrases of the music library were partly created in a similar way to the scales. Tree structures with two branches on each level contained short rhythmic phrases. The first two branch levels were written in 4/4 and the rest in 2/4, for variation. Similar to the scales, the three categories, solo, mixed, and accompaniment were applied to the rhythms too. The other part of the rhythmic construction was based on the prosody analysis of the tweets. Rhythmic phrases consisting of 2 to 9 symbols – notes or rests – were written for five different meters, 2/4, 3/4, 4/4, 5/8, and 6/8. As mentioned above, whether the prosody or the rhythm trees were used for a form was hard-coded into the form structures.

3.8.3. The Form Structures

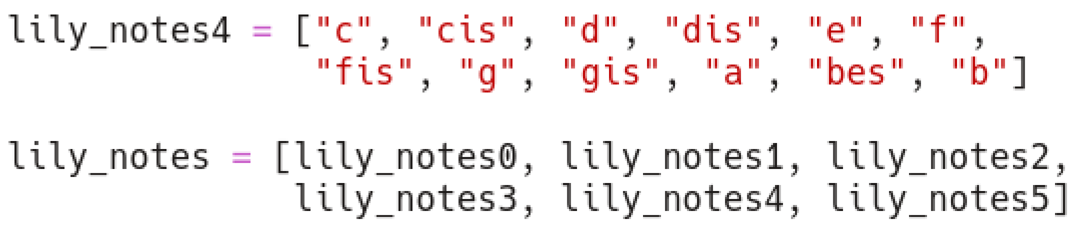
The nine distinct form structures contained the rest of the information for one block of music. Figure 10 is a Python dictionary of one such form. The first dictionary key – the word “form” – gives information on the source of rhythm – prosody or trees – and the orchestration for four sub-blocks. After the word “prosody” or “trees” there is a Python list with two empty strings and a tuple. These three elements denote the three roles, solo-mixed-accompaniment, in respective order, and concern the melodic scales and the rhythms – when these were constructed from trees and not prosody. The empty strings denote that the categories solo and mixed are not used and all four sub-blocks use the category accompaniment where we have a tuple with strings with names of instruments. Perc5 and Perc6 are the percussion groups 5 and 6. There were six different percussion groups performed by two musicians. This form structure is a rather still one and was aimed at the neutral sentiment.

The Python dictionary key “melody” determines the accidental types, sharp or flat – e.g. whether the score should have a B flat or an A sharp – and refers to the index of a list containing six lists with the twelve tones in Lilypond notation. Figure 11 demonstrates an example of one of these six lists and the main list. The value of the key “melody” in Figure 10 refers to the 5th element of the list `lily_notes` in Figure 11, which is the list `lily_notes4`, the top list of Figure 10. The notes of the `lily_notes4` list of Figure 11 are written in Lilypond notation and refer to the notes C, C#, D, D#, E, F, F#, G, G#, A, Bb, and B. The key “dynamics” gives a dynamic range for each sub-block along with a source for determining these dynamics. In this case the dynamics were between *p* and *mp* for all four sub-blocks (there were eight different dynamics, *ppp, pp, p, mp, mf, f, ff, fff*. The indexes 2 and 3 correspond to *p* and *mp*). The “distribution” key determines the overall percentage of the duration of each sub-block. “perc_dist” determines the percentage of the use of each instrument in the given percussion group. The “tech” key provides techniques for each instrument together with a percentage value for each technique. In this case, in the first sub-block, the clarinet plays non legato for the whole sub-block, 30% harmonics, 10% multiphonics (MP), 10% glissando, and 5% key. These percentages did not accumulate to 100% so not every single note had a technique assigned to it. The semantics around the techniques – caret and double quotes – are part of the Lilypond syntax and were placed inside the Python dictionary for easier use inside the algorithm. Finally, the “tempo” key provides the tempo in BPM for the whole form block.



```
form3={'form':{0:['prosody',['','',('Cl','BSax','Vln','Vla')]],
               1:['trees',['','',[('Cl','Bsax','Vln','Vla'),'Perc5','Perc6']]],
               2:['prosody',['','',('Cl','BSax','Vln','Viola')]],
               3:['trees',['','',[('Cl','Bsax','Vln','Vla'),'Perc5','Perc6']]]},
       'melody': 4,
       'dynamics': {0:[2,3,'sentiment'],
                    1:[2,3,'sentiment'],
                    2:[2,3,'sentiment'],
                    3:[2,3,'sentiment']},
       'distribution':{0:20,1:40,2:20,3:20},
       'perc_dist':{1:{'Perc5':[40,20,20,10,10],'Perc6':[20,30,10,10]+[5]*5},
                    3:{'Perc5':[40,20,20,10,10],'Perc6':[20,30,10,10]+[5]*5}},
       'tech':{'Cl':{0:[('^"non leg"',100),('^"harm."',30),('^"MP"',10),('^"gl."',10),
                         ('^"key"',5)],
                     1:[('^"leg"',100),('^"harm."',30),('^"MP"',10),('^"gl."',10),
                         ('^"key"',5)],
                     2:[('^"non leg"',100),('^"harm."',30),('^"MP"',10),
                         ('^"gl."',10),('^"key"',5)],
                     3:[('^"leg"',100),('^"harm."',30),('^"MP"',10),('^"gl."',10),
                         ('^"key"',5)]},
               'BSax':{0:[('^"non leg"',100),('^"harm."',10),('^"MP"',30),('^"gl."',10),
                           ('^"air"',5)],
                       1:[('^"leg"',100),('^"harm."',10),('^"MP"',30),('^"gl."',10),
                           ('^"air"',5)],
                       2:[('^"non leg"',100),('^"harm."',10),('^"MP"',30),('^"gl."',10),
                           ('^"air"',5)],
                       3: [('^"leg"',100),('^"harm."',10),('^"MP"',30),('^"gl."',10),
                            ('^"air"',5)]},
               'Vln':{0:[('^"non leg"',100),('^"s.t."',15),('^"s.p."',15),
                          ('^"harm."',30),('^"gliss"',20)],
                      1:[('^"leg"',100),('^"s.t."',15),('^"s.p."',15),('^"harm."',30),
                          ('^"gliss"',20)],
                      2: [('^"non leg"',100),('^"s.t."',15),('^"s.p."',15),
                           ('^"harm."',30),('^"gliss"',20)],
                      3:[('^"leg"',100),('^"s.t."',15),('^"s.p."',15),('^"harm."',30),
                          ('^"gliss"',20)]},
               'Vla':{0:[('^"non leg"',100),('^"s.p."',30),('^"harm."',20),
                          ('^"gl."',25),('^"ric."',10)],
                      1:[('^"leg"',100),('^"s.p."',30),('^"harm."',20),('^"gl."',25),
                          ('^"ric."',10)],
                      2:[('^"non leg"',100),('^"s.p."',30),('^"harm."',20),
                          ('^"gl."', 25),('^"ric."',10)],
                      3:[('^"leg"',100),('^"s.p."',30),('^"harm."',20),('^"gl."',25),
                          ('^"ric."',10)]}},
       'tempo': '65'}
```

Figure 10: A Python dictionary for the third form structure of Data Mining / Live Scoring.



```
lily_notes4 = ["c", "cis", "d", "dis", "e", "f",
               "fis", "g", "gis", "a", "bes", "b"]

lily_notes = [lily_notes0, lily_notes1, lily_notes2,
              lily_notes3, lily_notes4, lily_notes5]
```

Figure 11: Python lists to determine the accidental types of notes.

The development of this library was equally based on the intuition of the composer and various rules that emerged through programming the algorithm for this work. By attending rehearsals of short compositions created by the algorithm from an early stage of the development, Chatzopoulou and I were able to determine the parts that would remain in the library and get further developed and the ones that would either be discarded or radically changed. As the goal was to create a soundtrack for Twitter, personal aesthetics became less apparent, and a general perception of sentiment expressed through music was the central focus.

3.9. The Algorithm

The algorithmic composition of *Data Mining / Live Scoring* was split in blocks in order to have the time to collect enough tweets to create a substantial volume of music, as explained in the "Definition of CAC" section. Another goal was to keep the time intervals between blocks rather short so that the audience would not have to wait too long to see their tweets projected. During the development of the algorithm, it appeared that the maximum number of tweets needed to create a block of music was thirty. The system needed about half a minute to analyse the maximum number of tweets, collect all the data from the music library for all instruments, send the data over a local network to the Raspberry Pi[45] computers, and finally for the Raspberry Pis to render and display the score – this is further explained in the next section. The resulting music produced by this process lasted about three minutes. These durations depended on the traffic on Twitter at each time window in between music blocks.

While a block of music was playing, the algorithm knew how long it lasted, and would download tweets until half a minute before the end of the current block, unless it had already downloaded thirty tweets before this time constrain was met, in which case it would start the score creation and rendering procedure earlier. When the algorithm finished downloading tweets it performed sentiment analysis and then it fed the accumulated normalised sentiment mapped to a range between 1 and 5, together with the density of the tweets, to the SVM classifier – as explained in the "Sentiment Analysis" section – in order to get a prediction of the form structure for the upcoming block. The overall sentiment valence would also determine which music scale would be used. The first musical element that was created was the rhythm for the whole block, then the melody for each instrument was created, and finally the dynamics and various techniques were added. Figure 12 shows a flowchart of the process.

A consequence of the functionality of the algorithm was that a tweet and a retweet would result in similar scores when these coincided in the same music block. This is because each block used one tree for the scales and the rhythms – unless the prosody was used – so a retweet was actually the exact same source as the original tweet. In the case of prosody the rhythm would most likely differ, still it would contain the same number of symbols. When designing the system we did not take this into account, but we perceived this as a positive side effect.

45 https://www.raspberrypi.org/

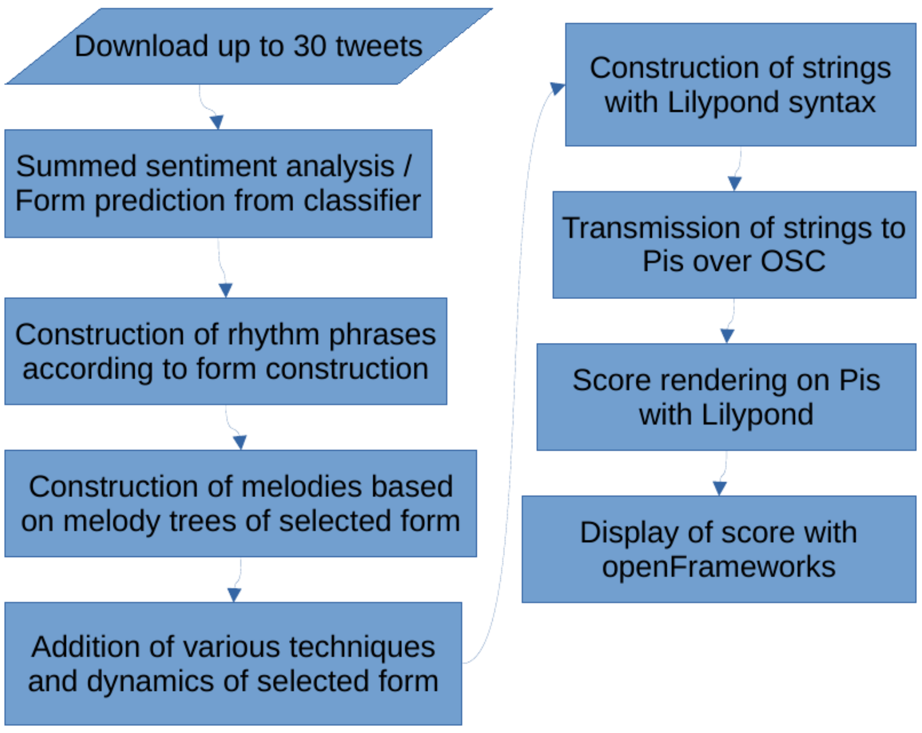

Download up to 30 tweets
Summed sentiment analysis / Form prediction from classifier
Construction of rhythm phrases according to form construction
Construction of melodies based on melody trees of selected form
Addition of various techniques and dynamics of selected form
Construction of strings with Lilypond syntax
Transmission of strings to Pis over OSC
Score rendering on Pis with Lilypond
Display of score with openFrameworks

Figure 12: Flowchart of the process of Data Mining / Live Scoring.

3.10. The Score Rendering System

The last technical aspect of *Data Mining / Live Scoring* that has not been analysed yet, is the system for rendering and displaying the scores. It consisted of the Lilypond music engraving environment in combination with an openFrameworks[46] application, both running on a Raspberry Pi mini-computer, one for each performer. The Raspberry Pi computer was chosen for its low cost and ease of integration to the hardware and scenic setup of the performance. In order to use any of the real-time systems mentioned in the "Live Scoring Software" section we would have needed one laptop for each performer as none of the software reviewed run on the Raspberry Pi – most do not run on Linux at all, the operating system of the Pi. With budgetary limitations in mind, one Raspberry Pi with

46 https://openframeworks.cc/

one monitor was a much lower cost solution than buying one laptop for each performer. Both Lilypond and openFrameworks run on the Pi. As mentioned in the “Live Scoring Software” section, when an engraving environment like Lilypond is combined with Python, which can handle text very efficiently, and the resulting score is fed to a program that can display PNG files, like openFrameworks, an interactive system can be created that is close to real-time.

3.10.1. Interactivity of the System

Each Raspberry Pi was receiving the Lilypond text from the main algorithm via OSC in a Python script. When all the text for the score of a block was received, the Python script called Lilypond and exported the score in a PNG file. An openFrameworks program was running inside the Pi. This program was receiving OSC messages from the Python script. These messages concerned when the next score was ready and what name the score file had. The openFrameworks program was also receiving OSC messages from the main algorithm which was doing all the counting and served as a conductor. The openFrameworks program displayed a blinking cursor at the beginning of the bar currently being played. This served to keep the musicians synchronised and to scroll the score when the bars were progressing so the musicians did not have to do this manually.

In order for the openFrameworks program to know where the beginning of each bar was, each score was rendered twice. One version rendered an additional note at the beginning of each bar. This was the C one octave above middle C for the G clef, and an E below middle C for the F clef. The second version rendered this additional note transparent. This second version was the one that was projected to the ensemble members. This difference between the visible and transparent notes in the two versions of the score enabled the openFrameworks program to locate the beginning of each bar by using the ofxOpenCV addon[47]. In case the first symbol of a bar was a rest, or a note close to this C (like B or D, one semitone below or above the additional C), the first version of the score rendered the rest or the note transparent and the second version – that was projected to the ensemble members – rendered the rest or note intact. In this case no additional note was rendered. Figures 13 and 14 show an example of such two versions of a score. In Figure 13 we can see that the beginning of every bar which starts with a note includes an additional C note, and every bar that starts with a rest renders this rest transparent. Note the fourth bar which contains a Bb and it is rendered transparent. The score in Figure 14 was displayed for the performers to sight-read. Note the second bar where the stem of the first note is longer than usual. This is because there is an additional C note above the visible F, which is rendered transparent.

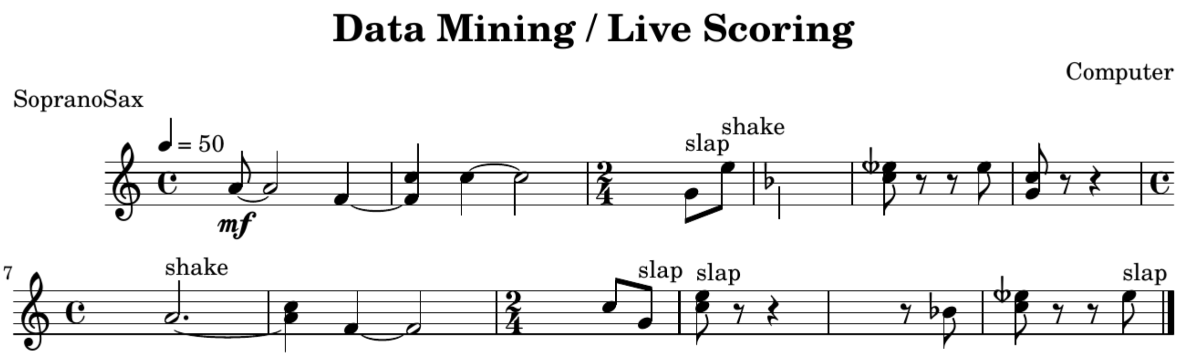


Figure 13: A score sample with transparent rests and visible additional notes.

47 OfxOpenCV is an OpenCV plugin for openFrameworks. OpenCV is a package for computer vision (https://opencv.org/)

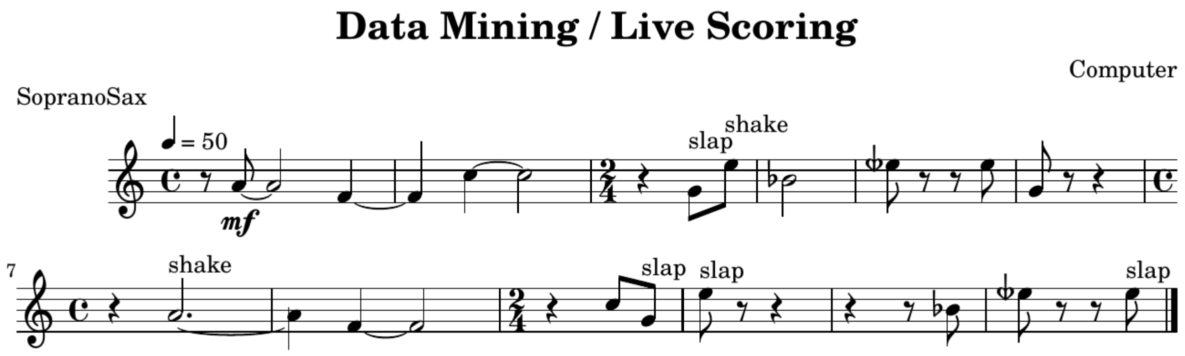


Figure 14: A score sample with visible rests and transparent additional notes.

3.11. Issues

This project resulted in two successful performances that the audience enjoyed and also engaged with by tweeting during this time. There were a few issues however, that will be addressed in future presentations. These issues concern the high volume of tweets received during the performance, trending hashtags overshadowing other hashtags that were also chosen at the beginning of each performance, an average sentiment valence which was floating between the central form structures, and occasional crashes of the Raspberry Pis.

As already mentioned, the algorithm downloads a maximum of thirty tweets for one block of composition. One block lasted approximately three minutes and the algorithm needed about half a minute to create the scores for all musicians. This left approximately two and a half minutes to download tweets. When the traffic on Twitter was high, the first thirty tweets were being downloaded much faster than the two and half minutes the algorithm had in its disposal. This resulted in many tweets being dropped. If the algorithm continued downloading tweets until the time it started rendering the scores, the upcoming block would become too lengthy. This problem would grow after every block, with the algorithm downloading more and more tweets, as it would have more and more time until the currently playing block was done. Apart from this, a lengthy block would distance the system even further from a real-time one, and the audience would have to wait too long to see their tweets projected. When tweets from the whole of Twitter were being dropped, this issue was not necessarily made apparent. But when tweets written by the audience were dropped, it became evident individually, to anyone who did not see their tweets projected. This had an immediate impact in the creative process, as one of the main goals was the participation of the audience, something that was also communicated, so all audience members were aware of the possibility to participate. It is possible that whenever this issue occurred, the appreciation of the audience regarding the interactivity of the performance, and the impact of their own input, was diminished.

The overshadowing hashtags presented another issue. A few minutes before the performance I logged on Twitter and checked the trending hashtags. I chose three of those plus the *#dataminінglivescoring* hashtag that I had created especially for this project. In the first performance I chose the hashtag of a Greek reality show which was being aired at the same time as the performance. This appeared to be a bad choice as the tweets with this hashtag were so many that they overshadowed all other hashtags, including the *#datamininglivescoring* which was used by the audience only. I had to remove this hashtag from the list during the performance. A good choice of hashtags is something that needs fine tuning. Contrarily to *TweetDreams*, I did not give prominence to the tweets with the *#datamininglivescoring* hashtag. Giving prominence to the hashtag of the performance could have provided a solution to this issue, as well as the previous one, as the tweets from the audience members only should be far less than the tweets of the whole of Twitter under trending hashtags.

When choosing many hashtags, hence many conversation subjects, there is a potential for the overall sentiment valence to average to some rather neutral value. This became apparent during the last phase of the preparation and affected the compositional result of the algorithm. The musical library of this project, consisting of nine different form structures, aimed at creating a composition with a certain level of variety. When the sentiment valence averaged to a neutral value, this variety was constrained, as only two or three central form structures were constantly being selected for an extended period of time. Fine tuning the training sets of the SVM classifier partly solved the issue, but this problem should be addressed in combination with the hashtag issue mentioned above. As the choice of hashtags needs fine tuning, so does the SVM classifier, and each time the classifier should be trained from scratch. An automation for creating training data sets could possibly provide a solution, but this would need to be thoroughly tested.

Finally, the temporal constraints forced by the composition of each block sometimes became problematic as a Raspberry Pi could reach a point where it did not have enough time to go through the whole score rendering process before the main algorithm signalled the beginning of the next block. This resulted in the Python script running in the Pi to signal the openFrameworks program to open the files for the next score, while these files did not yet exist. This caused the openFrameworks program to crash and leave its respective musician without a score. A possible solution might be to create musical bridges which would be deployed only when a block was finished but not all Pis were done with rendering the next score. These bridges should be of open form and length in order to leave as much time as needed to the Pis.

3.12. Conclusions

Data Mining / Live Scoring is a live algorithmic composition based on input from Twitter. Its aim was to create a composition for acoustic ensemble which would serve as a soundtrack for the tweets, the analysis of which determined the composition. Twitter served well both as random input and for the participation of the audience, as audience members were tweeting during the performance. Some issues arose during the development and the performances of this project, but the overall outcome was successful since the general rehearsal and the two presentations resulted in three distinct and complete iterations of the work with common elements, giving the impression of three different works written by the same composer. If I were to collaborate with another compositional co-creator the resulting pieces would probably bear little or no resemblance to the ones already realised.

The connection between the music and the text of the tweets was deliberately quite abstracted as this project did not aim to make an explicit direct sonification of the incoming data. This means that there was deliberately no direct way for the audience to control the musical outcome. Additionally, the resulting music aimed to serve as a soundtrack for Twitter, similarly to soundtrack for film, where the mood of the tweets was abstractly depicted and augmented by the music.

The audience engaged well during both performances with many tweets using the *#dataminininglivescoring* hashtag going through the algorithm. My goal was the engagement of the audience through tweeting as a source of amusement. This was indeed reflected by the content of the tweets provided by the audience, most of which had a humoristic tone, at the same time expressing a fascination for the overall experience[48]. It is worth noting that repeated attendances would most likely not result any kind of deeper understanding of

48 https://twitter.com/search?q=datamininglivescoring&src=typed_query

the relationship between the tweets and the music. Still a different music composition that would result by a different set of tweets would provide a similar, but different experience.

3.12.1. Contribution to Knowledge

Data Mining / Live Scoring aims to provide a compound example of a conceptual music artwork, where different techniques and approaches of existing works are combined in this project. In the *Twitter Input in Related Works* section of this chapter, a significant number of existing works that utilise Twitter is presented. Bearing in mind the limited number of these works that apply sentiment analysis, *Data Mining / Live Scoring* aims to reinforce this small group of artworks. It aims to contribute to knowledge in this specific field by projecting the approach and resources used to realise this analysis.

The project is not limited to Twitter and sentiment analysis, but goes further to create an algorithmic composition. Again, there is a significant number of works that apply algorithms in the compositional process with the use of computers, but not many do this in the context of an acoustic music ensemble. The *Music Library* section of this chapter aims to provide insight to my approach to algorithmic composition for an acoustic ensemble, by analysing how the music library was composed, what elements it consisted of, and how these elements were selected by the algorithm. The Live Scoring part of this project also contributes to knowledge by providing an alternative to the existing software through the source code of the openFrameworks application that projected the scores, and an analysis of the approach to the interactivity of this system.

3.13. Future Work

Inspired by the *Deep Learning With Audio* course[49] from the SOPI research group, I am planning to further develop this project and utilise their Python3 version of the *pyext* Pd external together with their Machine Learning agents for creating melodies. This version should provide a more sophisticated score generation mechanism than the current one. The collaborating composer will not have to write a library with melodic and rhythmic trees, but only short segments of music. These will be classified according to the various sentiment values and will be fed to the agent, where the SVM classifier will decide which music segment will be used, based on the sentiment analysis. This way, part of the existing work will be combined with these new elements. In this case, since Pd will be used to load Python code inside *pyext*, the scores can be generated with the *notes* Pd external that uses Lilypond.

49 https://github.com/SopiMlab/DeepLearningWithAudio

Chapter Four

Live Coding on a Modular Synthesiser

This chapter is about a significant work of musical development combining a tangible artefact, a hardware module called Code, that enables live coding on the 3dPdModular modular synthesizer system, which is a large-scale independent standalone hardware project that I have personally developed, much of the work on this system maturing outside and alongside the PhD study, and a music piece produced with it. Live coding with a computer and live patching a modular synthesiser are two practices that have many things in common (Hutchins, 2015, p. 147), especially when live coding[50] is undertaken within a visual programming environment, like Pd.

When patching a hardware modular synthesiser, the user changes the flow of electricity within the system of the instrument. When coders write code live, they change the connections between various code abstractions, which bears similarities to plugging and unplugging cables between various hardware modules. This connection can become more evident when using a system such as Eriksson's Automatonism (Eriksson, 2019), in which Pd abstractions have been built following the hardware modular synthesiser paradigm.

Several Eurorack[51] modules that support coding to some extent have been produced. To the best of my knowledge, the only module that supports writing code live is Teletype by monome (Crabtree and Kelli, 2021). Teletype was the inspiration to create the Code module for my previously devised 3dPdModular system. The system is programmed

50 or live patching, to use the preferred nomenclature.
51 Eurorack is the most popular modular synthesiser format initiated in 1996 by Doepfer Musikelektronik.

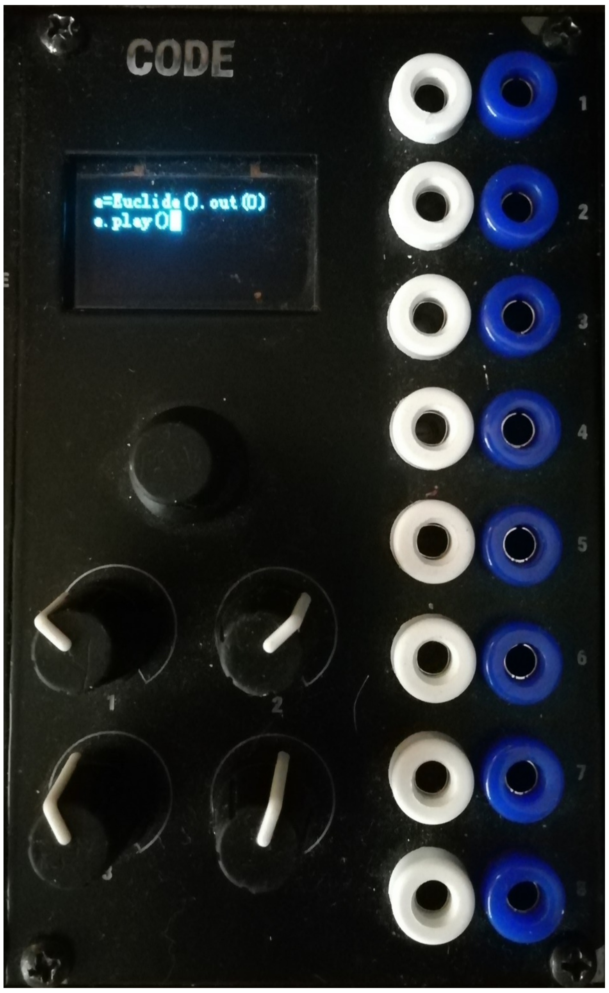
entirely in Pd, but my goal was to create a module that would support writing Python code, as this is my main textual programming tool. Luckily, Olivier Bélanger, the developer of the Pyo module had already created a Pd external object that ports Python with Pyo into Pd, called *pyo*~. Even though this object does not fully support live coding, but achieves a level of integration of Python and Pyo into Pd, using *pyo*~ as a starting point, I was nevertheless able to implement support for live coding according to my specific requirements.

The main idea of Code is twofold. Firstly, it aims to expand the capabilities of a hardware modular synthesiser, which is limited by the number of modules present in the system. When writing computer code, the only limitation of the user is the CPU (White, 2019, p. 71), and with modern computers this limit is met only after introducing a very large number of processes into an algorithm. By being able to import any of the Pyo classes[52], a wide range of Unit Generators, filters, effects, sequencing mechanisms, and others are available to the user, filling the gap created by the absence of hardware modules for specific processes.

The second part of the idea of Code is expressed in the words of Thor Magnusson: “There is little fun in watching a stressed programmer designing algorithms for minutes before a simple sine oscillator is applied in the playback of a silly melody” (Magnusson, 2011, p. 503). This issue has been addressed by several programming languages built for live coding. These languages have been mentioned in the “Live Coding” subsection of the “Software Concepts” section in the first chapter of this thesis. Most of these languages though aim at rhythmic and melodic loop-based music. The Code module aims to address this issue with a different approach. By combining live coding with live patching, the user

52 http://ajaxsoundstudio.com/pyodoc/api/classes/index.html

can take advantage of the tangibility of patching modules together, such as oscillators and filters, and make changes in the sound by taking short breaks from writing code.



```
e=Euclide().out(0)
e.play()
```

Figure 15: The Code module.

4.1. The 3dPdModular System

The 3dPdModular system is a hardware modular synthesiser system running on a single Raspberry Pi computer and a Teensy micro-controller, an Arduino compatible micro-controller for controlling sensors and physical computing in general. This system was inspired by the *rePatcher*[53], by OpenMusicLabs, a now discontinued shield for the Arduino, turning the latter into a patch-bay matrix for Pd or Max. This system started from a personal need to develop a musical instrument that would provide a satisfactory interface for live performance of electronic music. I did not find the popular laptop and MIDI controller setup good enough for live performance, mainly due to the limited resolution in MIDI controllers – seven bits of resolution providing 128 discrete steps in the range of a potentiometer – and the lack of visual mapping of hardware to its functionality – potentiometers and sliders in MIDI controllers are usually only numbered sequentially on their panels, while in software they are being mapped to all sorts of different functions. I decided to start developing a hardware modular synthesiser based on these tools, and the 3dPdModular system was gradually created.

This system is fully integrated within itself. In order to use it in combination with other hardware – e.g. a Eurorack modular system – a specific bridging module is necessary. All the signals going through the wires that connect the various modules are digital pulses and do not transfer audio or other analogue signals from one hardware module to another. The Teensy micro-controller detects connections between the various modules and sends the information to the Raspberry Pi. The latter runs a Pd patch that includes software versions of the hardware modules. When a connection is detected by the Teensy and sent to the Raspberry Pi, the same connection is being done in the software.

Such an integrated system may not be easily combined with other – analogue – hardware, but it has a number of unique characteristics due to its digital nature. A characteristic module of this system is Clone which clones the software of any combination of modules connected to it. The hardware of the connecting modules is used as the control

53 http://www.openmusiclabs.com/projects/repatcher/index.html

interface for all clones, with Clone determining which clone instance is being currently controlled. This way the user can apply polyphony or other effects without needing to physically possess the same hardware modules multiple times. Another characteristic of this system is metadata that can pass through the connections between the various modules. This metadata concern the nature of signals – e.g. whether a signal is audio, Control Voltage (CV), a trigger signal, etc. – and it is used to minimise the hardware needed for some modules. If, for example, a module behaves differently when an incoming signal is CV or audio – e.g. a Voltage Controlled Amplifier (VCA) uses logarithmic curves for the amplitude control of audio and linear curves for CV –, metadata can automate the change in this behaviour. This makes a module more compact and less prone to error.

3dPdModular can save patches in textual form that can be transferred to a mobile phone over a bespoke application, called Patches. This application can display descriptive text concerning connections between modules and visual representations of the positions of potentiometers. Figure 16 shows a screenshot of the application with the potentiometer positions of the module Trapezoid. By using this feature, a user may remake a patch with minimal effort.

Regardless of the type of the software signals, the signals passed through the physical cables are of the same nature – digital pulses –, so a generic module can be programmed to receive and output any type of signal in any of its inlets and outlets. Generic modules for other systems, like the Eurorack, have a fixed number of audio and a fixed number of CV inputs and outputs, and their configuration cannot be changed.

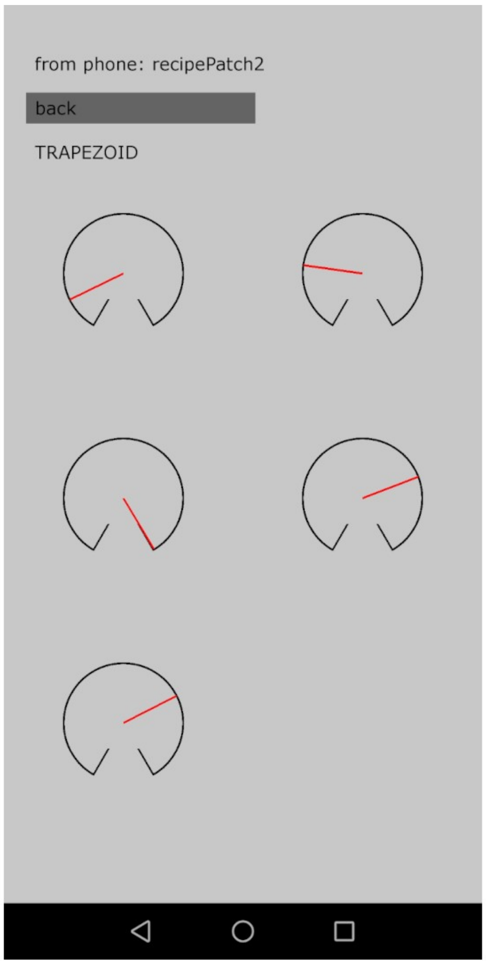


Figure 16: Screenshot of the Patches mobile application.

4.2. Related Work

Both commercial and non-commercial projects that combine computer programming or live coding with modular synthesizers in various ways exist. In this section I will mention projects that, to the best of my knowledge, are closer to the Code module than others.

4.2.1. Non-Commercial Projects That Combine Live Coding with Modulars

Among the non-commercial projects that combine live coding with modular synthesizers, The Force by Ryan Ross Smith et al. is probably the most relevant (Smith et al., 2016). This is an Arduino device that connects a computer with a modular synthesizer. The coder writes openGL shaders and sends data to the Arduino over OSC, which in turn converts this information to CV signals through a 12-bit DAC[54] and sends them to the modular. This communication is duplex, as the synthesizer can send CV signals to the Arduino device that are being digitised and sent over OSC to the computer. This project is close to Code as it creates a one-to-one live coding and live patching system. The two components of the system though are remote, in contrast to Code where both are integrated into the modular synthesizer.

Another project that can combine these two practices has been realised by Haddad et al., who created a Eurorack module that can connect various modular synthesizers remotely, as long as each synthesizer has one of these modules in their setup and these are connected to the Internet (Haddad et al., 2019). Even though the main goal of this project is to connect remote synthesizers, live coding is possible through a browser-based interface where the coder can write JavaScript code to send audio signals to connected modules.

A third project that is more loosely related to Code is a hardware module created by Joseph Paradiso that connects to his massive modular synthesizer (Mayton et al., 2012). This project is more loosely connected because there is no live coding involved, but remote users can control parameters of this module through a browser-based graphical

54 Digital to Analog Converter.

interface. Through this interface, connected users can also listen to the modular synthesizer in real-time.

4.2.2. Commercial Projects That Combine Code with Modulars

There are several commercially available modules that may be programmed by the user in a variety of languages. Built in the Eurorack format, these modules are integrated in the most popular modular synthesiser system, with hundreds of manufacturers and thousands of modules being produced and distributed in the market. This fact alone differentiates them from Code, but still there are certain similarities that make them relevant in the context of this chapter.

Bela Salt is a Eurorack module produced by Bela (Bela, 2021), the audio shield for the BeagleBone Black mini-computer. Bela can provide a stereo input and output, along with a number of analogue and digital sensor inputs and outputs. The Eurorack module version of Bela provides four potentiometers, four CV inputs, four CV outputs, two trigger inputs, and two trigger outputs, along with the audio stereo input and output, a fixed configuration that cannot change. It can be programmed in a variety of languages, including Pd, SuperCollider, C++, and Python with the Pyo module. Live coding is not possible, as the only option is to program the module before using it.

OWL Modular is a derivative product of the OWL, a programmable foot pedal (Rebel Technology, 2018). It is very similar to the Bela, in that it is a programmable Eurorack module that can be programmed in a variety of languages, including C++, FAUST, Pd and Max Gen patches. Its inputs and outputs are of fixed type, with a stereo audio input and output, five CV inputs, and one trigger input and one output. Like the Bela, this module does not support live coding.

Another similar module is the QU-Bit Nebulae (QuBit Electronix, 2021). This module is mainly a granular sampler, but also a DSP platform for uploading various programs. It can be programmed in Pd, Csound, and SuperCollider. Since it is mainly a granular sampler, its panel layout is labelled following this functionality. When a user uploads a different file in one of the supported languages, most likely the panel layout will not fit the functionalities of the potentiometers and buttons. Live coding is not an option.

Finally, Teletype is the most relevant module, since it does support live coding. This module supports both predefined scripts and writing code live. It includes eight trigger inputs, four trigger outputs, and four CV outputs, again a fixed configuration. It uses a simple bespoke language which can control the triggering of the first four outputs, and the voltages in the last four outputs. This module provides functionality for sequencing but not for audio signal processing, since it does not provide audio inputs or outputs.

Code can only be programmed in Python, and, compared to the other modules mentioned above, that could be seen as a drawback. It is also part of an integrated system and cannot be combined with other systems without extra hardware. When used within the 3dPdModular system though, it provides flexibility in various ways. Its eight inputs and eight outputs can receive and output any type of signal – triggers, gates, CV, or audio. The module can provide functionality for audio processing, sequencing, and signal generation. Like Teletype, it supports live coding, but it uses a popular, general-purpose and mature programming language, instead of a domain-specific language that is aimed at audio programming only, or a bespoke language like the one Teletype uses, limited to the functionalities of the trigger and CV inputs and outputs. With Code, the user can take advantage of the extensive library of Python modules, whether these are designed for audio processing or other functionalities. This way the user can include techniques such as Machine Learning, Data Mining, or others supported by the various modules of the

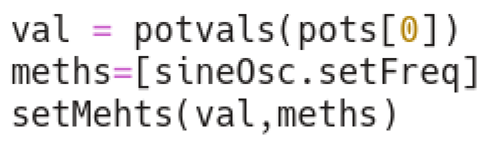
Python repositories, with minimal coding since these modules encapsulate most of the code for carrying out their tasks.

In contrast to Teletype and following the live coding convention of the coders sharing their screen, the Code module supports projecting what is being typed on an external monitor or projector. This projection simulates the Python interpreter inside a computer's terminal and is realised with an openFrameworks application launched when the user chooses to project their performance. Another feature of Code is the capacity to save live coding sets for future use as pre-defined scripts, as well as capturing the projection of the code and saving it as a video file, for documentation.

4.3. Fast prototyping

A distinction between live coding and live patching concerning prototyping could be made. Both practices make use of tools, the modular synthesiser for the latter and a programming language for the former. In this context, the more acquainted the user is with the respective tool, the faster the user can prototype. However, when compared to live coding, the tangibility of the hardware modular synthesiser provides a faster way to input values to a certain function. Turning a potentiometer is a lot faster than typing computer code. The Code module provides a minimal tangible interface that can be imported to the code it executes. This interface consists of four potentiometers that can get arbitrary ranges and curves, and can be used to provide input to various functions of the Pyo classes. For example, when the user creates an oscillator in Code and uses another hardware oscillator to modulate the frequency of the former, they can use one of these potentiometers to control the frequency of the Code oscillator in order to get a satisfactory result. In this case, where Code is used in combination with other hardware, if this process is done by writing code, it will likely take more time.

An additional feature to this interface is mapping potentiometer values to certain functions and freeing the respective potentiometers. In the example above, when a satisfactory sound has been created, the user can write the code in Figure 17. In this example, the output value of the first potentiometer – that is supposed to be the first element of a Python list called `pots` – is mapped to the `setFreq()` method of a sine wave oscillator. The frequency of the sine wave oscillator will remain fixed after the `setMeths()` function call and the first potentiometer of Code's hardware will be freed and available for other uses. This feature is operable with up to four potentiometers and up to four methods for mapping, all with one `setMeths()` function call.



```
val = potvals(pots[0])
meths=[sineOsc.setFreq]
setMehts(val,meths)
```

Figure 17: Example of the setMeths() function call.

4.4. Challenges

Developing the Code module posed a number of challenges. The main challenge was the fact that the *pyo~* Pd external does not support live coding, as it was built for importing existing Python scripts. The user can create variables and objects, and call functions live by sending specific messages to the *pyo~* external, but creating functions and classes within Pd is not possible. Since *pyo~* integrates Python with Pyo into Pd, I thought it was easier to find a way to write Python code live and upload it to the object, rather than find another way to import Python with Pyo into Pd.

A feature of the *pyo~* external that proved to be helpful is the ability to upload Python scripts on top of others. The user can upload one script and then upload another without deleting the previous one – e.g. the first script can be a square wave oscillator and the second one a filter with the oscillator of the first script as its input. By utilising this feature, I could create short Python scripts, one for every line of code, or for every function, loop, or class definition, and upload them to *pyo~* whenever I hit the return key on the keyboard. This was made possible by translating the raw bytes received from the program that detects the key strokes on the keyboard to alphanumeric characters, and placing them in text files with the appropriate indentation[55]. These files are saved with the .py suffix as *pythonized_lineX.py* where X is an incrementing number. This number is sent to the Pd abstraction of the Code module via OSC and the current Python script is uploaded to *pyo~* when the return key is hit.

Another challenge was the use of a display on the module's panel to give visual feedback to the programmer, and how to properly display what is typed, preferably in a similar way to that of a Python interpreter inside a computer's terminal. This interface is fairly simple, yet it provides very good visual feedback and a small set of features to the programmer (whether the current line is a single one, or part of a function or class, and being able to scroll through the previously typed lines). The 3dPdModular System provides an interface using the Inter-Integrated-Circuit (I2C) protocol, to use an external micro-controller in case one is needed. In the case of Code, an external micro-controller is needed to control the display. The micro-controller receives the typed characters in their ASCII form as bytes, and displays them one next to the other, until it receives the return

55 Python's syntax is based on indentation as it does not use curly brackets to encapsulate code structures.

key byte. Every typed string is saved to memory, so the user can scroll through the previously typed lines, as in the computer's terminal Python interpreter. One issue is the size of the display, which is rather small, and therefore not so easy for the programmer to see. This means that each line has a very limited number of characters, namely twenty. Even though writing short lines when programming can be a good practice, and the small size of the display could be a reason to push the user to adopt such a coding style, sometimes it is impossible to include a single line of code in twenty characters, simply because a calling class might have a long name, or take many parameters. For example:

```
sl=SineLoop(freq=100,feedback=.1)
```

The line above has 33 characters, even though the name of the object used is minimal (sl, for SineLoop). If used intact, this line would be broken in two lines in the Code module. Having space for only displaying five lines at a time, it makes such long lines impractical, plus scrolling back to previously typed lines would result in this line being displayed in two different parts.

A solution to this problem was to write a class for each one of the Pyo classes with a shorter name for the class itself, and shorter names for all its functions. This way the line above changes to the following:

```
sl=SL(fr=100,fb=.1)
```

The line above has 19 characters. One drawback is that in order to send it to an output, an extra line has to be written, which is the following:

```
sl.out(0)
```

Instead of calling the `out()` method at creation time and forcing the Code module to split the line in two, as it would not fit in the display, it is preferable to separate this method call and write a new line. It results in cleaner and more readable code, easy to recall with the use of the arrow keys.

A last challenge was the projection of the typed code in real-time. The 3dPdModular system runs on a Raspberry Pi, and not all graphics libraries are compiled for this architecture. Pd’s Gem graphics library could be a choice for this task, but there are two disadvantages using it. First, it is maintained but not actively developed and has dependencies that are not always updated. Secondly, the curly brace character cannot be used inside Pd. The curly brace character denotes a Python dictionary, and it is very likely that the user of Code will want to type it. This reason excludes another Pd library for graphics, Ofelia, which is actively maintained and developed.

The above restrictions led to the choice of openFrameworks, because it does not have any character restriction, it is compiled for the Raspberry Pi and it is easy to setup. Again the OSC protocol was used, not only when the return key was pressed, but for every single character typed. I wrote an openFrameworks application that receives the ASCII bytes via OSC and displays them in a similar fashion to that of the on-board display of the module. A projected screen can include much more than twenty characters, so in case a line breaks on the on-board display, this will not happen in the projection, making the code look less ugly and easier to understand for the audience.

4.5. “Latitude”: The Live Coding / Live Patching Performance

In this section I will reflect on the performance based on the Code module, from a technical, conceptual, and musical point of view. The performance with the Code module aims to support its twofold concept, the extension of the functionality of the hardware of the 3dPdModular system, and the fast creation of sound with a small number of lines of code typed by the user. It also aims to introduce methods outside the realm of music. The modules present in the performance are two oscillators, one dual VCA, a clock divider, the Code module, and the main module of the 3dPdModular system which provides the audio inputs and outputs. With this limited hardware setup, I aimed to produce a performance with a variety of sound processes greater than those provided by the rest of the hardware, and at the same time offering a solution to avoid writing an excessive amount of code before starting to produce sound.

The non-musical method used was geolocating randomly chosen capital cities through the Google Maps API for Python. The latitude and longitude values of each searched capital were used as MIDI note values to control the pitch of the two oscillators of the system. Later on, the previously selected coordinates were combined with the current ones, with the former controlling various timbral parameters of the oscillators. The use of the Python module that provides the Google Maps API was pre-defined and imported at the beginning of the performance. This was required to avoid sound distortion, as loading the Google Maps module takes some time and requires processing power. After this module was imported, I created a Euclidean rhythm generator, provided by the Pyo Python module. This generator is based on the Euclidean algorithm to produce rhythms with their onset patterns being distributed as evenly as possible (Toussaint, 2005). Its output was sent to the clock divider hardware module to create polyphony in the resulting rhythm. By controlling the internal Attack-Decay envelopes of the two oscillator modules, I was able to create a rhythmic pattern with two voices in a short amount of time. Randomness was introduced to the patterns of the Euclidean rhythm generator at a later stage with a few more lines of code. Phase modulation was applied to one of the oscillators with the other one being the modulator. This was done in the hardware of the synthesiser.

The next phase introduced the geolocation of randomly chosen cities. In a pre-defined Python script, a random country was chosen from a list provided by the *pycountry* Python module, and its capital was retrieved with the *countryinfo* Python module. The result was fed to the `googlemaps.Client.geocode()` method, and the latitude and longitude values were isolated from the output. These values were output from the hardware outlets of the Code module and input to the frequency inlets of the two oscillators. The chosen capital together with its latitude and longitude values were displayed on the projection of the code, revealing this information to the audience. Eventually, two delay lines, two distortion units, and two frequency shifting effects were introduced, one for each oscillator input. These three processes used the hardware potentiometers of the Code module to control some of their parameters.

In this performance, the Code module provided a total of seven processes, not present in the rest of the hardware setup – a Euclidean rhythm generator, two delay lines, two distortion units, and two frequency shifters –, extending the functionality of the synthesiser. It also provided a geolocation mechanism available in Python – mainly for purposes other than music – that was easily imported into the module. By utilising Python's *multiprocessing* module for diffusing processes across the cores of the computer's processor, the CPU usage on the core where Pd run was not strained by the geolocation Python module. This was necessary to avoid dramatic dropouts in the sound whenever a city's coordinates were queried. The hardware potentiometers of the module were also utilised, making the process of tuning various parameters faster than just typing values. The combination of coding and hardware proved to be effective with the various tasks of the performance equally distributed between live coding and live patching.

The starting point of the concept of this work is the global issue of mass migration and refugees, which results in prejudice developing in local communities[56]. Under these circumstances, the origin of people plays a crucial role in how they are being treated by others. This performance associates randomly selected capital cities with sound, by mapping their latitude and longitude coordinates to frequencies and other timbral elements. *Latitude* questions our notion of localism by producing unexpected sounds, where the sound produced by the coordinates of a city known to be beautiful might be disturbing, while the sound produce by an infamous city might be more pleasant.

Since the latitude and longitude coordinates do not have any social meaning by themselves, controlling the flow of data within the synthesiser system gives a certain character to these coordinate numbers. With these coordinates being tightly connected to the name of a city, the sonic character created by the operation of the synthesiser is tied to that city during the performance. The development of the performance aims to create a unified, unsettling atmosphere for any city that might be chosen, as a form of critique on a global social issue of our times, that is likely to leave a mark in history.

The music of this performance started as a rhythmically-led musical piece, with random variations introduced to the patterns from an early stage. A playful but harsh texture is achieved with the frequencies generated by the randomly queried capital cities. Modulating the phase of one oscillator with the other, a technique I am very fond of, provides this harsh character to the sound. This is further intensified with the delay, distortion, and frequency shift effects. A linear development of style can characterise this performance. It can be separated into three main sections that are smoothly bridged from one to the next. At times the work is rhythmically-led with steady patterns, at times random, at times ambient and closer to noise music.

56 This statement is based on personal experience in Greece, where for an extended duration, immigrants and refugees seem to be trapped while on their way to northern Europe.

4.6. Issues

Even though Code is a functional module that can successfully support live coding, there are some issues that could be addressed to improve the hardware and software components of this module. An evident issue is the small size of the display, that makes reading the code difficult. A larger display can solve this issue, but that would require a complete redesign of the hardware and the circuitry. This is included in my current plan for further development of the 3dPdModular system, but it has currently not been realised.

As mentioned in the Challenges section of this chapter, the small display size limits the number of characters displayed in a single line. The solution to this problem was to write aliases for the Pyo classes with shorter names for the class names, their variables, and their methods. Using short names can render the code written live less understandable and more difficult for the audience to follow the coder's train of thought. Classes like `SuperSaw`, `Pattern`, or `MoogLP` can be self-explanatory to an audience with some knowledge on electronic music creation. The short aliases for these classes are `SSaw`, `Pat`, and `Moog`. These names are, of course, not as self-explanatory as the original ones.

The next issue is the speed at which the characters are being transferred to the micro-controller controlling the display. This transfer is done with the I2C protocol, that is transferring bytes over one wire in duplex mode. For this and other reasons that concern electronics, this protocol is considered to be rather slow. Combined with all the data that need to be transferred from the main micro-controller of the 3dPdModular system to the Raspberry Pi, and vice versa, sending character bytes to the micro-controller that controls the display is a bit slow. This results in some character bytes not being transferred even though typed and saved to the Python scripts, when the user types fast. For example, the user might type the following:

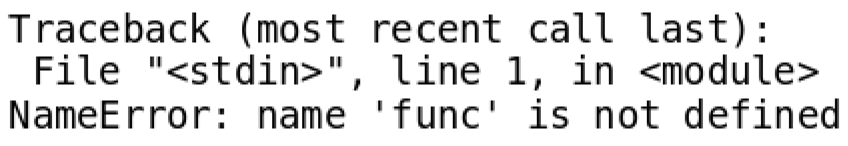
```
def a_func(i):
```

But the display will show the following:

```
def a_func():
```

In this case the argument that is supposed to be passed to the function is not visible in the display, but it has been passed to the saved Python script loaded to the *pyo~* external. In this case, the user cannot know whether this is really the case or whether they failed to type `i`. A solution is to clear the current line of code, but due to the speed issue, sending the backspace character will occur at a slow speed. When live coding, this speed issue can prove to be problematic, as the live coder is likely to be typing fast since they are familiar with the computer keyboard, and typing fast is desirable in order to address the prolonged audience waiting in live coding sessions. Applying threading to the Arduino code that runs in the Teensy micro-controller is a possible solution, since the I2C communication will be isolated in its own thread and will not be interrupted by other routines necessary for the system. This will likely increase the speed at which characters are sent from the main Teensy to the one controlling the display.

Lastly, feedback to the user when bad code occurs is limited. When writing Python code in a Python interpreter inside a terminal, the computer provides detailed feedback when the user writes invalid code. For example, if the user calls a function called `func()` which has not been defined yet, the computer will provide the feedback in Figure 18, called a traceback in Python nomenclature.



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'func' is not defined
```

Figure 18: Error traceback in a Python interpreter.

In a similar case, when this error occurs in the Code module the *pyo~* object only prints the following line in Pd’s console:

```
Bad code in file <pythonized_line0.py>
```

The “Bad code” console printing occurs when the *pyo~* object returns an error code with the value 2. This is accessible through the object’s source code – its C file. I modified this code and created an extra outlet in the object that outputs this error code. When the value 2 is sent out this outlet, the display of the module prints “Bad code!”. This feedback is very limited and provides only minimal information to the user. Retrieving more detailed feedback from *pyo~*’s source is not a trivial task, and even if this is achieved, with the current slow speed of data transfer to the Teensy that controls the display, sending detailed feedback on bad code is not feasible.

4.7. Conclusions

Code is a module for the 3dPdModular system that provides flexibility and extensibility to the system by importing the Python programming language with the Pyo module, both in the context of loading pre-defined scripts and in the context of live coding. This chapter focused on the live coding capabilities of the module. When compared to live coding with a laptop, live coding with Code might seem limited, or slow. Limited due to its minimal editing capabilities, and slow due to the low responsiveness of its display when the user types fast. Despite these problems, Code can have great artistic potential in the context of a modular synthesiser as it introduces another dimension in improvisation with such an instrument. It also introduces a vast pool of functionalities, most of which are outside the musical realm. From Machine Learning to Data Mining, or even web browsing or sending emails, Code can enrich a performance, enabling artists to combine disciplines in a unified artwork based on the modular synthesiser.

This module, unlike other commercial programmable modules, can be programmed in one language only, thus it is not a polyglot module. It does use Python, a language rarely found in DSP, and, to the best of my knowledge, found in only one of the relevant modules. It is also limited to the 3dPdModular system, and cannot be combined with other systems without extra hardware. It is though, to the best of my knowledge, the only module for a synthesiser system that supports live coding in Python, and that can project the code, following the “show us your screen” convention of live coding.

4.7.1. Contribution to Knowledge

The contribution to knowledge of the Code module lies both on the technical aspects of both the hardware and the software, and the conceptual approach of the performance realised with it. As far as the technical side is concerned, Code approaches live coding from a different perspective, as it uses Python, a language that is rarely, if at all, used in this practice. As with the *Echo and Narcissus* opera, it introduces the use of the Pyo module for live coding, a module that, despite its effectiveness, remains rather unexploited in this artistic practice. The technical problems that arose during the development of this module provided a flexible way to address the issue of writing code live, resulting in an approach that, even though it can be considered unorthodox, proved to be effective in enabling live coding within an interface that does not really support it. The problem-solving philosophy applied to the realisation of this project aims to inspire readers to invent their own approaches, even if unorthodox, to solve their own technical problems.

The *Latitude* performance utilised an extra-musical component as a key element of the work: the latitude and longitude coordinates of randomly chosen capital cities used as MIDI notes for two hardware oscillators of the 3dPdModular system. From a conceptual point of view, this approach can provide a useful example for musical or even mixed media artworks, by associating sound with the selected cities, thus aiming at creating connotations between perception of localism and the various locations. From a more technical perspective, *Latitude* proved the effectiveness of Python, as all the needs of this project, whether musical or extra-musical, were covered by modules that are available in this language.

4.8. Future Work

Further development of Code is included in my plans. Apart from addressing most of the issues mentioned in the previous section, future plans for Code include supporting more programming languages. Python is not the only language that may be imported into Pd. For example, Lua, JavaScript, and Scheme are already imported, with Lua port being at a stable state, and the other two still at a beta stage. Some can provide an easy way to write code live, and some must be approached in a similar way to the current way Code works with Python.

The main 3dPdModular Pd patch supports only audio signals between modules. The three languages mentioned here function on the control domain without receiving or outputting audio signals, but that can be remedied by converting audio to control for input and control to audio for output. This conversion will introduce sample-inaccuracy, but that will probably be very subtle. Pd’s block size is 64, and with this size, sample blocks last only 1,45 milliseconds, rendering most of sample-inaccuracy inaudible. Further plans also include extending the hardware of the module by adding another four potentiometers, for a total of eight, as well as adding four push-buttons for manual triggering of events, and four LEDs to convey additional visual feedback.

The *Latitude* performance can also be further developed. The performance realised for this research used a limited number of modules deliberately, to demonstrate how the Code module can extend the 3dPdModular system, by providing functionalities not present in the hardware. In a future performance of *Latitude* I will use more modules to have greater variety in the sounds that can be created, and more hardware control on them. A possible addition to the performance is to randomly associate the latitude and longitude coordinates to words that relate to the character of a city, like “unemployment”, “infrastructure”, or “criminality”. This idea aims to augment the concept of the synthesiser operator functioning as a kind of free speech and fake news controller, by providing random data for the various cities.

Chapter Five

Conclusions

By concluding my research, I have come to a point where I have grown to a Python programmer to the same extent as a Pd programmer. Before I started this research I had authored a book on Pd and Arduino programming for interactive musical projects, and created the freestanding 3dPdModular system, programmed in Pd. These two rather large-scale works indicate my focus on Pd as a main tool for music creation. Python came at a later stage, and through this research, I was able to explore its creative potential through the realisation of the three works of this portfolio. This research helped me develop both as a programmer and an artist. The unconventionality of the works presented here, supported by the use of Python, has highlighted my artistic vision, that of creating mixed media works that are both conceptually rich, and capable of producing a complete and compound artistic composition while introducing new concepts and elements not known to be found in musical works.

The use of Python has proven to be highly efficient and inspiring. Its simple syntax has helped me approach textual programming and develop my skills in this field, besides providing a suitable language for the development of the libretto for *Echo and Narcissus*. Its vast volume of native and external modules for various tasks, whether related to music or not, have both supported and inspired my research. Apart from the Pyo module, which is the central Python module I have used, an array of modules for various functionalities has been available, all within the same software. From network communication with the OSC modules – either between different machines or within the same computer –, through

Data Mining, prosody analysis, geolocation, to Machine Learning, both technical and conceptual aspects of my work have been supported by the use of Python.

5.1. Contribution to Knowledge

The contribution of this research is twofold. On one hand, this research has highlighted the use of Python in creative coding practices, a programming language that is rarely used in such fields. On the other hand, it has promoted an unconventional approach to composition and the various source materials. The practical outcome of this research aims to contribute to knowledge by acting as a series of examples for mixed media art, with a focus on the code's conceptual perspective, rather than its purely technical. The works created can be thought of as immutable artefacts, in the sense that, even though new performances of them may have differences compared to archived ones (arguably with the exception of *Echo and Narcissus*), the overall outcome will fall under a certain concept and will bear similar aesthetics. Therefore, the code of this research does not apply to other works, intact. Nevertheless, the three works of this research provide both conceptual and technical examples for mixed media art.

From a technical perspective these works are valuable as examples of the use of Python as the central, if not the only tool. The use of Pyo, a very efficient and intuitive Python DSP module, is highlighted throughout the research. *Echo and Narcissus* and *Latitude* demonstrate a rich set of audio and control processes available in Python through Pyo, together with other Python-enabled technologies, including networked communication and multi-process spawning.

Apart from these strictly technical capabilities, the portfolio takes advantage of other strong points of the Python language. *Latitude* and *Data Mining / Live Scoring* make extensive use of various features not related to music, such as geolocating and tweet mining. In *Echo and Narcissus* I exploited a subtler characteristic of Python to adapt the libretto to this language, that of its syntax simplicity and extended use of English words. Utilising a Pythonic approach to programming has opened new creative spaces that domain specific languages do not offer. While the latter offer the typical functionality that musicians require for computer music, exploring a generic programming language simultaneously creates opportunities and imposes constraints, thus allowing new and novel approaches to creative musical and inter-disciplinary practices to be explored. The Pythonic elements mentioned in this paragraph that fall outside the strictly technical scope, bridge the gap between the technical and conceptual strands of the works of this research.

By creating mixed media musical works that introduce elements reaching far beyond common music practices, I aimed to produce a kind of a conceptual model for artistic creation. Implementing the extra-musical Pythonic features in my works functions as a kind of blueprint for using a multitude of extra-musical sources, and highlights the ways in which these can become the central focus of an artwork's concept. I could also refer to the various sound textures of *Echo and Narcissus* and *Latitude* that were made possible using the Pyo module, but I perceive this to be highly subjective. On the other hand, unconventionality can be expressed objectively by employing techniques that are not commonly used in a certain field. Therefore I prefer to focus on the conceptual side of these works and not its aesthetic one.

This research aims to contribute to knowledge by giving insight into the realisation process and the source code of these works, as a source of inspiration to other artists. One does not need to be a Python programmer to conceive original ideas for musical or other artworks. In this sense, the contribution of the twofold nature of this research – technical and conceptual – is at the same time combined and separable.

5.2. Future Work

By realising the works of this portfolio I have become confident in the use of Python in music contexts. I have already submitted proposals to festivals and institutions to realise two future works based on Python and various techniques used in this research. Both proposals are based on Twitter and Machine Learning. The first includes the 3dPdModular system and the Code module. It is a one-act opera for soprano and modular synthesiser, where the tweets will constitute the libretto and their sentiment analysis will determine the character of the music, as this will be produced by various modules controlled by a Machine Listening agent. This project will bring together the three works of this research, as it is an opera based on Twitter, where the music is produced by the 3dPdModular system, controlled by the Code module.

The second proposed work is a dynamic prose based on an ancient Greek tragedy that will be created by chatbots. This prose will be hosted on Twitter and the chatbots will interact with other Twitter users and exchange ideas on the text of the original play. This work will be accompanied by an in-situ sound installation where the tweets will be read out loud with Text-To-Speech synthesis and will be accompanied by electronic music, controlled by a Machine Listening agent, again based on sentiment analysis.

Both proposals are grounded on more extended concepts than this simple description, each connected to a certain theme. It is beyond the scope of this chapter to analyse these concepts, therefore I provide only a minimal outline to demonstrate the impact that the present research is having on my artistic development. To conclude this chapter and the thesis itself, I would like to include the last line of the libretto of *Echo and Narcissus*:

```
os.system('poweroff')
```

Appendices

A USB drive accompanies this thesis with video documentation and the code of each of the three works of this research. These can also be accessed online. Links are provided below.

Appendix I – *Echo and Narcissus*

Live coding poetry opera by artist group Medea Electronique.

Katerina Maniou – soprano

Marios Sarantidis – bass baritone

Video documentation of the opera premiere can be viewed at the following link:

https://drive.google.com/file/d/1ddDUtv7o8x4ciASvbQpVwoOXxf5ayX17/view?usp=sharing

The video of the premiere has a better view of the live writing of the libretto, but poor sound quality. For a better sound quality, a multi-camera documentation of the general rehearsal is available at the following link:

https://www.youtube.com/watch?v=1Btt4am2S2k&ab_channel=OnassisFoundation

The code of the opera is located at the following link:

https://drive.google.com/drive/folders/1zQKz-rJHopok779FJVDD_jV16GelHPuI?usp=sharing

Appendix II – *Data Mining / Live Scoring*

Live algorithmic composition for acoustic ensemble.

Nicoleta Chatzopoulou – music library composer

ARTéfacts Ensemble:

Spyros Tzekos – Clarinet

Guido De Flaviis – Saxophones

Kostas Seremetis – Percussion

Thodoris Vazakas – Percussion

Evgenios Zhibaj – Violin

Ilias Sdoukos – Viola

Louizos Aslanidis – Direction, Visuals

Video documentation of the general rehearsal is available at the following link:

https://drive.google.com/file/d/166QdTgD7TzW9fGNL-g66859NVX9fVL7y/view?usp=sharing

The code of this project is located at the following link:

https://drive.google.com/drive/folders/1IFKMco467HmbxgELbYtks44LkYLSdZhz?usp=sharing

Appendix III – Code module for the 3dPdModular system

Video documentation of the performance of *Latitude* realised with the Code module and the 3dPdModular system is at the following link:

https://drive.google.com/file/d/1I6YJW9iivxJBRU75DD6P0PQ0XRbK9pc3/view?usp=sharing

The code of the module is located at the following link:

https://drive.google.com/drive/folders/1b5Hfw2CVk-Bnv3qObtAY7ph4mNmWJeqf?usp=sharing

Bibliography

Aaron, S. and Blackwell, A.F. (2013) ‘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*. New York, NY, USA: Association for Computing Machinery (FARM ’13), pp. 35–46. doi:10.1145/2505341.2505346.

Agostini, A. and Ghisi, D. (2013) ‘Real-Time Computer-Aided Composition with bach’, *Contemporary Music Review*, 32. doi:10.1080/07494467.2013.774221.

Alt, F. *et al.* (2010) ‘Creating Meaningful Melodies from Text Messages’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Sydney, Australia, pp. 63–68. doi:10.5281/zenodo.1177713.

Alvarez, C.I.D.L. (2017) ‘The Intrinsic Mutability of Code Poetry Uncovers New Notions of Poetic Design’, *Philippine Humanities Review*, 19(1). Available at: https://journals.upd.edu.ph/index.php/phr/issue/view/630.

Ames, C. (1981) ‘Protocol: Motivation, Design, and Implementation of a Computer-Assisted Composition for Solo Piano’, in *Proceedings of the 1981 International Computer Music Conference, ICMC*.

Ash, K. (2012) ‘Affective States: Analysis and Sonification of Twitter Music Trends’, in *Proceedings of the International Conference on Auditory Display, ICAD*. Atlanta, GA, USA, pp. 257–259.

Ash, K. and Stavropoulos, N. (2011) ‘Livecell: Real-Time Score Generation Through Interactive Generative Composition’, in *Proceedings of the International Computer Music Conference, ICMC.* Huddersfield, UK.

Baird, K.C. (2005) ‘Real-Time Generation of Music Notation via Audience Interaction Using Python and GNU Lilypond’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Vancouver, BC, Canada, pp. 240–241. doi:10.5281/zenodo.1176695.

Bela (2021) ‘Bela Salt’. Available at: https://shop.bela.io/collections/modular/products/salt.

Bélanger, O. (2016) ‘Pyo, the Python DSP Toolbox’, in *Proceedings of the 24th ACM International Conference on Multimedia*. New York, NY, USA: Association for Computing Machinery (MM ’16), pp. 1214–1217. doi:10.1145/2964284.2973804.

Boren, B. *et al.* (2014) ‘I Hear NY4D: Hybrid Acoustic and Augmented Auditory Display for Urban Soundscapes’, in *Proceedings of the International Conference on Auditory Display, ICAD*. New York, USA.

Bourotte, R. and Kanach, S. (2019) ‘UPISketch: The UPIC idea and its current applications for initiating new audiences to music’, *Organised Sound*, 24(3), pp. 252–260. doi:10.1017/S1355771819000323.

Bresson, J. (2014) ‘Reactive Visual Programs for Computer-Aided Music Composition’, in *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pp. 141–144. doi:10.1109/VLHCC.2014.6883037.

Buongiorno Nardelli, M. (2015) ‘materialssoundmusic: a Computer-Aided Data-Driven Composition Environment for the Sonification and Dramatization of Scientific Data Streams’, *Journal of Scientific Conference Proceedings* [Preprint].

Candy, L. and Edmonds, E. (2018) ‘Practice-Based Research in the Creative Arts: Foundations and Futures from the Front Line’, *Leonardo*, 51(1), pp. 63–69. doi:10.1162/LEON_a_01471.

Carlson, K. (2009) ‘How Prosody Influences Sentence Comprehension’, *Language and Linguistics Compass*, 3, pp. 1188–1200. doi:10.1111/j.1749-818X.2009.00150.x.

Cherston, J. *et al.* (2016) ‘Musician and Mega-Machine: Compositions Driven by Real-Time Particle Collision Data from the ATLAS Detector’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Brisbane, Australia: Queensland Conservatorium Griffith University, pp. 78–83. doi:10.5281/zenodo.1176012.

Coduys, T. and Ferry, G. (2004) ‘Iannix. Aesthetical/symbolic visualisations for hypermedia composition’, in *Proceedings of the Sound and Music Computing Conference*. Paris, France.

Crabtree, B. and Kelli, C. (2021) ‘Teletype’. Available at: https://monome.org/docs/teletype/.

Dahl, L., Herrera, J. and Wilkerson, C. (2011) ‘TweetDreams : Making Music with the Audience and the World using Real-time Twitter Data’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Oslo, Norway, pp. 272–275. doi:10.5281/zenodo.1177991.

Daudin, C. *et al.* (2009) ‘The Guido Engine a toolbox for music scores rendering’, in *Proceedings of the 2021 Linux Audio Conference, LAC*. Parma, Italy.

De Pra, Y. and Fontana, F. (2020) ‘Programming Real-Time Sound in Python’, *Applied Sciences*, 10, p. 4214. doi:10.3390/app10124214.

Drymonitis, A. (2015) ‘Introduction to Arduino’, in *Digital Electronics for Musicians*. Berkeley, CA: Apress, pp. 51–96. doi:10.1007/978-1-4842-1583-8_2.

Eacott, J. (2011) ‘Flood Tide See Further: Sonification as Musical Performance’, in *Proceedings of the International Computer Music Conference, ICMC*. Huddersfield, UK, pp. 69–74.

Eaton, J., Jin, W. and Miranda, E. (2014) ‘The Space Between Us. A Live Performance with Musical Score Generated via Emotional Levels Measured in EEG of One Performer and an Audience Member’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. London, United Kingdom: Goldsmiths, University of London, pp. 593–596. doi:10.5281/zenodo.1178756.

Egozy, E. and Lee, E.Y. (2018) ‘*12*: Mobile Phone-Based Audience Participation in a Chamber Music Performance’, in Luke Dahl, T.M., Douglas Bowman (ed.) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Blacksburg, Virginia, USA: Virginia Tech, pp. 7–12. doi:10.5281/zenodo.1302655.

Elblaus, L., Falkenberg, K. and Unander-Scharin, C. (2011) ‘EXPLORING THE DESIGN SPACE: PROTOTYPING “THE THROAT V3” FOR THE ELEPHANT MAN OPERA’, in *Proceedings of the SMC 2011 - 8th Sound and Music Computing Conference*. Padova, Italy.

Endo, A., Moriyama, T. and Kuhara, Y. (2012) ‘Tweet Harp: Laser Harp Generating Voice and Text of Real-time Tweets in Twitter’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Ann Arbor, Michigan: University of Michigan. doi:10.5281/zenodo.1178249.

Eriksson, J. (2019) *Automatonism: towards dynamic macro-structure in generative music for modular synthesisers.* PhD Thesis. Birmingham City University.

Fober, D., Orlarey, Y. and Letz, S. (2012) ‘INScore An Environment for the Design of Live Music Scores’, in *Proceedings of the 2021 Linux Audio Conference, LAC*. California, USA.

Forero, J. 2021. ‘Code, Poetry and Freedom’, in *Proceedings of the 9th Conference on Computation, Communication, Aesthetics & X,* 261–77.

Funk, T. (2018) ‘A Musical Suite Composed by an Electronic Brain: Reexamining the Illiac Suite and the Legacy of Lejaren A. Hiller Jr.’, *Leonardo Music Journal*, 28, pp. 19–24. doi:10.1162/lmj_a_01037.

Gardner, R. (1971) *Notes*, 28(2), pp. 304–304.

Ghose, S., and Prevost, J. J. (2020). ‘Autofoley: Artificial Synthesis of Synchronized Sound Tracks for Silent Videos with Deep Learning’, *IEEE Transactions on Multimedia*, 1(1). doi: 10.1109/TMM.2020.3005033

Gimenes, M., Largeron, P.-E. and Miranda, E. (2016) ‘Frontiers: Expanding Musical Imagination With Audience Participation’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Brisbane, Australia: Queensland Conservatorium Griffith University, pp. 350–354. doi:10.5281/zenodo.1176020.

Haddad, D. D., and Paradiso, J. (2019) ‘The World Wide Web in an Analog Patchbay.’ in Marcelo Queiroz and Anna Xambó Sedó (eds.) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Porto Alegre, Brazil: UFRGS. pp. 407–410. doi:10.5281/zenodo.3673013.

Hajdu, G. and Didkovsky, N. (2012) ‘Maxscore - Current state of the art’, in *Proceedings of the International Computer Music Conference, ICMC*. Ljubljana, Slovenia, pp. 156–162.

Hermann, T. *et al.* (2012) ‘Tweetscapes – Real-time Sonification of Twitter Data Streams for Radio Broadcasting’, in *Proceedings of the International Conference on Auditory Display, ICAD*. Atlanta, GA, USA, pp. 113–120.

Hödl, O. *et al.* (2017) ‘Design Implications for Technology-Mediated Audience Participation in Live Music’, in *Proceedings of the 14th Sound and Music Computing Conference*. Espoo, Finland.

Hödl, O., Kayali, F. and Fitzpatrick, G. (2012) ‘Designing Interactive Audience Participation Using Smart Phones in a Musical Performance’, in *Proceedings of the International Computer Music Conference, ICMC*. Ljubljana, Slovenia, pp. 236–241.

Holden, D. and Kerr, C. (2016) *./code –poetry*. Available at: https://code-poetry.com/.

Hoos, H. *et al.* (1998) ‘The GUIDO Notation Format a Novel Approach for Adequately Representing Score-Level Music’, in *Proceedings of the 1998 International Computer Music Conference, ICMC*. Ann Arbor, Michigan.

Hopkins, S. (1992) ‘Camels and Needles: Computer Poetry Meets the Perl Programming Language’, in *Proceedings of the USENIX Winter 1992 Technical Conference*. San Francisco, USA, pp. 391-404.

Hutchins, C. *et al.* (2014) ‘Soundbeam: A Platform for Sonyfing Web Tracking’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. London, United Kingdom: Goldsmiths, University of London, pp. 497–498. doi:10.5281/zenodo.1178810.

Hutchins, C.C. (2015) ‘Live Patch / Live Code’, in *Proceedings of the First International Conference on Live Coding*. Leeds, UK: ICSRiM, University of Leeds, pp. 147–151. doi:10.5281/zenodo.19346.

Jessop, E., Torpey, P.A. and Bloomberg, B. (2011) ‘Music and Technology in Death and the Powers’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Oslo, Norway, pp. 349–354. doi:10.5281/zenodo.1178051.

Kelly, E. (2011) ‘Gemnotes: A Realtime Music Notation System for Pure Data’, in *Proceedings of the 2nd International Pd Convention*. Weimar, Germany.

Kiritchenko, S. and Mohammad, M.S. (2016) ‘Happy Accident: A Sentiment Composition Lexicon for Opposing Polarities Phrases’, in *Proceedings of the 10th edition of the Language Resources and Evaluation Conference (LREC)*. Portorož, Slovenia, pp. 1157–1164.

Kirkbride, R. (2021) *FoxDot*. Available at: https://foxdot.org/.

Knees, P., Pohle, T. and Widmer, G. (2008). sound/tracks: Real-Time Synaesthetic Sonification of Train Journeys. In *Proceedings of the 16th ACM international conference on Multimedia (MM ’08)* New York, USA. pp. 1117-1118.

Koutsomichalis, M., and Gambäck, B. (2018). ‘Algorithmic Audio Mashups and Synthetic Soundscapes Employing Evolvable Media Repositories’, in *Proceedings of the International Conference on Computational Creativity.* Salamance, Spain.

Lazzarini, V. *et al.* (2016) *Csound*. Springer. Doi:10.1007/978-3-319-45370-5.

Lin, H.-Y., Lin, Y.-T., Tien, M.-C., and Wu, J.-L. (2009). ‘Music Paste: Concatenating Music Clips Based on Chroma and Rhythm Features’, in *Proceedings of the 10th International Society for Music Information Retrieval Conference*. Kobe, Japan: ISMIR. pp. 213-218. doi: 10.5281/zenodo.1415758

Lindborg, P. (2017) ‘Pacific Belltower, a sculptural sound installation for live sonification of earthquake data’, in *Proceedings of the International Computer Music Conference, ICMC*. Shanghai, China.

Liu, B. and Zhang, L. (2012) *A Survey of Opinion Mining and Sentiment Analysis*. Boston, USA: Springer.

Louzeiro, P. (2018) ‘The Comprovisador’s Real-Time Notation Interface (Extended Version): 13th International Symposium, CMMR 2017, Matosinhos, Portugal, September 25-28, 2017, Revised Selected Papers’, in, pp. 489–508. doi:10.1007/978-3-030-01692-0_33.

Lutz, M. (2010) *Programming Python.* 4th edn. Sebastopol, CA: O’Reilly.

Magnusson, T. (2011) ‘The IXI Lang: A SuperCollider Parasite for Live Coding’, in *Proceedings of the International Computer Music Conference, ICMC*. Huddersfield, UK.

Manaris, B. *et al.* (2018) ‘JythonMusic: An Environment for Developing Interactive Music Systems’, in Luke Dahl, T.M., Douglas Bowman (ed.) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Blacksburg, Virginia, USA: Virginia Tech, pp. 259–262. doi:10.5281/zenodo.1302575.

Marino, G., Serra, M.-H. and Raczinski, J.-M. (1993) ‘The UPIC System: Origins and Innovations’, *Perspectives of New Music*, 31(1), pp. 258–269.

Mayton, B., Gershon D., Nicholas J., and Paradiso, J. (2012) ‘Patchwork: Multi-User Network Control of a Massive Modular Synthesizer.’ in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Ann Arbor, Michigan: University of Michigan. doi:10.5281/zenodo.1178345.

McCartney, J. (1996) ‘SuperCollider, a New Real Time Synthesis Language’, in *Proceedings of the International Computer Music Conference, ICMC*. Hong Kong, China, pp. 257–258.

McKell, C. (2016) ‘Sonification of Optically-Ordered Brownian Motion’, in *Proceedings of the International Computer Music Conference, ICMC*. Utrecht, The Netherlands, p. 524.

McKenzie, D. (1991) *Notes*, 47(3), pp. 962–963.

McLean, A. (2021) *Tidal Cycles*. Available at: https://tidalcycles.org/docs/.

Mercer-Taylor, A. and Altosaar, J. (2015) ‘Sonification of Fish Movement Using Pitch Mesh Pairs’, in Berdahl, E. and Allison, J. (eds) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Baton Rouge, Louisiana, USA: Louisiana State University, pp. 28–29. doi:10.5281/zenodo.1179138.

Michel van der Aa (2016) ‘Blank Out’. Available at: https://www.vanderaa.net/.

Mikalauskas, C. *et al.* (2018) ‘Improvising with an Audience-Controlled Robot Performer’, in, pp. 657–666. doi:10.1145/3196709.3196757.

Mikulska, M. (1981) ‘Some Remarks on Computer-Assisted Composition’, in *Proceedings of the 1981 International Computer Music Conference, ICMC*.

Morawitz, F. (2016) ‘Molecular Sonification of Nuclear Magnetic Resonance Data as a Novel Tool for Sound Creation’, in *Proceedings of the 2017 International Computer Music Conference, ICMC*. Utrecht, The Netherlands, p. 524.

Morgan-Mar, D. (2018) *Piet*. Available at: https://www.dangermouse.net/esoteric/piet.html.

Neuhaus, M. (2000) ‘The broadcast works and audium’.

Nienhuys, H.-W. and Nieuwenhuizen, J. (2003) ‘Lilypond, a System for Automated Music Engraving’, in *In Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*. Firenze, Italy.

Oliverio, J. and Pair, J. (1996) ‘Design and Implementation of a Multimedia Opera’, in *Proceedings of the International Computer Music Conference, ICMC*. Hong Kong, China, pp. 202–203.

Park, S. *et al.* (2010) ‘Online Map Interface for Creative and Interactive Music Making’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Sydney, Australia, pp. 331–334. doi:10.5281/zenodo.1177877.

Poonguzhali, R. *et al.* (2018) ‘Sentiment Analysis on LinkedIn Comments’, *International Journal of Engineering Research & Technology IJERT (ICONNECT)*, 6(7), p. 415.

Puckette, M. (1997) ‘Pure Data’, in *Proceedings of the International Computer Music Conference, ICMC*. Thessaloniki, Greece, pp. 37–41.

Puckette, M. (2004) ‘A divide between “compositional” and “performative” aspects of Pd’, in *Proceedings of the 1st International Pd Convention*. Graz, Austria.

Pulido-Prieto, O. and Juárez-Martínez, U. (2017) ‘A Survey of Naturalistic Programming Technologies’, 50(5). doi:10.1145/3109481.

QuBit Electronix (2021) *QuBit Nebulae*. Available at: https://www.qubitelectronix.com/shop/nebulae.

Ramstrum, M. and Lemouton, S. (2003) ‘Realtime Performance Strategies for the Electronic Opera K…’, in *Proceedings of the International Computer Music Conference, ICMC*. Singapore.

Rebel Technology (2018) *OWL Modular*. Available at: https://www.rebeltech.org/product/owl-modular/.

Ritchie, D. et al. (1981) ‘C PROGRAMMING LANGUAGE.’, *Western Electric engineer*, 25, pp. 14–29.

Rohrhuber, J. and de Campo, A. (2011) ‘Just-in-Time Programming’, in Wilson, S., Cottle, D., and Collins, N. (eds) *The SuperCollider Book*. Cambridge, Massachusetts: MIT Press, pp. 207–236.

Samaruga, L., Silvani, D. and Saladino, I. (2021) ‘SuperCollider library for Python’. Available at: https://github.com/smrg-lm/sc3.

Schoon, A. and Dombois, F. (2009) ‘Sonification in Music’, in *Proceedings of the International Conference on Auditory Display, ICAD*. Copenhagen, Denmark.

Smith, R. R., and Lawson, S. (2016) ‘Closing the Circuit: Live Coding the Modular Synth.’ in *Second International Conference on Live Coding (ICLC)*. Hamilton, Canada.

Straebel, V. and Thoben, W. (2014) ‘Alvin Lucier’s Music for Solo Performer: Experimental music beyond sonification.’, *Organised Sound*, 19(1), pp. 17–29.

Suchánek, J. (2020) ‘SOIL CHOIR v.1.3 - soil moisture sonification installation’, in Michon, R. and Schroeder, F. (eds) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Birmingham, UK: Birmingham City University, pp. 617–618. doi:10.5281/zenodo.4813226.

Thompson, W., Russo, F. and Sinclair, D. (1994) ‘Effects of Underscoring on the Perception of Closure in Filmed Events’, *Psychomusicology: A Journal of Research in Music Cognition*, 13. doi:10.1037/h0094103.

Tin Men and the Telephone (2020) Available at: https://tinmenandthetelephone.com/

Tome, B. et al. (2015) ‘MMODM: Massively Multipler Online Drum Machine’, in Berdahl, E. and Allison, J. (eds) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Baton Rouge, Louisiana, USA: Louisiana State University, pp. 285–288. doi:10.5281/zenodo.1179184.

Toussaint, G. (2005) ‘The Euclidean Algorithm Generates Traditional Musical Rhythms’, in *Proceedings of BRIDGES: Mathematical Connections in Art, Music, and Science*. Banff, Alberta, Canada, pp. 47–56.

Turchet, L. and Barthet, M. (2018) ‘Demo of interactions between a performer playing a Smart Mandolin and audience members using Musical Haptic Wearables’, in Luke Dahl, T.M., Douglas Bowman (ed.) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Blacksburg, Virginia, USA: Virginia Tech, pp. 82–83. doi:10.5281/zenodo.1302687.

Unander-Scharin, C. *et al.* (2014) ‘The Vocal Chorder’, *Interactions*, 21(6), pp. 14–15. doi:10.1145/2662967.

Unander-Scharin, C., Höök, K. and Elblaus, L. (2013) ‘The Throat III: Disforming Operatic Voices through a Novel Interactive Instrument’, in *CHI ’13 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery (CHI EA ’13), pp. 3007–3010. doi:10.1145/2468356.2479596.

Wakefield, G., Smith, W. and Roberts, C. (2010) ‘LuaAV: Extensibility and Heterogeneity for Audiovisual Computing’, in *Proceedings of the Linux Audio Conference, LAC*. Utrecht, The Netherlands.

Wang, G. and Cook, P. (2003) ‘ChucK: A Concurrent, On-the-fly Audio Programming Language’, in *Proceedings of the International Computer Music Conference, ICMC*. Singapore.

White, A. (2019) ‘ANALOG ALGORITHMS: GENERATIVE COMPOSITION IN MODULAR SYNTHESIS’, in *Proceedings of the ACMC 2019*. Melbourne, Australia: Melbourne, Monash University, pp. 68–73.

Xambó, A. and Roma, G. (2020) ‘Performing Audiences: Composition Strategies for Network Music using Mobile Phones’, in Michon, R. and Schroeder, F. (eds) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Birmingham, UK: Birmingham City University, pp. 55–60. doi:10.5281/zenodo.4813192.

Xenakis, I. (1972) *Formalized music, thoughts and mathematics in composition*. Bloomington: Indiana University Press.

Zhang, L., Wu, Y. and Barthet, M. (2016) ‘A Web Application for Audience Participation in Live Music Performance: The Open Symphony Use Case’, in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Brisbane, Australia: Queensland Conservatorium Griffith University, pp. 170–175. doi:10.5281/zenodo.1176147.

zmölnig, Io. and Eckel, G. (2007) ‘Live coding: An overview’, in *Proceedings of the International Computer Music Conference, ICMC*. Copenhagen, Denmark, pp. 295–298.