The Algorithmic Composition Explorer

Using a learning environment to enable users to experience intrinsic feedback on musical algorithms

Stephen Gardener

Master's programme in Music, Communication and Technology

Department of Music

Norwegian University of Science and Technology

Department of Musicology

University of Oslo

Spring 2022

Abstract

This thesis proposes a novel design for an interactive system that introduces people to algorithmic composition. There are many libraries and environments for algorithmic music, with little consistency in terms of programming language and software framework. For those that have little or no programming knowledge, or those that just want to get a tactile understanding and quick feel of a selection of different algorithmic approaches, there are few options. To use most existing algorithmic composition tools users are still required to carry out some implementation or integration work, which also require some expertise in computer music frameworks. This thesis proposes a system that attempts to fill that gap.

The primary aim is to build a system that takes a learning through practice approach, allowing learners to get an understanding of how a particular algorithm works by experimenting with its parameters and listening to the results. If a lack of technical knowledge should not be a barrier to the use of the tool, the system needs to be widely available, and not requiring of any complex installation or setup. Finally, the system is designed such that it can be built upon and expanded, and released as open-source software. There are limits on what can be built as part of a thesis such as this, so the design must provide a framework for future expansion.

Acknowledgements

I would like to thank my supervisor Stefano Fasciani for all of his help, support and advice throughout this course, for pointing me in lots of interesting directions, and for being so responsive to all of my questions. Thanks also to my fellow MCT students who made this course, and the whole MCT programme, a lovely experience. Finally, huge thanks to my wife Olaug and my son Emil for being patient with me during the long coding and writing sessions of the past few months. Yes Emi, we can play Minecraft together again now.

I dedicate this work to our brilliant dog Chilli, who passed away during the writing of this thesis.

Thesis links

The Algorithmic Composition Explorer: https://algorithmic-composition-explorer.com

Source code: https://github.com/stega/algorithmic_composition_explorer

Blog post: https://mct-master.github.io/masters-thesis/2022/05/15/stephedg-algorithmic-composition-explorer.html

1. Introduction 1
- 1.1 Aims 2
- 1.2 Target Audience 2
- 1.3 Key Contributions 3
- 1.4 A Short Note About Computers 3
- 1.5 Structure & Overview 3

2. Background 5
- 2.1 A Brief Introduction to Algorithmic Composition 5
- 2.2 An Overview of Algorithmic Composition Key Techniques 6
- 2.3 Algorithmic Composition Pedagogy 10
  - 2.3.1 Learning through practice 10
  - 2.3.2 Extrinsic and intrinsic feedback 11
  - 2.2.3 Computational thinking 12
- 2.4 Related works 13

3. Design & Methodology 17
- 3.1 Languages & Frameworks 17
  - 3.1.1 Python 18
  - 3.1.2 Javascript & the Web 19
- 3.2 Computer representations of music 22
  - 3.2.1 Representation of pitch 23
  - 3.2.2 Music engraving 23
- 3.3 The overall design of the system 24
- 3.4 Algorithm selection 25

4. Implementation 28
- 4.1 Building the Algorithmic Composition Explorer 28
- 4.2 Selected Algorithms 33
  - 4.2.1 Guido 33
  - 4.2.2 Musikalisches Würfelspiel 37
  - 4.2.3 Markov Chains 41
  - 4.2.4 Tintinnabuli 44
- 4.3 Extending the Algorithmic Composition Explorer 46
  - 4.3.1 Adding a new algorithm 47
  - 4.3.2 Extending the Score class 49
  - 4.3.3 Adding a New Instrument 50

5. Evaluation 51

5.1 The User Interface 51
5.2 The Algorithms 51
5.3 Browser compatibility 53
5.4 Mobile performance 54
5.5 Website performance 55
6. Conclusion 58
References 60

1. Introduction

Algorithmic composition has a long and rich history, and encompasses a wide range of approaches and techniques. It can been defined simply as the application of formal rules to generate musical material and ideas.

While it is a well known tool for creating new music and ideas, it can also be a valuable approach in teaching music and composition. Being able to describe music in terms of algorithms can contribute to a learners understanding, helping focus their attention on the structure and organisation of that music. The application of algorithmic thinking to music can be used to understand form and style. And algorithmic composition can be used as a vehicle for concept development, with the algorithms producing raw material that is then edited and assigned musical meaning by the learner (Falthin, 2012).

Like algorithmic composition itself, the use of algorithmic composition techniques in music education has a long history, and has been dated back to Guido around 1000 CE (Neirhaus, 2009). Modern educational approaches tend to focus on learning to code, as this offers a good opportunity to delve deeply into different algorithmic approaches, as well as develop new ones. For those that want just a taste of what algorithmic composition can offer, or those that want to get an understanding of a selection of approaches in order to compare the results that can be expected, there are less options available. Comparing the output of different approaches would require bringing together and setting up multiple tools, which would be difficult and potentially off-putting for a beginner. Certainly, some of the programming environments could be set up to be used in this way, but even setting up a programming environment can be challenging to new students. What is needed is a tool that is accessible to anyone, without the need for installing or configuring software, and without the need for in-depth technical knowledge. Such a tool would allow the learner to listen to a selection of algorithms and compare their output. It would also offer the ability to manipulate selected parameters of the algorithms, allowing the learner to gain a deeper understanding of how that algorithm works. Ideally, it would also allow the user to take any generated ideas with them, so they can incorporate them into their own work. Finally, the tool should be expandable. By building a solid foundation that can be built upon, the tool can grow, having new algorithms incorporated over time.

1.1 Aims

This thesis proposes and develops a proof-of-principle system for introducing a selection of algorithmic composition approaches to learners that have little or no technical background. The system - the Algorithmic Composition Explorer - has the following attributes:

- It is implemented using ubiquitous, standards-compliant technologies that allow it to have the greatest reach, while requiring little previous technical knowledge on the part of the user.
- It adopts a “learning through practice” approach, allowing users to manipulate and compare different algorithmic approaches, offering a visual representation of the generated music as well as audio playback.
- It allows users to take away their generated musical ideas in the form of MIDI scores and downloadable music notation.
- It is open-source and ready to be built upon and expanded with more algorithms, export formats, and visual representations of the music.

1.2 Target Audience

It has already been stated that a lack of technical knowledge should not be a barrier to the use of the tool. This should not mean to exclude those that are more technically inclined however - setting up and experimenting with existing solutions would still require some time and effort even for those that are more technically competent. If someone wants to experiment with and compare a variety of algorithms with minimal effort, they can be considered as part of the target audience.

There is also the question of musical knowledge. If a system deals with aspects of music theory and notation, there is an assumption that a certain musical background is required. While true to a lesser extent, the aim is to build an environment where learning happens via the interaction with the system. If intrinsic motivation and feedback is designed into the system, it should be enough to try things out and observe the results.

This is a key element to building an experiential learning environment, and will be discussed more in Chapter 2.3.

1.3 Key Contributions

The key contributions of this thesis are the following:

- The introduction of a novel approach to building an experiential learning environment for algorithmic music composition.
- The design and implementation of a widely accessible open-source tool for learning about and experimenting with algorithmic composition approaches.
- The establishment of an open framework that can be further built upon and expanded.

1.4 A Short Note About Computers

While this thesis proposes a software system, it's important to note that there are many historical and contemporary algorithmic composition techniques that do not require the use of a computer. As one of the aims of this paper is to open up the world of algorithmic composition to those with little technical knowledge, it is important to convey that it is not necessary to be a accomplished programmer or have an advanced understanding of mathematics in order to apply algorithmic thinking and techniques to composition. The application of algorithmic thinking to music is an interesting and effective discipline, regardless of the involvement of computers and complex mathematical theories.

1.5 Structure & Overview

Chapter 2 presents an introduction to algorithmic composition, its history and the pedagogical approaches related to it. The introduction also looks at related works, how they compare to this thesis's proposal, and how the Algorithmic Composition Explorer builds on and extends these current approaches. In Chapter 3, the proposed system design is outlined, and an overview of possible approaches and tools is detailed. How

music can be represented digitally is also covered, as well as options for rendering of notation. Finally, the criteria for choosing the algorithms are explained, along with the algorithms that were finally chosen. Chapter 4 delves into the implementation details, of the software overall, as well as how the specific algorithms were tackled. The ability to build on and extend the software is an important part of this proposal, and so the chapter ends with how new algorithms, instruments and other components can be added to the system. Finally, Chapter 5 includes discussion on the various aspects of the project that worked, and challenges that are yet to be addressed, for both the individual algorithms, and the system as a whole.

2. Background

2.1 A Brief Introduction to Algorithmic Composition

Algorithmic composition has a long and rich history, and encompasses a wide range of approaches and techniques. At its core, algorithmic composition is a method of music making in which music is generated based on a set of rules that the composer has defined (Sweet, 2014). It is these sets of rules - a sequence of instructions - that is the algorithm, the term being adopted from the field of computer science around the halfway mark of the 20th century (Burns, 1997). Indeed, algorithmic composition is often associated with the rise of the digital computer, but manual and analog musical algorithms have a long history (McLean & Dean, 2018).

Some of the earliest known compositional algorithms date back to the eleventh century, with Guido of Arezzo, a music educator and Benedictine monk, being an early pioneer. Over the following thousand years, many more compositional systems were developed, from games of chance using dice to arrange, combine, and compose both lyrics and musical work, to a focus on combinatorial arithmetics and music's close affinity with numbers.

The digital computer was first put to compositional use in the 1950s, but many of the pre-digital algorithms were equally applicable - the general assumption being that if the rules of a task can be formalised, they can also be written into hardware such as pin cylinders and punch cards (Magnusson, 2019).

This formalisation brings up an important point regarding the teaching of composition. In the 1680s, Leibniz argued that the rules for composition are so well defined that anyone can be instructed to compose following certain rule sets. Cope (2015) goes so far to say that all composers use algorithms while composing whether they are aware of doing so or not - "all composers are algorithmic composers" (Cope, 2015).

If we are all, consciously or subconsciously, composing with algorithms, it would make sense that there would be almost as many different approaches to algorithmic composition as there are composers, and that classifying them into "methods" or "schools of thought" is going to be a difficult undertaking. In addition, a composer may employ a

variety of approaches to implementing algorithms and use a combination of different techniques in a single piece of music (Manzo, 2021). But as Manzo (2021) wrote, “a good idea for a composition is a good idea for a composition, and a conversation about the mechanics of realising these ideas through algorithms can be interesting and enriching.”

2.2 An Overview of Algorithmic Composition Key Techniques

This section will offer a very brief history of algorithmic composition, and an overview some of the most important techniques and methods. Algorithmic composition can be applied to music and sound at many levels. While there have been multiplicity of interesting approaches that work with timbre, instrumentation and sonification, this thesis will be focused on the generation of notes - of pitch and rhythm. This overview of techniques will reflect that bias.

It has already been mentioned that algorithmic composition can be traced back to Guido of Arezzo in the eleventh century. Guido’s vowel to pitch algorithm is the first technique implemented in the Algorithmic Composition Explorer, and will be looked at in more detail in Chapter 4.2.1.

In the fourteenth century, composers of Ars Nova started to introduce new rules around rhythm and pitch. Isorhythm is where a rhythmic sequence, called the *talea*, is mapped onto a pitch sequence, called the *colores*. This technique is called *isorhythm*, and is implemented as a component in the Algorithmic Composition Explorer, as can be seen in Chapter 4.2.4.

Music evolved in the Renaissance and into the Baroque, with the rise of counterpoint and polyphonic forms such as the canon and the fugue. The strict rules and algorithmic nature of the styles of the time make them good candidates for automated composition, using rules-based algorithmic approaches, in computer aided composition or using techniques such as genetic algorithms (Acevedo, 2005).

During the classical period, a technique known as *Ars Combinatoria* became very popular. This approach of composing music using chance to arrange pre-written melodies

has been attributed to Mozart, and is discussed in more detail in the chapter on Musikalisches Würfelspiel in Chapter 4.2.2.

In the twentieth century, Arnold Schoenberg built a new framework for the systematic composition of strictly formalised music. Twelve Tone Technique or Serialism gave all twelve chromatic tones equal importance, and by doing so, began the move away from tonal music that had dominated traditional music up until that point. A piece of serialist music consists of multiple rows of notes, with each row containing all twelve chromatic notes in a certain order. Within a row, repetition of notes was not permitted, and every note had to be used (Aschauer, 2008). Various transformation techniques can be applied to subsequent rows, including *inversion*, where the intervals between the notes are reversed; *retrograde*, which involves reversing the whole row; and *retrograde inversion*, where the inversion is played backwards. Serialism was taken even further by Stockhausen when he applied serial methods not just to pitch, but also to rhythm and dynamics, as well as introducing elements of chance into his compositional processes (Simoni, 2003).

Chance was also explored by John Cage, who would use dice, coin flips and even divination games in his compositions to introduce randomness. Cage would also use natural phenomena to determine the direction of his work. This can be seen in his *Atlas Eclipticalis* (1961), which was composed by laying notation paper on top of astronomical charts and placing notes where the stars occurred (Schwartz, 1993).

Computers were first used for algorithmic composition in the middle of the twentieth century, with Lejaren Hiller and Leonard Isaacsons Illiac Suite in 1957. Hiller, together with Robert Baker, was also involved in the creation of MUSICOMP, one of the first automated composition systems. This was written as a library of subroutines, and allowed the re-combination of these routines in a wide variety of ways (Alpern, 1995). This is the principle of Modularity, a fundamental concept in engineering which says that “systems should be built from cohesive, loosely coupled components (modules)” (The Modulariy Principle, n.d.)

Another pioneer in the early use of computers in composition was the Greek / French composer Iannis Xenakis (1922 - 2001). He had a keen interest in mathematics which

he applied to his compositions through stochastic processes, using probability theory in the selection of musical parameters (Simoni, 2003).

As the use of computers became more widespread from the 1970s and onwards, a range of techniques evolved that wouldn't have been possible without their technical assistance. In the next section, we shall look at the main categories of algorithmic composition, including several that developed alongside the rise of the computer.

Categories of Algorithmic Composition

As stated in Chapter 2.1, attempting to classify algorithmic techniques is not an easy task. An algorithm can be a complete approach to composing a piece of music - serialism could be seen as an example of this. But more commonly a hybrid approach is taken, with algorithmic music being composed using a combination of multiple different methods. The following represents groupings that have been used to categorise algorithmic compositions and approaches.

Aleatoric methods involve the use of chance or randomness. Mozarts Musikalisches Wuerfelspiel is an early example of using chance, an approach adopted and developed by composers such as John Cage in the twentieth century. For true aleatoric methods, the probability of any event happening is the same for all events - there is no preference for one result over another.

Then there are the methods which use *probability*. Xenakis was a pioneer here with an approach he called *stochastic music,* which involves the use of randomness together with statistical principles (Aschauer, 2008). With *stochastic* methods, a result is calculated using random values that are mapped to a given probability distribution. These methods can produce discrete and continuous values, allowing us to work within the discrete pitches of the chromatic scale, or the continuous values of velocity. Unlike purely aleatoric methods, stochastic methods allow for more control over the output while still working with random data. Statistics and probability are an important element of many algorithms, including for example Markov Chains, which will be discussed in more detail in Chapter 4.2.3.

When considering chance, there has also been made a distinction between *deterministic* and *indeterministic* approaches. With deterministic approaches, the use of any

aleatoric elements is part of the compositional process - once the music has been written, it is fixed and will always be played the same. Indeterministic methods on the other hand leave elements of chance to the performance itself. For example, the composer might provide music notation, but the arrangement of the notation is left to the performer.

Rules-based methods and *constraint systems* allow the application of certain rules to musical material. These rules can be widely applicable, covering general music theory models such as counterpoint and voice leading, or more targeted and focused towards a specific idea or intention, depending on composers musical goal. The rules-based approach is the basis of what has been called *computer-aided composition.* These are software systems that allow composers to experiment with and refine their musical ideas by applying rules that enforce particular music theories or styles.

Artificial Neural Networks are another popular approach that would not have been possible without the aid of computers. These mimic a simplified model of the human brain, and offer the ability to learn from supplied musical examples without the need for the creation of complex rules on the part of the composer or programmer (Aschauer, 2008). Similarly to Markov Chains (see Chapter 4.2.3), ANNs work best when imitating the style of the training examples, and can be put to good use developing rules for use by other systems that rely on rules, such as genetic algorithms.

Genetic algorithms take their inspiration from the biological world. These methods imitate procedures found in the evolution of life, such as selection, reproduction and mutation, with the goal of allowing the exploration and evolution of, in our case, musical ideas. Genetic methods excel at search and optimisation tasks when dealing with a very large search space, and music composition can be understood as an optimisation process where a composer "searches in the space of all possible music compositions that he is able to compose, for one such that satisfies his own artistic criteria" (Dostál, 2013 p.936)

A genetic algorithm works by stepping through several stages. First, the current population will be *evaluated* for fitness, and individuals will be *selected* according to a set of rules set by the composer. This set of rules is called the *fitness function*, and is a key component in determining the success of a genetic algorithm. The fitness function can

employ various techniques including those discussed above such as neural networks and rules-based methods, in order to find the 'fittest', or most musical, candidates. In the *reproduction* stage, the selected candidates from the previous stage are chosen and 'bred' together, using a procedure called *crossover* to ensure a suitable mixing of ideas. During the last stage, *mutation*, randomness is used to introduce subtle changes, and the results are fed back into the cycle, allowing the process to continue (Aschauer, 2008).

2.3 Algorithmic Composition Pedagogy

This section will look at some pedagogical theories and approaches that are particularly relevant for the building of an Algorithmic Composition Explorer. Learning environments, learning through practice and computational thinking will be discussed in relation to learning algorithmic composition.

2.3.1 Learning through practice

Diana Laurillard (2012) has identified several ways students can learn - through acquisition, enquiry, discussion, practice and collaboration. Learning through acquisition is when we read, listen to lecturers or podcasts, or watch videos. It's a relatively passive way of learning, but widely used. Learning through enquiry is where the learner makes use of resources that provide searchable access to knowledge, data, information and ideas. Enquiry learning is a more active form of learning, where the student can follow their own line of enquiry. Learning through discussion is based on Vygotskys (1978) ideas around the social aspect of learning - the idea that peer discussion plays a significant role in learning at all levels of education. Learning through collaboration can be loosely defined as when two or more people learn something together, building a shared public knowledge.

Finally we come to learning through practice. Learning through practice, or experiential learning, is an essential part of the learning experience because it "invites the learner to adapt their conceptual understanding to the task at hand, and then reflect on what the

experience means for how they might modulate their understanding” (Laurillard, 2012 p.162). In other words, setting up an environment in which the learner can practice allows them to check and refine their current understanding based on these experiences.

In 1980, Papert identified three different use-cases for computers in education:

1. as tutorials, where the computer serves as a kind of “mechanised instructor”
2. as a tool for accomplishing a specific task
3. as a concept he called *microworlds*

Papert defined a microworld as an environment designed to afford the learning of some system or set of concepts and powerful ideas (Papert, 1980). Laurillard argues *“that all forms of learning through practice require a practice environment that has this property of affordance, so the microworld is a significant idea to make use of”* (Laurillard, 2012 p. 54).

This microworld, or learning environment, provides the essence of learning through practice, and the intrinsic feedback that it requires.

2.3.2 Extrinsic and intrinsic feedback

Extrinsic feedback, or feedback from the teacher, and intrinsic feedback, that from the environment, play different roles in learning.

Extrinsic feedback is when a teacher provides guidance or criticism on a learners performance. The learner can then choose to follow the advice in order to improve. Intrinsic feedback is a natural consequence of the action the learner has taken. The learner can use intrinsic feedback to work out how to improve without teacher intervention.

Extrinsic feedback is potentially the more efficient of the two as it guides the learner to the correct answer, while intrinsic feedback needs to be correctly interpreted. On the other hand, extrinsic feedback that is too helpful can reduce the learners own active reflection. Overall, intrinsic feedback, while harder to create, is more effective when designed correctly (Laurillard, 2012).

2.2.3 Computational thinking

Deleuze (2006) states that ideas are intrinsically linked to their mode of expression, such that *"I cannot say that I have an idea in general. Depending on the techniques I am familiar with, I can have an idea in a certain domain, an idea in cinema or an idea in philosophy"* (Deleuze, 2006 p.415). In our case, we are talking about musical ideas expressed as algorithms, and so according to Deleuze we would need to be familiar with algorithmic thinking as well as music before we can express ourselves in this way.

Filimowicz and Tzankova (2017) have written about computational literacy as a two-step process. The first step requires an ability to understand and modify code. The second step, and often the more challenging, translates ideas and concepts into code.

Computational thinking is the second step in this process. While it has been stressed elsewhere in this thesis that compositional algorithms have been in existence for far longer than computers, coding can still be an ideal medium for expressing such ideas.

Computational thinking can be defined as the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms (Aho, 2012). It was a term introduced by Papert (1980), who believed that children could learn by teaching the computer through programming. It has become a method of formalised problem solving, and the process can be broken down into the following four steps.

- **Decomposition** - This step involves breaking down data, processes, or problems into smaller, more manageable parts.
- **Pattern Recognition** - Here we analyse the data and identify similarities and connections among its different parts.
- **Abstraction** - abstraction is about identifying the most relevant information needed to solve the problem, and eliminating the extraneous details.
- **Algorithmic thinking** - Finally, we develop a step-by-step process to solve the problem so that the work is replicable by humans or computers.

(McVeigh-Murphy, n.d.)

We have established the importance of computational thinking and its importance for education today, but what is its relevance to music and composition?

Musical expectations are a core phenomenon of music cognition (Tillmann et al., 2014). Our brains naturally build expectations based upon previously acquired knowledge of musical rules (Zatorre & Salimpoor, 2013). For centuries, composers have taken advantage of this fact to formalise compositional structures, and it has been argued that the development of musical form naturally leads to algorithmic approaches to composition (Edwards, 2011). So musical form grew out of how we listen to and respond to music, and the application of computational thinking can help us deconstruct, understand and recreate the music we listen to. By generalising our understanding of the different forms and approaches, we can build algorithms that apply some of those rules. While the ability to emulate certain styles is arguably the least interesting aspect of algorithmic composition from an artistic perspective, developing the ability to express and formulate ideas and rules in an algorithmic manner opens up a world of possibilities and musical places to explore.

2.4 Related works

A Platform for Algorithmic Composition on p5.js by Chan Jun Shern [1]

This is an interactive and explorable web tutorial using P5.js. P5.js is a Javascript interpretation of Processing, built on top of the Web Audio API, and will be discussed in further detail in Chapter 3.1.2. Chan has built a tutorial that allows the user to learn about many different algorithmic approaches. It runs in the browser, and makes use of animations and other visualisations to explain musical principles and demonstrate algorithms. While it covers a range of algorithms, its interactive examples stop short of letting the user learn through practice. Instead the examples are there to support the written explanation and code, rather than provide any experiential learning opportunities. This is the opposite of the approach proposed for the Algorithmic Composition Explorer, where the textual explanations are there to pose questions and prompt exploration of the al-

[1] https://junshern.github.io/algorithmic-music-tutorial/

gorithm. It is this focus on the algorithm itself - on learning through practice - that is the key aspect of the Algorithmic Composition Explorer.

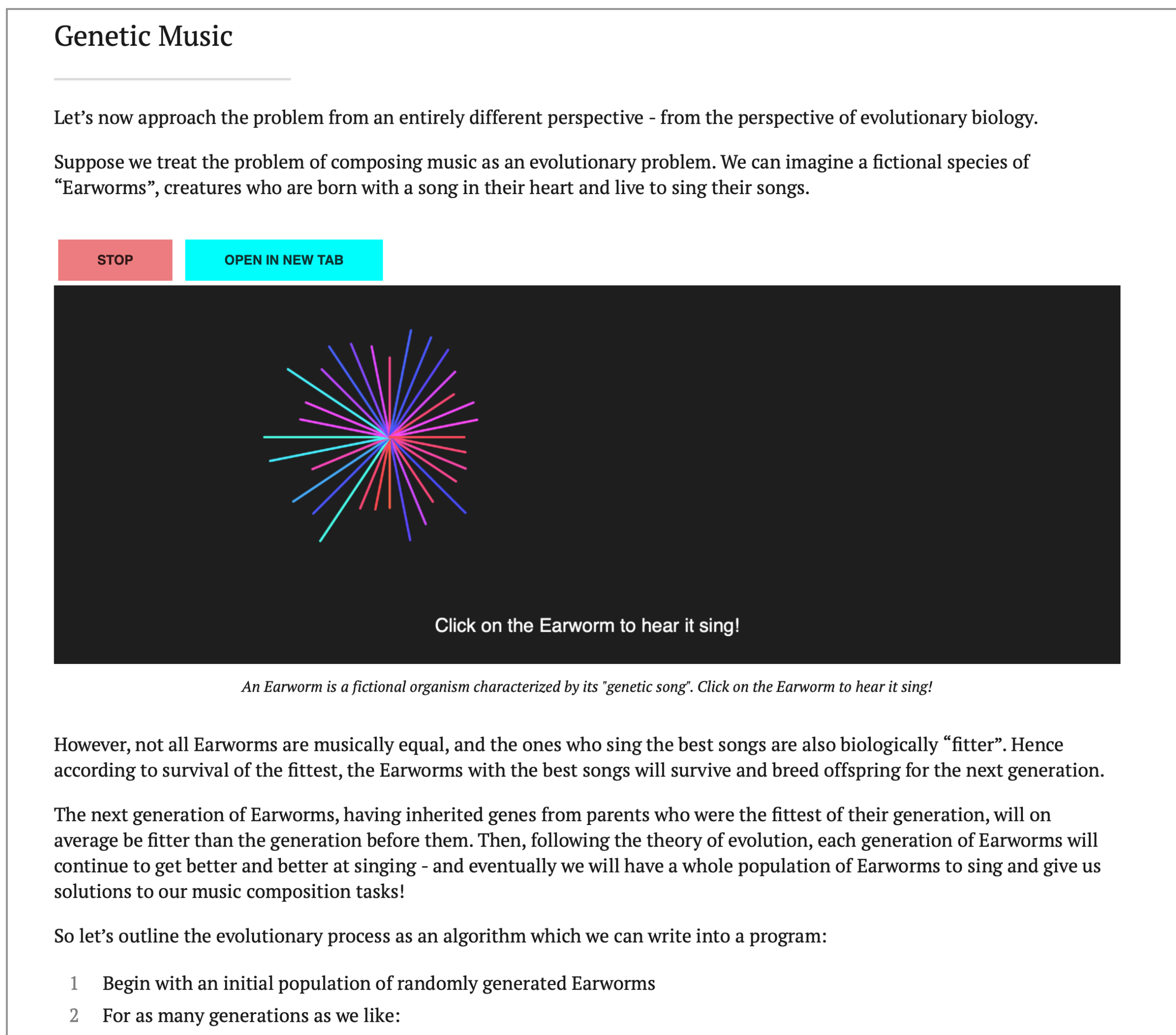

Figure 2.1: A Platform for Algorithmic Composition on p5.js by Chan Jun Shern

How Generative Music Works by Tero Parviainen [2]

Parviainens audio-visual primer on generative music is a web tutorial and presentation he gave at Ableton Loop. It is written using Tone.js (see Chapter 3.1.2), and looks at many common techniques of generative music. It features interactive Javascript visualisations for works by composers such as Steve Reich, Terry Riley and Brian Eno. As it

[2] https://teropa.info/loop/#/title

was designed to be a presentation, the user must step through the algorithms and explanations one by one, there is no opportunity to explore here. The visualisations are very effective however, and work very well to aid the understanding of the concepts being explained.

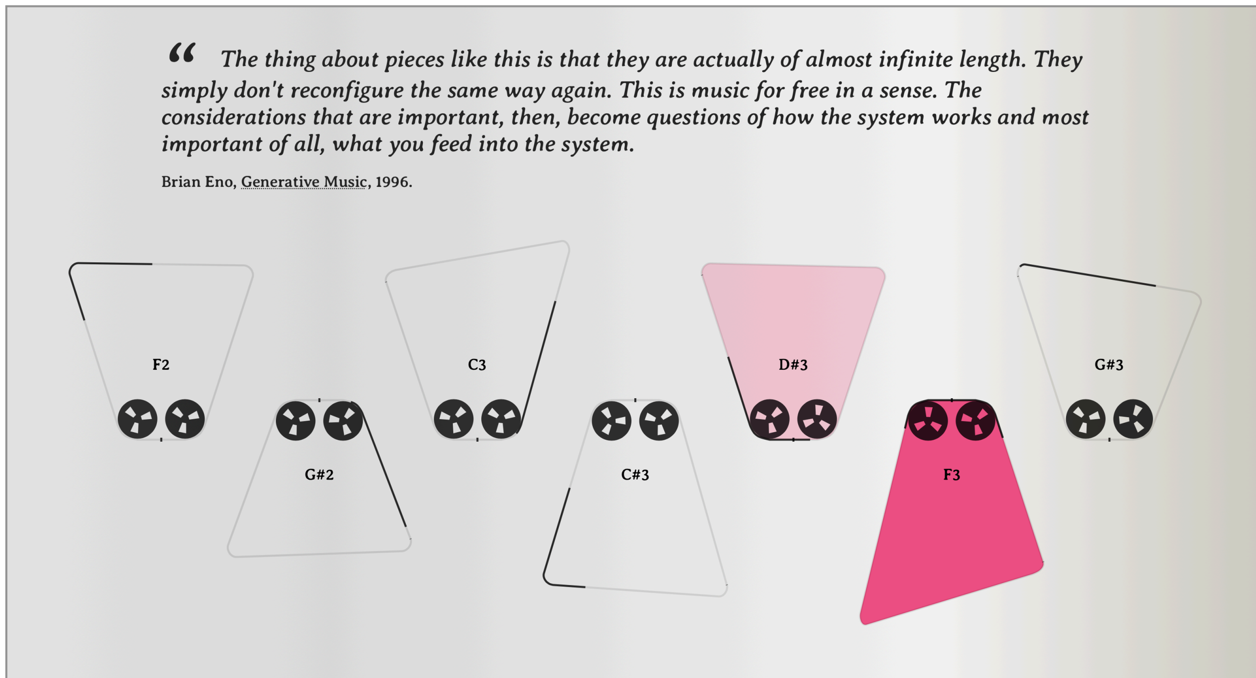

Figure 2.2: How Generative Music Works by Tero Parviainen

Parviainens also wrote an in-depth tutorial[3] explaining how he used Tone.js to build the presentation, focusing in on three examples taken from it. The tutorial is focused on learning Javascript, Web Audio API and Tone.js in order to make music, but also delves into algorithmic composition, in particular the history and background of the algorithms chosen.

The limitations I pointed out for Chans “Platform for Algorithmic Composition on p5.js” apply here as well. It is an effective system for introducing algorithmic composition and bringing together examples of multiple algorithms, but the options for working with the algorithms are limited.

What is missing here is an opportunity for a user to learn about algorithmic methods through interacting with them. For those that have worked through these tutorials, and

[3] https://teropa.info/blog/2016/07/28/javascript-systems-music.html

now want to try out some of the methods themselves, there are several large technical hurdles that would need to be overcome. The first is the need to install and set up one of the many excellent environments that offer algorithmic composition libraries such as Max, Music21 or SuperCollider. The learner would need be comfortable coding / working in that environment, or have the time and motivation to become familiar with it. They would then need to code the algorithm that they were interested in. If they were wanting to compare several algorithms, then the motivation and effort required increases again. For those that already have the motivation, time and technical expertise, these are excellent environments to work in and learn from. For those that just want to dip their toes in, the cost of entry is likely to be too great.

The Algorithmic Composition Explorer bridges that gap. It takes people whose curiosity has been piqued, perhaps by Chans or Parviainens tutorials, and lets them experiment with a selection of algorithms. It even allows them to take the results with them, and incorporate them into their own compositions.

The next section looks at the design of such a system.

3. Design & Methodology

The three main aims for the Algorithmic Composition Explorer are:

1. To be widely available and requiring little in the way of previous technical knowledge. Avoiding the need for any installation or setup to get started.

2. To allow users to experiment with and compare different algorithmic approaches, and offers a visual representation of the generated music as well as audio playback. Allowing users to take the generated music with them and incorporate it into their own work.

3. To provide a platform that can be built upon and extended with new algorithms, instruments and components.

For the first aim, an overview of potential tools, frameworks and languages is needed before development can begin. Choosing the most appropriate combination of language and framework is key for this step.

The overall design of the software needs to satisfy the requirements listed in the second aim - allowing users to experiment with, visualise and playback algorithmically generated music. Another important consideration here is pedagogy, and how the design can take into consideration the current best practices in this field.

Finally, for extensibility, there are two aspects that need to be taken into consideration. First is the chosen language and framework. How established is it? What are the associated best practices? Are there official standards that need to be adhered to? Second, there needs to be considered the potential directions the software could take. When it comes to future development, certain design decisions could open up particular avenues of exploration, and shut off others. This needs to be considered and incorporated into the design of the system.

3.1 Languages & Frameworks

The considerations when approaching the choice of technology for this project are:

- what tools currently exist that could be used for building an algorithmic music system
- what tools satisfy the requirements for learner interaction, visualisation and playback
- what tools would best allow the result to be shared widely
- which tools offer the most potential for future expansion

This section will give an overview of the current best options I found for building an Algorithmic Composition Explorer. While far from exhaustive (it focuses on only a handful of Python and Javascript frameworks), it looks into a few promising environments that could provide a solid base on which to work, and allow easy sharing of the results.

There are many software libraries and frameworks that focus on simplifying coding for a particular use-case, for example - music analysis, creative coding, browser-based interactive music (live-coding) and music composition. While none of those I came across had been specifically developed for teaching algorithmic composition, all can support it to a greater or lesser extent. As environments for learning music through coding - and coding through music - there are many interesting approaches available.

3.1.1 Python

Python is a popular choice for learning music and programming. It is widely used, has a simple, approachable syntax and a much shallower learning curve than many languages. At the same time, it is a powerful, well designed language, and fully object-orientated. One issue for those new to programming (and this is certainly not unique to Python) is that the initial setup and configuration of the environment can be troublesome for those new to programming. One option here is to use Google Colab[4]. Google Colab offers a way to write and execute Python directly in the browser, avoiding many of the installation and configuration issues that can cause problems for beginners. Once interest has been piqued, the individual can move over to running Jupyter Notebook or Python directly on their computers, offering the ability to dive into the code, and opening up the potential of the environment.

[4] https://colab.research.google.com/

Isobar

Isobar is a Python library designed for "creating and manipulating musical patterns, designed for use in algorithmic composition, generative music and sonification" (*Isobar*, n.d.). It offers a rich library of tools specifically designed for algorithmic composition, with many inspired by SuperCollider, the live coding audio synthesis platform. Its focus is very much on composition however, and it does not generate any audio on its own. Using Isobar would require integrating it with a separate library for audio synthesis. It has been in development for over 10 years.

Music21

The python library Music21 (Music21: A Toolkit for Computer-Aided Musicology, n.d.) is a "toolkit for computer-aided musicology". The focus here is on analysing scores, but it is a flexible tool and has been used for algorithmic composition (Shiihs, 2013). In addition, it can be used together with Avro, a library that adds various algorithmic functions for isorhythmic constructions and minimalistic approaches (Dimitrov, 2020/2021). Music21 is a popular, well supported and long running project, having been in active development since 2006, with the current version 7 being released in September 2021.

3.1.2 Javascript & the Web

Javascript, the language of the Web, has been becoming more and more popular in recent years, and as of 2020, was the most commonly used language, with Python coming in third (Most Used Languages among Software Developers Globally 2021, n.d.). One of the advantages of Javascript to first time programmers is that it can be run directly in the browser, reducing the potentially complex setup and configuration processes that can hinder getting started with other languages. In addition, the ubiquity of the Web means that Javascript runs everywhere, offering the ability to easily publish and share creations widely, adding to its attractiveness. As a result of this popularity, there can now be found a large number of libraries and frameworks for music, including for algorithmic composition.

Web Audio API

The Web Audio API was first proposed by Chris Rogers in 2010 (Web Audio API Proposal from Chris Rogers on 2010-06-15 (Public-Xg-Audio@w3.Org from June 2010), n.d.). The Web Audio API provides a *"powerful and versatile system for controlling audio on the Web, allowing developers to choose audio sources, add effects to audio, create audio visualisations, apply spatial effects (such as panning) and much more"* (Web Audio API - Web APIs | MDN, n.d.). While the focus is on generating and manipulating audio itself (as apposed to providing a platform for composition), there are several frameworks that have been built on top of the Web Audio API that make working with the API more straightforward.

P5.js

Processing[5] is one of the original creative coding environments. It began as a Java based library, but has developed into an arts-oriented approach to learning, teaching, and making things with code. This approach gave rise to P5.js[6] and it's audio library P5.js-sound[7]. P5 is a library *"with a focus on making coding accessible and inclusive"*, and manages to be both approachable and powerful. It is a Javascript alternative to Processing, and is supported by the Processing Foundation. Like Processing, the core of P5 is focused on drawing and the visual arts, but there is a rich collection of libraries that can be used to extend on this core functionality. P5.js-sound library adds Web Audio functionality, building on top of the Web Audio API. The focus is weighted more towards generating and manipulating audio than composition. It uses a Phrase / Part / Score framework for creating musical sequences, which is a functional, if not optimal, solution to composition.

---

[5] https://processing.org/

[6] https://github.com/processing/p5.js

[7] https://github.com/processing/p5.js-sound

Tone.js

Tone.js[8] is a Web Audio framework for creating interactive music in the browser. It sits on top of and simplifies the Web Audio API, and adds components such as prebuilt synths and effects, in addition to offering digital audio workstation features like a global transport for synchronising and scheduling events. Javascript is a single-threaded language, and so theres no way to guarantee that a function will be called at an exact moment in time (JavaScript Systems Music, n.d.). Tone.js improves on the Web Audio API by executing callbacks slightly before their scheduled triggering time, passing in the exact time the function should be triggered. So Tone.js's timeline can offer sample-accurate scheduling, allowing sounds to be played at exact intervals. For composition itself, Tone.js allows you to string sequences of notes together using the Part and Sequence objects. Importing MIDI files is possible, with Tone.js using its own JSON format for representing pieces of music, with scientific pitch notation (SPN) used for the actual pitches. The JSON representation has not been designed to be easily human-readable or writable however, and it doesn't follow any existing notation standard such as LilyPond or MusicXML. Regarding my own experience with Tone.js, I found it interesting to work with, with a clean and expressive syntax for audio tasks. Like the Web Audio API on which it's built, it's at its best best when dealing with audio, but it's global transport and sequencing options also make it suitable for working with algorithmic composition.

The Holy Grail

Python combined with Isobar or Music21 offers a lot of power together with well established, sophisticated libraries for algorithmic composition. Combined with Google Colab or Jupyter Notebook, some of the issues that can arise when setting up and running code can be mitigated, and a Python based system would allow users to more easily delve into the code, offering a lot more power for those that want to go deeper. This added power does come at a cost however. Set-up is still not seamless - it requires some effort and technical knowledge for troubleshooting any issues that come up, especially with regard to managing dependancies.

---

8 https://tonejs.github.io/

Javascript code lacks the user friendly nature of Python. For those that aren't looking to jump into the code however, a Javascript app running in a browser offers a much better user experience. The combination of the Web Audio API's audio synthesis and processing capabilities and Javascript make for a very powerful platform. Javascript is the language of the Web, it can be run everywhere, on any computer, smartphone or tablet. This means that we can share not only the musical output of any system we build, but the system itself, simply by sharing a URL. Such a Web site would require no installation or setup, and little previous technical knowledge to be used. And the popularity of Javascript, and the Web in general, ensures a good potential for future development. For these reasons, it was Javascript that was chosen to build the Algorithmic Composition Explorer.

3.2 Computer representations of music

Any system that supports a particular musical task will need some way to represent the music itself. The goal of that system will necessarily dictate the nature of that representation - a system for musical analysis will have different requirements of its data than that of a system used for outputting graphical notation. MIDI is possibly the most well known of digital music representations, but there have been many approaches to how the primary attributes of music - including pitch, duration, dynamics, timbre etc. - can be represented. Selfridge-Field (1997) classifies representations as different 'codes', including sound related codes, musical notation codes, codes for data management and analysis, and representations of musical patterns and processes.

The main focus for many of the frameworks listed above was on the generation and manipulation of sound, with the representation of music in a clear, well-defined format a secondary consideration. There have been many attempts to solve this issue, with various "standards" in use today, including MusicXML, Helmholtz, Scientific Pitch Notation, ABC, Lilypond and many others. This section will discuss the different approaches that have been taken, and their use.

3.2.1 Representation of pitch

The notes of the western chromatic scale can be notated using either letters or numbers. Helmholzt notation was proposed by Hermann von Helmholtz in 1863, and uses a combination of the letters A-G and sub and super prime markers (‘ and ,) to designate octave. It is still commonly used in research, as well as in notations such as ABC and Lilypond. Scientific pitch notation (SPN) uses a combination of the letters A-G, plus a number to indicate octave. SPN is generally seen as being easier to read than Helmholtz, but there has been inconsistency in the number used for middle C, where C3, C4 and C5 have all been used to denote middle C (Cakewalk - SONAR LE Documentation - Fretboard Pop-up Menu, n.d.), (Logic Pro 9 User Manual: Display Preferences in Logic Pro, n.d.). Finally there is MIDI notation, which uses the integers 0 - 127 to represent pitch. Integers offer the opportunity to apply various mathematical functions to pitch, making them a good choice for use in algorithmic composition.

3.2.2 Music engraving

Being able to visualise the music was an important consideration when building the Algorithmic Composition Explorer. This section lists the options I found for embedding music notation into a Web page.

Vexflow[9] is a very powerful engraving library for javascript and the Web. It offers a depth of control over the display of the music that other libraries cannot match, but this does come at a cost of added complexity. For example, the API has been designed so that there is only one bar / measure per stave - a new stave needs to be created for each bar, which would then be appended onto the previous stave.

Verovio[10] is an engraving library which was designed for use with the Music Encoding Initiative (MEI) format, but now supports multiple formats including MusicXML, HumDrum, MuseData and ABC. There is a javascript app available which can be embedded into a Web page, outputting an SVG image of the music.

---

9 https://www.vexflow.com/

10 https://www.verovio.org/

ABCjs[11] is a library that focuses on playing and rendering music in ABC notation. It is a lot more forgiving when rendering music than the other libraries, being more tolerant of inaccurate notation such as incorrectly positioned bar-lines. I also found it easier and quicker to get started with.

3.3 The overall design of the system

Having reviewed the language and framework alternatives and options for representing and visualising music, the overall design of the system can now be considered.

How a codebase is structured and organised is an important consideration when dealing with a system that could grow and become potentially large. Javascript and Tone.js can be an effective combination, but don't in and of themselves offer anything in the way of overall structure. Javascript frameworks such as Angular and React offer more potential for structuring a larger codebase, but these frameworks can get complex very quickly.

The ability to modularise code is essential. Being able to compartmentalise functionality makes for a much more easily understood codebase, allows for code re-use, and can make the inevitable changes and introduction of new features more manageable. Building for the Web revolves around three main technologies - HTML for the page layout and content, CSS for styling those pages, and Javascript for adding functionality. However, by itself, HTML doesn't offer the ability to modularise its code.

There are a class of frameworks called static site generators that offer structure and modularisation with less complexity than some of the larger Javascript frameworks. Static site generators such as Jekyll allow you to build a Web site by combining multiple smaller snippets of content into single HTML files during compile time. The resulting output is a static website that can be run on any Web server, without requiring a separate application server for running backend code. This keeps the website fast and responsive, and greatly simplifies deployment.

[11] https://abcjs.net

Jekyll supports the use of *partials* - snippets of HTML that can be inserted into pages, and *layouts*, which are page templates into which content and partials are added. Layouts would give the ability to have a selected number of page templates throughout the app giving a consistent user experience. Partials would allow the creation of re-useable interface components, such as sliders to control parameters like BPM, and select boxes to choose notes and scales. This modular nature would not only provide a solid platform for future development, but a consistent and well organised environment for the experience of learning through practice, and of Papert's microworlds (see Chapter 2.3.1).

Another aim of this thesis was to build something that is widely available. Building for the Web goes a long way in supporting this aim, but it is important to remember that people access the Web from many different devices. If the software doesn't work or display well on mobile devices, then a large section of the target audience could be excluded. The Mozilla Developer Network defines responsive Web design as *"a set of practices that allows web pages to alter their layout and appearance to suit different screen widths, resolutions, etc."* (Responsive Design - Learn Web Development | MDN, n.d.). Fortunately, there are many CSS frameworks that support responsive Web design. For this project, the Bootstrap CSS framework was chosen, due to its strong support for responsive design, as well as its familiarity to the developer. Bootstraps library of styles and components make it easy to implement a design that will look and function consistently across different browsers, platforms and devices.

3.4 Algorithm selection

The target audience has been defined as those wanting to learn more about algorithmic composition, but don't have the technical knowledge to set up a development environment and start coding. With this in mind, I wanted to choose algorithms that were simple enough for learners to get an understanding of through adjusting parameters and figuring out what's going on under the hood. As discussed in Chapter 2.3.2, intrinsic feedback is a natural consequence of the action the learner has taken, and needs to be designed into the system. So there needs to be a clear relationship between the chang-

ing of an algorithms parameter and the results that are heard. Any instructions provided should encourage exploration and ask questions, as apposed to simply explaining what an algorithm is doing.

I also wanted the choice of algorithms to reflect algorithmic composition's long history. Presenting algorithms that either pre-date or don't require the use of computers could help provide an alternative and possibly more approachable view of algorithmic composition. And moving from simple algorithms towards the more complex provides a smoother transition to the world of algorithmic thinking, which could otherwise be seen as intimidating.

With this in mind, starting at the beginning made sense here. Guido of Arezzo's vowel-to-pitch algorithm is one of the earliest known, and while simple to comprehend, adds an interesting relationship between the source text and music produced.

Mozarts Musikalisches Würfelspiel was an algorithm that was almost discounted, due to the lack of control the user has over the music produced. However, it is a good example of using randomness to generate music, and could encourage the learner to think about form and style, and what makes the song sections work together.

Like Mozarts dice game, the Markov chains algorithm also works with previously prepared music, but the algorithm itself is more complex in this instance. The underlying concept is understandable however, and much can be learned about this algorithm simply by varying the source material and adjusting the order.

Arvo Pärt's Tintinnabuli takes a step back from Markov chains in terms of the complexity of the algorithm, but opens up a world of exploration. While this is an approach that can be more easily grasped, the results are more customisable and potentially more easily transferable to the learners own compositions.

While the time available limited this prototype to the above four approaches, several other algorithms were considered. Likely to be the first choice in any future development would be Cellular Automata, due to the interesting nature of the problems it poses. The music generated can vary over time and can continue indefinitely, so an interesting challenge could be where in the process the music is taken from. How could this choice be presented to the user? Would using a visual representation of the resulting

patterns be an option for choosing the range of notes to be rendered? In addition, developing an interface that allows users to change the rules or add new rules could also be an interesting problem to tackle.

4. Implementation

This chapter details how the Algorithmic Composition Explorer is implemented, and how related challenges have been addressed. The chapter starts with the general approach that was taken when building the software, covering the main components such as the Score and Audio Player classes, before going into more detail around how each algorithm was implemented. The Algorithmic Composition Explorer is available at https://algorithmic-composition-explorer.com, while its source code is available at https://https://github.com/stega/algorithmic_composition_explorer.

4.1 Building the Algorithmic Composition Explorer

Jekyll[12] was chosen as the framework on which everything would be built on. As discussed in Chapter 3.3, Jekyll allows the building of a static website by bringing together a combination of layouts and snippets of HTML, Javascript and CSS, offering the ability to modularise and organise the code. This was going to be important when building a library of components that could be reused in different algorithms.

[12] https://jekyllrb.com/

The HTML files were split into three main directories - the layouts, the algorithms and the partials, as shown in Figure 4.1.

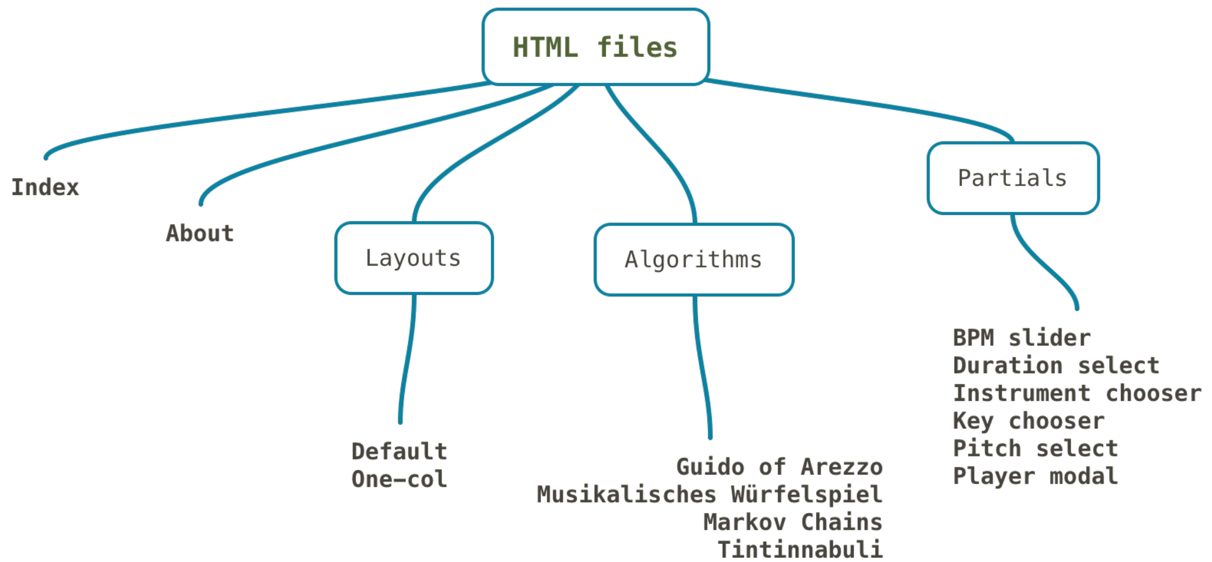

Figure 4.1: the HTML file structure

The Javascript files containing functionality specific to the algorithms themselves were kept together under the `algorithms` directory, separate from the shared functionality, such as the Score class and the different instruments, as shown in Figure 4.2.

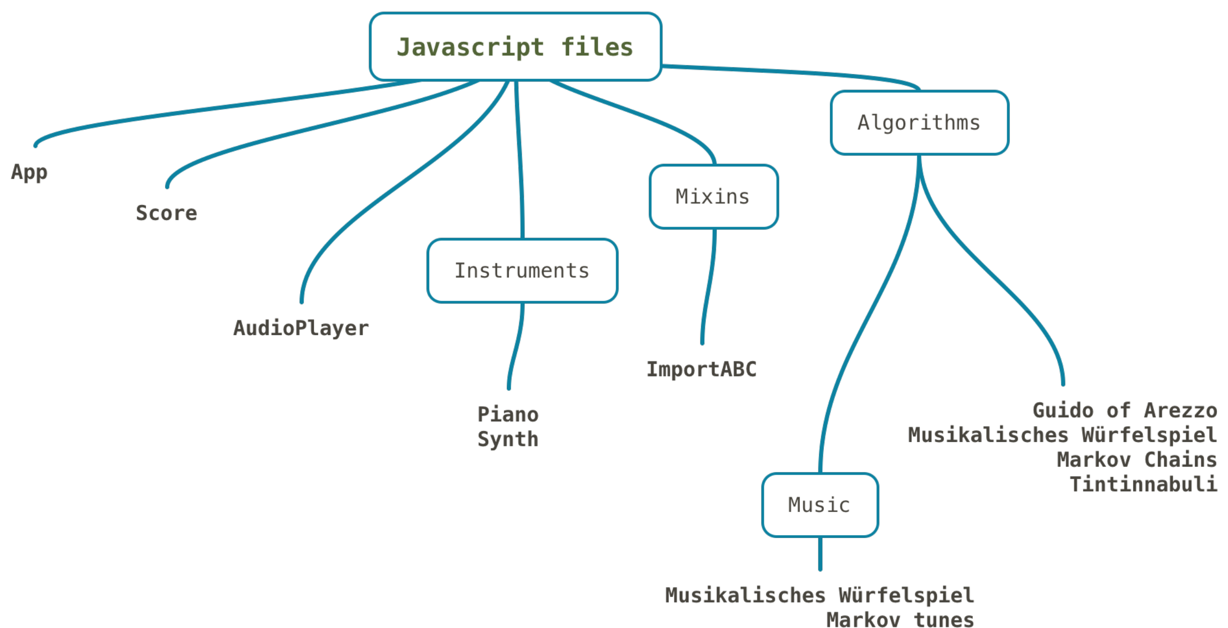

Figure 4.2: the Javascript file structure

Visually, the UI needed to include a list of algorithms to choose from, an area for the description of the chosen algorithm, the instructions for using it and some code / pseudo code examples. The algorithm itself needed to be a HTML form, displaying the various controls / components that would allow for the manipulating of the selected algorithms various parameters. This can be seen in Figure 4.3.

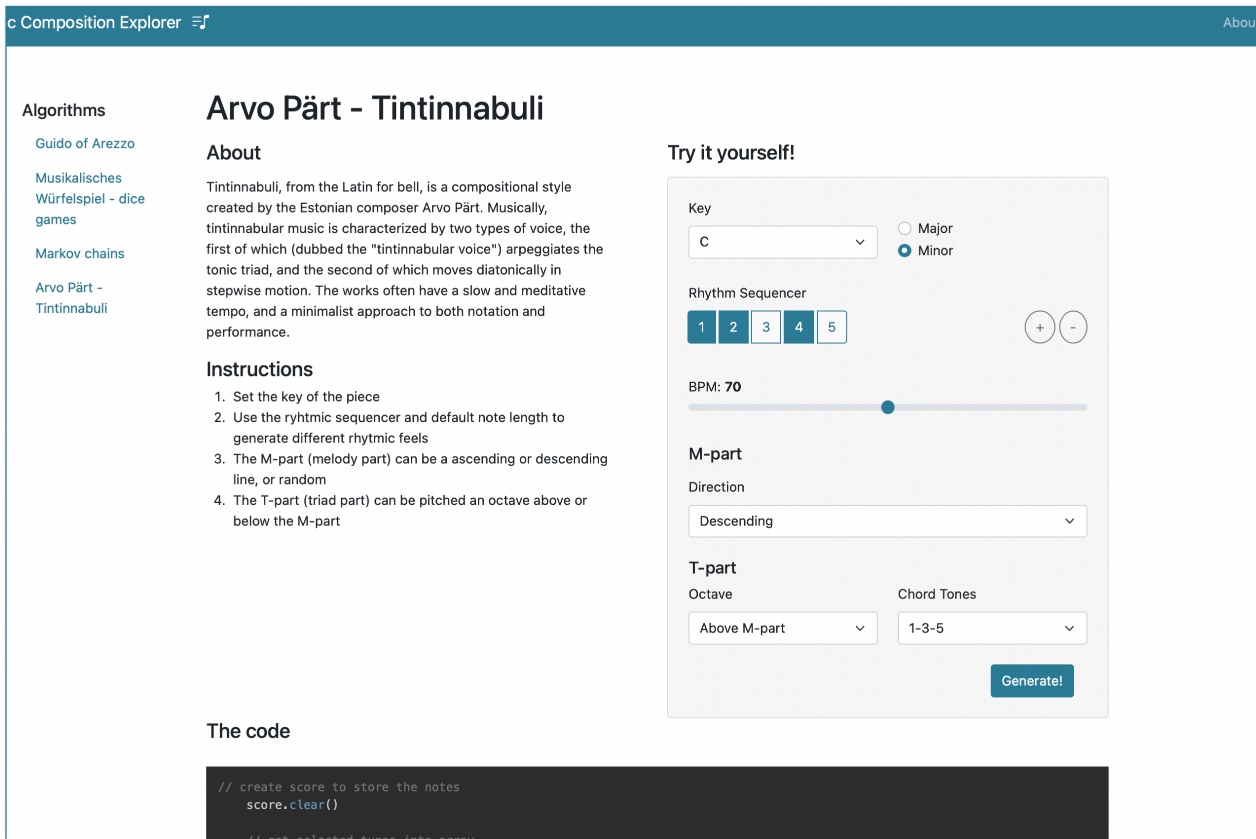

Figure 4.3: The main user interface

For the graphical representation of the music, the ABCjs library was used. This greatly reduced the time needed to build the prototype by providing the functionality to both visualise and play-back any generated music that was in the ABC notation format. The next major component that was required was somewhere to store and manage the generated notes and music. This was to be the responsibility of the `Score` class. ABC notation is based on Helmholtz pitch notation, so this was used when storing note information in the Score. Using Helmholtz notation allows for the easy conversion to ABC, and therefore access to ABCjs's convenient engraving and playback features. An example of the output from ABCjs can be seen in Figure 4.4.

Figure 4.4: The Player modal, showing rendered notation and playback options

The Score object would also need to store the musics attributes and meta-data, including the title, meter, base note length, tempo and key signature. The notes themselves were initially stored in an array of strings, each denoting the pitch and duration using Helmholtz notation. As development continued, it became apparent that this approach to storing notes wasn't going to be flexible enough.

While the Helmholtz approach to pitch notation is well established in research and was an effective way to get started, it was not as intuitive to work with in code, and could make certain musical operations difficult. In addition, while Helmholtz worked well with ABC.js, the more flexible sound generation options that were available in libraries such as Tone.js were preferable for this project. Tone.js uses scientific pitch notation to rep-

resent pitches, and so it made sense to store notes using this approach. Tone.js however didn't offer the straight forward approach to sheet music engraving that ABC.js offered. Both approaches would need to be supported to fulfil the requirements laid out in Chapter 3.

Another issue was that the storage of both pitch and duration in a single string made working with timing and rhythm more difficult than it needed to be. The note array needed to be changed to an array of dictionaries, with each containing separate pitch and duration fields. Chords also needed to be represented, and for this, an array of pitches were used. In ABC notation, each note follows the last, which means that the note onset time isn't recorded. An alternative approach, and one taken by other solutions such as MIDI, is that each note is given a start time relative to the beginning of the score. This was added to the Score class, formatted using the `toBarsBeatsSixteenths()` method provided by Tone.js.

Finally, algorithms that generated multiple voices or parts needed to be supported. A multi-dimensional array was used for this, so each voice could contain an array of note event dictionaries. An example showing the final note representation object can be seen in table 4.1.

Voice 1	{ time: 0:1:0, pitch: C4, duration: 4n }	{ time: 0:2:0, pitch: A4, duration: 4n }	{ time: 2:0:0, pitch: G4, duration: 2n }
Voice 2	{ time: 0:1:0, pitch: G3, duration: 2n }	{ time: 0:2:0, pitch: [C3,E3,G3], duration: 2n }	

Table 4.1: Note representation in the Score class

For playing back the generated music, the ABC.js library was initially used. While this worked well for the most part (there was an unresolved bug where playback could be double triggered after the initial page load), it offered little flexibility when it came to sound. For playing back audio, Tone.js synthesis and sample playback abilities were far in advance of ABC.js. A separate AudioPlayer class was built around the Tone.js

`Transport` object. Both the score to be played, as well as the instrument to be used, could be passed into the AudioPlayer for playback. Functions for creating a sample-playback instrument using piano samples, as well as a two oscillator poly synth were written. Both can have delay and reverb effects added, and while not comprehensive, work well and can act as a template for further developing this aspect of the application. Expanding the player to allow for the use of different instruments for scores with multiple voices would be a relatively simple undertaking.

ABC.js and Tone.js are just two of the third party Javascript libraries that the Algorithmic Composition Explorer relies on, and more will be introduced as the discussion turns to the algorithms themselves. Managing multiple Javascript libraries for modern Web applications can be a complex process. A build tool such as NPM can be used to manage external libraries, as well as to prepare them for deployment. Jekyll, being a Ruby based tool, uses Bundler to manage its dependencies, and integrating NPM adds an extra layer of complexity when it comes to bundling the assets. To keep things simple, the CDN JSDeliver was used for loading any third party javascript libraries that were needed in the code.

4.2 Selected Algorithms

4.2.1 Guido

Guido was a 10th century Italian monk, best known today for the invention of staff notation which would replace neumatic notation and eventually develop into the Western music notation that we know today. However, his work and theories in music were of a very practical nature, designed to aid singers in the learning process. Guido's ideas and techniques focused on finding easier, more efficient ways to learn music, and he would transform the way music was taught (Reisenweaver, 2012). He is also credited with creating the first algorithmic composition technique, which was introduced as an approach to teaching composition, with the goal of transmitting creativity and sensitivity to the learner (Miller, 1973). This combination of algorithmic approaches and pedagogy makes Guido an ideal starting point for the Algorithmic Composition Explorer.

Type or paste text to be analysed here

Vowel	Pitch	Note Length
A	C	1/16
E	Eb	1/8
I	F	1/4
O	G	1/2
U	Bb	1

BPM: **140**

Generate!

Figure 4.5: The Guido algorithm interface

The algorithmic approach Guido developed was a method for the automatic conversion of text into melodic phrases. The idea was that the vowels of a selected piece of scripture would be mapped to tones. The idea is an example of a translational algorithm - an approach that takes an input and translates this into musical information based on a set of rules. It can also be classified as a deterministic algorithm, meaning that with any given input, it will always produce the same output.

For this implementation, each vowel would be mapped to a chosen pitch and rhythm. Two translation tables were therefore needed, one to lookup pitch, and one to lookup

the rhythm. As the user is to supply the pitch and rhythm information, these tables need to be built each time the algorithm is run, using the user provided data.

The code in Figure 4.6 shows how the pitch translation table is built using user-supplied data.

```
function buildPitchTable() {
  return pitchTable = {
    'a': document.querySelector('#pitch-select-a').value,
    'e': document.querySelector('#pitch-select-e').value,
    'i': document.querySelector('#pitch-select-i').value,
    'o': document.querySelector('#pitch-select-o').value,
    'u': document.querySelector('#pitch-select-u').value
  };
}
```

Figure 4.6.

The algorithm is then a simple matter of working through the provided text, and fetching the appropriate pitch/duration information whenever a vowel is encountered. The note data is then appended to the score.

```
function mapPitches(text) {
  // create lookup tables for pitches and note duration
  pitchTable    = buildPitchTable();
  durationTable = buildDurationTable();

  // loop through text & convert vowels to pitches/durations
  words = text.toLowerCase().split(' ');
  words.forEach(function(word, index) {
    for (let char of word) {
      if (isVowel(char)) {
        duration = durationTable[char];
        pitch    = pitchTable[char];
        score.addNote(pitch, duration);
      }
    }
  });
}
```

Figure 4.7.

Once the algorithm was working, the code was refactored for easier readability and to allow re-use of components - for example, the pitch and duration select options were refactored into their own `_includes` files.

The BPM slider, as visible in Figure 4.5, was added after the fact to allow the user to set the desired tempo of the resulting music. This BPM slider was developed during the writing of the Markov Chains algorithm. Once it had been refactored out into the `_includes` directory, it could be re-used, and so was introduced here to allow for a little more control over the output of the algorithm.

4.2.2 Musikalisches Würfelspiel

Musikalisches Würfelspiel, German for 'musical dice game', was a system for using dice to randomly generate music from a set of pre-composed snippets. It was a popular game in 18th century Europe, with the most famous example (and the one used in this algorithm) being attributed to Mozart (Zbikowski, 2002).

	I	II	III	IV	V	VI	VII	VIII
2	96	22	141	41	105	122	11	30
3	32	6	128	63	146	46	134	81
4	69	95	158	13	153	55	110	24
5	40	17	113	85	161	2	159	100
6	148	74	163	45	80	97	36	107
7	104	157	27	167	154	68	118	91
8	152	60	171	53	99	133	21	127
9	119	84	114	50	140	86	169	94
10	98	142	42	156	75	129	62	123
11	3	87	165	61	135	47	147	33
12	54	130	10	103	28	37	106	5

Figure 4.8: Number table for measures 1-8 of Musikalisches Würfelspiel

The game consisted of two tables: a table of numbers and a notation table which contained the snippets of music (see Figure 4.8 & 4.9). For each column in the number table, two dice would be rolled to choose a random row for that column. The number from the chosen row would then be used to find the corresponding measure of music in the notation table. As the player steps through each column in the table, the resulting snippet of music would be added to the last, generating a completed piece of music once the player reached the end of the table (Zbikowski, 2002).

* The notes with stems down are used for the first ending.
** The notes with stems up are used for the second ending.

Figure 4.9: Extract from the notation table for Musikalisches Würfelspiel

While it seems that the dice rolls mean randomness is driving this algorithm, there is actually very little left to chance. The success of such games relied on their creators having a good understanding of the music's form and style, and Mozart ensured that each set of measures would flow nicely from one to the next. (Zbikowski, 2002; Manaris & Brown, 2014).

For this algorithm, pre-transcribed snippets from Musikalisches Würfelspiel were sourced from a GitHub repository[13]. The music was formatted for the notation library VexFlow, and so it needed to be re-formatted to work with Tone.js, but this was a quicker and easier task than transcribing from the original notation would have been. A javascript object was created, with each property corresponding to a measure that could be chosen. This was added to the `_includes/music` directory, with the file then being imported into the main javascript file using Jekyll's include directive.

As mentioned earlier in this section, while the choice of measure at a particular point in the composition is random, the selection of measures available at that point has been carefully crafted beforehand by the original composer. To ensure that the correct mea-

[13] https://github.com/timmydoza/mozart-dice-game.

sures would be associated with the correct table number, Mozarts original table number ordering also needed to be used. Whereas Mozart used two tables for parts 1 and 2, and the process included a repeat of the first part, here the first part is played through only once in order to keep things simpler for the learner to understand.

There are two main tasks for this algorithm. The first is the setting up and managing of the grid of numbers, and the second is the assembly of the composition from the selected grid numbers.

The grid numbers were provided in an array, with the grid being created dynamically on page load, as can be seen in Figure 4.10.

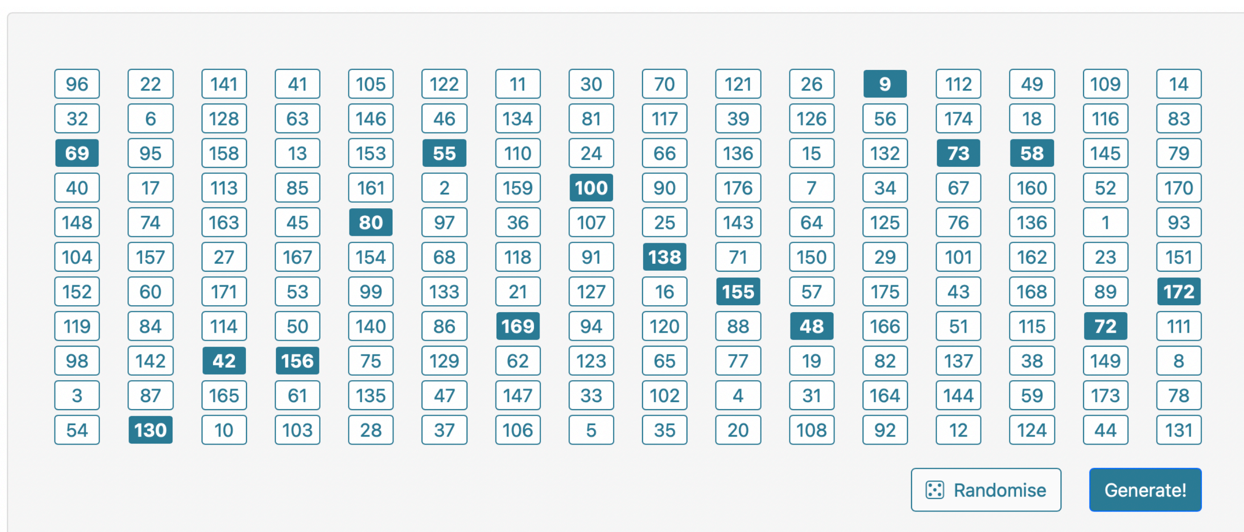

Figure 4.10: the musikalisches würfelspiel interface

By default, the top row of measures is automatically selected. Clicking any number in the grid will select that measure. To ensure only one measure in each column could be selected at a time, an event listener was added to de-select any already selected rows in the same column.

A random button (see figure 4.10) was also provided to automatically choose a random row in each column.

The second task was to create a Score object from the selected measures. This involved stepping through the columns and fetching the measure that corresponded with each selected row. For the first algorithm, Guido, the code only needed to work with a

single melody line. Chords were not required, and the music was played by a single voice. Mozarts composition consisted of two parts however, and each part could contain chords. The Score class needed to be updated to support both chords and multiple voices, as outlined in Chapter 4.1.

```
function arrangeTune() {
  // step through each column, and fetch the corresponding
  // measure from the noteTable
  for (col=0; col<16; col++) {
    if(col === 7){
      // use fixed ending for final measure of first section
      var measureNotes = noteTable[177]
      parseNotes(measureNotes.treble, 0)
      parseNotes(measureNotes.bass, 1)
    } else {
      // fetch the column
      column = document.querySelectorAll(`div.kol-${col}`)
      // find the selected row in the column
      column.forEach(
        function(col) {
          if(col.querySelector('.dice-num-selected')){
            // grab the number of the selected row
            num = col.querySelector('.dice-num-
selected').innerHTML
            // fetch & parse the measure from the note table
            var measureNotes = noteTable[num]
            parseNotes(measureNotes.treble, 0)
            parseNotes(measureNotes.bass, 1)
          }
```

Figure 4.11: building a Score from the selected measures

Another feature of Mozarts composition is that measure 8 only has one possibility, regardless of the outcome of the dice roll. This is reflected in the code which checks for this and loads measure 177 at that point in the composition.

Musikalisches Würfelspiel is another example of a deterministic algorithm - selecting the same rows in the table will always result in the same output.

4.2.3 Markov Chains

With Musikalisches Würfelspiel, the algorithm itself is based on a simple dice roll. What makes the system work is that a human composer has put together a collection of suitable phrases beforehand. If dice rolls were used to randomly pick individual notes, the result is unlikely to be very musical. There are ways to use randomness and probabilities to generate scores that are much more likely to be musical however. One approach is to use stochastic methods, as discussed in Chapter 2.2. Markov chains are another approach.

Markov chains use probability, but unlike stochastic methods, the probability of a particular event happening is based on the previous event, or state. For each change in state of a system, the probability of what that next state will be is calculated. In a musical context, the algorithm could analyse a set of songs and calculate the probability of any particular note following another in those songs. So the state transitions in this case could be the seen as the change from one note to the next.

Markov chains also have an *order* parameter. The order sets the size of the chunk of musical information the algorithm is dealing with at each step - the larger the chunk, the larger phrases it looks at when predicting the next note. Larger chunks, or *n-grams,* result in music that sounds closer to the source material that was used to train the algorithm.

When developing this algorithm, it made sense to find a Markov chains library rather than spending too much time reinventing the wheel. For this, the Javascript library js-

markov[14] was used. It works with numerical and text data, and has a simple API for adding states, setting the order and training.

Before the js-markov library can be used, a formatter is needed that prepares the music for analysis. The pitch and duration of each note will be considered together by the algorithm, so the formatter would need to group each note event into a string containing information about both. Each note event could then be considered as a 'word' by the Markov algorithm.

The user would start by selecting one or more songs for processing (see Figure 4.12). To keep things simple, a selection of songs from two unrelated genres of music were found. The first set were traditional Welsh folk songs, and the second were taken from video games. Limiting the musical choices and offering two distinct sets was a deliberate choice to help the learner get a better understanding of how the algorithm works, and what it can be good for. The learner can compare the results between choosing songs from a single genre verses mixing between the two, seeing which produces the most musical output.

The selected songs were retrieved from the ABC Notation website[15]. The six chosen songs were added to a file in the `_includes/music/` directory and made available using the Jekyll `include` tag. Importing the ABC-formatted music required writing a set of functions to parse Helmholtz notation and convert it to a Tone.js compatible format, which uses scientific pitch notation. As the importer was likely only needed for this algorithm, the code was eventually extracted into a Javascript MixIn that could then be mixed-in to the Score object only when required. This kept the Score class more readable and focused while still keeping the import functionality available when needed. In addition, this mix-in approach provides a framework for adding importers for other formats, such as MIDI or MusicXML, at a later date.

Each song can be previewed, allowing the user to hear what is being sent into the Markov chain in order to compare it with the generated output. As mentioned previously, the Markov algorithm also requires an *order* parameter which is used for dividing the songs into *n-grams*. A slider was added (see Figure 4.12) to allow the user to set this to

---

[14] https://github.com/EdThePro101/js-markov

[15] https://abcnotation.com/tunes

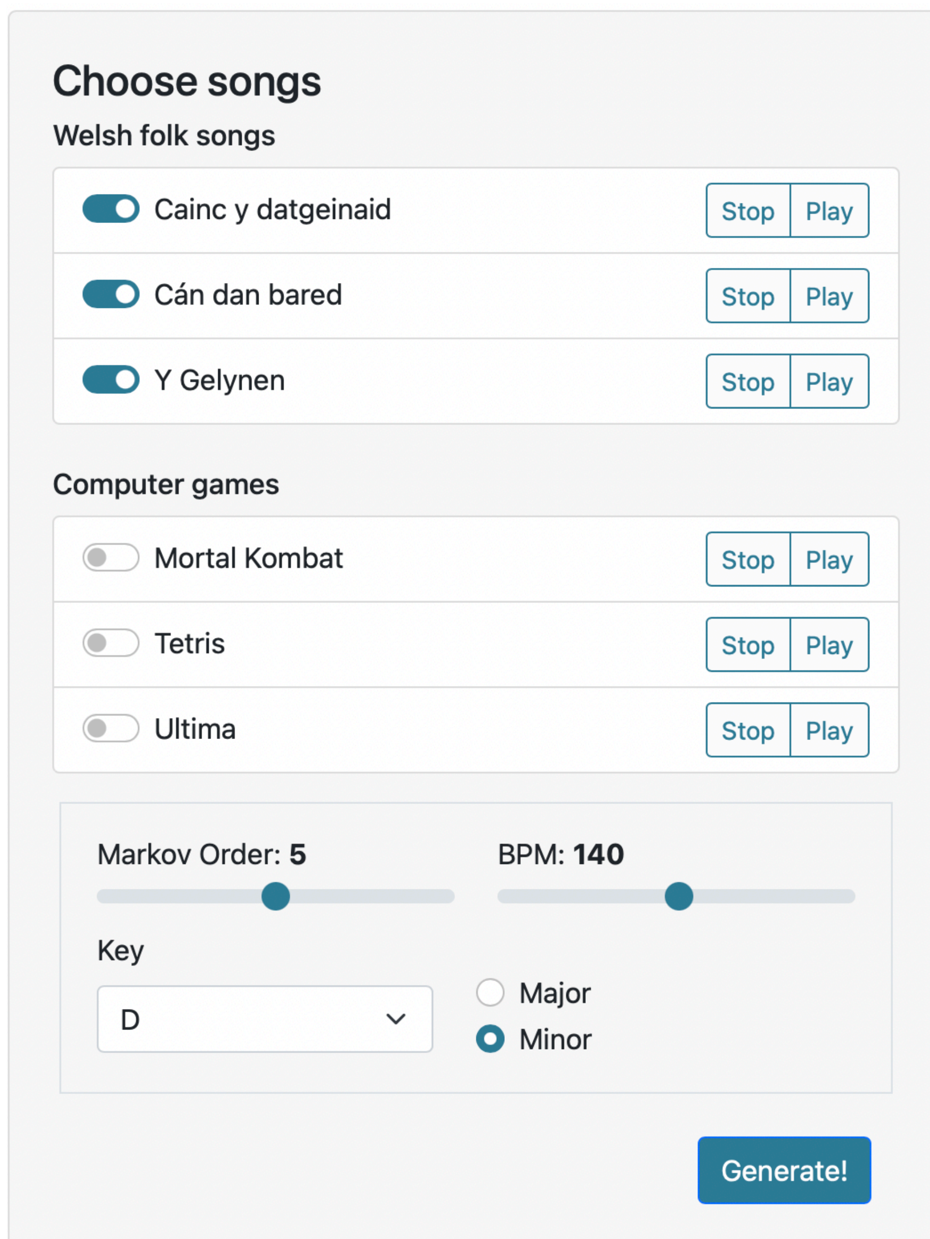

Figure 4.12: the Markov Chains interface

a value between 3 and 9, with higher values producing output with longer phrases taken from the inputted songs.

As the songs are in different keys, the output could get very un-musical. A function to align all generated notes to a particular scale was needed. Tonal.js[16] is a Javascript library for music theory. Among its many functions is a method for fetching all the notes from a particular scale. An `applyKey()` method was added to the Score class. This

[16] https://github.com/tonaljs/tonal

steps through the notes in the score, checking if the note is contained in the scale fetched from Tonal. For those notes that lie outside of the scale, the note is sharpened or flattened until it is in key. A `key-chooser` component was then created in the `_includes` directory, which would allow the user to select the scale and scale type that would be sent into the `applyKey()` method.

4.2.4 Tintinnabuli

Arvo Pärt is an Estonian composer who developed a compositional technique called Tintinnabuli, introduced in *Für Alina* in 1976. Tintinnabuli, from the Latin word for bells, has been characterised as serene and enigmatic, sounding both modern and ancient at the same time (Zivanovic, 2012).

There are two elements that work together in the tintinnabuli style. The M-part represents a melodic line, often moving stepwise through a scale. The second part, the T-part, consists of a note taken from a major or minor triad. The triad can be placed above or below the M-part, or even alternating between the two (Chikinda, 2011).

The basics of the algorithm are relatively straight forward. Step through the notes of a scale, while playing a random note from the scales triad (e.g. notes 1, 3 and 5) underneath. There can be many permutations however. For the M-part, the scale notes can traversed in an ascending, descending, or ascending and descending fashion. They could also just be stepped through at random. The T-part could be played an octave below, or an octave above, the M-part. The chord tones could be extended to include the 7th, 9th or 13th notes.

The interface designed for the tintinnabuli algorithm can be seen in Figure 4.13. For the initial generation of the scale, the `key-chooser` component created for the Markov Chains algorithm (see Chapter 4.2.3) was used together with the Tonal.js library to retrieve the corresponding notes. Once the scale notes have been stored in an array, they can be manipulated using Javascript's built-in array functions such as `reverse()`, or by writing custom functions.

For many pieces, Pärt introduced a rhythmic element to the music. To allow for these rhythmic variations in the Algorithmic Composition Explorer, a sequencer component was created. This consists of a series of steps that can be activated or deactivated, defining a repeated rhythmic pattern for the generated music. Each step corresponds to the note division set in the Score objects noteLength attribute, set to a quarter note by default. To give the possibility of generating more interesting rhythmic patterns, plus and minus buttons were added to the sequencer component, allowing the user to determine the length of the rhythmic sequence. At the same time, the M-part will be stepping through it's eight notes independently of the programmed rhythmic pattern. This technique of applying a rhythmic pattern to a pitch sequence is called *isorhythm*. The incorporation of odd pattern lengths combined with different combinations of active steps allows for the creation of interesting patterns, where the rhythm repeats at a different point than that of the melodic contour.

Figure 4.13: The Tintinnabuli interface

After setting the BPM (which has been constrained to a lower range of between 30 and 100, to reflect the natural calmness of much tintinnabuli music), the user can then work with the parameters of the M and T parts. The direction of the M-part can be set to ascending, descending, ascending/descending or random. This was mostly implemented using standard array methods, although Javascript doesn't have a built in function to randomise an array, so this needed to be coded. The T-part can be placed an octave above or below the M-part, and can use extended chord tones. The transpose method provided by Tonal.js was used for the former. The latter a function that would step through the chosen chord flavour (triad, 7th, 9th, etc.), extracting the notes required from the given scale.

```
// takes a scale, and returns the chord tones
// flavour defines notes required - [1,3,5] etc.
function generateChordTones(scale, octave, flavour) {
  var chordTones = []
  flavour.forEach(i => {
    note = Tonal.Note.transpose(scale[i-1], octave)
    chordTones.push(note)
  })
  return chordTones
}
```

Figure 4.14: generating chord notes

4.3 Extending the Algorithmic Composition Explorer

One of the aims of this thesis was to build a platform which can be built upon and added to over time. The main way the system can be extended is by expanding on the core functionality of the application - adding new algorithms. In addition, there is much scope for adding new instruments and developing the sound capabilities of the system. Another potential avenue of development could be the inclusion of new export options or visualisations. As discussed in Chapter 3.3, the organisation and modular nature of the code makes it easier to build upon and support any future development. This sec-

tion will detail how a new algorithm, instrument or component can be added to the system.

4.3.1 Adding a new algorithm

To add a new algorithm, the following steps would be required:

Step 1: Add the algorithms page to the sites navigation menu. The `_data/algorithm_nav.yml` file contains a list of the page links and names used by the site menu. Adding a new entry here is as simple as adding the following code

```
- name: Twelve-tone Method & Serialism
  link: /algorithms/serialism.html
```

Step 2: Add the HTML page for the new algorithm to the `/algorithms/` directory. Jekyll uses Front Matter to set parameters, so the HTML file needs to begin with the following code:

```
---
layout: default
title: Twelve-tone Method & Serialism
js: serialism
---
```

The *layout* parameter defines which layout file, contained in the `/_layouts/` directory, should be used for this algorithm. There are currently two layouts - `default` and `one_col`. The `default` layout consists of the algorithms description and its interface (a HTML form) in the first row, and a code example in the second. The `one_col` layout is for algorithm interfaces that require the full width of the page. The second row then contains the algorithms description and code example. New layouts can be created as needed, but consideration should be given to providing a consistent experience for the user. See https://github.com/stega/algorithmic_composition_explorer/blob/main/_layouts/default.html for how layout files are constructed.

The *title* parameter is used to set the page title.

The *js* parameter needs to be set to the name of the Javascript file that will be created in step 3 below. The system uses this parameter to load the required Javascript for the algorithm, avoiding the need to load unnecessary scripts.

As mentioned when describing layouts above, there are three sections in the HTML file:

- the algorithms interface, consisting of a HTML form
- the description of the algorithm, and instructions for use
- a code / pseudo-code example

A Jekyll plugin called *ContentBlocks*[17] allows the separation of the sections so they can be inserted into the correct places in the layout file. Two `contentfor` blocks are used - `{% contentfor algorithm_description %}` and `{% contentfor code %}`. Placing the appropriate content in each block will ensure it's inserted into the correct location on the page. See https://github.com/stega/algorithmic_composition_explorer/blob/main/algorithms/guido.html for a complete example of this.

Finally, the notation viewer and audio player are displayed in a modal window. The code for this is found in the `player-modal.html` file in the `/_includes/` directory. Use the `{% include player-modal.html %}` directive to add it to the page. The modal can be triggered using a HTML button as shown in Figure 4.15.

[17] https://github.com/rustygeldmacher/jekyll-contentblocks

```
    <!-- GENERATE NOTES BUTTON -->
      <div class='row my-4 px-3'>
        <div class='col-12 text-end'>
          <button class="btn btn-primary"
                  id="genNotes"
                  data-bs-toggle="modal"
                  data-bs-target="#playerModal">
                Generate!
          </button>
        </div>
      </div>
    {% include player-modal.html %}
```

Figure 4.15: Button to trigger the Audio Player modal

Step 3: Create the algorithms main Javascript file in the `/assets/js/algorithms/` directory. This is a plain Javascript file, using `window.addEventListener('load')` to register callbacks for the GUI components. The `generateNotes()` function can be used to build out the score. Notes can be added to the Score using the `addNote()` method, and calling `render()` on the score object will display the score as notation in the Audio Player modal window.

4.3.2 Extending the Score class

When adding a new algorithm, it might be that the Score class also needs to be updated. For functionality that can be used across algorithms, adding new methods to the Score is the recommended approach. However, for functionality that pertains to a specific algorithm, or for functions that make logical sense to be grouped together, the code can be written in a separate Javascript *mixin* file. A mixin provides methods that implement a certain behaviour, but these are not used alone, they are used to add their behaviour to other classes (*Mixins*, n.d.). This approach keeps the core functionality of the Score class in one place, resulting in a more focused and easier to manage class,

and yet still gives us the opportunity to extend that class with new functionality when required. A mixin was used to add ABC notation importing functionality to the Score class for the Markov Chains algorithm. That particular mixin can be viewed at https://github.com/stega/algorithmic_composition_explorer/blob/main/assets/js/mixins/import-abc-mixin.js. Mixins can be used by using the `Object.assign()` method, as show in Figure 4.16.

```
// add import ABC functionality to Score
Object.assign(Score.prototype, importAbcMixin);
var score  = new Score("Markov Chains");
```

Figure 4.16: Adding a mixin to the Score class

4.3.3 Adding a New Instrument

Adding a new instrument requires the following steps:

Step 1: Add a new instrument file to `/assets/js/instruments/` Instruments are plain Javascript files that consist of a single function that returns a Tone.js instrument. The existing instruments have some reverb applied, and accept an optional delay parameter, in order to provide a basic demonstration how effects can be used. While it was out of scope for this thesis, Tone.js allows for the building of very sophisticated instrument and effect chains, and there is much potential here for future development.

Step 2: Add the instrument to the `instrument-chooser.html` component in the `includes` directory. This component is used to select between instruments, and an option for the new instrument needs to be added here. The instrument-chooser component also allows the passing in of an instruments name as a parameter, so an algorithm can be set to use a particular instrument by default.

Step 3: Finally, the new instrument needs to be added as an option in the `getInstrument()` function in `/assets/js/app.js`. This function instantiates and returns the instrument the user has selected using the `instrument-chooser` component.

5. Evaluation

This chapter considers the various technical challenges encountered while building the Algorithmic Composition Explorer. It discusses the implementation approach taken for the different algorithms, as well as looking at various performance metrics and considerations.

5.1 The User Interface

For the stated use case, the system works well and I believe meets the aims set out at the beginning. I deliberately focused on the functionality and spent little time on the visual presentation, aside from that which was required to make the application usable. The result is something which works, but aesthetically it falls behind what could be expected of a modern Web app. This aspect of the software could certainly be improved by someone with a greater knowledge of UX and graphic design.

5.2 The Algorithms

Guido of Arezzo

Starting with Guido's text-to-notes algorithm made sense, offering a simple and easy to comprehend approach. It is a novel approach to generate random musical ideas, and the ability to tie pitches and note durations to vowels offer some interesting opportunities to try and sculpt the music generated. The output is very much at the mercy of the inputted text however, and as such it is a difficult algorithm to control or predict. Repeated attempts could certainly produce interesting and usable ideas however.

One interesting aspect of this algorithm is that the integral nature of the text to the outputted results can imbibe the generated music with meaning. If the words have meaning for the learner, the generated ideas will perhaps have an extra significance. As well as providing an easily understandable introduction to algorithmic composition, Guido's algorithm could also encourage thinking about what else could be used as source ma-

terial for generating ideas. Phone numbers, car number plates and weather reports from Mars could all be put to use as idea generators when applying such an algorithm, with each idea generated having a special connection with its source.

Musikalisches Wuerfelspiel

Mozarts Musikalisches Wuerfelspiel offers something a little different. The pre-written nature of the music makes it harder to incorporate into the learners own work. However, as a way of shifting focus from individual notes to phrases, there is a lot that can be learned. Randomness can be an effective tool regardless at which level of abstraction it is applied. Questions such as how can chunks of music be re-combined, the role the form or style of music plays in what works and what doesn't are important considerations and learning opportunities for the user.

A question I had when developing this algorithm was how would it work if the user was allowed to upload their own snippets of music? Could this be used in conjunction with the other algorithms, where their output could be added to the grid? Columns could be then assigned different scales or even tempos, giving the phrases either some continuity, or allowing the transposition of ideas as the song progresses. This approach shares much with the digital audio workstation Ableton Live, which can also be configured to randomly trigger a collection of phrases arranged in a grid.

Markov chains

Compared to the previous algorithms, Markov chains are potentially a little harder to understand for the user. They are known for their ability to re-create existing styles of music, but creating something original using this algorithm can be harder. I wanted to bring the learners attention to this aspect of Markov chains by providing two distinct styles of music that can be used as source material, and allowing the mixing and matching of the different songs.

Regarding the development, while using the markov-js library made this aspect of the algorithm quick to develop, there were other parts that were more time-consuming and technically challenging. Writing the ABC importer for the songs that were to be used in the algorithm required a lot of iterations, as each song would introduce new set of ABC

notation features that hadn't been considered, or introduce issues that needed to be handled by the importer. The importer as it stands covers only aspects of the ABC standard, enough for this particular use case, but it is likely to fail reasonably quickly if opened up to allow user uploads of ABC files.

This implementation with its fixed set of source music, while interesting from a learning perspective, is limited for those that want to use it to generate ideas for their own music. The logical next step for this algorithm would be to add the ability for the user to upload their own songs. As discussed above, the current ABC importer would need to be more thoroughly tested before being used in this way. The development of other importers for formats such as MIDI would also open up opportunities for this algorithm. Another option could be to make use of Tone.js's MIDI functionality, and allow the learner to use a MIDI keyboard to play notes that can then be used as source material.

Tintinnabuli

Tintinnabuli is probably the algorithm that offers the most when it comes to giving the user control over the output. Despite this, the algorithms intrinsic nature will always shine through, and it is capable of creating some beautiful and beautifully simple results. While could be seen as a deterministic algorithm, some randomness has been introduced in the T-part with regard to the chord tone that is chosen at any particular step. Even with the parameters left unchanged, running this algorithm multiple times can generate scores with a noticeably different feel.

Aside from the algorithm itself, the sequencer was a new component that was added to introduce some variation. The idea is that this encourages the user to think about the interplay between rhythm and melody, and the interesting effects that can be created when they don't line up.

5.3 Browser compatibility

The CSS framework Bootstrap was used for styling and layouts in the Algorithmic Composition Explorer. This is a widely used and mature framework, having been in development for 10 years, and has built in cross-browser support for its layout and UI

components. To check the constancy of the user interface across browsers, I ran tests in the following desktop browsers:

- Safari 15
- Chromium 89
- Firefox 100
- Opera 86
- Edge 101

In terms of the user interface, the app displayed as expected across all browsers.

Web Audio API support includes all browsers with the exception of Internet Explorer. In Safari however, I noticed that the audio output was lower than in the other browsers. Safari was mainly used during development, and when testing in other browsers, the synthesiser instrument was distorting. Tone.js instruments have a volume parameter which is set to 0dB by default, and lowering this prevented the distortion.

5.4 Mobile performance

During development, I was mainly using the Safari browser on the desktop. I would check that page layout scaled down to mobile dimensions at regular intervals using the Responsive Design Mode built into Safari. Once the app had been uploaded to the Web, it was tested on an iPhone 8 and an iPhone X.

In terms of functionality, the app works well - it successfully generates and plays back audio, and the synthesiser, piano and effects are working as they should.

The notation rendering provided by ABC.js could be improved. It currently scales the score to the width of the container, and on a mobile device, this results in a score that is scaled down to the extent that is becomes difficult to read. There are options for wrapping notation offered by ABC.js that requires a predefined width. One improvement that could be made here is to use the Javascript `screen.availWidth` property to fetch the current screen width, and pass this into the ABC.js render function. Bars per line and note spacing could all be adjusted to improve notation display on mobile devices. The downside of this approach is that the same layout will be used when using the

download score functionality. So any notation PDFs created on the phone will not read very well when subsequently viewed on larger screens.

Generally speaking, layout and styling on mobile devices can be difficult to get right. Bootstrap helps here, offering options for hiding selected content on certain screen sizes, and adjusting parameters like padding and margins depending on the device. These options have been used judiciously throughout development to retain the usability on devices with smaller screens. Using a responsive approach will always be a compromise however, as layouts and forms that work well on one particular screen size will often be less suitable on another. The alternative is to build specifically for each device, but this can greatly increase the amount of development time and expertise needed, and can shackle further development.

5.5 Website performance

I ran the Algorithmic Composition Explorer through website audit tests from two providers - Geekflare[18] and Dotcom-Tools[19]. These are tools that help you test your website from multiple locations and on different devices with various levels of network performance. I used the page for the Tintinnabuli algorithm in the tests, and tested for desktop performance, as well as mobile, which simulates a slightly slower 4G network. The results can be seen in Figures 5.1 to 5.4.

Geekflare showed an average page load time of 0.5 seconds for both desktop and mobile. Dotcom Tools showed an average page load time of 0.73 seconds on desktop, and 0.88 seconds on mobile phone. Geekflare measured the *first contentful paint*, a measure of how quickly content is displayed, as higher than the page load. This suggests that while the files are downloading quickly, the page rendering is slower than it could be. Dotcom Tools results show the opposite however - there the first contentful paint has been measured at 0.5 seconds, a lower value than the 0.73 second page load time that it measured. Nevertheless, these results are under the 3 seconds load time recommended as best practice by Google (An, 2018).

---

[18] https://geekflare.com/website-audit

[19] https://www.dotcom-tools.com/website-speed-test

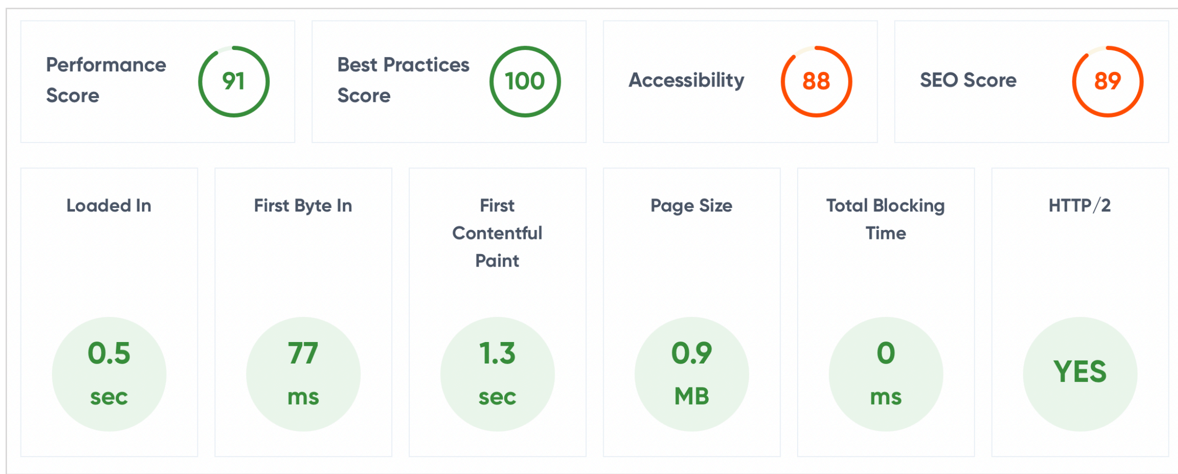

Figure 5.1: Geekflare desktop performance

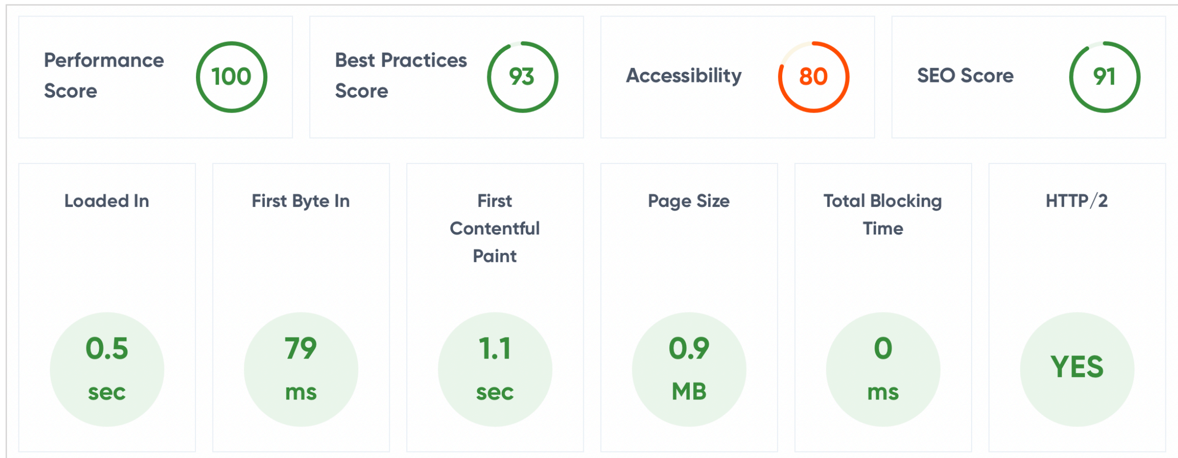

Figure 5.2: Geekflare mobile performance (4G)

First Visit Results — Checks Complete: 1 of 1 Location — Errors From: 1 Location

Location	Perf...	Full Page Load	First Meaningful Paint	Network	DOM Complete	Page Size	
London	Fast	0.73 s	500 ms	214 ms	0.65 s	863 KB	Details ›
Average Duration		0.73 s	500 ms	214 ms	0.65 s		

Figure 5.3: Dotcom Tools desktop performance

First Visit Results — Checks Complete: 1 of 1 Location — Errors From: 1 Location

Location	Perf...	Full Page Load	First Meaningful Paint	Network	DOM Complete	Page Size	
London	Fast	0.88 s	0.70 s	554 ms	0.80 s	862 KB	Details ›
Average Duration		0.88 s	0.70 s	554 ms	0.80 s		

Figure 5.4: Dotcom Tools mobile performance (4G)

HTTP Archive is an open source project that collects a series of reports on Web performance. Their findings show that the median page load times are 3.7 seconds for desktop, and 9.1 seconds for mobile (HTTP Archive, 2022). When measured against their data, the Algorithmic Composition Explorer performs significantly faster than the average.

6. Conclusion

For this thesis I proposed and set out to build a proof-of-principle system for introducing a selection of algorithmic composition approaches to learners that have little or no programming knowledge. I wanted to emphasis experiential learning, and build an interactive environment that focused on learning through practice and experimentation.

To have the most impact, it needed to be widely available, and any barriers to entry needed to be minimised. By building for the browser, the system is widely available. Good page load times, cross browser compatibility and mobile support means that the system is accessible by anyone with an Internet-connected device.

Finally, the system needed to provide a solid foundation for further development. The language and tools used were chosen not only due to their open-source nature, but also because they were popular, well established and supported, based for the most part on standards such as the Web Audio API. With much of the work having gone into the foundations and core structure, the system has been designed to be easy to expand and build upon. The systems modular nature mean that developers interested in adding a new algorithms will not have to consider functionality beyond what they want to work on. The provided Score class, the instruments, audio player and rendering functions, as well as the structure and framework of the whole Web site, all work together to provide a platform on which new ideas can be implemented easily.

Future work

Due to the limited timeframe of this thesis, there are many aspects of the software I would have liked to have developed further. Implementing more algorithms is perhaps an obvious one. As discussed in Chapter 3.4, Cellular Automata would be an interesting algorithm to tackle next. There are also many improvements that could be made to the existing algorithms. For example, allowing the user to upload songs to the Markov Chains algorithm, or even being able to populate the Musikalisches Würfelspiel grid with user-created phrases.

The ABC importer works, but it requires further testing if users are to be allowed to upload their own ABC music. An importer for MIDI was also looked into, as was implementing some of the functionality offered by Tone.js/MIDI. Being able to input notes using a MIDI controller would be an interesting way to feed the Markov Chains algorithm with data, or add phrases to Musikalisches Würfelspiel.

Regarding the rendering of notation using ABCjs, there is room for improvement. While an alternative like VexFlow is more complex to work with, it produces more accurate notation and would be a superior choice.

Aside from the technical improvements, user evaluation is also necessary to further support the validity of the learning-through-practice approach.

During development I found myself wanting to take ideas generated and incorporate them into my own compositions. This lead to several brainstorming sessions where I explored how to adapt the system more for this purpose. I have spoken much about the modularisation of the code, but this is an approach that would be interesting to take with the interface as well. Creating a 'playground', where different algorithmic components can be combined in order to build a customised algorithm without the need for coding would be an interesting next step. In some ways, this could be seen as moving the focus away from learning about different algorithmic composition approaches and more towards creating an environment for algorithmic composition itself. This thesis has very much been focused on the learning through practice approach, and I believe learners could only benefit from more opportunities to do just that.

To summarise, the Algorithmic Composition Explorer is a novel experiential learning environment for algorithmic composition. It is a widely accessible, browser-based tool for experimenting with different algorithmic composition approaches. Finally, it is a framework that can be further built upon and expanded.

My hope is that the Algorithmic Composition Explorer creates a spark for people curious about this fascinating subject, helping them realise the creative potential of the application of algorithms in creating new musical ideas, and inspiring them to dig deeper.

References

Acevedo, A. (2005). Fugue Composition with Counterpoint Melody Generation Using Genetic Algorithms. 3310, 96–106. https://doi.org/10.1007/978-3-540-31807-1_7

Aho, A. V. (2012). Computation and Computational Thinking. Computer Journal, 55(7), 832–835. https://doi.org/10.1093/comjnl/bxs074

Alpern, A. (1995). Techniques for Algorithmic Composition of Music.

An, D. (2018). Think with Google. Retrieved 14 May 2022, from https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

Aschauer, D. (2008). Algorithmic composition [Thesis]. https://repositum.tuwien.at/handle/20.500.12708/11036

Cakewalk—SONAR LE Documentation—Fretboard pop-up menu. (n.d.). Retrieved 1 April 2022, from https://www.cakewalk.com/Documentation?product=SONAR%20LE&language=3&help=Notation.07.html

Chikinda. (2011). Pärt's Evolving Tintinnabuli Style. Perspectives of New Music, 49(1), 182–206. https://doi.org/10.7757/persnewmusi.49.1.0182

Deleuze, & Lapoujade, D. (2006). Two regimes of madness : texts and interviews 1975-1995 (p. 415). Semiotext(e) ; Distributed by MIT Press.

Dimitrov, G. (2021). Arvo [Python]. https://github.com/georgesdimitrov/arvo (Original work published 2020)

Dostál. (2013). Evolutionary Music Composition. In Handbook of Optimization (pp. 935–964). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-30504-7_37

Editor, T.-B. T. (2018, December 13). If Curious, Then Learn: A Brief Intro to Algorithmic Thinking. Tech-Based Teaching: Computational Thinking in the Classroom. https://

medium.com/tech-based-teaching/if-curious-then-learn-a-brief-intro-to-algorithmic-thinking-ba683bf44994

Edwards, M. (2011). Algorithmic composition: Computational thinking in music. Communications of the ACM, 54(7), 58–67. https://doi.org/10.1145/1965724.1965742

Falthin. (2012). Creative structures or structured creativity. Examining algorithmic composition as a learning tool. Norges musikkhøgskole. http://hdl.handle.net/11250/172357

Fernandez, J. D., & Vico, F. (2013). AI Methods in Algorithmic Composition: A Comprehensive Survey. Journal of Artificial Intelligence Research, 48, 513–582. https://doi.org/10.1613/jair.3908

Filimowicz, M., & Tzankova, V. (Eds.). (2017). Teaching Computational Creativity. Cambridge University Press. https://doi.org/10.1017/9781316481165

HTTP Archive: Loading Speed. (2022). Retrieved 14 May 2022, from https://httparchive.org/reports/loading-speed

Isobar. (n.d.). Retrieved 5 May 2022, from http://ideoforms.github.io/isobar/

JavaScript Systems Music. (n.d.). Retrieved 31 March 2022, from https://teropa.info/blog/2016/07/28/javascript-systems-music.html#understanding-timing-in-tone-js

Knuth, D. E. (1974). Computer Science and Its Relation to Mathematics. The American Mathematical Monthly, 81(4), 323–343. https://doi.org/10.2307/2318994

Laurillard, D. (2012). Teaching as a Design Science: Building Pedagogical Patterns for Learning and Technology (1st ed.). Routledge. https://doi.org/10.4324/9780203125083

Logic Pro 9 User Manual: Display Preferences in Logic Pro. (n.d.). Retrieved 1 April 2022, from https://help.apple.com/logicpro/mac/9.1.6/en/logicpro/usermanual/index.html#chapter=44%26section=6%26tasks=true

Manaris, & Brown, A. R. (2014). Making Music with Computers. Chapman and Hall/CRC. https://doi.org/10.1201/b15104

McVeigh-Murphy, A. (n.d.). Computational Thinking, Algorithmic Thinking, & Design Thinking Defined. Retrieved 25 March 2022, from https://equip.learning.com/computational-thinking-algorithmic-thinking-design-thinking

Miller. (1973). Guido d'Arezzo: Medieval Musician and Educator. Journal of Research in Music Education, 21(3), 239–245. https://doi.org/10.2307/3345093

Mixins. (n.d.). Retrieved 1 May 2022, from https://javascript.info/mixins

Most used languages among software developers globally 2021. (n.d.). Statista. Retrieved 7 March 2022, from https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

music21: A Toolkit for Computer-Aided Musicology. (n.d.). Retrieved 7 March 2022, from https://web.mit.edu/music21/

Papert, S. (1980). /Mindstorms: Children, computers, and powerful ideas/. Basic Books.

Papert, S. (1980). Mindstorms, children, computers, and powerful ideas. New York, NY: Basic Books.

Reisenweaver. (2012). Guido of Arezzo and His Influence on Music Learning. Musical Offerings, 3(1), 37–59. https://doi.org/10.15385/jmo.2012.3.1.4

Responsive design—Learn web development | MDN. (n.d.). Retrieved 4 May 2022, from https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design

Selfridge-Field, E. (1997). Beyond MIDI: The handbook of musical codes. MIT Press.

Shapiro, I., & Huber, M. (2021). Markov Chains for Computer Music Generation. Journal of Humanistic Mathematics, 11(2), 167–195. https://doi.org/10.5642/jhummath.202102.08

Shiihs. (2013, August 24). A touch of music: Algorithmic composition: generating tonal canons with Python and music21. A Touch of Music. http://a-touch-of-music.blogspot.com/2013/08/algorithmic-composition-generating.html

Simoni, M. (2003). Algorithmic Composition: A Gentle Introduction to Music Composition Using Common LISP and Common Music. SPO Scholarly Monograph Series. https://doi.org/10.3998/spobooks.bbv9810.0001.001

Tabesh, Y. (2017). Computational thinking: A 21st century skill. Olympiads in Informatics, 11(2), 65–70. doi:10. 15388/ioi.2017.special.10

The Modulariy Principle. (n.d.). Retrieved 7 May 2022, from http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/ood/principles/Modularity.htm

Tillmann, B., Poulin-Charronnat, B., & Bigand, E. (2014). The role of expectation in music: From the score to emotions and the brain. WIREs Cognitive Science, 5(1), 105–113. https://doi.org/10.1002/wcs.1262

Vygotskij, Cole, M., John-Steiner, V., Scribner, S., & Souberman, E. (1978). Mind in society : the development of higher psychological processes (p. 159). Harvard University Press.

Web Audio API - Web APIs | MDN. (n.d.). Retrieved 7 March 2022, from https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

Web Audio API Proposal from Chris Rogers on 2010-06-15 (public-xg-audio@w3.org from June 2010). (n.d.). Retrieved 31 March 2022, from https://lists.w3.org/Archives/Public/public-xg-audio/2010Jun/0010.html

Zatorre, R. J., & Salimpoor, V. N. (2013). From perception to pleasure: Music and its neural substrates. Proceedings of the National Academy of Sciences of the United States of America, 110(Suppl 2), 10430–10437. https://doi.org/10.1073/pnas.1301228110

Zbikowski. (2002). Conceptualizing Music. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780195140231.001.0001

Zivanovic. (2012). Arvo Part's Fratres and his tintinnabuli technique. Universitetet i Agder; University of Agder. http://hdl.handle.net/11250/138506