CITY
UNIVERSITY OF LONDON
EST 1894

# THE DESIGN OF NOTE-BASED ALGORITHMIC SYSTEMS THROUGH THE USE OF MENTAL MODELS

Gilberto dos Santos Agostinho Filho

Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Submitted December 2021

City, University of London
Department of Music

# Acknowledgements

I would like to thank all of the people who have supported me along the way, both professionally and personally, towards completing my PhD. I am incredibly grateful for your unrelenting support, patience, and love, without which none of this would have been possible.

To my supervisor, Newton Armstrong.

To all members of staff of the Music Department of City, University of London.

To my former supervisor, Luboš Mrkvička, and all members of staff of the Academy of Performing Arts in Prague.

To my composition teachers, Mário Ficarelli and Otomar Kvěch.

To my music teachers, Ana Cristina de Paula, Andrew Channing, Christine Ludwig, Hynek Farkač, Michal Rataj, Miriam Němcová, Paulo Rydlewski, Sérgio de Carvalho Oliveira, and Vlastislav Matoušek.

To my dear friends, Andrew Simmons, Beatriz Botelho Gomes, Christine Dysers, Daniel Edward Hough Monteiro, Gabrielle Messeder, Jonathan Higgins, and Soosan Lolavar.

To my partner, Tabitha Poulter.

To my stepfather, Antônio Murillo Zamora.

To my sisters, Ana Lívia dos Santos Agostinho and Viviana dos Santos Agostinho.

To my parents, Eliana Isabel Teixeira and Gilberto dos Santos Agostinho.

# Abstract

This practice-based research, which consists of this dissertation, a portfolio of compositions, and a programming library for Python, explores the role played by metaphors and mental models in providing a framework for algorithmic composers to operate within. When working with algorithms, the composer approaches the act of music creation through an algorithmic lens which, in turn, can often suggest specific compositional ideas. To put it simply, in order to compose with algorithms, the composer must think *algorithmically*, which, in turn, affects how they conceive their musical ideas in the first place. We are shaped by our tools.

This research argues that this algorithmic musical thinking is often realised with the aid of metaphors and mental models. These cognitive mechanisms are crucially important not only for algorithmic conceptualisation but also for their potential to suggest specific ways of organising and manipulating algorithmic ideas, directly influencing the final artwork. As such, this work presents a novel way of approaching algorithmic composition, one in which the composer is consciously designing specific (and possibly idiosyncratic) mental models. From this perspective, composition becomes an investigation of the potentials of these mental models, aligning itself with the notion that algorithmic music can serve as a form of exploration to reach musical results not entirely planned *a priori*.

This approach is utilised throughout the accompanying portfolio of compositions, which is also analysed in this dissertation. The mental models employed in these compositions—particularly those involving musical repetition—are connected to a specific set of aesthetic concepts that underpins this research. The accompanying programming library, Auxjad, provides classes and functions written in Python that directly implement these specific mental models. This library is publicly available online under a permissive free software license, allowing other composers to adapt and incorporate its code into their own practices.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

> *Life imitates art.* We shape our tools and thereafter they shape us. These extensions of our senses begin to interact with our senses. The new change in the environment creates a new balance among the senses. No sense operates in isolation.
>
> —John M. Culkin (1967, p. 70)

A fundamental notion when composing algorithmic music is that algorithmic thinking often suggests specific mental frameworks for the composer to operate within. While the composer selects and manipulates the tools used in their compositional process, they are also affected by this act of selection and manipulation. In other words, in order to work with algorithms, one must be able to *think* algorithmically, which, in turn, affects how one formulates their algorithms. This is an important epistemological notion that also holds true in relation to programming in general; as Chowning (1996, p. xii) profoundly remarks, 'programming concepts can suggest functions that might not occur to one outside of the context of programming.' Likewise, compositional ideas developed while working with algorithmic music might not occur to a composer engaged with a more traditional and non-algorithmic approach.

This research focuses on how working with computers can lead to abstractions of musical processes and materials that are algorithmic-specific. It is particularly centred on the role played by metaphors in algorithmic conceptualisation: that is, how they can aid us in interpreting abstract and complex ideas through more palpable mental constructions. Metaphors are a crucial mechanism for creating the mental models that enable us to better grasp abstract concepts (Lakoff & Núñez, 2000, p. 39). Mental models, which Norman (2013, p. 26) defines as 'the conceptual models in people's minds that represent their understanding of how things work', can help us not only to comprehend an idea but also suggest specific ways of organising and manipulating them. When working with algorithmic composition, metaphors and mental models thus serve the critical role of devices for framing epistemological conceptions. By being able to grasp a complex abstract idea through a concrete mental model, the model itself becomes substantially more malleable and may suggest new forms of operation that might not have been thought of originally.

This practice-based research—consisting of this dissertation, a portfolio of compositions, and a programming library—presents a novel way of approaching algorithmic composition, one in which the composer's starting point is the conscious design of specific (and possibly idiosyncratic) mental models. That is, the main characteristics of mental models are conceived beforehand, and the act of composing becomes an exploration of the potentials of these mental constructs—an idea that has broader applications to other forms of algorithmic art. It is important to emphasise that this is not a linear process since, during the design of a composition, the composer may go back to the metaphors and mental models in order to reevaluate and adjust them accordingly. In other words, metaphors and mental models do not only impact the algorithmic design but can also be influenced back by it, as shown in the diagram of Figure 1.1. As will be demonstrated in Chapter 5 with my *Cartographies* series (2017–2020), a single mental model can be used as the departing point of multiple pieces, each forming a unique sound world but which, nevertheless, could not have been thought of with traditional stochastic methods. This approach is not only discussed in this dissertation but also applied in practice in the compositions that make up my portfolio, as well as implemented in my programming library, Auxjad (Agostinho, 2021).

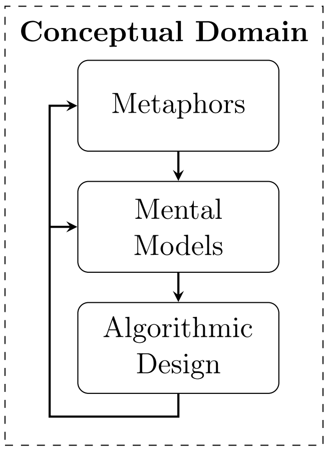

**Figure 1.1:** Diagram of the conceptual domain of my work

One of the starting points of my investigation into the application of mental models to algorithmic music is the research done by Lakoff & Núñez (2000), who apply ideas from cognitive sciences in order to formulate a general theory about how our minds engage with mathematical thinking. Lakoff & Núñez's main argument is that sophisticated and complex mathematical ideas are often conceptualised in our minds through metaphors, building on the previous research done by Lakoff & M. Johnson (2003). These metaphors can map complex abstract concepts into much simpler and more concrete ideas, which are often based on objects that belong to our physical world, thus evoking our experience of them. This mapping between abstract concepts and concrete mental models makes the process of interacting with mathematical ideas substantially more graspable and intuitive, as one can think in terms of these physical objects more instinctively.[1]

Grounding abstract ideas using more concrete and tangible metaphors have implications in fields far beyond the domain of mathematics. In the field of music, one can frame a musical concept such as a tonal scale as a container of ordered elements that is transformable through specific procedures such as transposition. Framing these abstract ideas in such a way can both facilitate musical thinking and lead to new avenues of musical thought. In my own practice as a composer, I use very particular—and somewhat idiosyncratic—mental models when conceiving my compositions. For example, in my *Cartography* series (see Chapter 5), I employ a mental model that I straightforwardly refer to as 'container'. This abstraction has very distinctive properties that dictate how its elements can be manipulated and how they are randomly selected by the piece's algorithm. Its formulation leads to a different approach towards stochastic techniques: musical entities are seen as objects that can be moved around in specific and idiosyncratic ways inside containers of fixed partitions, which are then used as the pool from which the random selections are made. In this same series of works, I also employ other mental models such as a looping window of constant size and the notion of 'input music', which is the piece of music generated by the algorithm for the sole purpose of being the source for the looping process, and which is thus never heard in its original state. These mental models enable me to more easily conceptualise my musical ideas while, at the

[1]One of the many concrete examples that Lakoff & Núñez write about relates to how we are able to conceptualise the infinite set of real numbers $\mathbb{R}$, which are values that can be expressed by an infinite decimal expansion, through a metaphor of points along a line. Each value is associated with a measurable distance from a fixed origin point in the line, and arithmetic operations are realised by shifting distances (Lakoff & Núñez, 2000, pp. 71–74). Therefore, through the easily understood notion of measuring distances along a line, one can grasp—and, most importantly, manipulate—an infinite set of numbers, each with an infinite number of digits.

same time, suggesting specific compositional routes to be taken.

This conceptualisation of mental models bears many similarities with the concept of abstraction in programming, which refers to the modelling of complex structures and behaviours through less complex ones. This is an essential idea when working with the object-oriented programming paradigm (Weisfeld, 2004, pp. 22–29). Objects are algorithmic entities that encapsulate both data and behaviour and which can be derived from others or combined together—through what is known as inheritance and composition, respectively—in order to form increasingly complex structures. Higher-level objects can be directly manipulated and allow for far more straightforward and direct ways to handle complex algorithmic structures.[2]

By working with programming languages that support object-oriented approaches, algorithmic composers can gradually build tailor-made toolboxes derived from the already available tools, making them more suitable for their own compositional needs. Object-oriented programming provides a very flexible environment for the user: classes written by others can be extended or altered, further customising one's tools. An example of this approach can be seen with my own Auxjad library (Agostinho, 2021), which consists of a collection of classes and functions written in Python intended to extend Abjad 3.4 (Bača et al., 2021) to better suit my own compositional needs. This library, which will be discussed in detail in Chapter 4, contains high-level implementations of many algorithmic procedures and mental models that I use in my compositions. These serve as the building blocks that I use to conceptualise and conceive my work.

## 1.1 Processes, Materials, and Aesthetics

One of the main preoccupations of my recent music is to create ambiguous and disorienting listening experiences while employing linear and strict algorithmic processes. Despite the linearity of these processes being evident to the eye when looking at a score, the resulting listening experience often suggests a far richer and more complex design. A significant factor for this comes from my use of repetition and, in particular, the 'looping window' mental model that I employed in many of my recent pieces. Loops are algorithmic processes by nature and, as such, are highly suitable for examination through algorithmic means. When used with the right set of looping parameters and input materials, these processes

[2]Similarly to how mental models and algorithmic design influence one another, designing and manipulating higher-level objects is often not an orderly and tidy process as the programmer frequently needs to reevaluate and adjust its lower-level parts during usage.

can sound less strict than their implementation might suggest. This further challenges the listener's ability to assess whether the music is changing or not, giving rise to a sense of disorientation and blurriness. These musical processes can often suggest specific musical materials to be used as input, materials that emphasise some of the processes' characteristics. However, the choice of musical materials can also influence the design of these processes, as the generated music results from the combination of both. This two-way relationship is represented by the diagram shown in Figure 1.2.

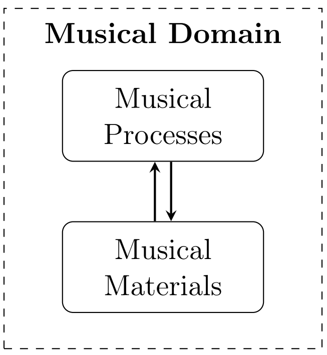

**Figure 1.2:** Diagram of the musical domain of my work

Besides this interest in perceptual disorientation, another significant element that informs the aesthetics of my practice is the notion of emergence. In the context of algorithmic music, emergence is the phenomenon that refers to complex high-level properties and behaviours of a system that cannot be accounted for solely by its parts and simple rules (Pearson, 2011, p. 108). The concept of emergence is present in many fields, including not only programming but also sciences and arts. One such example from biology is the behaviour of flocks of birds: although each individual bird in a flock reacts only to its own stimuli and surroundings, the flock itself behaves as a single coherent entity that moves as one, despite lacking any centralised decision-making. In the case of my music, complex emergent structures can arise from interactions of the simple algorithmic processes that create these pieces. Emergent structures predominantly occur at the borders of consecutive looping windows, where materials that were not originally consecutive are heard side by side. In the right circumstances, such as when elements display pitch and temporal proximities (Bregman, 1990, pp. 455–528), these now consecutive notes and chords can end up grouped by our ears as one single linked structure. In such cases, emergence results from the same mental model initially employed to intensify the listener's disorientation. At the same time, emergence can also contribute to the overall sense of disorientation of a piece: when emergent structures appear at the borders of looping windows, they help mask the location of these borders, further obfuscating the linearity of the looping process taking place. The mutual influence between these aesthetic concepts is represented in Figure 1.3.

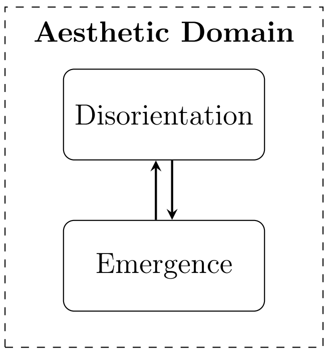

**Figure 1.3:** Diagram of the aesthetic domain of my work

My work also displays a predilection for fragile musical materials—namely, those that are soft and slow. In my music, this type of material serves not only an aesthetic purpose but also a perceptual one: they contribute to the disorientation caused by the use of near-repetition procedures, making the resulting music more challenging to be grasped solely by ear. Similarly to the case of emergent structures, the edges of the looping windows are often not apparent to the listener due to these fragile materials, and, as such, they can help mask the linearity of the musical process that is constantly unfolding in the background. Therefore, there is a clear link between these aesthetic and musical notions: repetition processes and fragile materials are employed for their ability to generate disorienting experiences and emergent structures, while these aesthetic principles also suggest specific materials and processes. This relationship is shown in Figure 1.4.

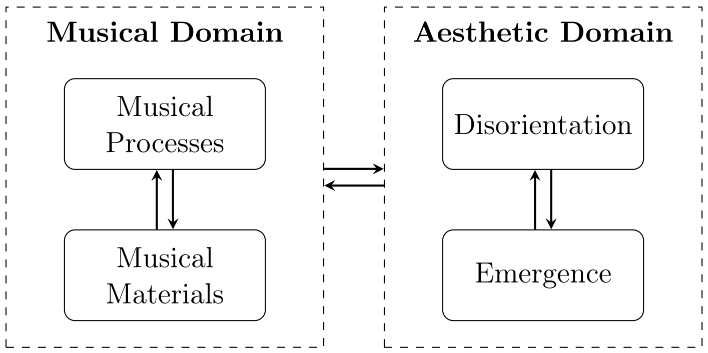

**Figure 1.4:** Diagram of influences between the musical and aesthetic domains of my work

These musical and aesthetic notions are also intrinsically dependent on the conceptual entities that formalise the algorithmic composition. Specific aesthetic concerns, musical processes, and musical materials are not only informed by the algorithmic process but can also suggest specific metaphors and mental models, as well as influence how they are reshaped during the algorithmic design phase. As such, these concepts form an entangled network: that is, each informs and is informed by all others, an idea represented in the diagram shown in Figure 1.5.

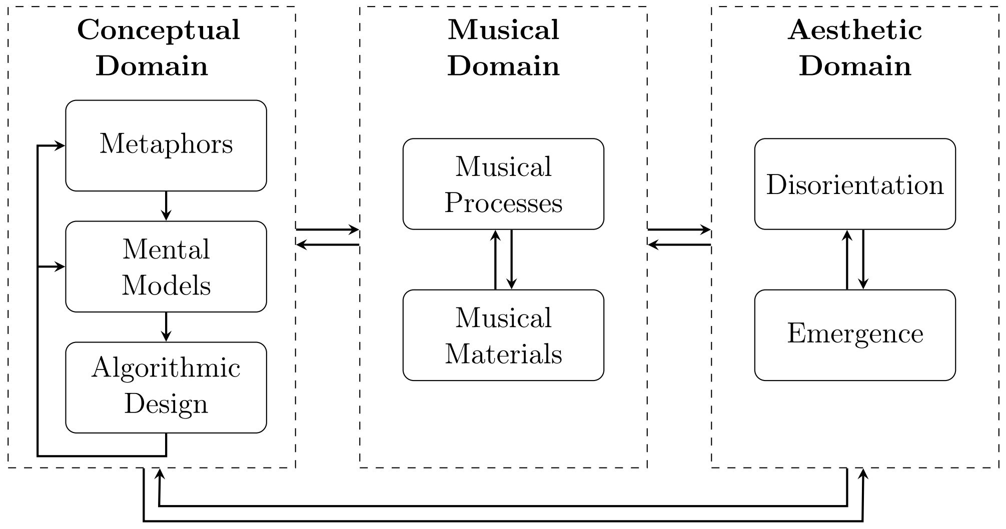

**Figure 1.5:** Diagram of influences between all domains of my work

## 1.2 Algorithmic Exploration

Algorithms allow us to create whole worlds in which artistic discourse can take place. Abstraction is a crucial component for shaping these worlds: it enables the creation of high-level entities—the objects the artist interacts with when working—while, at the same time, keeping the lower-level implementation and mechanisms out of sight. While abstractions serve as a fundamental programming technique for structuring code, mental models and metaphors serve as equivalent notions for how our minds conceptualise these entities. Roberts & Wakefield (2018, p. 303) argue that abstraction is not a mere 'structural convenience' but can actually lead to a 'model of the world' which the musician interacts with during their creation process. This notion is supported by Rohrhuber, Campo, & Wieser (2005), who write:

> Programming languages allow the formulation of such algorithms, not only for the computer to actualize them, but at the same time, to maintain a

discourse with a model, a portrait of some world with its own rules.

This interwoven relationship between the algorithmic world and our conceptualisation of it emphasises a critical aspect explored in the research of Hamman (2000c, pp. 7–8): algorithms are never ideologically neutral. As such, Hamman argues that technology becomes embedded in the algorithmic work itself and thus informs part of its aesthetics. Because of this, the technological origin of these works become an intrinsic part of the final artwork. This aligns algorithmic music closer to conceptual art, where neither the process of creation nor the object of art itself can be disassociated from one another (LeWitt, 1967a). Some authors, such as Nake (2010) and Mohr (quoted in Hattrick & Mohr, 2012), go as far as to argue that algorithmic art and conceptual art are, in fact, two highly similar attitudes to art, mainly due to their use of recipes for realisation, focus on processes, and neglect of materiality. Both argue that the main difference between these two approaches lies in the hierarchy between concept and realisation: while conceptual art gives more weight to the former, algorithmic art considers both as essential aspects of the final artwork (Nake, 2010; Hattrick & Mohr, 2012).

An essential aspect of algorithmic art is that it enables artists to create works that could not otherwise be conceived using non-algorithmic techniques. This is a consequence of the idea that the algorithms can suggest specific frameworks within which one operates—as such, working with algorithmic music becomes an exploratory activity guided by heuristics, particularly when the algorithmic systems are not deterministic. Essl (2007, p. 108) notes that computers enable him to transcend a 'limited personal horizon', while Brün (2004, p. 120) writes that he 'learns' from the aesthetics of his results. Code thus became part of the compositional process as well as of the composition itself (Magnusson & McLean, 2018, p. 262). Such 'formalised abstractions' (a term used by McLean & Dean, 2018a, pp. 5–6) extend our musical thinking and, in the process, expand our musical horizons. As such, music created with algorithmic means is inherently algorithmic in nature: algorithms are not merely tools used for applying musical processes to the composition materials, but they are also part of the nature of these very processes and materials.

In this context, the computer can be considered as means for expanding personal horizons, enabling composers to reach musical results outside the scope of non-algorithmic music. Because of this, the algorithmic compositional process becomes fundamentally interactive: the musical results will contain elements that were not pre-planned by the composer—particularly when working with generative methods—and therefore, the composer is constantly shifting between tweaking the system and evaluating its partial results. Even in a deterministic context, it is impossible for someone to foresee all details of a complex algorithmic composition. As Oberholtzer (2015, p. 246) states, ‘Any sufficiently complex, but finite, fixed pattern of values is liable to be indistinguishable to a listener from a random sequence.’ This brings algorithmic music towards the post-human territory: the composer’s perspective is expanded by the machine, allowing for previously unreachable results that are created through a form of symbiotic collaboration. Algorithmic music, and more generally algorithmic art, can therefore become a significant method for investigating the relationship between people and technology.

## 1.3 Notes on Terminology

My work is aligned with what Landy defines as ‘note-based music’, which is the type of algorithmic music focused on generating music scores for instrumental performance, as opposed to ‘sound-based music’, whose primary focus lies in audio manipulation (2008). I am particularly interested in automatic systems that, once set in motion, can run by themselves, requiring no further intervention by the composer (Reich, 2002b, p. 34; Eno, 1996, pp. 330–331). When describing this approach, the term ‘generative music’ will be used according to the categorisation done by Levtov (2018), who defines ‘generative’ as referring to algorithmic music that runs without user input, in contrast with the terms ‘reactive’ and ‘interactive’, which refer to music that responds to environmental input and music that requires end-users to directly interact with the algorithm in order to produce output, respectively. Although the focus of this dissertation resides on the type of music that is the most relevant for my own compositional practice, the ideas explored are applicable to other types of algorithmic music too.

These definitions are particularly necessary in the field of algorithmic music since much of the terminology used carries historical and aesthetic connotations with them and often have slightly different meanings depending on the composer or group of composers being addressed (Ariza, 2005, p. 4; McLean & Dean, 2018a, pp. 5–6; Roads, 1996). As such, I will avoid the terms ‘computer music’, ‘stochastic music’, ‘computer-aided composition’, and ‘automatic music’ concerning my work, although, occasionally, I will use them in relation to composers who used that terminology to describe their own work. The word ‘automatic’ will be used in relation to automatic systems or automatic processes, while ‘stochastic’ will be used in the context of random procedures.

## 1.4 Dissertation Structure

Chapter 2, 'Why Algorithms Matter', sets out the context of this research project, identifying key literature sources. There are five key points that this chapter will focus on. First is the notion that algorithmic composition can be approached as a form of exploration, particularly when algorithmic systems are designed using non-goal-oriented strategies. Next is the idea that technology embeds itself into the final artwork, becoming an integral part of the aesthetics of the result. This chapter also discusses the different approaches composers can take when designing algorithmic systems, giving particular attention to generative systems. Next, it examines how algorithmic thinking takes place and how it relies on metaphors and mental models as fundamental aids for constructing compositional frameworks. Finally, the interdisciplinary relevance of this research topic is addressed, together with its relation to other art forms.

Chapter 3, 'Aesthetic Dimensions', focuses on the set of aesthetic concepts that underpins my compositional practice. Individual sections are dedicated to each of the four main aesthetic concepts I explore in my music, namely 'slippage', 'fragility', 'emergence', and 'liminality'. This chapter establishes how these concepts developed from my looping processes and their relationship to repetition, a key element of my recent music. It also demonstrates how these concepts interact with one another, as well as how they can be used for creating music that is ambiguous and perceptually disorienting despite being composed using linear algorithmic processes. These concepts will be framed from the point of view of perception and listening experience, exploring their relation to Gestalt grouping mechanisms.

Chapter 4, 'The Auxjad Library', focuses on the methodology employed in my programming library, Auxjad. It discusses how mental models can be formalised as code through object-oriented programming and how they can be used to build ever-more complex algorithmic entities. The main tools that I use to compose my music, namely LilyPond (Nienhuys & Nieuwenhuizen, 2003) and Abjad (Bača et al., 2015), will be introduced together with my own programming libraries, lilypondLibrary (Agostinho, 2019) and Auxjad (Agostinho, 2021). The latter is the focus of a large section of this chapter in which I first discuss the motivations behind it and then illustrate its capabilities by going over some of its members.

Chapter 5, 'Commentary on My Music', consists of an extended discussion on the music that makes up my portfolio of compositions. It starts with an overview of the methodology used in these works, particularly the definition and characteristics of the ‘container’ mental model, which was the starting point of my *Cartographies* series. This chapter discusses how this specific mental model led to particular ways of approaching and formalising the works in *Cartographies* and how it is connected to the aesthetics of these pieces. Musical repetition, and in particular the use of the ‘looping window’ mental model, are also the focus of the discussion. Selected pieces from this series will be analysed, together with works written shortly after it, in order to illustrate the ideas explored in this dissertation up to that point.

Lastly, Chapter 6, ‘Summary and Conclusion’, contextualises the research findings of this project and demonstrate how my portfolio aligns with the ideas explored in this dissertation. Possible future directions for this research are addressed, both in terms of writing as well as composition.

In combination with this written dissertation, this practice-based research project also consists of a portfolio of compositions and a programming library written in Python, which will be discussed in depth in Chapters 4 and 5, respectively. My portfolio of compositions informs and showcases many of the ideas discussed in this text, addressing questions such as how mental models can shape compositions and how the specific set of aesthetic concepts addressed in this text can be articulated in practice. Table 1.1 lists all compositions submitted as part of this doctoral project, with their year of composition, instrumentation, duration, and availability of recording as of December 2021. Links to all scores and available recordings can be found at `https://github.com/gilbertohasnofb/PhD_portfolio`. With my programming library, Auxjad (Agostinho, 2021), I sought to demonstrate how the mental models used in my work could be implemented using an object-oriented approach and to develop a practical toolkit of compositional tools for my own use. This library is publicly available online, released under the MIT License, granting permission to any other composers to adapt and distribute its code as they wish. Auxjad’s source code and documentation page are available in the links `https://github.com/gilbertohasnofb/auxjad/` and `https://gilbertohasnofb.github.io/auxjad-docs/`, respectively.

| **Composition** | **Year** | **Instrumentation** | **Duration** | **Recording Available** |
|---|---|---|---|---|
| *Cartography #1* | 2017 | Pno. Vib. | 4'30" | ✓ |
| *Cartography #2* | 2017 | Pno. | 6'30" | ✓ |
| *Cartography #3* | 2017 | Gtr. | 3'00" | ✓ |
| *Cartography #4* | 2017 (rev. 2020) | Hp., Vla., Fl. | 14'00" | |
| *Cartography #5* | 2017 | Vln., Pno. | 4'00" | ✓ |
| *Cartography #6* | 2017 | Pno. (4 hands) | 8'00" | |
| *Cartography #7* | 2018 | 4 Electric Gtr. | 11'00" | |
| *Cartography #8* | 2018 | Fl., Sax, Vln., Vcl., Acc. | 8'30" | ✓ |
| *Cartography #9* | 2018 | Cl., Vla., Vib., Pno. | 9'00" | ✓ |
| *Cartography #10* | 2018 (rev. 2020) | Mar., Vib., Pno. | 8'30" | |
| *Cartography #11* | 2018 | Pno. | 7'00" | ✓ |
| *Cartography #12* | 2019 | Fl., Cl., Vln., Vla., Vcl. | 10'00" | ✓ |
| *and thereafter they shape us* | 2019 | Vcl. | 14'30" | ✓ |
| *adrift* | 2020 | Pno. | 36'00" | |
| *what holds them together* | 2020 | Portative Org. | 14'00" | ✓ |
| | | **Total:** | 2h38'30" | 1h21'00" |

**Table 1.1:** List of compositions submitted as part of this doctoral project

# Chapter 2

# Why Algorithms Matter

Algorithms are everywhere. Ever since the so-called 'Algorithmic Revolution' took place in the 1930s (Weibel, 2007; 2004), technology has silently taken over most aspects of our lives, becoming entrenched in both social and political spheres. Since the 1960s, artists of various disciplines have employed algorithmic ideas in their practice, not only for their potential for generating open-ended exploratory approaches to art-making (Eno, 1996, p. 330) but also for the epistemological framework they provide for artistic investigations (Hamman, 2004, p. 121).

This chapter will focus on the questions raised by the use of algorithms as well as the literature on the topic that supports my research project. It will discuss some of the motivations for working with algorithms in the context of art creation and the aesthetic and epistemological questions that this type of practice raises. Then, it will consider various approaches taken when designing algorithmic music systems, and in particular generative systems. This algorithmic thinking will then be connected with the notions of metaphors and mental models and will examine how these can affect the formalisation of algorithmic systems in the context of arts. Finally, this discussion will be expanded to consider interdisciplinarity, exploring the relevance of algorithmic thinking in other artistic disciplines such as architecture, painting, photography, sculpture, and literature.

## 2.1 Algorithmic Composition as a Form of Exploration

One of the biggest allures of working with algorithmic composition is that it facilitates the creation of musical results that were not entirely planned *a priori*. Algorithms may not only suggest specific aesthetic and epistemological questions, but they also enable artists to achieve results that would not be imaginable without the computer. This is not solely due to the computer's ability to promptly carry out complex computations—which are orders of magnitude faster than manual calculations—but also because working with algorithms constitutes an entirely different framework for an artist. Therefore, the relationship between composer and computer is one of extension, as opposed to a merely utilitarian link: the device widens the composer's artistic horizons, allowing them to create musical results that could only be conceived by working within an algorithmic framework. Concerning this extension enabled by the computer to the composer, Ariza (2005, p. 2) writes:

> It is an error to think of algorithmic composition as a replacement of humans with music-writing machines. The techniques of algorithmic composition are employed at a great variety of compositional levels, often in a complex mixture of algorithmic procedures and human choice, and always bound by musical interpretation of a human composer. The levels and mixture may be so intermingled as to make crisp distinctions impossible. Often, a composer's original materials are modified by composer-designed processes. Often, the composer is expanded, not removed. Fixed materials, algorithms, and the tuning of algorithms all become compositional materials.

This 'tuning of algorithms', described by Ariza above, constitutes an essential aspect of working with algorithmic music. While a composer might initially write a system with concrete musical ideas in mind, most algorithmic systems—particularly generative ones such as those employed in my creative practice research—will produce a complex output that the composer cannot entirely predict. One of the reasons for this is that such systems often employ processes whose sheer combinatorial complexity goes beyond what the human mind is capable of calculating within a reasonable time frame. Furthermore, generative systems that make use of stochastic processes or randomly generated material will produce output that is the result of random procedures, which, by design, cannot be wholly anticipated prior to generation. For example, in relation to his own generative systems, Brian Eno (1996, p. 330) writes that 'the point of [his systems] was to make music with materials and processes I specified, but in combinations and interactions that I did not.'

Therefore, one of the main advantages of working with algorithms is that 'the autonomous system does all the heavy lifting; the artist only provides the instructions to the system and the initial conditions' (Pearson, 2011, p. 4). This is perhaps the most important technological aspect enabling the exploration of artistic territories beyond the limits of more traditional approaches. When unshackled of the limitations of manual implementation, artists can quickly explore a vast number of different processes, materials, initial conditions, and constraints before committing to a specific result. The intermediary results from this experimentation phase may well suggest new paths to be explored for the final artwork, paths that were not visible beforehand. As Nierhaus (2010, p. 56) describes it, 'Even simple algorithms stretch beyond our cognitive resources, because of lack of precision, lack of speed, or limited working memory. So, the only way to predict the result is to execute the algorithm.'

The algorithmic composition process, therefore, will often require a high degree of experimentation and heuristics, creating a feedback loop of delineation of algorithmic processes and evaluation of the output. In my own practice, it is often at this experimental phase of the compositional process that the work starts to take shape: specific characteristics of the musical output can be amplified or dampened, alternative models can be tested out, parameters can be coupled and decoupled, single processes can be extended as multiple different procedures, and compositional elements can be multiplied or deleted altogether. By the end of this process, the final work can bear little semblance to the original conception of the system and its initial output. In relation to this 'expanded investigation', Essl (2007, p. 108) writes,

> [the use of computers] enables one to gain new dimensions that expand investigation beyond a limited personal horizon. From this basis, algorithms can also be regarded as powerful means to extend our experience—they might even develop into something that may be conceived as an 'inspiration machine'.

Algorithms thus become an extension with which artists create. They are not mere implementations of pre-conceived materials or pre-defined processes but instead are entities that interact with the artist, allowing for unseen and unforeseen materials, processes, and, above all, results. When working with computers, 'code becomes increasing [sic] important to the creative process, in generating surprising results that are otherwise beyond the imagination of the programmer' (Magnusson & McLean, 2018, p. 262).

Supper identifies three distinct categories for classifying algorithmic music: modelling traditional non-algorithmic procedures, modelling original algorithmic procedures, and using algorithms from other disciplines (2001, p. 48). The first category, algorithmic music that models traditional non-algorithmic procedures, was the most prevalent in the beginnings of computer music when composers implemented established compositional approaches such as the twelve-tone system or Fuxian species counterpoint. Examples of works created with these procedures include some of the early experiments by Lejaren A. Hiller & Leonard M. Isaacson (Supper, 2001, p. 49; L. A. Hiller & Isaacson, 1958), Gottfried Michael Koenig's programs *Projekt 1* and *Projekt 2* (Essl, 2007, p. 115; Koenig, 1983, p. 30), and the work of David Cope, who uses algorithms to compose tonal music which imitates the styles of specific Baroque and Classical composers (1991; 2004). The second category, modelling original algorithmic procedures, is perhaps the most representative of the three and the one utilised most frequently in my compositional practice. It includes the majority of computer music works by early pioneers such as Herbert Brün (2004) and Iannis Xenakis (1992), as well as works such as George E. Lewis's *Voyager* (1987), a real-time improvisation system (Lewis, 2000; Steinbeck, 2018), Karlheinz Essl's *Lexikon-Sonate* (1992–2020), a piano sonata for real-time computer-controlled piano (Essl, 2018), and Clarence Barlow's *Çoğluotobüsişletmesi* (1975–1979), generated using the author's own AUTOBUSK system (Barlow, 1990; Supper, 2001, pp. 49–50). The final category relates to music created using algorithms from other disciplines, such as those inspired by natural phenomena. Hanspeter Kyburz's *Cells* (1993–1994) is an example of such composition; it makes use of L-systems, which are generative self-similar systems that can be used to model the morphology of plants (Supper, 2001, pp. 50–53). Some of Xenakis's pieces can also be considered as being part of this category. These include works such as *Pithoprakta* (1955–1956), which uses equations from statistical mechanics of gases, and *N'Shima* (1975), which implement ideas of Brownian motion (Xenakis, 1992, pp. 15–18, xiii). The work of Agostino Di Scipio, in particular his *Audible Eco-Systemic Interface* project, is another example fitting of this category. Di Scipio implements what he calls 'bio-cybernetic principles', biologically inspired algorithms for modelling dynamic interactions of audio signals (Di Scipio, 2003; Eldridge & O. Brown, 2018).

Naturally, there are multiple ways of categorising a field as vast as algorithmic music. Arguably, Supper's last two categories—modelling original algorithmic procedures and using algorithms from other disciplines—can be considered subcategories of a single larger one, defined by an intrinsically-algorithmic approach to music. Nierhaus similarly defines only two categories of algorithmic music which he names 'style imitation' and, somewhat controversially, 'genuine composition methods' (2010, pp. 259–260). From my own perspective as a composer, the notion of the algorithm embedding itself in my musical work is of significant importance to me. The main focus of this creative research lies in the investigation of how algorithms affect my music and my musical thinking rather than simply reaching pre-defined goals. As Shanken (2014, p. 13) succinctly describes:

> The computer's most profound aesthetic implication is that we are being forced to dismiss the classical view of art and reality which insists that man stand outside of reality in order to observe it, and, in art, requires

the presence of the picture frame and the sculpture pedestal.

Harper's conception of composition as manipulation of musical objects also reinforces this idea (2011, pp. 83–87). By abstracting musical material as objects that can be manipulated as well as interact with one another, Harper effectively invites composers to think in terms of relationships of objects and processes, both of which are fundamental to algorithmic thinking. What distinguishes Harper's musical objects from traditional music parametrisation (such as the division of sound into pitch, duration, dynamic, and timbre) is that they can accommodate relative and subjective entities as well (2011, pp. 88–91). Moreover, his objects can be joined together, creating new higher-level ones in the process; music is then built through the relationships between increasingly complex objects. This conception is highly suitable to be implemented using computers and very closely models the approach utilised throughout the accompanying portfolio.

## 2.2 Aesthetics of Algorithmic Art

An important principle of technological art forms, such as algorithmic music, is that technology is not aesthetically neutral and, as such, embeds itself in the final artwork. By making use of technology, artists engage with its aesthetic questions, which cannot be completely disassociated from the final artworks. In other words, technology becomes an intrinsic part of the artistic output. In relation to algorithmic music, these ideas are supported by Hamman (2000c, pp. 7–8), who writes:

> A musical work can no longer be accounted for purely through examination of the acoustical experience it engenders or the formal structure it may exhibit. As technological, the work constitutes both the result and the technological forms by which the result was realized. These include the particular technical tools plus the attitude of the subject under whose unfolding those tools were taken up and applied. To equate music solely with the results of its productive activity is to disembody the result from its technique—it is to fetishize the musical work, converting it from a catalyst for experience into a commodity to be traded within an economy, whether financial or ideological.

For Hamman, technology is an integral part of the artwork and thus constitutes a critical element of the resulting aesthetic posture.[1] This approach

[1]When writing about Xenakis's *ST/10-1, 080262* (1962), Keller & Ferneyhough go as far as to argue that, due to its stochastic nature, understanding this composition's underlying mechanisms is a *prerequisite* for understanding the piece itself (2004, p. 161). Although I myself do subscribe to Hamman's view that technology becomes an intrinsic part of the resulting

requires understanding technology as both social and cultural phenomena instead of a simple utilitarian tool used to reach pre-defined goals. Based on the ideas of Andrew Feenberg, Hamman writes about two contrasting approaches to the use of technology in arts, namely 'instrumental theory' and 'substantive theory' (2004, p. 116). Instrumental theory considers technology merely as a tool that exists beyond any social and cultural influences. From this point of view, technology is intrinsically apolitical and stands apart from society. Substantive theory comprises an opposite approach: technology is anything but neutral and can serve as a tool for social and political dominance, which in turn can lead to a total rejection of technology (Hamman, 2004, p. 116). Hamman then defines a third approach which he calls the 'critical theory of technology'. It states that the relation of technology and society is malleable, as opposed to the immutable notion used by both preceding theories. From this point of view, the social, cultural, and political contexts in which technological tools are created become a fundamental aspect for our understanding of them (Hamman, 2004, pp. 116–117). It is through the lens of the critical theory of technology that I consider my own use of algorithms.[2]

For Hamman, composers working with algorithmic systems will often challenge the instrumental view of technology that is commonplace outside the algorithmic art world (2000c, p. 8). Algorithmic composers can use technology in ways that transcend the simplistic 'problem-solving tool' metaphor. In doing so, technology helps inform the context in which the artwork is created and determine both cognitive and epistemological aspects of the resulting piece. As Hamman (2000c, p. 8) writes,

> technology *preserves* the problematic of compositional process rather then [*sic*] attenuating it, while the technical thing is transformed from an object for the social mediation of cognitive and epistemological activity to an object through which humans explicitly and experimentally participate directly in the shaping of that activity.

These points bear parallels to some of the core ideas of McLuhan's writings on

music and is strongly connected with its aesthetic results (2000c, pp. 7–8), I do not believe that the listener must understand the intricacies of the algorithmic mechanisms by which these pieces are created in order to understand and appreciate the resulting music. To me, the acknowledgement that a work is the result of a purely technological process, one that engages with generative and stochastic processes and whose automatism is dependent on computers, is enough to inform the listener of the context by which the piece was created and help guide them towards a different way of listening.

[2]It is important to point out that the notion of technology not being aesthetically neutral is not a recent idea; as Roads points out, this idea had already been raised in relation to algorithmic music by Herbert Brün in as early as the 1960s (1996, p. 857).

medium theory: technology, as an integral part of the artwork, also becomes part of the artwork's *message*. McLuhan (1964, p. 23) writes, 'the personal and social consequences of any medium—that is, of any extension of ourselves—result from the new scale that is introduced into our affairs by each extension of ourselves, or by any new technology.' To consider algorithmic music solely as the sounding result that emerges at the end of the creation process is to selectively disregard the technology that enabled such music: in other words, it is to ignore part of the artwork's message as well.

Another important aspect that informs algorithmic art is the fact that algorithms have become pervasive entities, permeating virtually every single aspect of our social and cultural lives. This phenomenon is the result of what Weibel calls the 'Algorithmic Revolution', a silent revolution which saw algorithms being disseminated into every corner of our lives. According to him, this revolution lay entirely behind us and went practically unnoticed, resulting in these invisible algorithms (Weibel, 2007; 2004). This latter idea is also supported by Hertz (2009, p. 59), who argues:

> Computers have become devices at once ubiquitous and practically invisible. Microprocessors hide in plain sight in cars and kitchens. Digital communications span the globe. The personal computer and the internet are only the most obvious sites of computational technology. Every medium bears its mark, as do the clothes we wear and the food we eat. It permeates our society, yet we are oddly oblivious to it. For a while, everything new and wonderful was 'digital'—now the term elicits little more than a yawn, the consumer's ultimate revenge.

Against this passive attitude towards technological progress, algorithmic artists can be instrumental in raising awareness with their audiences, inviting them to engage with certain aspects of technology that would commonly go unnoticed—or merely elicit a 'yawn'. When technology becomes embedded in artworks, it invites the audience to acknowledge and react to it, which in turn can challenge preconceived notions that they may have. Uliasz makes the case that through 'misuse' and 'appropriation', an artist can overcome pre-established perspectives of a given technology, purposefully shifting it towards an anti-capitalist and non-commodified aesthetics (2017). She writes:

> In my practice, I identify my fundamental aim to engage with such a 'perversion' through working with pre-existing structures and systems. Through a practice of misuse, decontextualizing technologies from a typical structure of information transmission and production is used to reveal the structure of the technology itself. My background in the humanities and lack of formal education or corporate involvement with information technology has required that I perceptually and ethically understand the tool before I

> might begin to figure out how to work with it. My role as an artist rather than a developer, as someone that is working from within an altogether different 'structure of participation,' is not to develop a technology in order to advance or complicate its function, but rather to develop an alternative use that may seem unintuitive or even counterproductive to some.

It is through these notions of misuse and appropriation that machines originally designed for calculations can become music boxes, that spreadsheets can be filled with poetry, that military technology can be used in art installations, and that artefacts of the computing world—the hardware, the cables, the screens, the noise—can become raw material for the creation of art. In the case of my own musical practice, the algorithm itself becomes a malleable material, one that is sculpted through the feedback loop of heuristic experimentation previously described in Section 2.1. As Burnham (2015, p. 115) states, 'In evaluating systems the artist is a perspectivist considering goals, boundaries, structure, input, output and related activity inside and outside the system.' He argues that while objects have fixed shapes, a system—and, therefore, an algorithm—'may be altered in time and space, its behaviour determined both by external conditions and its mechanisms of control' (Burnham, 2015, p. 115). This concept of a malleable system that can be altered externally (such as by an artist) reinforces the notion that algorithmic works result from, and can only with, a two-way interaction between artist and technology.

Such preoccupations were already present in the artworks and writings of the early pioneers of algorithmic art. Reichardt, for instance, highlights the non-utilitarian use of computers to produce art in the early 1970s (1971b, p. 8), while, during the same period, Brün writes about the possible roles of the composer in a technological era (2004, pp. 163–176). He also argues that composition systems can only become meaningful when they are based on human decisions during their creation (Brün, 1969, p. 119), emphasising the importance of understanding these systems as extensions of the composer and not as isolated tools. Around the same time, Bense wrote extensively on what he called 'generative aesthetics', which can be understood as the aesthetics resulting from the application of generative methods to computer art together with the epistemological questions that it raises (1971, pp. 57–58). Bense also had some more contestable ideas about computer aesthetics, particularly relating to the role played by randomness in computer art. Reichardt writes that 'Professor Max Bense [. . .] has pointed out that randomness involved in computer graphics replaces that aspect in art which is described as intuitive. Thus the randomizing procedures in computer technology are analogous to an artist's intuition', an idea which she finds questionable (Reichardt, 1971b, p. 89). On the other hand, she does point out that such efforts show that 'attempts are being made to find equivalence between human activities in the sphere of creativity and the realization of those activities with the aid of a cybernetic device' (Reichardt, 1971b, p. 89).

It becomes evident from these texts that a major preoccupation of early practitioners of the field was to relate technical aspects of algorithmic music to non-technological approaches to composition. On this point, Burnham warns of the dangers of 'craft-fetishism', which, according to him, are the basis of modern formalism (2015, p. 114). He argues that relevant artists should seek means of production for their artwork that are closer to their own society, engaging with it in the process. For him, 'the artist becomes a symptom of the schism between art and technics. Progressively the need to make ultra sensitive judgements as to the uses of technology and scientific information becomes "art" in the most literal sense' (Burnham, 2015, p. 114). The development of algorithmic composition can also be approached using Sontag's notion of 'new modes of sensibility' (2018). In her essay from 1965, Sontag identifies a transformation in the role of art that emerged as a reaction to the technological developments of society. She then describes how the perceived chasm between arts and science—what she calls a conflict between 'two cultures'—is but an illusion, with artists contemporary to her challenging these boundaries and developing an entirely new sensibility towards art-making, one that bridges the gap by transforming the role of art altogether, often by embracing these technological developments (Sontag, 2018, pp. 39–41). This is an important idea that is very relevant to this day, with a similar position articulated by Hamman (2000c, p. 2), who writes that 'technological art penetrates technology *as means*. In this way, its activity understands the technological as aesthetic, and vice-versa.'

## 2.3 Systems of Algorithmic Music

### 2.3.1 General and Composition-Specific Systems

Composers working with algorithmic music systems commonly adopt one of two different design approaches: they can either create general systems that will be used for multiple different compositions or design a system tailored for a specific work (Bown & Martin, 2012, p. 9).

General systems are designed to be flexible: the composer interacts with the system through an interface that can be used for setting parameters and initial conditions, which, in turn, will affect the musical output (Ariza, 2005, p. 18). These systems tend to implement general compositional ideas and are designed to create more than one different piece of music. Some composers publicly release their general systems for others to use, suggesting that they may not view the system as part of any resulting artwork but rather the composer's selected subprocesses and chosen initial parameters. Notable examples of this approach include Koenig's *Projekt 1* and *Projekt 2* systems (commonly referred to as PR1 and PR2), used to compose his *3 ASKO Pieces* (1982) and *60 Blätter* (1992), respectively (Koenig, 1983, p. 27; Ariza, 2005, pp. 46–47; Clarke, Dufeu, & Manning, 2020, pp. 59–60); Xenakis's *Stochastic Music Program* (SMP), used to compose the *ST* series of compositions written in 1962, which include *ST/10-1, 080262*, *ST/48-1, 240162*, *Atrées*, and *Morsima-Amorsima* (Ariza, 2005, p. 46); and Barlow's *AUTOBUSK* program, used to compose *variazioni e un pianoforte meccanico* (1986) and *Orchideae Ordinariae or The Twelfth Root of Truth* (1989), among other pieces (Barlow, 1990, p. 168; Hajdu, 2016, p. 182).

These programs are all examples of 'composition agnostic' systems: they do not contain procedures or musical operations specific to a single work but rather general functions that can be used to create many different works by anyone who uses them. Composers using such systems will then choose, during the compositional process, what algorithmic procedures will be used and also set parameters to values of their liking.[3] Although fundamentally responsible for the musical output, these programs are not *themselves* the artwork, even though they will invariably suggest particular aesthetic and technical preferences (e.g. Koenig implemented serial procedures in his system, Xenakis's system is specifically designed for stochastic processes, Barlow's system uses evaluation functions based on his own musical theories to manipulate and select material). Even though composers often write these programs for their own use, some made their systems freely available to others, emphasising that these are general music systems that any composer can use to create *their own* works. The aforementioned systems, Koenig's PR1, Xenakis's SMP, and Barlow's AUTOBUSK, are all publicly available, with Xenakis having already released the source code of his software back in 1965 (Ariza, 2005, p. 46; Barlow, 2000).

Regarding the operation of these general systems, Xenakis (1992, p. 144) poetically writes:

> With the aid of electronic computers the composer becomes a sort of pilot: he presses the buttons, introduces coordinates, and supervises the controls

[3]Regarding Koenig's PR2, Berg (2009, p. 79) writes that it 'can be considered an open system which required the user to construct a structural formula and specify a great deal of input data (about 60 questions needed to be answered). Input was not only extensive, it was also quite complicated.'

of a cosmic vessel sailing in the space of sound, across sonic constellations and galaxies that he could formerly glimpse only as a distant dream. Now he can explore them at his ease, seated in an armchair.

While Xenakis's systems are complex cosmic vessels to be piloted by anyone with access to them, other composers have approached algorithmic systems in a different way: they create their own personal vessels which, after much testing, are to be entirely run on autopilot, with all buttons and levers hidden away from any prospective passengers. These are composition-specific systems designed by a composer to create a specific work. In this case, it becomes difficult to differentiate the system from the artwork itself: the choices made when designing the algorithm—such as those related to musical processes, input material, system constraints, and initial values—are hardcoded in the system itself. Therefore, these systems are often highly idiosyncratic rather than being focused on realising any other compositional ideas, be they of the designers themselves or of other composers (Bown & Martin, 2012, pp. 9–11; G. Wang, 2017, pp. 72–73). Generative systems used to create indeterminate performances or ever-changing series of works are also included in this category since these are not general tools to be used by others but rather tailor-made machines designed with specific compositions in mind.

Works that make use of such systems include Jem Finer's *Longplayer* (1999), a deterministic thousand-year-long algorithmic composition (Longplayer, 2019); Agostino Di Scipio's *Audible Eco-Systemic Interface* project, a generative system that interacts with its external environment (Di Scipio, 2003; Eldridge & O. Brown, 2018, p. 232); Karlheinz Essl's *Lexicon-Sonate* (1992–2020), a real-time composition for computer-controlled piano (Essl, 2007, pp. 122–124; Essl, 2018); Josiah Wolf Oberholtzer's note-based works *Aurora* (2011), *Plague Water* (2014), and *Invisible Cities* (2014–2015), generated with Abjad (Oberholtzer, 2015); Tom Johnson's *Automatic Music* (1997) and *Tilework* series (2003), which use numeric algorithms and tiling techniques in their musical processes (T. Johnson, 1998; 2011; 2006); Brian Eno's *Music for Airports* (1978), originally created with a system of tape loops of different lengths (Essl, 2007, pp. 121–122); and Kevin C. Baird's *No Clergy* (2004), a real-time score synthesis composition (Baird, 2005).

Despite this separation of two contrasting approaches, it is important to note that composers working with composition-specific systems will likely reuse ideas from their past works. They may reuse compositional ideas as well as code itself. This is the case of my own music: each piece is written as a standalone program whose execution will generate only that very composition. However, I have written two open-source libraries that provide the framework for my algorithmic compositions (see Chapter 4). The first, lilypondLibrary (Agostinho, 2019), allows users to generate LilyPond scores using Fortran syntax. It is a very general library that does not contain implementations of musical processes, serving only as a bridge between Fortran and LilyPond. The other, Auxjad (Agostinho, 2021), serves as a musical toolbox for algorithmic composers and, as such, contains implementations of specific mental models and algorithmic processes frequently used in my compositional research but which are general enough to be of interest for others. Both libraries are publicly available online under permissive free software licenses and can be used and adapted by other composers, a common approach taken by many other algorithmic composers (see Section 4.5).

Musical systems may also be designed with multiple specific compositions in mind, and in which case, the system itself may evolve as each piece is composed. This approach can be found in Bernhard Lang's Computer-Aided Design for Musical Applications system, commonly referred to as CadMus. In addition to various cutting and looping processes, a substantial part of CadMus consists of cellular automata algorithms that can be used to manipulate musical material (Dysers, 2019, pp. 57–62). Originally written in 1997, Lang reworked and expanded his system when composing the first piece of the *Monadologie* series (2007–present) and used it in every composition of this series until abandoning it with *Monadologie XXIX* (Dysers, 2019, p. 57). This blurred boundary between a single evolving system and a series of different pieces is acknowledged by Lang himself, who argues that 'the *Monadologies* are all versions of a single composition' (Lang as quoted in Dysers, 2019, p. 57).

It is important to recognise that algorithmic music can also be designed and implemented without the aid of computers or other technological apparatuses. Pieces such as Steve Reich's *Clapping Music* (1972), Arvo Pärt's *Spiegel im Spiegel* (1978),[4] and John Cage's *Music of Changes* (1951) can all be fully defined with automatic procedures and are thus suitable for algorithmic implementation with computers, even though the composers did not use them (Reich, 2002a; Supper, 2001, p. 48; Shvets & de Paiva Santana, 2014; Pritchett, 1996, pp. 78–88).[5,6] In relation to his process pieces, Reich (2002b, p. 34) writes that 'once

[4]In the case of Arvo Pärt, he has adopted the term 'computer music' to describe his compositions that employ algorithmic methods, despite not using computers in his compositional process (Supper, 2001, p. 48).

[5]Cage's piece would, of course, result in a different score due to the intrinsic randomness of his chance operations.

[6]A notable early example of an almost entirely automatic composition is Pierre Boulez's *Structures Ia* (1951), often considered the first work of total serialism. Despite nearly all

the process is set up and loaded it runs by itself.' Any distinction between composition and algorithm is completely absent in the case of Process Music, with Reich (2002b, p. 34) arguing that he composes 'pieces of music that are, literally, processes.'

### 2.3.2 Designing Music Systems

The designer of an algorithmic music system must always address the questions related to what Roads calls the 'representation issue': how is the musical data represented in the program, how is this data displayed in the interface, and, in the case of general music systems, what controls are made available to the user (1996, pp. 856–857). In other words, the representation issue is, at its heart, a question of musical mapping: how are musical elements mapped into data structures and how can those be manipulated by the system itself and its users. This mapping is of utmost importance for a system designer since it acts as the interface between the musical objects and computer code.

Specific mappings can also suggest specific ways of working. For types of music that are naturally parametrised (particularly those that use distinct and decoupled parameters), this mapping can be very straightforward. An example of this can be seen in the work of serial composers, who employed simple numeric mappings and manipulations. This was done by first mapping a source of basic material (usually a pitch-class set) into a series of numbers. These numbers could then be manipulated, with the result mapped back into musical parameters such as pitch, duration, dynamics, and articulations (Boulez & Cage, 1995, p. 101; Wuorinen, 1994, p. 130). This type of mapping can be very easily implemented with a computer, as composers have done since the 1950s (Ariza, 2005, p. 65; L. A. Hiller, 1981, p. 12).

However, this does not mean that more complex musical objects—and, therefore, more complex mappings—cannot be accomplished using computers. As Ariza (2005, p. 139) writes:

> Many early CAAC [computer-aided algorithmic composition] systems, and numerous modern systems, treat musical parameter values exclusively as data symbols. This data is either numeric (such as time or pitch values) or symbolic (such as dynamic symbols or performance articulations). With object-oriented programming, it is possible to model musical materials and procedures as specialized objects. Although not necessarily offering

of its parameters being processed by deterministic serial procedures, the piece is not fully automatic because one parameter, namely register, was arbitrarily selected by the composer on a note-by-note basis, as opposed to being implemented using a serial procedure (Brindle, 1987, pp. 25–33).

computational advantages, such techniques allow for more intuitive controls, transformations, and interactions of musical material.

As opposed to the simpler mapping of parameters into numbers or symbols, object-oriented programming allows the composer to map complex musical entities and their behaviours into a single structure known as a ‘class’. This allows for higher-level manipulations, as these entities resemble the musical structures we traditionally use in music, as opposed to working with arbitrary numerical mappings. A class serves as a blueprint for creating multiple instances of these entities. These instances contains attributes (i.e. their data) as well as methods (i.e. the functions used to manipulate this data). All data manipulations are realised by invoking the methods available in an instance, as opposed to using functions defined outside the class, a concept known as encapsulation (Weisfeld, 2004, pp. 10–21; Pearson, 2011, pp. 112–125). Classes allow programmers to construct increasingly complex structures: they can ‘inherit’ attributes and methods from a so-called parent class as well as be made of instances of other classes, a technique known as ‘composition’ (Weisfeld, 2004, pp. 21–27). Working with object-oriented programming allows for a much more natural mapping of musical concepts into computer code (Treviño, 2013, pp. 12–23). This is because objects provide both representational advantages over simple numeric data processing and allows for complex interactions between them (Ariza, 2005, p. 139). These ideas will be discussed in more detail in Chapter 4, which will also show how these techniques can be used to model high-level musical ideas (Bača et al., 2015; Oberholtzer, 2015, pp. 11–17).

Composers can use similar objects to the ones described above even outside the realm of computer-aided composition, such as by approaching composition using the extended definition of musical objects proposed by Harper. For him, musical objects can be both ‘nouns and verbs’, so that a single parameter, a whole musical note, or even a whole piece of music can all be considered as objects, as well as any processes such as reverb or tempo changes (Harper, 2011, pp. 83–87). This concept is as powerful as it is simple: it provides a direct framework for approaching the problem of musical mapping into code, and it translates especially well into object-oriented code.

Interestingly, Harper also includes relative objects in his definition of musical objects. These include simple subjective ideas such as ‘harsh’ or ‘loud’ as well as more complex and flexible instructions such as ‘play music for twenty seconds’ (Harper, 2011, p. 88). These can still be very useful when designing algorithmic processes or selecting constraints for a system; for instance, a system might use a simple lookup table to store arbitrary harshness values for a list of instruments and their several playing techniques, a list subjectively created by the composer. This table can then be used by the system when making algorithmic decisions during the system's runtime.

For Harper, a musical object can be anything that holds or transforms musical information: individual note parameters, small-scale structures such as notes and musical cells, large-scale structures such as phrases, sections, and form, as well as any musical process and function. These can be applied not only to other objects but also to themselves. Effectively, musical objects are the building blocks of *any* musical abstraction. This is of crucial importance since, as Roberts & Wakefield (2018, p. 303) write, 'Abstractions are not merely structural convenience: through their constraints and affordances, abstractions effectively present a *model* of a world with which a live coder maintains discourse.' Their remark is valid not only for live coders but also for all algorithmic composers.

By engaging with systems through objects, artists also engage with a particular framework within which they operate. Malleable and interactive objects are more accessible to be abstracted by our minds than vectors and matrices of purely numeric or symbolic data. This very process of abstracting concepts can lead to particular ways of thinking, suggesting specific procedures and materials in the process. Therefore, the formalisation of the system and the act of composition become more closely aligned than in more data-driven approaches to algorithmic music. Even the simple act of naming objects, attributes, and methods affect the system's affordances, which will affect how users perceive and approach them. As Magnusson & McLean (2018, p. 262) argue:

> High-level pattern languages are useful as they are minilanguages or high-level systems that provide bespoke and often idiosyncratic ways of thinking and performing music. In the design of pattern systems, the naming of the functions suggests affordances: they are linguistic abstractions of processes that may or may not be easy to write in a standard language. [...] The system's method names thus become semantic entities in the compositional thinking of the composer or performer. They outline the scope of the possible.

These affordances are crucial in establishing the system's relevance and significance to the composer as they signal the system's structure and information potential (Brün, 2004, p. 207).

### 2.3.3 Generative Systems

Composers have explored rigorous systems for generating music much before the first experiments in algorithmic composition from the 1950s. One of the first known examples dates back to the 11$^{\text{th}}$ century when, in his *Micrologus de* *disciplina artis musicae* (ca. 1030), Guido d'Arezzo described a rigorous method for converting any text into melodic lines. This is done by using a fixed mapping between the vowels of the text's syllables and musical pitches (Palisca & Pesce, 2001). Despite the deterministic nature of this procedure, the resulting music will vary according to the input text and is carried out automatically. Johannes Ockeghem's 15th-century *Missa cuiusvis toni* is another early example of a work in which the composer is partially ceding control, creating a predecessor to Eco's notion of an open work (1989). This mass is notated without clefs, allowing performers to interpret the same music in any of four distinct musical modes: Phrygian, Mixolydian, Lydian, or Dorian (Mengozzi, 2008, p. 56). Some other early examples also show a more direct use of indeterminacy. Wolfgang Amadeus Mozart's apocryphal *Musikalisches Würfelspiel* (1787) requires dice throws to select and combine measures of pre-composed music, while John Clinton's *Quadrille Melodist* (1865) consists of a deck of cards containing musical fragments that are freely combined by the pianist (Edwards, 2011, pp. 60–61).

However, musical indeterminacy would only gain a prominent role in composition with the work of John Cage, who started experimenting with chance operations in the 1950s (Pritchett, 1996, pp. 70–71). Cage's ideas were born from his studies of Zen Buddhism: for him, chance operations provided a method for removing himself from his music, avoiding his own 'individual taste and memory' (Griffiths, 2011, pp. 26–30). Cage's approach was highly influential, reaching not only his close group of American composers but also Europe's leading serial composers, such as Karlheinz Stockhausen and Pierre Boulez, both of whom experimented with notions of indeterminacy in their works (Griffiths, 2011, pp. 30–33, 107–13).

Although computers are not the only available tools capable of generating random data—Cage himself initially relied on coin tosses (Griffiths, 2011, p. 26)—their main strength lies on their speed: computers can 'flip' many simultaneous coins at extraordinary speeds, allowing for vastly more complex random procedures than those carried out manually. Most programming languages provide a built-in pseudorandom number generator (often abbreviated with the acronym PRNG). PRNGs are deterministic mathematical functions that output seemingly random numbers; although the resulting sequence of numbers from a PRNG is deterministic, these functions are designed to ensure that certain statistical properties are respected over long sequences (Park & Miller, 1988). In other words, for most purposes, the output of these functions will be indistinguishable from factual randomly generated sequences, particularly for the use cases found in generative art systems (Pearson, 2011, p. 52).[7]

Randomness has been an integral technique used by algorithmic composers since the very first experiments of computer music. In what is considered the first computer-aided composition, *Illiac Suite* (1956) for string quartet, Lejaren Hiller and Leonard Isaacson used random and statistical procedures such as Markov chains and random walks (Edwards, 2011, p. 61). Xenakis further developed these ideas through what he termed 'stochastic music', i.e. music created using strict probabilistic procedures often inspired by scientific disciplines such as physics, statistics, and mathematics (1992, pp. 1–42). Although his early stochastic compositions, such as *Pithoprakta* (1956) and *Achorripsis* (1957), were laboriously implemented without computers, Xenakis gained access to an IBM-7090 mainframe computer in Paris in the early 1960s, which he used to formalise and implement the methods employed in his earlier pieces, leading to his renowned ST computer algorithm, completed in 1962 (Xenakis, 1992, pp. 131–144; Essl, 2007, pp. 116–117). This algorithm was used to create compositions including *ST/48, 1-240162* (1962) for large orchestra, *ST/10, 1-080262* (1962) for ten soloists, *Atrées* (1962) for ten soloists, and *Morsima-Amorsima* (1962) for four soloists (Xenakis, 1992, p. 144). Xenakis's stochastic music marks an important moment in the history of computer music, shifting the focus from technical experimentation towards the creation of ambitious artworks composed for the concert hall. For him, the computer was not merely a tool for generating music artefacts but rather a new avenue for artistic experimentation which transcended a more simplistic rationalist discourse (Hamman, 2004, p. 121). According to Hamman (2004, p. 121), 'automation provided [Xenakis with] a framework for epistemological investigation concerning the very nature of sound and music.'

Any algorithmic system that employs randomness is able to output different results at each execution, allowing composers to engage with Eco's idea of the open work (1989). Even pieces that exist as a single score can be thought of as being 'conceptually open' since the system that created it is also able to create multiple other versions of the same composition. In relation to his piano piece *Çoğluotobüsişletmesi*, realised with the AUTOBUSK software (Barlow, 1990), Clarence Barlow states that 'The actual piece is only one of many possible realizations' (Supper, 2001, pp. 49–50). This is because AUTOBUSK is a

[7]Some specific fields may require more robust random number generators. For instance, regular PRNGs, such as those found in Python's `random` module, are not secure enough for cryptographic use. Python 3.6 introduces the module `secrets` for generating cryptographically strong random numbers.

generative system capable of outputting randomised MIDI data for a fixed set of parameters. Therefore, in this case, Brian Eno's 'ever-changing music' (1996, pp. 330–332) is replaced by a system with 'never-ending potential', i.e. a system capable of producing a nearly infinite pool of musical realisations from which the note-based composer must select. A single piece such as *Çoğluotobüsişletmesi* exist as a single realisation and is, therefore, a closed work, but the system that was used to generate it can be considered itself as a form of open work since each new execution will produce different sounding that nevertheless obey the same set of rules (Eco, 1989, pp. 1–4). Keller & Ferneyhough similarly argue that individual compositions in Xenakis's *ST* series 'might be understood as instantiations of a general model' (2004, pp. 161–162). Concerning this approach, Essl (2013, p. 299) remarks,

> by transforming a compositional idea into a more generalized abstract model I could use this 'formula' for obtaining an infinite number of structural variants of the same piece. In so doing, the traditional concept of a deterministic and untouchable work (an opus magnum et perfectum) is transformed into a fluid and open process which can be expressed by an algorithm.

Generative techniques such as these allow composers to work with defined procedures and materials but let the system create their combinations and interactions, resulting in music that is, ultimately, unforeseen. This is exactly the technique used by Eno in his generative systems, which are based on probabilistic rule-sets to generate music that is new at each execution. He writes that part of his enjoyment of this type of music came especially from 'the knowledge that the music I was hearing at any given moment was unique, and would probably never be heard in exactly that way again' (Eno, 1996, pp. 330–331). This method of 'growing' rather than composing music shows a strong parallel with the approach taken by generative visual artists, as Pearson (2011, p. xviii) highlights:

> Generative art isn't something we build, with plans, materials, and tools. It's grown, much like a flower or a tree is grown; but its seeds are logic and electronics rather than soil and water. It's an emergent property of the simplest of processes: logical decisions and mathematics. Generative art is about creating the organic using the mechanical.

Therefore, generative techniques transform the role of the composer into that of a gardener who grows their systems while heuristically exploring its potential, becoming 'an audience to the results' in the process (Eno, 1996, p. 5). This idea of a composer relinquishing control and becoming an audience to their own work requires a different mentality than that commonly found in more traditional musical environments. This mentality is very candidly described by Dahlstedt

(2001, p. 122) while writing about his relation to his own generative music during the compositional process:

> When confronted with a large body of material, such as a MIDI file or a sound file coming from a program of mine, I get very mixed feelings. I have a slight feeling I did not write that music, and yet I am quite sure no one else did. I designed the algorithm, implemented it and chose the parameters, and still I feel alienated. Mentally I am just a consumer of the music, because I could not predict the results of my algorithms. [. . .] This feeling of alienation has to be eliminated, and the cure is to listen and listen again, until the material is assimilated by the mind, incorporated into my intuition making my circle of imaginable music expand. This process of assimilation takes a lot of time, but it is absolutely necessary. Without it, I will not be able to assemble the material in a meaningful way, and more importantly, I will not have any moral right to put my name on it. Even if I do not change much in the generated structure before putting a title on it and publishing it, I have made it a part of me. I have become the composer by changing myself to accommodate to the result.

Algorithmic composition allows the composer to modify any minute part of a system and immediately observe how all parts will then interact and how the resulting music will change. This experimental approach further emphasises the exploration aspect of algorithmic composition previously discussed in Section 2.1: as they work, the composer must delve into previously unseen lands, scrutinising and mapping its potentials, but never entirely sure of what lies ahead. When working with generative systems, the composer is invariably faced with an inescapable duality: although they are responsible for the creation of the system and may, at any point, modify any of its parts, they are also unable to predict its full generative potential at any given point—i.e. they cannot foresee all possible results. As such, algorithmic composition becomes a cyclical heuristic activity of creation, exploration, and evaluation.

In the case of my own systems developed as part of this research, which employ stochastic procedures for the generation of input material as well as for some of its transformation procedures, I commonly fix their random selections in place by utilising what is known as a 'random seed'. A random seed ensures that all functions that generate random numbers will output the same sequence of values at every execution of the program; in other words, each possible value of the random seed is mapped into a single series of the randomly generated output. Although each different seed will produce completely different musical results, the composer can fix any given result by setting this random seed to a specific value, making the musical output of the system reproducible. As such, one can consider the whole, 'unfixed' system as a network of musical potentials, while setting different seeds and observing the results as akin to looking for an *objet trouvé*, that is, a found object that will constitute the final work without any alteration (Chilvers & Glaves-Smith, 2009, p. 521).

### 2.3.4 Software for Note-based Algorithmic Composition

There is a vast body of literature concerning software that can be used to generate note-based algorithmic music. The advent of GNU LilyPond has been of particular importance to the field (Nienhuys & Nieuwenhuizen, 2021; 2003). LilyPond is an engraving program that takes plain-text files as input and compiles them into a score. Due to its plain-text-based nature—as opposed to other notation software such as Sibelius, Finale, and Dorico—LilyPond's input can be manipulated algorithmically: that is, any programming language can be used to generate plain-text files containing LilyPond syntax, which can then be compiled into a score and output as a PDF file. This feature led to the proliferation of multiple software packages and APIs that act as intermediaries between users and LilyPond. These include *Abjad* for Python (Bača et al., 2021; 2015), *Fosc*, *Fomus*, and *LilyCollider* for SuperCollider (Armstrong, 2021; Barros, 2018; 2019, respectively), *notes* for Pure Data (Oliver La Rosa, 2018), and my own *lilypondLibrary* for Fortran (Agostinho, 2019). LilyPond itself comes with a built-in Scheme interpreter, allowing users to write Scheme functions directly in their LilyPond input file (Nienhuys & Nieuwenhuizen, 2003).

Abjad has been crucially important to this research project and, together with my own library Auxjad (Agostinho, 2021), forms the basis of my current compositional method (see Chapter 4). Abjad's API makes full use of Python's object-oriented model of programming, and, as such, its objects can then be manipulated in very natural ways by composers (Bača et al., 2015, pp. 165–167), leading to a modular approach to algorithmic composition. Details of the operation of Abjad can be found in Oberholtzer's PhD dissertation (2015), as well as information on his *Consort* library, which includes classes and functions that extend Abjad beyond its original capabilities (2015, pp. 159–217).

Other notable libraries and software capable of generating music notation but which do not use LilyPond for engraving include *Bach* (Agostini & Ghisi, 2015), *MaxScore* (Didkovsky & Hajdu, 2008), and the *DECIBEL Scoreplayer* (Hope & Vickery, 2015), all for Max/MSP, *Gemnotes* for Pure Data (Kelly, 2011), and IRCAM's standalone *OpenMusic* (Assayag et al., 1999, pp. 64–71). OpenMusic tends to be particularly popular among composers who generate and manipulate materials in the context of computer-assisted composition, while the software mentioned above are prominent in the field of real-time notation.

## 2.4 Algorithmic Thinking

A crucial source for this research project—and one that has greatly influenced my recent series of compositions *Cartographies* (see Chapter 5)—is the work of Lakoff & Núñez (2000). The authors build on the previous research by Lakoff & M. Johnson (2003), particularly on the notion that the human brain uses conceptual metaphors as tools for grasping the complex phenomena it experiences (2003, pp. 3–6). In *Where Mathematics Comes From*, Lakoff & Núñez apply these ideas to mathematics and use cognitive science theories to study how our minds engage with and construct mathematical thinking. Their main argument is that every complex mathematical theory is built upon other less complex notions, which are built using several types of metaphors (Lakoff & Núñez, 2000, pp. 1–11). 'Grounding metaphors' serve as the primary type for mental constructs as they yield basic concepts that are immediately graspable. Examples include abstracting sets as containers, its members as physical objects, and operations such as addition and subtraction as adding or removing objects to or from a collection (Lakoff & Núñez, 2000, pp. 52–53). Another type of metaphor, 'linking metaphors', deals with more abstract ideas and yields higher-level concepts. They are used, for instance, when abstracting the real numbers as points on a continuous line. This is by no means a trivial construct but one that enables very natural manipulations of real numbers, despite these consisting of an infinite set whose majority of members are irrational (Lakoff & Núñez, 2000, pp. 52–53). Lakoff & Núñez (2000, p. 39) describe:

> Metaphor, long thought to be just a figure of speech, has recently been shown to be a central process in everyday thought. Metaphor is not a mere embellishment; it is the basic means by which abstract thought is made possible. One of the principal results in cognitive science is that abstract concepts are typically understood, via metaphor, in terms of more concrete concepts.

These ideas have ramifications that go far beyond the application of cognitive sciences to pure mathematics. Every composer engaging with algorithmic music must deal with the 'representation issue' (Roads, 1996, pp. 856–857), that is, the problem of mapping musical ideas into programming structures to be implemented using a computer. This mapping is realised using metaphors such as those described by Lakoff & Núñez (2000). By framing the representation issue and other algorithmic and musical constructs using high-level metaphors, the composer can become aware of specific types of solutions that were not initially conceived. As such, digital artists do not simply code with ones and zeroes—those are not the raw materials of their artworks. Instead, they create, relate, and manipulate high-level entities that map to concepts in their fields, leading to a much more natural way of working. This idea of creating high-level entities that then serve as building blocks is a fundamental principle of programming, particularly when working with the object-oriented paradigm (Weisfeld, 2004, pp. 129–136), which will be further explored in Section 4.5. In his foreword to Roads's *The Computer Music Tutorial* (1996), Chowning (1996, p. xii) writes,

> there is a less tangible effect of programming competence which results from the contact of the composer with the concepts of a programming language. While the function a program is to perform can influence the choice of language in which the program is written, it is also true that a programming language can influence the conception of a program's function. In a more general sense, programming concepts can suggest functions that might not occur to one outside of the context of programming. This is of signal importance in music composition, since the integration of programming concepts into the musical imagination can extend the boundaries of the imagination itself. That is, the language is not simply a tool with which some preconceived task or function can be accomplished; it is an extensive basis of structure with which the imagination can interact, as well.

This is a profound remark. Computers are often reduced to mere tools, simple problem-solving machines that should, at best, go unnoticed or, at worst, cause as little hassle as possible (Hamman, 2000c). Chowning's conception of programming as a different mental paradigm adds another layer of complexity on top of the notion of a programmer simply giving orders to the machine: the characteristics and behaviours of the device itself influence back the programmer's approach. The resulting interaction between them, therefore, becomes a feedback system of mutual influence.[8] This bi-directional relationship between system and artist is also noted by Brün (2004, p. 177), who remarks:

> The link between the computer system and the composer of music is 'The Program.' Composers may think of themselves and their minds and their ideas in any way they please, until they decide to use the computer as an assistant. From that moment on, composers must envisage themselves, their minds, and their ideas as systems, since only systems can be translated into that language, the program, which can generate their analog appearances in the computer.

Such two-way exchange between programmer and computer can only become possible if interface designers do not consciously limit the possible modes of

[8]According to Di Scipio, a concrete example of this can be observed in the electroacoustic works of Xenakis and Brün. He writes that, 'In some of their work, "sound synthesis" and "music composition" become one and the same', leading to what he calls 'emergent sonorities', that is, sound properties that emerge from the interaction of algorithmic processes (Di Scipio, 2002, pp. 23–24).

interactions available in their software or programming language. Such limitations—often implemented in the name of 'user-friendliness'—can end up reducing a programmer into a mere 'user', bounded not by the limits of their technological means but rather by arbitrary decisions imposed by someone else (Hamman, 2000c; 2004). Even though a user-friendly approach may seem innocuous at first, it severely limits the modes of interaction that are available, particularly in the context of artistic creation and innovative thinking. It discourages experimentation and provides a 'caged' interaction between software and user. Hamman (2004, p. 118), a fierce critic of such user-friendly interfaces, argues:

> By simplifying the interface, computer interface designers have reinvigorated the mechanistic legacy that computers threatened to undue when they forced the computer user to communicate by writing computer programs. [...] 'Ease-of-use' has triumphed over representational flexibility, while the GUI (graphical user interface) has replaced programming as the primary means for communication.

Exchanging flexibility for ease-of-use can severely limit the number of interaction modes available between artist and machine. Hamman writes that such an approach emphasises an instrumental view of technology, transforming the computer into just a 'device'. This leads to users thinking in terms of 'expected' results, effectively neutralising their subjective thinking, which is essential for non-goal oriented activities such as creating digital art. In a similar argument to both Lakoff & Núñez and Chowning, Hamman argues that interface metaphors are fundamentally connected with the system's affordances, thus affecting how we conceptualise the system and communicating how we interact with it (Hamman, 1999, p. 92). This dependency of the user on the interface is made very explicit by Ehn (1988, p. 164), who writes:

> [The] functionality of a computer artifact is a relation between the user and the artifact, something that is found in the use activity, not just a property of the artifact. The user interface is both form and function, in the sense that it from the user's point of view conditions not only *how* but also *what* can be done through the artifact. The functionality comprises the remaining possible actions when the user's intentions have been constrained by the user interface.

It is thus fundamental for the artist to not only be aware of these limitations that are imposed on them but also to attempt to break free from any imposed modes of reasoning that may limit their subjective thinking. The main danger of user-centric approaches is that it risks locking its users into specific patterns of activity and cognition, severely affecting the activity of using the computer as well as any planning and thinking related to it (Hamman, 2000b). Such user-friendly environments have their uses, particularly when applied to goal-oriented tasks, but they are seldom appropriate for creative and subjective work (Hamman, 2000b).

Hamman also argues that when composers do not get involved with software development, they end up with tools that are handed over to them and which can 'carry huge ideological and epistemological payloads to which the composer must accede', possibly leading to a commodified approach to music production as well as to the propagation of the system designer's own ideologies (2000a, pp. 91–92). An obvious way for artists to fight back against these imposed ideologies is by directly engaging with programming languages or programming environments, thus becoming the designer of their own systems and bearer of the responsibility for the ideological weight of their designs (Hamman, 2004, pp. 118–119).

An interesting parallel can be drawn between Hamman's arguments above and Lakoff & Núñez's conclusion that mathematics is inherently non-Platonic. Lakoff & Núñez argue that the building blocks for mathematical thought arise from human thinking alone and, thus, are bound by the human intellect (2000, pp. 1–3); in other words, mathematics is a human construct devised by us for our own use. Similarly, by framing the design of computer systems as a form of human activity, Hamman focuses his attention on this activity's social and political aspects. Systems are built upon ad hoc, arbitrary, and possibly idiosyncratic values imposed by their designers. This arbitrariness is emphasised by Brün (1969, p. 119) in the following passage:

> Without the concept of systems, the concepts of relevance and significance are meaningless. They are equally meaningless with regard to so-called 'universal' or 'natural' systems, in which everything is as it is and could not be otherwise because that is the way it is, 'it' being everything. For anything to be of relevance to something, to be of significance to someone, a system has to be created; an artificially limited and conditioned system has to be imagined and then defined.

All systems are but 'artificial' ones, constructed with mental models and metaphors that allow us to grasp their behaviour. As Hamman (1999, p. 102) remarks, 'the computer is itself a tool for the construction of tools—tools with which one might generate epistemological frameworks for imagining and solving problems of compositional significance.' By becoming their own toolmakers, it is the artists themselves that define what problems are of compositional significance to them and in what epistemological frameworks they will operate.

## 2.5 Interdisciplinarity

Algorithmic thinking is a highly interdisciplinary activity, touching disciplines from both sciences and arts. As such, this dissertation will draw from sources from a wide range of fields other than music, including programming, mathematics, cognitive science, visual arts, and architecture, among others. This section will focus on the interdisciplinary literature that has not been discussed in the preceding sections.

Algorithmic visual arts have a similar origin to algorithmic music, with their early practitioners being contemporaries to composers such as Hiller, Xenakis, and Brün. A key figure in this field is Manfred Mohr, who has worked with generative drawings and digital art since the 1960s (Hattrick & Mohr, 2012). He has given an extensive number of interviews in which he discusses computer art in general, often repeating his unwavering position that algorithmic art is nothing more than Conceptual Art 'that gets realised' (Hattrick & Mohr, 2012). His work indeed bears conceptual similarities to that of Sol LeWitt (1967a), who writes that 'The idea becomes a machine that makes the art.' However, it is the ideas themselves that are the main artistic agent for LeWitt's conceptualism, while Mohr's interest lies in their execution. Mohr writes, 'it doesn't matter where you start. The starting point is arbitrary, but what happens then is not arbitrary' (Hattrick & Mohr, 2012). Regarding the abstract nature of his processes—the driving force behind his work—he remarks that 'These "conceptual rules" are not necessarily based on already imaginable forms, but often on abstract and systematic processes. They are parametric rules, which means that at certain points in the process, choices have to be made as to which way a calculation should continue' (Mohr, 2000, p. 441).

Frieder Nake is another pioneer algorithmic visual artist and a contemporary to Mohr. Nake also shares Mohr's stance that algorithmic art is, by definition, always conceptual. However, Nake (2010, p. 57) argues that concepts must always be clear descriptions of operations in digital art as, otherwise, they cannot be translated into programming code for mechanical execution. Therefore, these are static descriptions of dynamic processes that must always be definable and executable, a constraint that non-algorithmic conceptual art is not bound by (Nake, 2010, p. 57). Nake often relates his ideas back to Sol LeWitt, one of the first proponents of Conceptual Art. The fact that one of Nake's texts is entitled 'Paragraphs on Computer Art, Past and Present'—a wordplay on LeWitt's *Paragraphs on Conceptual Art*—must not go unnoticed. Nake (2010, p. 58) writes:

> The algorithm is the concept in its strictest form of description. [...] Some years before Sol LeWitt wrote this in 1967, algorithmic art had already eliminated the skilled craftsman. We see: algorithmic art is the final form of art in times of industrial production. Beyond all craftsmanship and aura, the work is produced automatically.

Sol LeWitt's ideas are indeed very relevant to artists working with algorithmic art, including music. His view of conceptual artworks as descriptions of blind mechanical processes (LeWitt, 1967a) is very much in line with the understanding of algorithmic art as a form of exploration (as previously discussed in Section 2.1). The notion of artists as generators of ideas is applicable to both algorithmic and conceptual arts. His writings on Serial Art are also very relevant to this research project since his approach is highly suitable for algorithmic exploration, as algorithms are excellent tools for exploring and exhausting all possible permutations of a single idea (LeWitt, 1967b; Reichardt, 1971b, p. 25). My own music often deals with serialised explorations of ideas through mechanical means, as can be observed in the multiple pieces that make up my *Cartographies* series; all pieces in this collection share common algorithms and mental models, and differ primarily in their initial conditions and constraints (see Section 5.1).

LeWitt's notions of serialisation and conceptualism can both be related to a genre of photography called 'deadpan'. Artists who work in this tradition aim at creating inexpressive and emotionless images that are detached from the photographer's own personal views and opinions (Cotton, 2009, pp. 81–113), an attitude that bears a strong similarity to Cage's attempts of avoiding individual taste and memory in his music (Griffiths, 2011, pp. 26–30). Deadpan photographers often use serialisation as a tool for the methodical exploration of their subjects (often vernacular in nature), creating encyclopaedic-like catalogues of images in the process—an approach that resemblances Sol LeWitt's notion of mechanical processes being blindly carried out (1967a). A notable parallel can be made between the deadpan attitude and the systematic and low-interventional compositional approach employed in my accompanying portfolio (see Section 3.5).

A contemporary approach to digital visual art can be found in Bohnacker et al., a comprehensive book on the Processing programming language and its capabilities for algorithmic art (2012). The authors not only write about specific technical elements of Processing but also showcase artworks created with it. These include thirty-five case studies that demonstrate many different artistic approaches to algorithmic visual art. Pearson is a similar source and focuses on practical concerns related to creating digital art with Processing (2011). Pearson's book concentrates on the use of randomness in generative art and how it can lead to the phenomenon known as emergence (see Section 3.4). He also showcases several digital artworks created with Processing. Generative strategies are employed by artists in numerous fields of algorithmic art, including that of algorithmic music. This importance of generative approaches is noted by Nake (2010, p. 58), who states that 'The algorithmic artist does nothing that is not generative. [. . .] The artist turned algorithmic is a generative artist by birth.'

As such, randomness plays a vital role in algorithmic art. In the context of music, although one can find earlier isolated examples of artworks that embrace some level of uncertainty (such as the previously mentioned *Musikalisches Würfelspiel*, apocryphally assigned to Wolfgang Amadeus Mozart), it was only in the first half of the 20th century that randomness became a topic for serious artistic exploration.[9] An example of this early use of chance can be found in the work of Jean Arp (also referred to by his German birth name of Hans Arp). Arp, a member of the Dada movement, would employ randomness to create aleatoric visual compositions. One of his techniques consisted of dropping torn pieces of painted paper onto a surface, with the final composition resulting from their random arrangement (Umland, Sudhalter, & Gerson, 2008, pp. 44–49); this approach can be observed in his *Collage With Squares Arranged According to the Laws of Chance* (1916–17). In music, the work of John Cage has been paramount for the popularisation of randomness as compositional technique. Pritchett writes extensively about Cage's work and includes detailed analyses of most of his major works (1996). This book also focuses on his early period, his transition from choice to chance, and his use of charts, coin flips, and the *I Ching*. Cage has had an immense influence in other artistic-related fields, including the philosophy of arts. This influence can be observed in Groos & Froitzheim's catalogue for the exhibition *[un]erwartet. Die Kunst des Zufalls*, which contains a diverse collection of texts about the philosophy of chance and its use in visual arts, literature, cinema, and music (Groos & Froitzheim, 2016). In relation to architecture, Verbeeck investigates how analogue randomness, such as that found in Jackson Pollock's drip technique and John Cage's chance operations, can be translated into digital strategies for architecture (2006). For him, randomness enables an exploratory approach to architectural design, as opposed to working with fixed rules set *a priori* and using the computer simply as a tool for carrying them out. He writes that, 'In the early phase of design[,] precision should not be required, only ideas, notions of, and wants are' (Verbeeck, 2006, p. 7).

[9]It is interesting to note that this renewed artistic interest in chance takes place at the same time as the deterministic views of classical physics are shaken by the discoveries of the late 19th and early 20th centuries, which led to an inherently probabilistic view of nature with quantum mechanics.

Generative and parametric approaches have a long history in relation to architecture. Woodbury writes about the use of parametric modelling strategies in architectural design. His focus lies on how algorithmic systems can be used for this architectural design and how they are created using mathematical functions to shape the final results. His view of parametric architecture as the design of dynamic systems offers a remarkable parallel to the ideas previously discussed in Section 2.1. He views parametric systems as tools that enable the architect to dynamically explore new sets of possibilities 'that are not practically reachable otherwise' (Woodbury, 2010, pp. 36–39). This idea echoes much of the motivation for the use of algorithmic systems expressed by composers such as Xenakis (1992, p. 144), Brün (2004, p. 177), and Barlow (Supper, 2001, pp. 49–50).

Generative systems can also be created without digital computers, as the work of Jean Tinguely illustrates. Tinguely created kinetic sculptures that bear many similarities to algorithmically designed systems. These works consist of electro-mechanical automata whose primary materials are found objects and pieces of junk, leading to artworks that naturally decay with time (Bek, 2004, p. 46). Some of Tinguely's sculptures have a natural musicality, with their mechanisms making rhythmic noises as they operate (Bek, 2004, pp. 44–45). He also made direct use of sound in some of his pieces, such as his *Méta-Harmonie* (1978–1985), a series of four colossal sound sculptures that employ noise-making mechanisms as well as musical instruments. Some of his artworks are literally self-destructing machines, ephemeral visual and sonic systems that eventually come to a halt (Chau, 2014, pp. 399–401). Chau interprets Tinguely's infamous *Homage to New York* (1960), a self-destructing artwork that was deliberately put on fire, offering a critique of the abundance of a consumerist society (2014, p. 401). Although algorithmic music is not ephemeral in the way that Tinguely's decaying work is, the relation between art and capitalism, methods of production, and commodification are often key points explored by algorithmic artists (Hamman, 2004; Shanken, 2002; Wieser, 2018; Vitalis, 2016).

In literature, the French group of writers and mathematicians known as Oulipo[10] would explore how constraints could be applied to the art of writing. They used formal and mathematical constraints to create literary works such as the '$N + 7$' rule, which requires replacing all nouns in a text by the seventh entry that follows them in a dictionary (Baetens, 2012, pp. 117–118), or the rule that the text should be written without a specific letter, such as Georges Perec's

[10]Sometimes stylised *OuLiPo*, this name comes from the contraction of *Ouvroir de littérature potentielle*, which is typically translated as 'Workshop on Potential Literature' (Buchanan, 2010b).

novel *La Disparition* (1969), written without the letter 'e'—the most frequent letter of the French language (Despeaux, 2015, p. 239). Christian Bök uses a similar constraint in his novel *Eunoia* (2001),[11] which employs a constraint that limits each chapter to use only one single vowel in all of its words (Baetens, 2012, p. 122; Despeaux, 2015, p. 239). Raymond Queneau, one of Oulipo's founders, wrote the combinatoric poem *Cent mille milliards de poèmes* (1961), which can be considered as a generative poem. It consists of a short ten-page booklet, each page containing 14 lines of text. The pages are cut into horizontal strips containing one line each, allowing each strip to be flipped separately from all others. This allows the reader to explore combinations of these lines: ten options for the first line, ten options for the second, and so on, until the fourteenth line. Although this is a very concise book from a physical standpoint, there are a total of $10^{14}$ possible combinations of all lines, resulting in 100,000,000,000,000 different poems. Oulipian constraints provided a strong influence for the self-imposed limitations employed in my *Cartography* series, in which each piece must be completely defined within a single A4 page (see Section 5.1).

All of these interdisciplinary sources point to the fact that algorithms, and, more specifically, the computer, can transcend a utilitarian instrumental view of technology: they can be used as more than mere calculators that retrieve concrete solutions to concrete problems. As such, computers do not exist in a cultural vacuum but rather within the realm of society and culture and, therefore, must be understood as one of its devices. The ubiquity of technology has made computers and algorithms virtually invisible (Hertz, 2009, p. 59; Weibel, 2004), leading to social and political dominance through such tools (Hamman, 2004, pp. 116–117). It is precisely for this reason that authors such as Uliasz have promoted the purposeful misuse of technology—a form of technical perversion—which can serve as an ethical approach to the use of algorithmic technology (2017). By using technology as their primary artistic tool, artists are addressing these issues above, critically engaging with capitalist and positivist aspects connected to such technologies. Similar to Lakoff & Núñez's views of mathematics being inherently non-Platonic (2000), algorithms can also be understood as human constructs that cannot be untangled from the context in which they are created—in other words, they carry in themselves their creator's values and beliefs. It is in this vein that Hamman argues that the act of programming a computer generates much more than simple source code: it generates human interaction through collaborations, research papers, and public forums, both online and offline

[11]It must be noted that Christian Bök is not a member of Oulipo, although his work is inspired by and often associated with the French group.

(Hamman, 2004, p. 119).[12] Similarly, Ehn argues that computers and their output should be considered enablers of communication, supporting individual and cooperative activities, and augmenting human capabilities (1988, p. 234). Lewis is yet another author supporting this non-instrumental view of computers, particularly concerning music creation; he writes (2009, p. 457):

> Understanding computer-based music-making as a form of cultural production obliges a consideration of the discourses that mediate our encounters with the computer itself. Increasingly, new imaginings of history, culture, and artistic practice are finding the computer at their centers, and particularly since the mid-1980s, digital technologies have served as a critical site for interdisciplinary exploration, accelerating the blurring of boundaries between art forms.

## 2.6 Conclusion

Algorithms provide a unique framework for artists to operate within. They enable a truly exploratory approach to art-making, one in which the autonomous system does all the 'heavy lifting' (Pearson, 2011, p. 4). Unshackled from the constraints of manual implementation, artists are able to explore multiple open-ended ideas and processes through a combination of experimentation and heuristics (Essl, 2007, p. 108). Through these interactions between human and machine, both agents shape one another: while the composer employs algorithms in order to implement their musical ideas, the algorithms can also suggest specific modes of thinking or routes of action. In other words, working with algorithms expands our way of conceiving ideas, which in turn become inherently algorithmic (Chowning, 1996, p. xii). As Culkin (1967, p. 70) remarks, 'We shape our tools and thereafter they shape us.'

Artists working in this manner will inevitably engage with epistemological questions concerning the role of algorithms in their art. That is, algorithms are not merely tools used to achieve a concrete goal but are instead an intrinsic part of the final artwork which cannot be entirely disregarded (Hamman, 2000c, pp. 7–8). Technology embeds itself into the final work and, as such, must be acknowledged by both artist and audience.

The following chapters will examine how these ideas about algorithmic art

[12] This social aspect of software development is particularly evident in relation to open-source and free software, of which I am an ardent advocate. This type of software often generates interactions on public forums that aim at an open discussion of ideas. Virtually all tools I use to create my compositions are open-source and free software; these include the Linux operating system, the Python, Fortran, and Pure Data programming languages, and software such as LilyPond, Abjad, LaTeX, and my own Auxjad and lilypondLibrary libraries.

relate to the specific methodologies and aesthetic qualities of my musical works. Chapter 3 will introduce the aesthetic concepts related to my music and discuss how they are connected to my algorithmic thinking. Chapters 4 and 5 will then explore how algorithmic thinking, and particularly the notion of mental models, influence my compositional approach and the implementation of my musical processes.

# Chapter 3

# Aesthetic Dimensions

In recent years, I have developed compositional methods that employ musical repetition as their primary operating element. In my pieces, repetition is used not just as a structural tool but also for its potential to affect the listener's perception of the music, creating a sense of disorientation during the listening experience. This disorientation is further intensified by the quiet and slow materials featured prominently in my work, as well as by the non-goal-oriented and emergent approaches to form and musical processes also employed.

This chapter will discuss the set of aesthetic concepts that underpins my recent work as a composer. It will focus on the notions of slippage, fragility, emergence, and liminality. It will trace the aesthetic origins of these concepts, discuss their relation to my algorithmic processes, and demonstrate how they are interrelated to one another in my compositional practice.

## 3.1 Compositional Context

For the past decade, I have primarily composed using algorithmic methods. My interest in algorithmic music started in 2011–2012, specifically when I began working on my composition *On the Origin of Pitches* (2012) for solo vibraphone (see Figure 3.1 for an excerpt of one possible version of this work). Heavily influenced by the ideas of John Cage and Brian Eno, I aimed to create an instrumental open-work whose content would completely change at each and every performance. The idea of music that rewrites itself at every hearing (Eno, 1996, pp. 330–332) was immensely attractive to me, especially since it also seemed to support the Cagean notion of silencing one's ego through the use of indeterminate procedures (Pritchett, 1996, pp. 60–62), a notion I was fascinated with at the time. Since *On the Origin of Pitches* is an algorithmic open-work, it does not exist as a single score but can instead be understood as the *solution space* of a given system: that is, it encompasses all possible musical realisations satisfying the composition's system of algorithmic constraints. To put it simply, every execution of those computer programs would generate a completely new score with music that had never been heard before. This is an idea I continued to explore and develop up until late 2017.

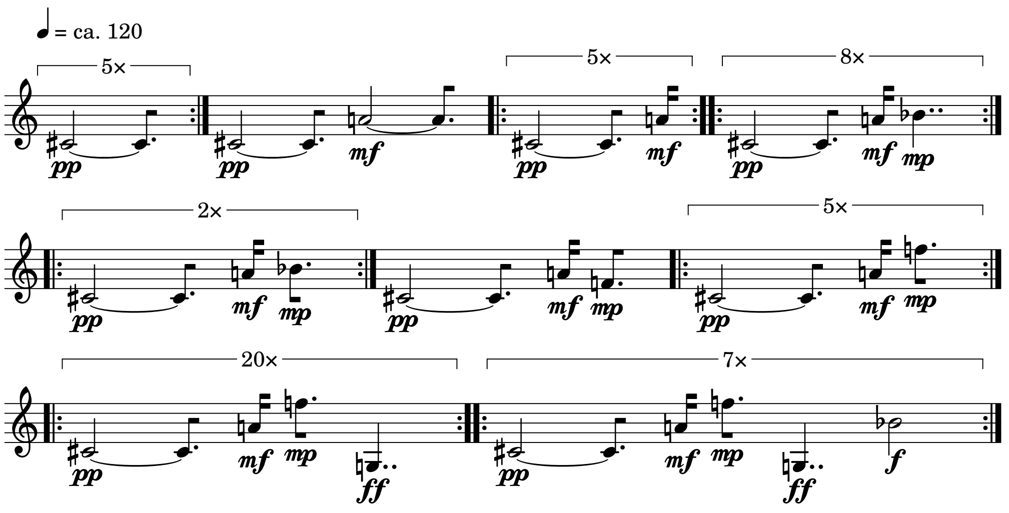

**Figure 3.1:** First systems of one possible realisation of *On the Origin of Pitches*

At that time, I began to support the notion that one cannot entirely separate technology from aesthetics in the context of algorithmic art. I promptly realised that algorithms were not mere tools for quick experimentation, as they also suggested aesthetic routes to be taken. I thoroughly embraced the generative and repetitive aesthetics of the pieces created with those methods and started exploring how algorithms help me compose new music that could not have been thought of by non-algorithmic means.

During this research project, my music has gone through a significant shift. Most notably, I have stopped composing algorithmic open works in favour of selecting a single fixed version for each composition. This decision was primarily driven by practical needs: first, the open-work approach required an unreasonable amount of effort from performers, who were required to discard a previously learned version of a piece and learn an entirely new score for each new performance; secondly, it limited the instrumental demands and notational approaches that could be explored in those compositions, as I needed to ensure that the scores were always fully playable and readable in all possible versions that could be generated by those systems. Despite not working in this manner any longer, generative aesthetics and automatic systems remain essential to my recent work.

With my current compositional methods, the final composition is but one selected version among infinite possibilities that a system could have generated. The difference is simply that, in this case, a composition is crystallised in the form of a single final score, with all other potential possibilities being left as unrealised contingencies of the solution space. This new approach solved most of the practical issues described above while, at the same time, allowing me to continue working with generative techniques and automatic systems.

Besides working with fixed versions of my scores, my recent compositions also sprung from different preoccupations than my earlier algorithmic works, in particular from questions of musical perception and algorithmic formalisation. My current research focuses on the relation between repetition-based algorithmic methods and the creation of disorienting and perceptually ambiguous music. I am particularly interested in the unstable perceptual phenomena that can arise from the interactions of strict algorithmic methods with fragile materials (which, in the context of my music, can be understood as those that are soft and slow). In my artistic practice, I explore how my algorithmic systems can maximise these perceptual instabilities and ambiguities as well as what types of aesthetic and emergent characteristics can arise from them.

In the works of the accompanying portfolio, I am concerned with two related notions that I refer to as ‘slippage’ and ‘liminality’. I use the term slippage to refer to the local disorientation caused by repetition-based algorithmic procedures, particularly when applied to fragile materials. Slippage is a local phenomenon: it arises when looped material undergoes slight changes at each iteration that are either entirely imperceptible or too difficult to pinpoint precisely. With subtle changes occurring at each loop, the listener is often left unable to grasp what has changed from the previous iteration of the process—if anything. This is further emphasised by the fragile materials employed in most of my works from these past four years, which further help to disguise the subtle changes present in this music and intensify the disorientation caused by the musical processes (Harrison, 2007, p. 33). The notion of liminality is used to describe the formal disorientation perceived in these pieces on a larger scale. Liminality is concerned with being in a transitory state, a condition of having left a place behind but without reaching the destination yet. It relates to notions of in-betweenness, unfolding, and becoming, which are evoked by these compositions as a whole. These recent works also engage with the concept of ‘emergence’, which describes the unaccounted for but observable complexity that arises from simple rules that constitute the algorithmic systems. In my music, emergence takes place in the form of musical structures that are perceived as aurally meaningful (i.e.

linked together by Gestalt principles of grouping) but which, in fact, arise from collisions of otherwise unrelated material. These often appear at the borders of consecutive looping windows and, as such, are unstable structures that can change unexpectedly with each iteration of the process. This phenomenon, in turn, further helps to mask the exact looping points of the algorithmic process, contributing to the local sense of slippage. The intention behind the pieces written as part of this research project was to compose music that is perceptually disorienting and ambiguous, music that creates non-linear listening experiences for the listener despite being generated using strict linear processes.

## 3.2 Slippage

The notion of 'slippage' describes a key perceptual aspect of my recent music: the disorientation caused by near-repetition procedures when applied to fragile materials. This section will investigate how this concept can be linked to the specific algorithmic processes and mental models employed in my work and how certain types of musical materials can emphasise it.

Slippage concerns the tension between a slow but constantly unfolding musical process and our inability to grasp its movement. This unfolding can become imperceptible to the listener in specific sonic contexts, presenting itself as identical repetition—somewhat akin to a *déjà vu* experience. Identical repetition can slowly lead to the illusion that sonic transformations are taking place in the looped material, as demonstrated by the research of Margulis (2014) and Deutsch, Henthorn, & Lapidis (2011). Although this psychoacoustic effect can significantly contribute to a sense of sonic disorientation by itself, my work tends to be primarily focused on near-repetitions, e.g. by using moving looping windows and other similar repetition-based processes. Even though these processes are continually unfolding in my music, the emerging sonic results can sometimes be locally indistinguishable from previous windows, creating the illusion of identical looping. Slippage is thus a local phenomenon, taking place on a moment to moment basis and engaging with our short-term memory; this will later be contrasted in Section 3.5 with the notion of 'liminality', a term used to describe the long-term disorientation and overall unfolding of a piece.

In my music, I am very interested in the friction created between the logical algorithmic processes that I employ and the perceptual disorientation and ambiguity that they create. The degree of this disorientation can vary from moment to moment within the same piece. In some moments, the process can become evident to the listener, allowing them to 'anchor' their perception on its mechanisms, leading to a low degree of slippage. In other situations, the exact nature of the algorithmic process can become difficult to pinpoint precisely, leading to a high degree of slippage. This often happens in situations that challenge the listener's memory, such as when consecutive windows are highly similar or when the processed materials are particularly fragile. The emergence of structures at the border of consecutive windows further contributes to the overall sense of slippage, as they further disguise the logical process taking place in the composition.

The musical processes found in my music are purposefully repetitive and slow. In some of my compositions, an overarching process is responsible for the form of the whole work. In other words, these pieces have a clear directionality in terms of form, going uninterruptedly from a specified initial state to a specified final one over the course of the whole work. This main process is the defining element of these types of pieces and can be clearly perceived by the listener, aligning these compositions with the long tradition of Process Music (Reich, 2002b, p. 34; T. Johnson, n.d.). However, the main difference between my approach and compositions usually associated with process music, such as Steve Reich's *Piano Phase* (1967) and Alvin Lucier's *I Am Sitting in a Room* (1969), is that I manipulate material with multiple parallel processes while the master process is being executed, as opposed to delegating all transformations to a single formative process.

A concrete example of this approach in my music can be observed in *Cartography #1*. Its form is dictated by a crossfade process applied to the density of notes of the piano and the vibraphone. Alongside the unfolding of this overarching process, the musical content of every single vertical moment is nevertheless decided through stochastic processes that select material from container-like objects common to all *Cartographies* (the container mental model will be discussed in detail in Section 5.1.2). However, most of my recent works—particularly those written after *Cartography #8*—are far less directional than this: rather than going from a state $A$ to a state $B$, these pieces start at a pre-defined initial state and then *evolve* through small transformations. There is no particular end state to be reached, even if the music must finish at some point; instead, it is the path itself that is the focus, with its ever-present kaleidoscopic transformations. These non-goal-oriented works can be well described by what Kramer (1981, p. 542) calls 'nondirected linearity':

> This music [. . .] is in constant motion created by a sense of continuity and progression, but the goals of the motion are not unequivocally predictable. I call this new species of musical time 'nondirected linearity'—a temporal

> mode unthinkable, even self-contradictory, in earlier Western music but quite appropriate given the breakdown of goal orientation in much of the music of [the 20$^{\text{th}}$] century.

Repetition is crucial to the types of processes I employ in my recent music and is the main element used to achieve this sense of non-linear yet constantly unfolding motion. I aim to achieve what Harrison (2007) defines as 'a sense of implied motion [. . .] but not of musical progress.' By constantly revisiting previously heard materials, looping windows and other near-repetition processes became essential techniques that I use to create this condition of unfolding without progress. Lang (2003) also views the use of loops and circular motion as a way of 'escaping linearity'; for instance, in his *Monadologie* series (2007–present), Lang employs subtle variations in his loops so that they 'seem to be drifting, as they aimlessly wander about without any particular goal or destination in mind' (Dysers, 2019, p. 104).

The repetition processes and input materials that I employ are purposefully designed to amplify this sense of drift and aimlessness. My use of slow-moving looping windows and other repetition-based mental models is crucial for achieving this, as they often give the impression that the composition is not moving at all on a local scale. With the right set of materials—particularly those that are quiet and slow—and suitable window and step sizes, the overall impression is of nearly identical or even literal repetition. The working principle of looping windows can be observed in Figure 3.2 (see Subsections 5.1.4 and 5.1.9 for their technical applications). This figure is a visual representation of the first five iterations of the looping window process, with the result shown at the bottom. At each iteration, the looping window is moved forwards by a specific step size and its content is appended to the final score. Although each consecutive output of the looping window looks very similar to the previous one—sometimes to the point of being indistinguishable from the surrounding ones—it is clear that the process will gradually move towards new territories. As this process advances, familiar material is gradually left behind and new materials are uncovered

The materials processed by these looping algorithms, here referred to as 'input music', are first generated by my computer programs. This notion is analogous to Roland Kayn's concept of the 'super-signal': a master source whose sole purpose is to serve as input material for an algorithmic process (Pickles, 2016, p. 75). I consider such input music as residing in the domain of 'pre-composition'. However, in the context of algorithmic composition, the distinction between what effectively constitutes composition per se, as opposed to pre-composition, is often unclear, with definitions varying substantially from composer to composer

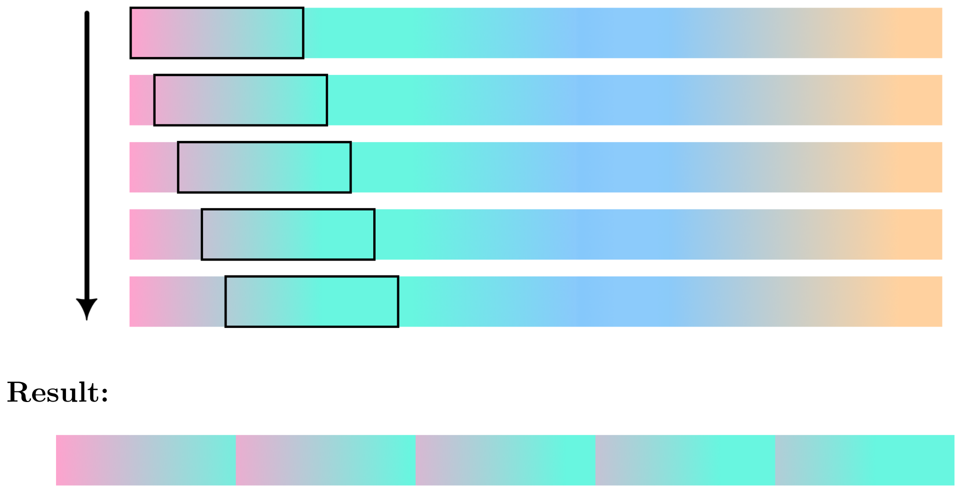

**Figure 3.2:** Visual representation of the looping window mental model

(Doornbusch, 2005, pp. 47–48). The input music used in my compositional approach can also be related to Xenakis's concepts of 'outside-time' and 'inside-time' structures. For Xenakis, a composer works with outside-time structures when generating raw non-temporal materials, which will later become musical events in the composition when articulated by time (1992, pp. 181–183). An example of this distinction, given by Xenakis (1992), is that of a pitch scale (an outside-time structure) becoming musical events in the form of a melody (an inside-time structure). In the case of my input music, it lies somewhere in between these two notions: on the one hand, it already contains temporal relations between its elements, as it is often generated as a fully-developed stochastic 'piece' which will serve the sole purpose of becoming the input material for the looping process; on the other hand, this music is never heard in its unadulterated form by the listener, and, as such, it does not take place in the same temporal reality as the final composition. In other words, the temporal relationships in the input music will be distorted by the looping process and will not be accessible to the listener in its unadulterated form. Nevertheless, such input music is created with specific materials that will emphasise certain characteristics of the processes later applied to them in the final output (see Section 3.3).

Moving looping windows are prevalent in Bernhard Lang's music and the films of Martin Arnold and Rafael Montañez Ortiz. These artists have used repetition to explore and examine materials borrowed from other sources. In his *Monadologie* series of compositions, Lang works primarily with borrowed musical materials[1] and offers them new readings by using looping and slicing procedures carried out using his software CadMus (Dysers, 2019, pp. 57–62; Lang, 2002). Similarly, both Arnold and Ortiz appropriate early 20th-century Hollywood films. By using looping techniques to manipulate the original narratives, Arnold and Ortiz bring out new meanings to otherwise mundane scenes (Dysers, 2019, pp. 144–148). While my use of loops shares many similarities with the techniques utilised by these artists—particularly the moving looping window technique used by Lang (2002)—my music does not feature appropriation or musical borrowing since the input music is always generated from within my own systems, prior to the looping processes. Moreover, the sonic results of my use of loops are substantially different to those found in the loop-based works of composers such as Bernhard Lang, Eric Wubbels, and Alex Mincek; this is primarily due to my choice of material, which tends towards the quiet and slow, and the time frames that my looping windows tend to operate on, which generally last for several seconds. These aesthetic decisions align my work closer to that of Bryn Harrison, who also tends to work with rather fragile materials that are repeated to generate disorienting musical textures. This approach leads to a substantially different sonic world than the jittery and sharp loops prominent in the work of Lang, Wubbels and Mincek.

Another repetition-based mental model that I recently began to employ in my compositions, beginning with *adrift* (2020), is what I refer to as 'fader'. A fader consists of a straightforward algorithmic process where a repeated musical cell has a single note added to or removed from it at each iteration. In the case of *adrift*, I combine two musical layers, each with its own fader. While one fades material in, the other fades material out, thus creating a stepwise metamorphosis between materials—i.e. a non-continuous crossfade (these mental models will be discussed in detail in Section 5.3). The fader mental model was partially inspired by a compositional technique described by Feldman (2000, pp. 193–194):

> One of the problems with variation in twentieth-century music is that they make the variation too obvious. You heard that it was a variation. I am interested now in a lot of music where the variation is so discreet, I would have the same thing come back again, but I would just add one note. Or I have it come back and I take out two notes. And I would vary the notes and keep the pulse, but very subtle.

However, an essential difference can be observed between my use of these

[1]Out of the 38 compositions that make up the *Monadologie* series until 2019, only four use original material written by the composer as input for his looping and slicing processes (Dysers, 2019, pp. 205–206).

mental models and Feldman's compositional approach: despite the shared interest in subtle variations at a local level, my compositional focus lies on the slow transformations created by algorithmic processes, while much of Feldman's music displays no such large-scale directionality (Feldman, 2000, pp. 134–139). My music often exhibits a sense of *overall* forward motion, even if, on a local level, the perception of such processes can become fuzzy and prone to slippages. This is especially true in the case of the crossfader, in which a specific musical cell morphs into another. Although the path taken by the process is stochastic and the slow and subtle changes can be imperceptible to the listener, the algorithm is ultimately moving from point $A$ to point $B$. Likewise, looping windows can also be considered directional since they linearly move from the beginning of the input music towards the end. Thus, it is the oscillation in how graspable these processes are, particularly on the local level, that gives rise to slippage.

Similar to Lang in relation to his *Monadologie* series, my interest lies in microvariations of musical material rather than in literal repetition (Dysers, 2019, p. 104). On a local level, my compositions are concerned with the recontextualisation of previously heard materials and our capacity to grasp their microvariations. In the case of looping windows, they create disorientation through their slow forward motion (which can result in nearly-identical repetitions) as well as by unfolding new musical structures—and, as such, creating friction at the borders of consecutive windows. At these borders, new relationships between materials can often arise, as the process causes a disjunction in the input music due to the backwards leap of the looping window process. These new relationships—emergent properties of both the process and the specific set of materials used—are often ephemeral: a note at the beginning of a given window might not be present in the next iteration of the process, or, alternatively, a new note might materialise at the end of a window. As such, these relationships create a type of emergent structural ambiguity: despite the strict process that can be readily grasped when looking at the musical score, the sonic outcome of the resulting music is substantially more ambiguous. This ambiguity results from emergent relationships often being perceived as structurally meaningful despite being the result of coincidental collisions of unrelated entities. This phenomenon is particularly powerful when such structures stand out from the surrounding texture, be it due to register, pitch content and interval, dynamic level, articulation, or melodic shape (Bregman, 1990, pp. 455–528; Tenney & Polansky, 1980).[2] Similar or identical structures can often appear in borders of multiple consecutive looping

[2]The notion of emergent structures and their relation to Gestalt grouping principles in my music is discussed in detail in Section 3.4.

windows, giving them the impression of being intrinsically interconnected to one another. In other words, loops give structural legitimation to sound objects that would otherwise simply slip by (Harrison, 2012, pp. 59–60).

The obfuscation of the exact looping points further amplifies these emergent relationships. The fragility of my materials—namely, their quietness and slowness—significantly contribute to these blurred boundaries. Although identical repetitions are not employed in my *Cartographies* series (its looping windows always move forwards at each iteration), the music often reaches moments of stasis during which near-repetitions can sound as literal loops. This takes place when a looping window moves forwards but finds no new notes in the input music nor leaves any note behind in relation to its last iteration. An example of this can be observed in the opening measures of *Cartography #11* (2018), shown in Figure 3.3. The score's notation shows that the musical content moves leftwards by one semiquaver (in this composition, the looping window has the length of a whole measure). However, the listener perceives the first five measures as identical loops lasting for fifteen semiquavers (one fewer than the actual size of the looping window) as no new notes enter the window, nor do any old notes leave it. The following three measures (measures 6–8) display the same property, as do the following two (measures 9–10).[3] This apparent stasis is a type of emergent behaviour that these systems display: although the algorithmic process constantly moves forward by a fixed step at one iteration per measure, the slow forward movement often seems to come to a complete halt before resuming motion. As such, the musical flow is perceived as being far more complex and less mechanical than the process behind it might initially suggest.

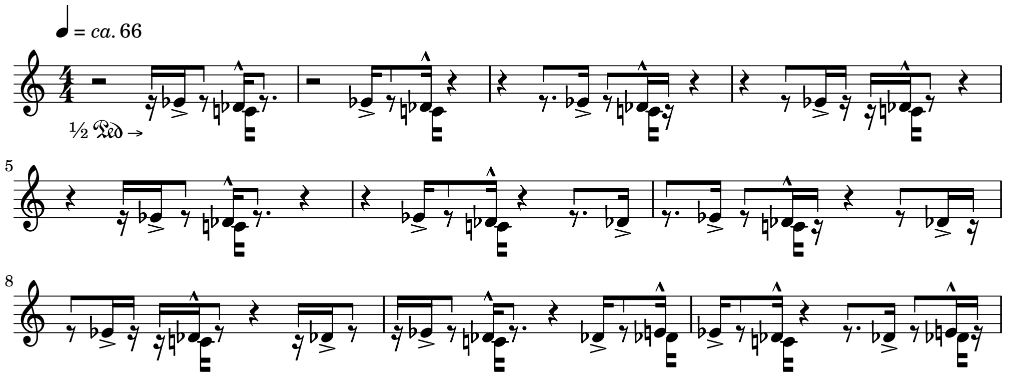

**Figure 3.3:** First systems of *Cartography #11*

These periods of stasis also serve another purpose: they create moments

[3]Figure 5.23 in Subsection 5.1.10 illustrates how these initial measures could be renotated to show the way they are effectively perceived.

of uniformity, tying the material together. While different notes in the input music are only related by sharing the same stochastic procedure that gave rise to them, looping techniques emphasise their inner relationships. That is, when the process gives the appearance of total stasis, these relationships are amplified to the highest degree of importance. As Sabbe (1996, p. 12) writes:

> Immediate repetitions (such as occur ever more in Feldman's late works) are often a means of establishing uniformity ('those huge, uniformly colored, hued sound surfaces of varying extension') over longer periods and thus creating the illusion of a permanently accessible, available, continually present, identical ideal object—an illusion, because even when remaining completely unchanged in notation, the sound object changes over time in perception.

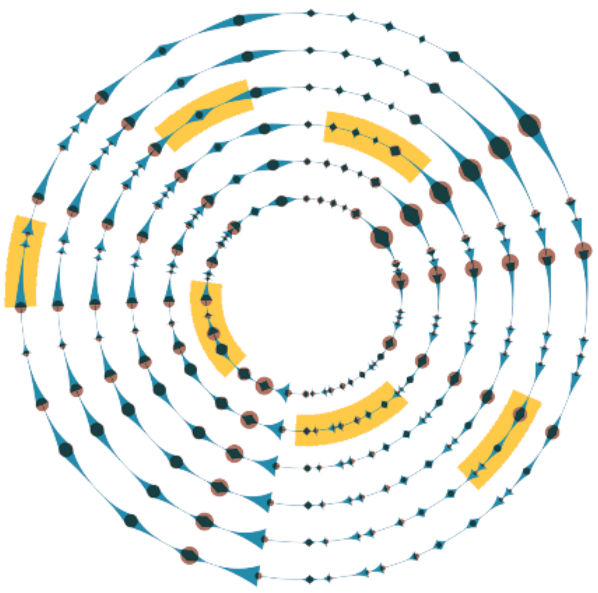

**Figure 3.4:** *Longplayer*'s looping windows at 10:49pm on the 7 July 2021; this image has been reproduced with the kind permission of Jem Finer and the Longplayer Trust

The emergent behaviours that arise from these looping windows are similar to those found in Jem Finer's *Longplayer* (2000), a sound installation designed to last for a thousand years. Finer first created six layers of sounds upon which he applied slow-moving looping windows. Using layers with different total lengths and looping windows of different sizes, Finer was able to calculate that the looping windows will be back at their initial position precisely one thousand years after the piece started. However, from the listener's perspective, this music often sounds static as nothing enters nor leaves any of the six asynchronous looping windows. Figure 3.4 shows a visual representation of these layers (Longplayer, 2019); each of the six layers is displayed as a concentric circle, while the looping windows, highlighted in yellow in each layer, slowly move clockwise. All six looping windows started at the 12 o'clock position when the sound installation began on 1 January 2000, and will align back at that position on 31 December 2999.

This approach thus gives rise to a duality in our perception of time: while the processes themselves must, by definition, have some degree of internal movement, they are also able to create nearly static situations, particularly when working with repetition, long durations, and materials that emphasise this staticity. Even a purely directional process, such as the looping windows of *Longplayer*, can give rise to music which, on the listener's scale, lacks *perceivable* forward movement. This is keenly observed by Frey (2004), who writes:

> It may easily be that, at the end of a performance of static music that has remained motionless, the listener is in himself no longer where he started out—just as, conversely, directed, mobile music that lays a path need not always take the listener along on a journey.

## 3.3 Fragility

In my recent work, I became interested in exploring the concept of fragility, particularly through the use of quiet and slow looped materials. These types of materials not only contribute to the sonic surfaces that arise in my music but are also an intrinsic part of the algorithmic processes I employ; they augment the sense of disorientation and slippage created through algorithmic processes and, in the process, create music that is sonically fragile.

Fragility can occur in music due to a broad range of phenomena; these not only include the presence of quiet and slow sounds but can also occur due to other less ostensible musical factors such as musical structure, psychoacoustic phenomena, choices of notation, and performance situations, among many others (Epstein, 2017). In her work on the typology of musical fragility, Epstein argues that repetition can act as a fragmentation tool which, in turn, can lead to structural fragility in a composition (2017, p. 44). Although my music does not display the same level of fragmentation observed in the works cited by her—namely Bernhard Lang's *Differenz/Wiederholung 1.2* (2002) and Eric Wubbels's *This is This is This is* (2009–2010)—my looping processes are nevertheless a source of musical discontinuity, causing instability in a piece's sense of directionality (Epstein, 2017, p. 44). This is then amplified by the quiet and slow materials I employ, which help mask the exact nature of the repetition-based algorithmic processes taking place in the background, thus contributing to the perceptual instability of the piece. In other words, these materials make the repetition process less obvious, more disorienting, and more challenging for the listener to grasp. It is not the structures themselves that are necessarily fragile, but rather the disorienting experience they present to the listener.

Repetition can initially work as a microscope, enlarging both material and its inner relationships: the listener's focus becomes free to wander around at each loop, and the ear can focus on, decompose, and analyse different aspects of the music at each loop, placing the musical content outside of time (Margulis, 2014, p. 7). However, literal repetition also affects how we perceive the content itself that is being repeated, particularly over long periods of time (Margulis, 2014, pp. 26–54; Sabbe, 1996, p. 12). According to Sabbe, the identical repetition of a musical object creates the illusion of permanent accessibility while, in fact, the object keeps changing with each repetition, even if its notation remains identical. This phenomenon is supported by the experiments of Deutsch, Henthorn, & Lapidis related to the so-called 'speech-to-song illusion': in their research using looped recordings of speech, they show that literal repetition affects how we perceive the content of these recordings, with the pitch and rhythmic content of the voice becoming more and more prominent as the semantic content loses perceptual importance (2011). This illusion is related to the notion of 'semantic satiation', a term originally coined by Jakobovits in 1962. He describes how continual repetition of a spoken word can result in a temporary 'detachment' of its meaning from its sound for the listener (Jakobovits, 1962, pp. 18–36); in other words, utterances temporarily become meaningless sounds devoid of syntactic content.

These types of auditory phenomena have been the focus of both psychoacoustic research and the practice of composition. Some composers have employed this directly in their pieces. An example of this can be observed in Tom Johnson's *Same or Different* (2004), a participatory composition in which all audience members are invited to say the words 'same' or 'different' out loud after hearing two similar or identical musical cells. As the cells grow, it becomes increasingly challenging to evaluate their likeness with a high degree of confidence, with the diverging shouts of the audience members testifying to how different people will experience repetition dissimilarly.[4] Another example of the compositional application of these psychoacoustic phenomena can be found in the early tape

[4]It is important to note that psychoacoustic phenomena might not be the only contributors to the audience's responses but that human psychology might also play a significant role in it. For instance, in *Same or Different* No. 7, all twelve pairs of cells are identical; however, an audience member might be less inclined to shout 'same' twelve times in a row, thinking they might need to adjust their responses due to their lack of statistical uniformity.

compositions of Steve Reich. In pieces such as *It's Gonna Rain* (1965) and *Come Out* (1966), Reich works with short loops of recorded speech that undergo a slow and gradual phasing process; as one listens to these pieces, the semantic content of the speeches gradually disappears as they, effectively, become melody (Simchy-Gross & Margulis, 2018, p. 1).[5] A similar principle can be observed in Gavin Bryars's *Jesus' Blood Never Failed Me Yet* (1971), although Bryars's loops are substantially longer and, therefore, display a less intense auditory illusion of loss of semantic content.

The uncertainty of whether a repetition is literal or not is also explored by Feldman, who famously writes that his use of repetition is a way of '"formalizing" a disorientation of memory' (2000, pp. 137–138); he compares his method to 'walking the streets of Berlin—where all the buildings look alike, *even if they're not*' (2000, p. 138). Feldman's solution for achieving this disorientation was to combine subtle variations in the repeated structures together with fragile materials. Somewhat surprisingly, quiet and slow materials can, in fact, pose a perceptual hurdle to the listener since they decrease musical contrast and, with it, mask structures that would otherwise be readily identifiable and serve as reference points for the ear. This type of music thus requires an additional commitment from the listener, as Harrison (2007, p. 33) points out:

> For me, working at a low dynamic volume has had the added effect of removing timbral differentiation as well as allowing me to create a softer palette. In all music which inhabits a quiet sound world, such as Feldman's, the sounds are no longer projected towards the spectator. Thus the listener is forced to bring something of themselves to the listening experience, to meet the sounds half-way and thus intensify the experience.

In my music, I have been especially influenced by the 'reverence for silence' of the post-Cagean composers of the Edition Wandelweiser (such as Jürg Frey, Eva-Maria Houben, and Michael Pisaro) and the 'reverence for quietness' of the post-Feldmanian composers whose music has been published by Another Timbre (such as Bryn Harrison, Laurence Crane, Catherine Lamb, Ryoko Akama, and Adrián Demoč, among many others). Be it through punctuations with extremely long silences, musical stillness, or repetition of quiet materials, a common element to the highly unique music of all these composers is that the experience of time while listening to them is seldom linear. It is often the case that this type of music creates a listening experience that is fragile and fleeting; as Frey (1996) writes about his own work:

[5]Interestingly, Simchy-Gross & Margulis point out that these two compositions by Reich predate the discovery of the speech-to-song illusion by science (2018, p. 1).

> With this music, we do not have a memory of moments of particular intensity after the concert. The situation is not at all shaped by memory. There is indeed the feeling that the music is already gone.

Besides working with quiet materials, I also tend to work with a very limited sound palette. My music often employs homogeneous sounds, and changes of playing technique are rare. When I employ changes of instrumental technique, they are always accounted for by the algorithm and are often used to create multiple structural layers rather than continuous transitions of timbre. Thus, these are not subtle slow variations of bowing angle or lip pressure; instead, they are sudden changes that emphasise the algorithmic nature of these works. I also have a predilection for simpler timbres, and my music seldom features any extended techniques at all. Part of my reason is to avoid too much timbral differentiation, which can help draw the listener 'inside' the work's texture (Harrison, 2007, pp. 32–33). My use of piano pedalling (which I usually require to be half or fully depressed for a whole piece) and predilection for long sustained notes can help to further increase the blurriness of my soft musical textures.

Stillness is another significant characteristic of the materials I work with; when coupled with the repetition-based processes I use, stillness helps create the gradual and sometimes imperceptible changes that my music often displays. It invites us to focus our attention on the more minor details of textures and structures. This, in turn, can serve as a form of 'musical misdirection': while our attention shifts towards the microstructure of these loops, global changes can sometimes go unnoticed. This type of fragility can thus emphasise a mode of listening in which sudden changes in our perception of musical context can occur; as Frey (2004) states, 'a seemingly static, monochrome sound gradually allows us to recognise that we are suddenly somewhere totally different.' The listener invariably oscillates between listening locally and globally, further contributing to the slippage displayed by the composition.

The relationships between fragility, staticity, repetition, and experience are also prevalent in other art forms. For instance, Post-Minimal artists often work with simple shapes that emphasise process and manipulation of material (Honour & Fleming, 2010, p. 853). The work of Agnes Martin, which has had a great influence on me, is one such example. Varnedoe (2006, p. 241) describes her work as essentially incorporeal, and her type of abstract painting as sensual rather than cerebral. He writes:

> In order to understand this art, you have to be there to feel the touch of the pencil, the lightness with which it hits the surface, to feel the subtlety of the tint. Martin's art is all about experience—on the part of both the artist and observer. (Varnedoe, 2006, p. 241)

[redacted]

**Figure 3.5:** Agnes Martin's *Untitled #5* (1994); image available at `https://www.tate.org.uk/art/artworks/martin-untitled-5-ar00177`

The focus is thus not on the object which constitutes the artwork itself but instead on the experience of observing it. This is evoked by the materials used by Martin: the pale shades of her palette, the repetition of lines and visual patterns, the tiny details of her technique revealed only at close inspection. An example of this type of painting can be seen in her work *Untitled #5* (1994), shown in Figure 3.5, in which she divides a large square canvas of over a metre and half in length in eleven bands, which are then filled one by one with a fixed sequence of three extremely pale tints. This methodical exploration of simple shapes using faded colours is typical of her work and which, similarly to music composed with fragile materials, draws the observer inside the painting: these are artworks that require a high degree of commitment on the part of the observer due to their highly subtle materials with low contrast. This commitment has a strong parallel with that observed by Harrison (2007, p. 33) regarding the requirements imposed on listeners by music that is very quiet and lacks timbral contrast.

## 3.4 Emergence

In the context of algorithmic design, emergence refers to the observed complex behaviours that can arise from a system made out of much simpler parts (Pearson, 2011, p. 108). These parts are often defined using straightforward rules that cannot account for all the system's complexity from an observer's perspective. It is important to emphasise that emergence is a property that relates to our perception of a system: all sophisticated patterns that may arise from simple interactions are, by definition, contained in the system itself, even if they are not *apparent* to us (Crutchfield, 1994, pp. 2–4).

Emergence can be widely observed in the natural world. Complex—and sometimes perplexing—collective behaviours can arise from the interaction of multiple individuals in situations where no one plays the role of a leader; examples of this include the formation flying of flocks of birds, the grouping behaviour of schools of fish, and the complex societies of ants that lack any central source of coordination (Crutchfield, 1994, pp. 1–2).

Although emergence can arise from random interactions in a generative system, randomness is not a prerequisite for its existence. In fact, many deterministic systems display emergent properties, such as is the case of chaotic dynamical systems (Crutchfield, 1994, pp. 2–3).[6] Deterministic musical processes can also display emergent properties, such as in Alvin Lucier's *Music for Piano with Slow Sweep Pure Wave Oscillators* (1992) and Phill Niblock's *Five More String Quartets* (1991). Both of these pieces employ glissandi that interact with other tones during their trajectory and, in the process, generate emergent auditory phenomena in the form of acoustic beats. The frequencies of these beats will vary in time since they are dependent, at every given moment, on the constantly changing frequencies of the glissandi tones and their surrounding pitches. Both of these pieces make use of multiple simultaneous glissandi that move at different rates, creating sonic results that are both complex and impossible to be precisely accounted for from simply looking at the pieces' scores. This complexity emerges

[6]In mathematics, the notion of chaos refers to dynamic systems whose behaviours are very sensitive to their initial states and appear to be completely random, despite the systems themselves being deterministic. With such systems, it is impossible to predict a specific long-term future state since even the slightest difference between initial states will lead to vastly different future ones. Despite this impossibility, the collection of all possible trajectories of such a system can reveal a patterned structure—an emergent pattern that arises from the system's chaotic nature (Strogatz, 2000, pp. 2–4).

from the local interactions that take place during the pieces' executions, while the scores are only concerned with notating their execution but not the sonic results.

Other examples of deterministic emergence can be found in the phasing pieces of Steve Reich, such as in his *Piano Phase* (1967), as well as the previously mentioned *It's Gonna Rain* (1965) and *Come Out* (1966). In *Piano Phase*, two pianists start by playing the same series of notes in unison at the same constant tempo. After several initial repetitions, the second pianist is instructed to speed up their tempo ever so slightly while the first pianist holds the same initial one. This change of speed marks the effective start of the phasing process: the notes of the second piano will begin to gradually move ahead of the first piano, desynchronising the music. At first, both pianos seem to continue to play in perfect unison, but soon thereafter, when the melodies of each piano have phased enough apart from each other, the combined sound is perceived as having a short echo. As the phasing process continues, this echo will be transformed into two separated but identical melodic lines that are delayed from one another. As these two lines continue to move apart, the attacks of one melody will, at some point, fall precisely in between consecutive attacks of the other, creating an interlocking pattern that will sound twice as fast as the original melody; these two melodic lines will then continue to move out of phase with one another, halting the perception of a perfect interlocking. The attacks of both melodies will later start to sync up again, although the pitches are now phased by one semiquaver. This rich listening experience, with its multiple perceptual threshold crossings, emerges from and can be wholly accounted for by a very simple process that is notated as shown in Figure 3.6.

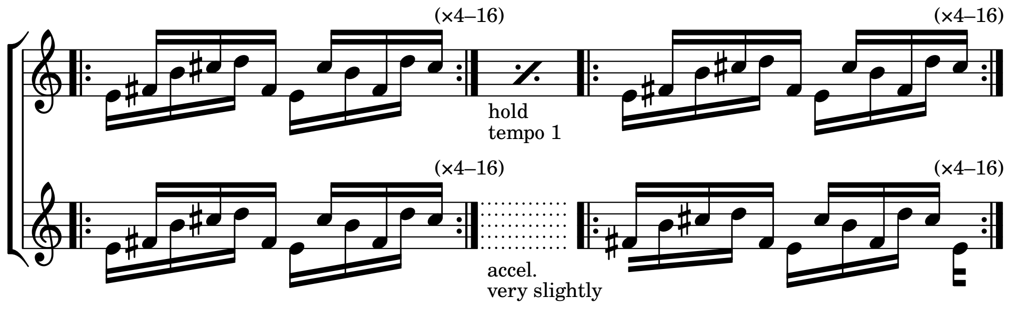

**Figure 3.6:** Excerpt from Steve Reich's *Piano Phase* (1967)

In my music, emergence manifests itself primarily as a consequence of my use of loops. As previously discussed in Sections 3.2 and 3.3, the looping techniques I use, coupled with the fragile materials I tend to work with, significantly contribute to an overall sense of disorientation in music that is otherwise created through very linear processes. Most of my pieces use looping windows with fixed lengths and step sizes, which could, in theory, lead to highly predictable sonic results. In reality, the listening experience offered by these pieces is far richer and subtler than their linear processes might suggest: the relatively long looping windows pose a challenge to our memory and thus make the identification of what is changing (if anything) difficult to assess, creating a disorienting listening experience. This is further reinforced by my use of short step sizes between consecutive looping windows, which often result in a perceived identical repetition of material. Therefore, by engaging with the listener's memory in this way, the superimposition of simple linear processes coupled with the right set of materials can, from a perceptual point of view, create non-linear and complex entities.

Emergent structures can also form at the borders of these looping windows. These are places where some elements of the input music interact with one another even though they are, in fact, not consecutive events in the input music. Thus, these constitute moments when new structural relationships emerge from the overarching process and cannot be completely accounted for solely from the original material. Notes and chords that are apart in the input music suddenly follow one another in direct succession, creating transient relationships in the process. These relationships shift and disappear as the elements drift apart; for instance, this can happen when the looping window leaves one of the elements behind or when it uncovers a new element that is heard in between the previous two. These new relationships often seize the listener's attention in an act of musical misdirection: depending on the properties of these new relationships, a strong link is perceived, and these elements are grouped as a single structural entity.

The laws that govern perceptual grouping were first studied by the psychologists of the Gestalt school in the early 20th century. They have shown that, among the many factors influencing how our mind groups visual entities together, proximity, similarity, and continuity are key attributes that influence the way we perceive links (Wertheimer, 2012). The law of proximity states that when observing a configuration of several entities with non-uniform distances from one another, closer entities will be perceived as belonging to the same group (Wertheimer, 2012, pp. 130–135). This can be observed in the first example shown in Figure 3.7: because of the wider gaps between some of the vertical rows of circles, an observer will perceive this configuration as three distinct columns of two by six circles each. The law of similarity describes how we will group objects with similar characteristics together (Wertheimer, 2012, pp. 135–139).

An example of this is shown in the second diagram of Figure 3.7, in which the yellow circles are grouped together while the pink ones form a second group. The law of continuity (sometimes referred to as the law of good continuation) states that objects in a path are perceived as a group that follows the smoothest possible path (Wertheimer, 2012, pp. 149–160). This can be observed in the third example of Figure 3.7, in which a series of circles is perceived as a wavy line.

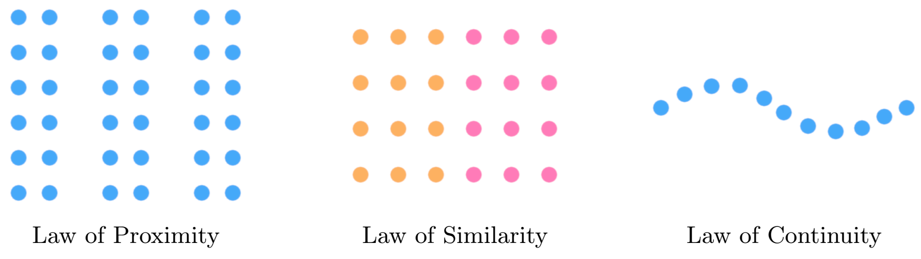

**Figure 3.7:** Gestalt grouping principles

Gestalt principles can be applied to musical structures as well, as demonstrated by the research of Bregman (1990).[7] In music, the primary aspects contributing to linkages are the temporal and pitch proximities between musical elements (Bregman, 1990, pp. 455–528; Tenney & Polansky, 1980). When writing about the application of Gestalt principles to music, Bregman uses the term ‘auditory stream’ when referred to the perceived groups of sounds, defining it as the ‘perceptual grouping of the parts of the neural spectrogram that go together’ (1990, pp. 9–10). He demonstrates that certain sequences of sounds can give rise to what he terms ‘auditory stream segregation’: that is, depending on the temporal and pitch properties of the sounds in a sequence, such as fast alternating high and low pitches, the listener may perceive them as two independent streams of tones instead of a single stream (1990, pp. 642–644).[8] This means that the closer in the pitch field that these subsets of tones are, the more likely they will be perceived as a single group—particularly if they are not too widely separated in time. In my music, such groupings become very prominent when their elements stand out from their surroundings, especially when some of their parameters are novel in a given musical passage; for instance, when new notes

[7]Interestingly, when describing his law of similarity and its relation to the law of proximity, Wertheimer uses musical pitches in some of his examples (2012, pp. 137–141), although they are much simpler than those explored by Bregman (1990).

[8]Composers in the Baroque period have widely used a technique known as ‘virtual polyphony’, which consists of a practical application of the notion of stream segregation: a single monophonic sequence of tones played by a solo instrument is perceived as two or more distinct musical voices through the alternation of high and low registers, thus creating the illusion of true polyphony in the process (Bregman, 1990, pp. 464–465).

appear in a register that had not been previously used, their linkage will be particularly strong. Since these emergent linkages occur at the transition points between consecutive windows, it is often difficult to perceive precisely where these borders are located when listening to this music, even if they are quite evident to the eye when looking at the score. I also tend to use other elements that further blur the location of these borders, such as nearly identical pitch content between consecutive windows, constant use of piano and vibraphone pedalling, long sustained notes, and notes tied across window boundaries. The overall impression is one of a slowly shifting landscape that moves at an irregular pace.

A concrete example of stream segregation and blurring of looping window borders can be observed in Figure 3.8, which shows an excerpt of measures 49–56 of *Cartography #11*, for solo piano, with each auditory stream notated by a different colour. In this example, the segregation between the two auditory streams is primarily due to pitch proximity, with the pitches G5 and A♭5 constituting one stream while E♭4, E4, and F4 constitute another. The marcato accents—the loudest articulation a note can have in this piece—that are present at the start of each group from measure 50 onwards further contribute to the perception of two separate streams, each of which displays relationships not present in the input music.

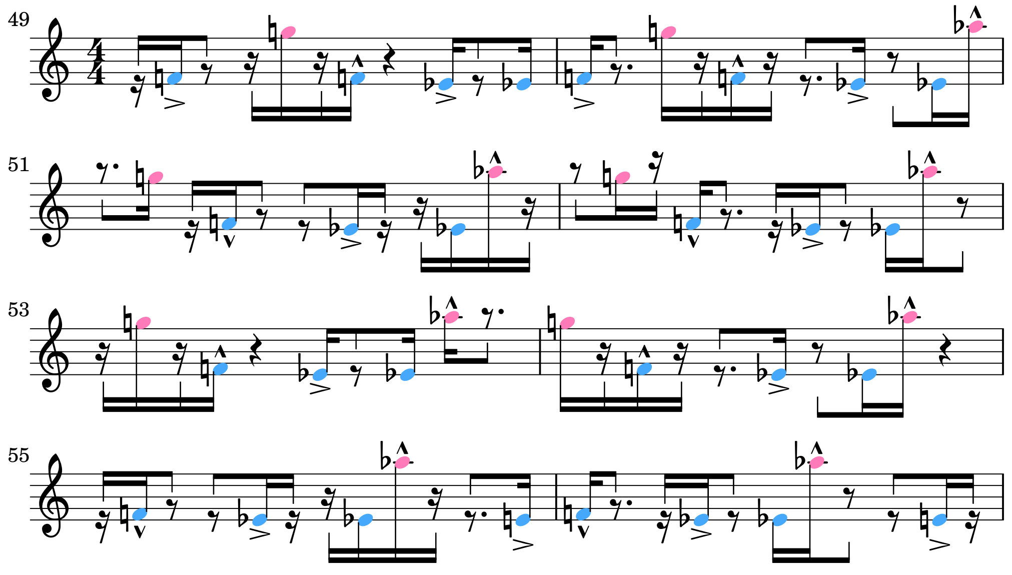

**Figure 3.8:** Auditory streams in *Cartography #11*, measures 49–56

Randomness can also play an important role in creating emergent structures. As previously discussed, a system is not required to employ randomness in order to promote emergent behaviours. However, randomness can be an excellent tool for creating emergent patterns as it can generate seemingly complex structures through straightforward procedures. A visual example of this can be observed in the so-called ‘Perlin noise’, a random function most often used to create two-dimensional textures that are then applied to three-dimensional models in 3D design (Pearson, 2011, pp. 57–59). This function initially creates a matrix of random numbers and then interpolates between neighbours in all directions, creating smooth gradients between the randomly generated values. Although this is a random procedure, the right set of parameters can promote emergent structures to develop from this noise function, as shown in Figure 3.9. The textures created with Perlin noise can show surprisingly natural and organic structures, which is part of their main attractiveness to 3D modellers (Pearson, 2011, p. 57).

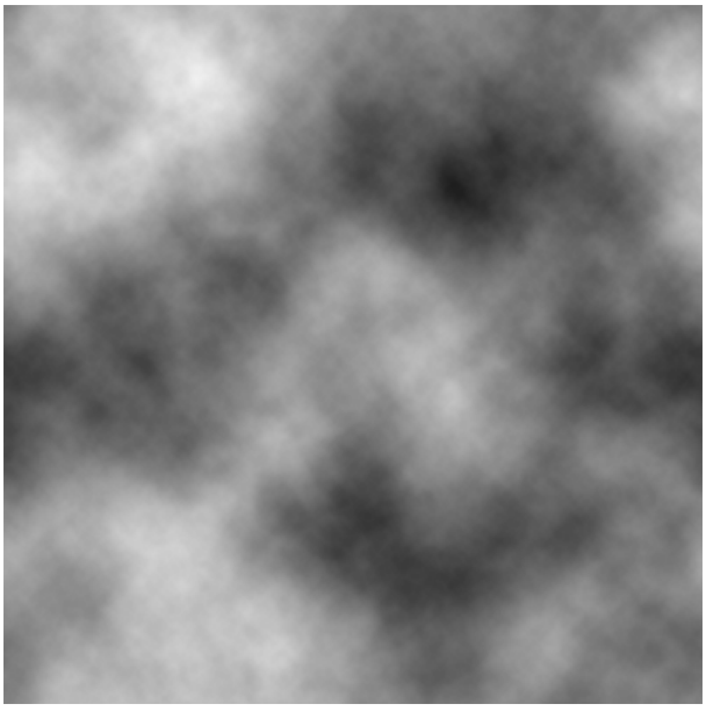

**Figure 3.9:** Example of Perlin noise

In my music, I often employ random procedures when generating the input music upon which the repetition-based processes will operate. These are heavily constrained random procedures which typically make weighted random selections from a pre-defined pool of options. By carefully selecting these weights and the elements available in these pools (or how the algorithm generates these elements, in case they are also random), emergent structures can also appear in the input music, similarly to the Perlin noise described above. The perception of these structures is once again tied to their component’s temporal proximity and intervallic characteristics (Bregman, 1990, pp. 455–528; Tenney & Polansky, 1980). Even prior to applying any looping process at all, this randomly generated and linear input music often displays identical or nearly identical repetitions of material that can be picked up by the ear. Calls and responses, quasi-canonic moments, and interwoven rhythms are all common musical structures that arise in this input music but which are not directly defined in the algorithms. Instead, their existence is fundamentally tied to how we perceive music: these are the result of our brains looking for patterns in otherwise random events. In my *Cartographies* series, I was particularly interested in exploring these emergent structures that arise from my usage of the mental model of a container (which will be discussed in Subsection 5.1.2).

Some deterministic procedures can also be perceived as 'random' due to their complex nature and seemingly unpredictable results—that is, unpredictable only to a human observer as, no matter how complex, a deterministic procedure will always be, by definition, entirely *calculable*. This is the case of Bernhard Lang's use of cellular automata in his *Monadologie* cycle of compositions (2007–present). Consecutive states in these compositions are generated through iterations of a cellular automata model, which are then mapped into pitch and rhythm. Although this technique is entirely deterministic, the results are chaotic enough to appear random to the listener (Dysers, 2019, pp. 104–115).

A visual example of emergent structures appearing from a straightforward random procedure can be observed in Sol LeWitt's *Wall Drawing #118* (1971), a work that exists as a set of instructions to be executed on a wall. The original instructions from the artist, as quoted in Russeth (2012, pp. 3–4), are:

> On a wall surface, any
> continuous stretch of wall,
> using a hard pencil, place
> fifty points at random.
> The points should be evenly
> distributed over the area
> of the wall. All of the
> points should be connected
> by straight lines.

Each realisation of this work will, of course, create a different-looking result. Figure 3.10 shows the result of a possible implementation of these instructions.[9] As can be seen in this realisation, the connections between those fifty points create regions of higher and lower densities of lines, in a visual result that is not too dissimilar from the Perlin noise example previously shown in Figure 3.9.

[9]For the code that generated this image, see Appendix A.

This rich result with a complex structure of lines and points emerges solely from the straightforward instructions given by the artist.

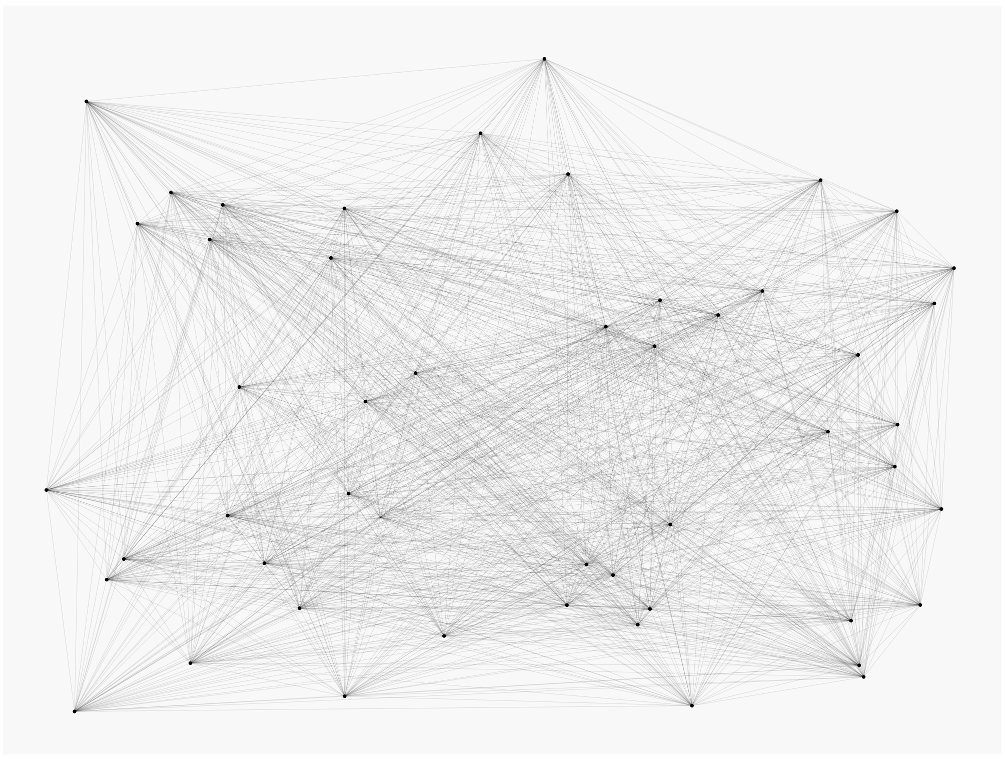

**Figure 3.10:** Possible realisation of Sol LeWitt's *Wall Drawing #118* (1971)

## 3.5 Liminality

I use the term 'liminality' to describe a large scale quality of my recent music. Whereas slippage relates to local phenomena—such as the disorientation caused by consecutive iterations of a looping window—liminality is concerned with the overall listening experience induced by the unfolding of the algorithmic processes. Liminality thus relates to transience, referring to the overall ambiguity and disorientation experienced when being on a threshold. As Buchanan (2010a) describes it, 'The liminal is the in-between, the neither one thing nor the other.'

The shape of my music is dictated by a series of states that undergo algorithmic transformations. A state can be understood as a snapshot of a moment in time, one that has the potential to morph into another by the unfolding of algorithmic processes. In most of my pieces, it is the trajectory from the initial state to the final one that defines the composition's shape and, therefore, musical form becomes an emergent property of these systems, as opposed to being imposed *a priori*. This approach of musical processes as agents of form helps emphasise the temporal mode that Kramer refers to as 'nondirected linearity' (1981, pp. 541–542; 1988, pp. 39–40). Kramer's concept describes music that has both inner motion and a sense of continuity, but that lacks an unequivocal goal towards which it moves to. Even though the term 'nondirected linearity' does not describe an attitude to musical form per se,[10] the emergent approach to form that I employ in my work produces music that lacks contrasting segments, which, in turn, helps to further underline its sense of continuity without any specific goals (Kramer, 1981, p. 542). In other words, by eschewing more dramatic formal models and allowing form to emerge out of the musical processes themselves, my compositions often seem to be in a single transient state for the duration of the whole work. However, it is important to note that even a composition written with multiple individual sections can lack this sense of destination, an idea supported by Frey (1996), who describes his use of different sections in his music as incidental:

> When this music has individual sections, they are not developed from one to the next, or linked by contrast—rather, the individual parts appear to be tied to each other by an invisible thread. This means that each idea for a new section must be a new beginning, dialectically unconnected with the previous section. In this case, there are only the sections which occurred to the composer, and these sections bring forth the identity of the piece.

Despite using algorithmic processes to shift between states, my music does not employ a narrative, be it musical or extra-musical, with the exact trajectory created by these processes being selected through stochastic procedures alone. This, in turn, emphasises their lack of final concrete goals: the focus of the pieces lies in the transformations themselves rather than in a clear destination to be reached. The processes and materials I employ in my recent work are never opposing one another, and thus there is no dialectic conflict to be resolved. I am particularly fond David Lang's description of Beethoven's vastly more dialectic music; he writes, 'When I listen to Beethoven, the emotional trajectory is incredibly erratic. That's not an emotional life I want. I like to remain in more or less one state' (David Lang as quoted in Davidson, 2014). The non-goal-oriented and ever-unfolding compositions of Morton Feldman are an excellent counterexample to this more dialectic music. While Feldman's music goes through transformations over time, the changes occur in such a way that the music flows without a clear sense of finality or a clear destination; as Coolidge

[10]That is, music that employs a more segmented approach to form can still make use of this temporal mode. Kramer himself uses the term to describe music that displays substantially more formal contrast than my own music (1981, p. 542).

(1988, p. 128) describes it: ‘Feldman’s music seems never to manipulate, as if never overdetermining its destination, so not rushing to insure the most ease (audience) in getting there. An absolutely non-rhetorical music [. . .].’

This lack of clear musical goals on a larger scale coupled with the ever-unfolding (but often ungraspable) inner motion of this type of music can give rise to what I describe as ‘liminal music’. Liminality is a concept from anthropology that characterises the transient aspects of certain rites of passage (Gennep, 1960, p. 11). The term has now become commonplace in subjects other than anthropology; for instance, Cotton (2009, pp. 94–95) writes about ‘liminal spaces’ concerning architecture and photography, defining it as ‘areas [that] exist where cracks in institutional or commercial definitions appear, and our sense of place is dislocated.’ This term is often applied to describe situations or locations that are intrinsically transitory and in which an individual feels displaced, such as crossing a hallway, staying in hotel rooms, driving on motorways, or taking a commercial flight—all of which have transient purposes but are never the destination in itself.[11] There exists thus a strong parallel between the notion of liminal spaces, places of transition and dislocation, and the repetitive, ever-unfolding music that attracts me, one that, as Harrison (2007) describes, gives the impression of implied motion despite lacking traditional musical progress. In my own music, liminality is interconnected with the notions of slippage and fragility; it is the sense of constant disorientation and ungraspability that emerges from the ever-present slippage and fragile materials that create these feelings of transition, aimlessness, and dislocation.

Liminality also plays an essential role in the algorithmic work of composers engaging with cybernetic systems, such as Roland Kayn and Agostino Di Scipio. Cybernetic music can be described as music produced by self-regulatory networks of electroacoustic devices through which audio signals can freely roam; these audio signals affect one another, creating complex emergent behaviours through

[11]In the case of the liminal space of an airport, Huang, Xiao, & S. Wang (2018) argue that airport lounges can create a sense of timelessness and placelessness that often affects the behaviour of the passengers who temporarily make use of it, and who may feel uninhibited by the apparent absence of social structures. The liminality of airports has been musically explored by Eno in his seminal *Music for Airports* (1978), conceived as music to be continuously played as a loop in the background of an airport lounge. Concerning the necessary characteristics to such music effective, he writes, ‘It has to be interruptible (because there’ll be announcements), it has to work outside the frequencies at which people speak, and at different speeds from speech patterns (so as not to confuse communication), and it has to be able to accommodate all the noises that airports produce. And, most importantly for me, it has to have something to do with where you are and what you’re there for—flying, floating and, secretly, flirting with death’ (Eno, 1996, p. 295). As such, Eno’s composition displays similar liminal characteristics to that of the environment it was written for.

feedback. These systems display an inherent goallessness and resemble living entities that are able to self-regulate; in the case of Kayn, the composer first sets up the machine (i.e. both the hardware wiring between the modules as well as what audio signal will go through it) before exploring it in real-time through experimentation with the system's interface. These are systems that are thus inherently liminal, chaotically transitioning between possible states but never reaching a final destination. Pickles (2016, pp. 74–75) writes:

> [Kayn equates] the systematic processes applied prior to the recording (the routing of the modulation matrix and the order of the initial sound signals from tape) with the systematic, human control processes he is utilising as the real time operator of the compositional system.

Although I do not employ real-time strategies in my work—these are not required as my focus is in generating notated scores—my systems are, nevertheless, entities that can also roam, producing endless different results in the process. Similarly to how Kayn is the real-time operator of his machines, so am I the performer of my systems when I tweak their metaphorical 'knobs and dials' in order to lead it towards certain directions, which are decided through a mixture of previous experience, musical intuition, and pure blind experimentation. A system can be tweaked not only by simple changes of variables and initial values but also by completely rewriting parts of the system on the fly. This is why a modular approach that uses multiple interrelated abstractions is fundamental for my compositional workflow, as certain musical entities and algorithmic processes can be modified independently from the others (see Chapter 4 for a methodological discussion on the topic).

Due to its inherent transience and in-betweenness, liminality also implies a sense of detachment, of leaving a place behind without having reached one's destination yet. A low-intervention approach to musical processes can further accentuate this experience of detachment: by letting the process take over and dictate the path to be taken in the composition, the composer is adding one more layer of detachment, one at the systematic level of the work. This attitude of using low-intervention is shared by many artists working with algorithmic systems, as previously discussed in Section 2.3. About his automatic techniques employed in *Discreet Music* (1975), Eno (1996, p. 330) writes that his interest lay in defining the music's materials and processes and then allowing their combinations and interactions to emerge from the algorithm. In the case of my own music, my interest lies in automatic processes that are carried out throughout a composition, slowly transforming and distorting its fragile sonic landscape and, in the process, creating a liminal sense of constant unfolding.

A parallel can also be made between this detachment caused by low-intervention coupled with systematic approaches and the ideas explored in the genre of photography known as 'deadpan'. This is a type of photography that is impassive and expressionless, in which the photographer stares at a subject with a detached and agnostic eye (Cotton, 2009, p. 81). While the photographer chooses the subject at which they point their camera, the photograph happens almost by itself, with the subject dictating the framing. Deadpan constitutes the quintessential non-goal-oriented approach to art, one that is all about the subject itself and not the photographer's own views about it, an attitude that bears a strong similarity to John Cage's attempts of avoiding individual taste and memory in his music (Griffiths, 2011, pp. 26–30). It is a low-intervention type of photography in which the photographer stares at something for what that something is, creating inexpressive and emotionless images in the process. Deadpan photography often lends itself to serial exploration (LeWitt, 1967b) through collections of related subjects upon which the deadpan look is cast (Cotton, 2009, pp. 82–83). The methodical and repetitive approach towards its subjects is an important characteristic of serialised deadpan photography, one which is also very close to my attitude towards my own work as a composer.[12]

Two notable examples of deadpan photography can be found in the works of Bernd & Hilla Becher and Rineke Dijkstra. The Bechers had an encyclopaedic interest in architectural photography; their better-known series of works consists of photographic catalogues of vernacular architectural structures photographed from an identical perspective. The notions of seriality and repetition are thus of primary importance to their work. Figures 3.11 and 3.12 shows examples from two of their series, *Water Towers* (1988) and *Pitheads* (1974), respectively.

Rineke Dijkstra is well-known for her series of deadpan portraits of people, often printed in near life-size. Her photographs display the characteristic unsentimental, systematic, and detached photographic style often associated with the deadpan aesthetic (Cotton, 2009, pp. 111–112). Figure 3.13 shows the photograph *De Panne, Belgium, August 7 1992* from her series *Beach Portraits* (1992–2002), a series which deals with a particularly liminal topic: the transition from adolescence into adulthood.

Algorithmic processes can, in fact, closely resemble this deadpan approach: by algorithmically inspecting and manipulating musical material, the primary focus of the composition can fall on the object itself, be it the mechanical process

[12]An example of this can be found in my series of compositions entitled *Cartographies*, in which I explore how the same mental model can be used to create unique but interconnected musical pieces. See Chapter 5 for an in-depth commentary on these pieces.

[redacted]

**Figure 3.11:** Bernd & Hilla Becher's *Water Towers* (1988); image available at `https://www.moma.org/collection/works/49624`

[redacted]

**Figure 3.12:** Bernd & Hilla Becher's *Pitheads* (1974); image available at `https://www.tate.org.uk/art/artworks/bernd-becher-and-hilla-becher-pitheads-t01922`

[redacted]

**Figure 3.13:** Rineke Dijkstra's *De Panne, Belgium, August 7 1992* (1992) from the series *Beach Portraits*; image available at `https://www.tate.org.uk/art/artworks/dijkstra-de-panne-belgium-august-7-1992-p78328`

that drives the composition, the generated material that is being processed, or a combination of both. This is similar to Fink's argument that Minimal and Process Music gives a 'deadpan attention to the "pure object"', particularly with its focus on combinatoric processes (2005, pp. 30–31). Through its strict and mechanical nature, algorithmic processes also enable a non-interventional and dispassionate attitude towards the musical subject, inviting explorations into the musical material itself.

## 3.6 Conclusion

Repetition has become one of the most important techniques I have used in my recent work and is employed throughout my accompanying portfolio. I use it not only for its structural properties but also as means for creating disorienting and ambiguous listening experiences. Repetition is generated in my music by linear and strict algorithmic processes that nevertheless sound as if they are non-linear. On a local level, this gives rise to slippage: that is, the listener is unable to fully grasp the process's constant movement and slow unfolding, with the resulting music occasionally giving the appearance of complete stasis. The fragile input materials that I tend to use—that is, those that are soft and slow—further contribute to this disorientation, as they decrease musical contrast, creating a lack of clear anchor points for the listener's ear.

Repetition also gives rise to emergent structures that are formed at the borders of consecutive iterations of my algorithmic processes. These are ephemeral structures created by the random collision of musical entities that are not present in the input music (that is, the music that serves as input for the looping processes). In the right circumstances, notes and chords at the borders of consecutive iterations of the process will be perceived as a single structure, further obfuscating the compositional process and increasing the overall disorientation experienced by the listener.

All these elements together can give rise to a liminal musical experience, one that comes from a sense of constant musical unfolding without ever reaching a final goal. As such, the focus of the pieces in my portfolio lies in the unfolding of the musical processes themselves as well as the disorientation they create. When designing my systems, I aim to maximise their potential for generating results that I consider interesting and engaging, such as those that display a high degree of emergent structures, perceptual fragility, and slippage, with the version selected as the final output discovered along the way through a process of heuristics and experimentation.

# Chapter 4

# The Auxjad Library

Auxjad (Agostinho, 2021) is an open-source Python library that provides auxiliary classes and functions for Abjad 3.4 (Bača et al., 2021). It serves as a compositional toolkit for the specific type of algorithmic thinking that underpins the accompanying portfolio. Auxjad contains high-level implementations of algorithmic processes that can be applied to any arbitrary musical material. These include traditional algorithmic processes such as looping, phasing, and shuffling, as well as implementations of compositional procedures that are specific to my own works (these will be discussed in detail in Section 4.6). Although the latter consists of specialised procedures tailored for specific compositions of mine, they are still general enough to be included in this collection in the hope that other composers might find them useful. An advantage of working with open-source, object-oriented code is that classes and functions can easily be adjusted or readapted to suit one's own particular needs (as will be discussed in Section 4.5).

This library was written with its practical application in mind, and, as such, it encompasses operations and musical metaphors used in my own practice. As a result, this collection closely reflects my compositional interests: most of its members were created with specific compositions in mind or, at least, originated from compositional experimentation, even if they did not end up being used in any of the works in my accompanying portfolio. Therefore, this library was never intended as a general collection of algorithmic processes. While a public release was not the primary motivator behind its development, the algorithmic processes implemented in this library likely have broader applications beyond my own compositional practice. I hope that other composers will find this library useful and incorporate parts of it into their workflow, be it by direct use or by borrowing and deriving elements from Auxjad, adjusting it to their own purposes.

Before examining the inner workings of the Auxjad library, this chapter will briefly introduce LilyPond and Abjad, as some knowledge of these tools is necessary for contextualising my own library. This will be followed by a discussion on the motivations behind the creation of the Auxjad library. This chapter will then overview some of Auxjad's main features, demonstrated with practical examples. This is not intended as a comprehensive documentation of this library; rather, the aim is to provide an insight into the relationship between my accompanying portfolio and the Auxjad library. For the complete documentation of all members of Auxjad's API, including all their methods, attributes, and arguments, please see the documentation page in the link `https://gilbertohasnofb.github.io/auxjad-docs/`. The documentation page contains hyperlinks to Abjad's own documentation, so all mentions of Abjad's classes and functions are linked to their individual documentation pages.

## 4.1 Music Notation With LilyPond

LilyPond (Nienhuys & Nieuwenhuizen, 2021) is a cross-platform, open-source music engraving software that uses plain-text files as source files for compiling publication-quality music scores in PDF or PostScript formats, among others. LilyPond is in many ways similar to SCORE (Smith, 2013), an older text-based music engraving software for the DOS operating system, and LaTeX (Lamport, 2021), a widely used typesetting system. These tools share a common design principle: by interpreting commands entered by the user on a command line or plain-text file prior to compiling the desired output, they all break with the vastly more prevailing WYSIWYG ('What You See Is What You Get') software paradigm.[1]

This separation between the user interface and output can be daunting at first, particularly for new users. All three programs mentioned above are often described as having a steeper learning curve than their WYSIWYG counterparts. However, as this section will show, there are many advantages to this philosophy

[1]In computing, the WYSIWYG paradigm is used to describe computer software in which the user interface displays the resulting formatted document in real-time. Examples of software employing this paradigm include music engraving programs such as MuseScore, Dorico, and Sibelius, and text editors such as LibreOffice and Microsoft Office. In contrast, programs such as LilyPond, SCORE, and LaTeX work within the WYSIWYM ('What You See Is What You Mean') paradigm, in which descriptive commands are used for formatting the documents which are later compiled.

of software interface design. The most obvious one is the precision of formatting: all information is entered as text and will reliably compile in the same exact manner. Parameters, such as positions, distances, and angles, are input as numerical values, which allows for utmost consistency. In contrast, WYSIWYG software often relies on mouse input (such as clicking and dragging) to change an element's position. This can lead to issues in the engraving consistency of a document, particularly when the positioning of elements is done solely by visual means. Another advantage of these text-based programs is that they do not need to perform in real-time and can thus use computationally more expensive algorithms (Evans, 2019, p. 14). For instance, this allows LilyPond to have a far better collision detection algorithm than most software using graphic user interfaces, or LaTeX to have a far more complex justification and hyphenation algorithms than most WYSIWYG office suites (R. Zinkstok & J. Zinkstok, 2020).

Although users can type its syntax using in any text editor, LilyPond has its own dedicated text editor named Frescobaldi (Berendsen, 2020). The main advantage of using Frescobaldi over a regular text editor is that its interface is divided into two halves, one showing the source code and the other the compiled PDF file (which can be recompiled at the click of a button or press of a key). This allows the user to see the previously compiled score while typing modifications in the editing pane. Frescobaldi's menus also offer shortcuts for much of LilyPond's syntax as well as notation snippets, enabling new users to more easily and quickly produce high-quality musical scores.

In LilyPond, blocks of music can be assigned to variables, which can then be reused at will. For instance, in the first code snippet below, four notes are entered directly in a staff block, while in the second, the same notes are assigned to a variable named `notes`, which is then used in the staff block. Both code snippets are equivalent and will produce the same output, shown in Figure 4.1.

```
\new Staff {
    c'4 d'4 e'4 f'4
}
```

**Listing 4.1:** Notes defined inside a `Staff` block

```
notes = {c'4 d'4 e'4 f'4}

\new Staff {
    \notes
}
```

**Listing 4.2:** Notes defined in a variable

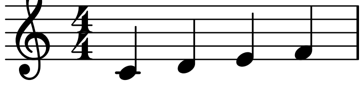
**Figure 4.1:** Example of LilyPond's output using a variable

The advantage of using variables in LilyPond is that they can be reused in later portions of the code as well as manipulated, such as by passing them as arguments to more complex functions. In the example below, the variable `notes` is used as argument for the functions `\repeat` and `\transpose`, resulting in the output shown in Figure 4.2.

```
notes = {c'4 d'4 e'4 f'4}

\new Staff {
    \repeat unfold 2 \notes
    \transpose c e {
        \repeat unfold 4 \notes
    }
    \transpose c d {
        \repeat unfold 2 \notes
    }
}
```

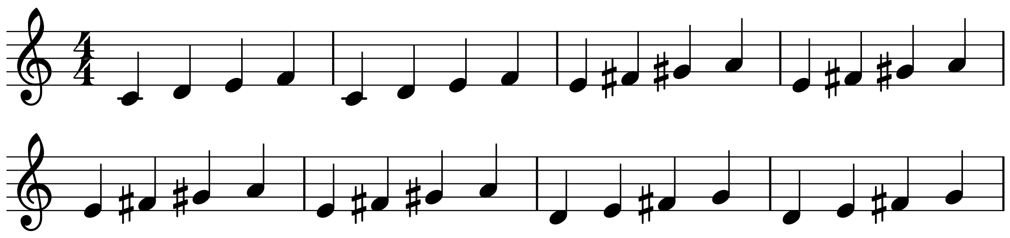
**Figure 4.2:** Example of LilyPond's functions applied to a variable

This modular approach to music engraving highlights another advantage of working with LilyPond: since the individual staves can be stored in their own variables, it becomes trivial to create multiple versions of a score by grouping these variables as required; this is particularly useful when extracting individual parts or creating reduced scores (Nienhuys & Nieuwenhuizen, 2003). LilyPond also allows for partial engraving (useful when dealing with large projects) and structuring a project using multiple music segments (e.g. by breaking a complex or longer score into several sections, each typeset into their own file).

LilyPond is also highly suitable for engraving contemporary music, as it can handle very complex rhythmic structures and make use of customised graphical objects (Evans, 2019, pp. 2–4). It has functions for drawing arbitrary 'paths', which can be combined with musical symbols or text characters from a regular typeface to create composition-specific notation. For instance, this can be used to create custom note-heads or add complex graphics to a score.

LilyPond comes with a built-in Scheme interpreter,[2] allowing users to write Lisp-like code directly in the source files of their scores (Nienhuys & Nieuwenhuizen, 2003). By using this feature, composers can write complex functions for handling both engraving and compositional procedures. Snippets of such functions have been collected, curated, and maintained over the years by the team behind the *LilyPond Snippet Repository* (Vigna, 2021).

## 4.2 Algorithmic Scores With lilypondLibrary

LilyPond uses a specific but consistent syntax that is read and interpreted from a plain-text input file. This is a crucial feature of LilyPond since plain-text files can be created not only by manually typing commands in a text editor but also programmatically. As long as this generated file complies with LilyPond's syntax and semantics, its code will compile and generate a score. This approach can then be generalised into a collection of functions and classes written in such programming language that are then each mapped to specific elements of LilyPond—such as notes, chords, rests, dynamics, time signatures, voices, and staves—effectively allowing a programmer to write music for LilyPond using any arbitrary programming language. I myself wrote a library with functions such as these, named lilypondLibrary (Agostinho, 2019), for the Fortran programming language. I used this library up until 2019, when I fully migrated to Python and Abjad.

The use of Fortran is undoubtedly an odd choice. This is an ageing programming language commonly used in fields that require vast numerical calculations, such as computational physics and engineering.[3] The single reason I worked with Fortran when writing this library (as well as for several years after that) is that it was the only programming language I was familiar with when I started working with algorithmic composition.

Fortran is categorised as an imperative programming language[4] that natu-

[2]A dialect of the Lisp programming language.

[3]The main attractiveness of Fortran for these fields is that it is cable of carrying out extensive numeric calculations extremely quickly. Nowadays, some programmers interface Fortran with other more modern programming languages, allowing them to use Fortran's speed for calculations while effectively writing code in a different programming language.

[4]More recent versions of Fortran do support object-oriented programming. However, from my own experience, its usage is far more cumbersome than with more modern languages such as Python or Java.

rally lends itself to a top-down approach. This programming paradigm can be contrasted with object-oriented programming, in which both data and functions are encapsulated in a single entity known as a class. The latter approach is far better suited for music representation (Pope, 1996), as it is far more natural to conceptualise musical notation as the manipulation of interrelated objects than sequential calls to functions. Classes serve as blueprints for making objects, which are referred to as instances, and can model any entity such as notes, chords, rests, time signatures, and dynamics, as well as higher-level structural concepts such as scales, pitch sets, rhythmic patterns, musical cells, voices, and staves. Each of these can be thought of as an object with both a state and a series of methods for transforming this state. An example of this would be a scale, with a state corresponding to its degree and mode, and methods for transposing it and altering its mode. These principles of object-oriented programming will be discussed in more detail in the following sections, particularly in Section 4.5.

lilypondLibrary's origins can be traced back to my composition *On the Origin of Pitches* (2012), for solo vibraphone. This was my first algorithmic piece, consisting of a single-page flowchart that dictated how to generate the score. Since it consists of a generative flowchart rather than a single performance-ready score, this piece exists in multiple equally valid versions as any score that is created using the flowchart is, in fact, a valid version of this piece. Initially, I created different versions of it by hand, using dice to make random selections and manually writing down notes on a score. Unsurprisingly, this type of manual transcription was extremely laborious and time-consuming, even with a short and simple solo piece.[5] It became evident that automating this process would save me from the laboriousness of manually transcribing the score and allow me to explore multiple versions of this piece far more quickly. As such, I created a Fortran library that mapped certain elements of LilyPond's syntax into Fortran subroutines, which could then be engaged with algorithmically. The example below shows the type of syntax that this library uses:

```fortran
program minimalexample
use lilypondLibrary
implicit none

```

[5]Multiple composers have addressed in the past the laboriousness of the manual implementation of generative ideas, particularly those working before computers became available for music-making. In a letter to Pierre Boulez dated from 1952, John Cage writes about the manual execution of his chance methods: 'you must realize that I spend a great deal of time tossing coins and the emptiness of head that that induces begins to penetrate the rest of my time as well' (Boulez & Cage, 1995, p. 133).

```
call HEADER(title="Minimal Example", filename="minimalexample.ly")

    call STAFF
        call NOTE(60, "4")
        call NOTE(62, "4")
        call NOTE(64, "4")
        call NOTE(65, "4")
    call END_STAFF

call SCORE(autoCompile=.TRUE.)

end program minimalexample
```

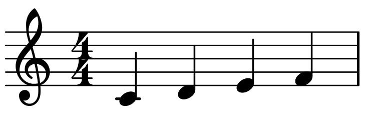

**Figure 4.3:** Minimal example of lilypondLibrary's output

In this example above, each note was entered one by one, showing no advantages over directly using LilyPond. In contrast, the example below showcases the main advantages of this programmatic approach: notes are not entered sequentially through individual commands but are instead created programmatically (in this case, via a loop that generates sixteen random notes between C4 and B5).

```
program randomnotes
use lilypondLibrary
implicit none

integer :: i, pitch
real :: r

call HEADER(title="Random Notes", filename="randomnotes.ly")

    call STAFF
        do i = 1, 16
            call RANDOM_NUMBER(r)
            pitch = FLOOR(r * 24) + 60
            call NOTE(pitch, "4")
        enddo
    call END_STAFF

call SCORE(autoCompile=.TRUE.)

end program randomnotes
```

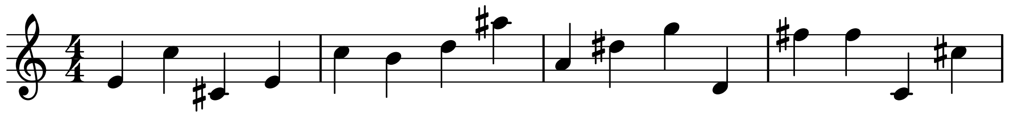

**Figure 4.4:** Random notes generated with lilypondLibrary

Even though lilypondLibrary is fully functional and can be used to create complex music—all compositions of mine up to and including the *Cartography* series were written in Fortran using it—this library does have some severe limitations. Its main limitation is that its subroutines, once invoked, directly write into an output file, requiring users to structure their music linearly: for instance, once a `NOTE` or `CHORD` subroutine is invoked, that note or chord will be written into the output LilyPond file and cannot be changed any longer. In other words, this library does not allow the user to revisit any musical entities with a second pass of operations, which consists of a major restrictive aspect. This is not to say that a better solution could not be achieved with Fortran, as these limitations are in part a product of how this library was designed.

The solution used in my practice to address the need to invoke subroutines linearly was to separate music creation from music notation. At that time, I would create matrices of numerical data representing musical parameters evolving in time. These matrices would later be mapped into musical representation using subroutines from this library. Consequently, my music had to be created with a very low-level approach, with these matrices and mappings needing to be formulated on a piece-by-piece basis.

Although this was undoubtedly a cumbersome approach to programming music systems—with its musical results being limited by what could be formulated within this environment—this was not detrimental to the final compositional results. As a matter of fact, it became very natural for me to approach algorithmic composition from within the limitations of these tools and to intuitively formulate the necessary mappings when writing those pieces. Effectively, I formulated my algorithmic and musical thinking through the lens of what was possible given my tools at the time, and these limitations became part of the technical and aesthetic frameworks within which I composed (Culkin, 1967, p. 70; McLuhan, 1964, p. 23). Despite my current workflow with Python, Abjad, and Auxjad being unquestionably better suited for the formulation and execution of my current algorithmic musical thinking, one should always be conscious that all of our tools carry ideological and epistemological payloads with them (Hamman, 2000a, pp. 91–92). This transition to a different set of tools—one that uses an entirely different programming paradigm—required a complete reformulation of my approach to both writing and thinking algorithmic music.

## 4.3 Abjad and Formalised Score Control

Abjad (Bača et al., 2021) is an open-source package for Python which computationally models music notation. Similar to lilypondLibrary, Abjad is dependent on LilyPond: it can be used to generate and compile files in LilyPond syntax so that users can create scores from scratch from the Python command line. A core concept in Abjad is what its developers termed 'formalised score control', which they define as 'the discipline of modeling and manipulating the typography of common practice notation programmatically via software.' (Oberholtzer, 2015, p. 3). Through this formalised score control, Abjad allows composers to build scores gradually (Bača et al., 2015, p. 163). As such, this package is not designed solely with algorithmic music applications in mind, as one can use it to typeset any music score. Nevertheless, Abjad provides an excellent framework for algorithmic composers interested in working with note-based systems.

Abjad is programmed using the object-oriented paradigm. Its implementation of musical concepts is done mostly through classes that encapsulate both the state of an entity and the available methods for altering this state. It models music notation as a tree of elements grouped in three distinct categories: components, indicators, and spanners. The nodes of the tree are made out of components, which can be either a container (used for modelling musical objects such as staves, voices, tuplets) or a leaf (which include musical objects such as notes, chords, and rests). Each of these nodes can have multiple indicators attached to them, such as clefs, dynamics, and tempo markings. Spanners, as their name suggests, span across multiple components, and include objects such as hairpins, slurs, and beams (Bača et al., 2015, p. 165; Oberholtzer, 2015, pp. 18–48).

Below is a comparison between LilyPond syntax and Abjad code, resulting in the same score shown in Figure 4.5. The first listing shows the LilyPond code, while the second contains the Python and Abjad code used to typeset the same score.

```
upper_staff = {
    c'4\p\<
    d'4
    e'4
    f'4\f
}

bottom_staff = {
    \clef bass
    \times 2/3 {
        <c e>2
        g,2
        r2
    }
}

\new Score <<
    \new Staff \upper_staff
    \new Staff \bottom_staff
>>
```

**Listing 4.3:** LilyPond syntax

```
>>> import abjad
>>>
>>>
>>> upper_staff = abjad.Staff([
...     abjad.Note("c'4"),
...     abjad.Note("d'4"),
...     abjad.Note("e'4"),
...     abjad.Note("f'4"),
... ])
>>>
>>> bottom_staff = abjad.Staff([
...     abjad.Tuplet((2, 3), [
...         abjad.Chord('<c e>2'),
...         abjad.Note('g,2'),
...         abjad.Rest('r2'),
...     ])
... ])
>>>
>>> abjad.attach(abjad.Dynamic('p'), upper_staff[0])
>>> abjad.hairpin('<', upper_staff[:])
>>> abjad.attach(abjad.Clef('bass'), bottom_staff[0][0])
>>>
>>> score = abjad.Score([
...     upper_staff,
...     bottom_staff,
... ])
>>>
>>> abjad.show(score)
```

**Listing 4.4:** Abjad code

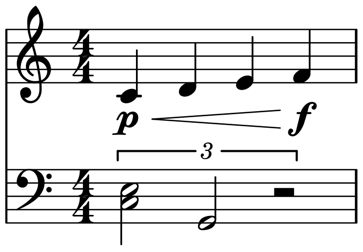

**Figure 4.5:** Example of a score generated by Abjad

This example above showcases a few of Abjad's classes and functions. Some of them, such as `abjad.Note`, `abjad.Dynamic`, and `abjad.Staff`, correspond to specific graphic objects in LilyPond. However, Abjad also has many classes and functions that do not map one-to-one onto any of LilyPond's internals. An example of this is `abjad.Selection`, a class used for grouping score components together so that they can be inspected, mutated, or assigned to an `abjad.Container`.

Two key concepts in Abjad are those of inspections and mutations. Inspection functions belong to the module `abjad.get` and are used to retrieve specific information about objects which is not necessarily part of their public properties. An example of this is `abjad.get.duration()`, which can be used to get the duration of individual components such as notes, chords, and rests, as well as the total duration of complex objects such as containers, tuplets, and selections. Mutations are a type of function that mutates the state of an object while producing no return value. In Abjad, these belong to the module `abjad.mutate`. An example of this type of function is `abjad.mutate.scale()`, which can scale up or down the duration of components, containers, and selections.

The previous code example also demonstrates another important feature of Abjad: scores can be created interactively and incrementally (Bača et al., 2015, p. 163). A composer working with Abjad can use the function `abjad.show()` to see the fully notated score of their composition as well as any sub-elements that make up this score (e.g. it is possible to compile just a subset of the total staves or a certain range of measures of a specific staff). They can also use the function `abjad.lilypond()` to check the LilyPond syntax of any of their score's components. These operations are only possible because the state of any object is accessible to be read and modified at any point during a program's execution, unlike the case of lilypondLibrary. Coupled with other built-in functions such as `abjad.select()`, `abjad.attach()`, and the functions in the modules `abjad.get` and `abjad.mutate`, Abjad enables the composer to interact with the musical material as they input the score. As the developers of Abjad themselves remark:

> We hope this will encourage composers working with Abjad to transition from working with lower-level symbols of music notation to modeling higher-level ideas native to one's own language of composition. (Bača et al., 2015, p. 167)

Because of all these features, the composer can use Abjad not only to gradually notate a pre-composed score but also to accomplish complex algorithmic manipulations of any arbitrary musical material, including those randomly created using generative processes. As a simple example of this, the user can, for instance, take any container of music, loop through its leaves, and randomise their pitches by algorithmically modifying the `abjad.Note.written_pitch` property of each individual leaf. This randomisation can be done using arbitrary rules, e.g. by using a user-defined function that gives more weight to certain pitch classes over others. Although manipulations such as these could be implemented directly in the LilyPond source files using its built-in Scheme interpreter (Nienhuys & Nieuwenhuizen, 2003), Abjad's object-oriented modelling of LilyPond's components allows for far more control, flexibility, and clarity. This is partially due to Python's straightforward syntax and powerful external packages, making Abjad an extremely capable tool for algorithmic composers working with note-based music.

## 4.4 High-Level Manipulations in Abjad

As discussed in the previous section, Abjad contains not only classes representing traditional notation concepts such as notes, chords, and dynamics, but also high-level classes and functions that can generate and manipulate material. To illustrate the latter, consider the simple example below in which a note is created with a C4 pitch (numerically represented in Abjad by the number `0`) and a duration of a crotchet (represented by the tuple `(1, 4)`).[6] Using these two variables as instantiation arguments for `abjad.Note`, Abjad creates the note `c'4`, as expected.

```
>>> pitch = 0
>>> duration = (1, 4)
>>> note = abjad.Note(pitch, duration)
```

[6]It is important to note that Abjad is very flexible regarding argument types for its classes and functions. Pitches can be entered as strings, integers, floats, as well as instances of `abjad.NumberedPitch` or `abjad.NamedPitch`; durations can be entered as strings, integers, floats, tuples, and instances of `abjad.Duration`. This is true for most of its classes.

```
>>> abjad.show(note)
```

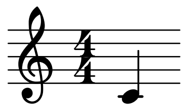

**Figure 4.6:** Example of `abjad.Note`

However, not all tuples of integers can be used to create valid durations for a single note. For instance, attempting to create a note with a duration of `(5, 16)` will raise an exception, as shown below.

```
>>> pitch = 0
>>> duration = (5, 16)
>>> note = abjad.Note(pitch, duration)
abjad.exceptions.AssignabilityError: not assignable duration: Duration(5, 16).
```

This error occurs because this duration is considered 'non-assignable': i.e. even with multiple augmentation dots, this duration cannot be reduced to a single note value. Instead, the composer must manually create the series of tied notes that add up to the total duration of `(5, 16)`. For instance, they can tie a crotchet to a semiquaver, as shown below.

```
>>> pitch = 0
>>> duration_1 = (1, 4)
>>> duration_2 = (1, 16)
>>> note1 = abjad.Note(pitch, duration_1)
>>> note2 = abjad.Note(pitch, duration_2)
>>> abjad.attach(abjad.Tie(), note1)
>>> staff = abjad.Staff([note1, note2])
>>> abjad.show(staff)
```

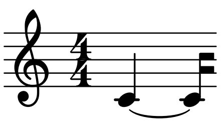

**Figure 4.7:** Manually creating notes with a non-assignable total duration

This is, of course, a very low-level approach since the composer needs to take care of individual leaves and ties. This becomes a problem if, for instance, one randomly generates durations, some of which may be non-assignable. In that case, a naive approach would be to create a function that checks for the assignability of each of those durations; in case they fail this check, another function could be used to break them down into multiple durations, which can then be individually checked for assignability. Evidently, this adds a layer of low-level complexity for the simple task of creating a note (or tied notes) with a specific total length. Thankfully, Abjad comes with higher-level classes and functions that can both generate and manipulate material. One such class is `abjad.LeafMaker`, which takes pitches and durations as input and automatically creates leaves by matching those together while taking care of any non-assignable durations in the process. Using the same variables from the previous example, we can see that `abjad.LeafMaker` creates a tied note with the correct total length, as shown in Figure 4.8.

```
>>> pitch = 0
>>> duration = (5, 16)
>>> maker = abjad.LeafMaker()
>>> leaves = maker(pitch, duration)
>>> staff = abjad.Staff(leaves)
>>> abjad.show(staff)
```

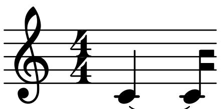

**Figure 4.8:** Example of `abjad.LeafMaker`

`abjad.LeafMaker` can take lists of values as well, as shown in the next example. This is particularly useful when working with generative strategies, in which pitches, durations, and other parameters are randomly generated by stochastic procedures. The output of this example, shown in Figure 4.9, illustrates another issue: although the individual length of each generated note is correct, their rhythm is not ideally notated. While all assignable and non-assignable durations are taken care of, the rhythm is poorly notated as `abjad.LeafMaker` only considers single notes when splitting them into multiple tied leaves, but it does not take the metric context into consideration. This makes the beat structure of the $\frac{4}{4}$ meter very unclear.

```
>>> pitches = [0, 2, 4, 5]
>>> durations = [(3, 16), (5, 16)]
>>> maker = abjad.LeafMaker()
>>> leaves = maker(pitches, durations)
>>> staff = abjad.Staff(leaves)
>>> abjad.show(staff)
```

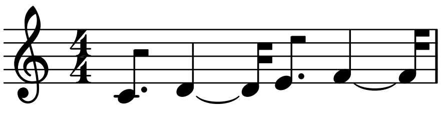

**Figure 4.9:** Output of `abjad.LeafMaker` with multiple pitches and durations

In order to improve its notation, the `staff` container can be mutated by another of Abjad's high-level members, `abjad.Meter.rewrite_meter()`. This is a recursive function that renotates rhythms according to a given meter. Applying this function to the output above creates a vastly more readable score, as shown in Figure 4.10.

```
>>> meter = abjad.Meter((4, 4))
>>> abjad.Meter.rewrite_meter(staff[:], meter)
>>> abjad.show(staff)
```

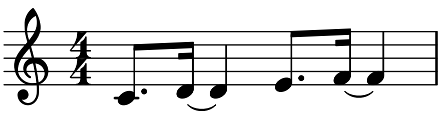

**Figure 4.10:** Example of `abjad.Meter.rewrite_meter()`

By using these higher-level members, the composer can focus on the act of composing the music itself while deferring the process of individual leaf creation and low-level rhythmic manipulations to Abjad. This also allows for the creation of increasingly complex classes, as shown with the members of Auxjad library discussed in Section 4.6 (`abjad.Meter.rewrite_meter()` is used in virtually every single core class of that library). Abjad thus enables me to focus exclusively on structural relationships of material when designing my generative procedures, knowing that the resulting music can be properly and easily notated.

## 4.5 Extending Abjad Through Object-Oriented Programming Principles

Two paramount principles of object-oriented programming are those of 'composition' and 'inheritance', which can be used to create increasingly complex (and thus higher-level) classes out of simpler ones. These concepts are fundamental to object-oriented design as they promote, among other things, clear structural relationships and hierarchies, code reuse, and minimise code redundancy (Weisfeld, 2004, pp. 129–136). A typical design strategy is to create base classes that are general enough to be parents of multiple other classes through what is known as inheritance. These classes may not be intended to be used directly by the user but can help create a clear structural tree of class hierarchies, which can be very useful for building computational models of music notation (Pope, 1996, pp. 56–57).[7] Examples of this in Abjad include `abjad.Leaf`, the base class for `abjad.Note`, `abjad.Chord`, and `abjad.Rest`, and `abjad.Container`, the base class of `abjad.Tuplet`, `abjad.Voice`, `abjad.Staff`, and `abjad.Score`.

Composition is often described using a ‘has-a’ metaphor: classes are said to employ composition when they contain instances of other classes (Weisfeld, 2004, pp. 129–130). An example of this would be a class that models musical notes. At its most basic, a note must have both pitch and duration, both of which could be modelled solely within the note class. However, these two concepts are general enough to be employed in other classes as well and, therefore, are better suited to being defined as their own classes. For instance, chords have multiple pitches, or instrumental ranges can be defined using two pitches, a minimum and a maximum value; durations can be used for notes, but also for chords and rests, as well as for a collection of those score components (measures, for instance). As such, modelling pitch and duration as their own classes allows the designer to define their behaviour in a single location and reuse them at will. Any modification to these classes will also affect all others making use of them.

On the other hand, inheritance uses an ‘is-a’ metaphor: a class inherits all methods and attributes from a parent class (Weisfeld, 2004, pp. 130–133). This is very useful when multiple classes share common characteristics and behaviours. In Abjad, the concepts of tuplet, voice, staff, and score all inherit from a base class named `abjad.Container`. This class is used as a container for score components, such as notes, chords, and rests. Its methods include, for instance, one that appends a new component at the end of the container. Since a class such as `abjad.Tuplet` inherits from `abjad.Container`, it inherits all of its attributes and methods as well; as such, `abjad.Tuplet` also offers the same method for appending new components.

These two notions are fundamentally important when designing a complex system such as the one needed for modelling music representation. They allow the creation of ever more complex classes on top of simpler ones, making them increasingly specific and leading to a web of hierarchies and shared behaviours (Weisfeld, 2004, pp. 129–134). As such, object-oriented programming languages

[7]However, it is important to note that deep class hierarchies, particularly those employing multiple inheritance, can lead to overly complex designs that are very difficult to understand, maintain, and extend. Therefore, a designer needs to strike a good balance when using these techniques to design the parts of a complex and expandable system (Weisfeld, 2004, pp. 179–192; Phillips, 2018, pp. 22–24).

are particularly suitable for representing musical notation (Pope, 1996). They also serve as mechanisms for extending a system, as users can create their own classes that inherit from, or are composed of, classes from other libraries. This enables users to accomplish more than simply create objects from blueprints designed for them. As such, any user of Abjad can create their own classes that inherit from any of Abjad's classes, a technique employed throughout in the Auxjad library. For example, `auxjad.NaturalHarmonic` and `auxjad.ArtificialHarmonic` inherit from `abjad.Note` and `abjad.Chord`, respectively; they model only the necessary note head alterations for each harmonic type while using Abjad's default behaviour of notes and chords.

Child classes will inherit all of the methods and attributes from a parent class. However, it is also possible to overwrite any attributes or methods inherited from the parent class. Programmers can then adjust the behaviour of child classes as necessary, according to their design purpose. As an example, consider the two classes below, named `Base` and `Child`.

```
>>> class Base:
...     def method_1(self):
...         print('Base method 1')
...     def method_2(self):
...         print('Base method 2')
...
>>> class Child(Base):
...     def method_2(self):
...         print('Child method 2')
...     def method_3(self):
...         print('Child method 3')
```

`Child` inherits all methods that are present in `Base`. But since `Child` overwrites `method_2()`, any instance of `Child` invoking `method_2()` will use the method as defined in `Child`, not `Base`:

```
>>> child_object = Child()
>>> child_object.method_1()
Base method 1
>>> child_object.method_2()
Child method 2
>>> child_object.method_3()
Child method 3
```

New methods and attributes can also be assigned to any classes, including those imported from different packages. Consider the example below, in which a function named `method_4()` is first defined and then assigned as a method to `Child`. All instances of `Child`, including those previously created, will now have access this new method:

```
>>> def method_4(self):
...     print('Extension method 4')
...
>>> Child.method_4 = method_4
>>> child_object.method_4()
Extension method 4
```

I have used this technique in Auxjad when adding new methods to some of Abjad's own classes, such as `abjad.Score` and `abjad.TimeSignature`. As a result, simply importing both Abjad and Auxjad will ensure that these new methods are added to Abjad's default classes. As shown in the next section, all mutation and inspection functions of Auxjad are also patched into Abjad's own mutation and inspection modules, i.e. `abjad.mutate` and `abjad.get`. This means that Auxjad's users do not need to recall what mutation or inspection functions belong to each library, as those are available on both namespaces. This type of technique is used by many Abjad users, as exemplified by Evans (2019, p. 13), who writes:

> Occasionally, I have found the need to tweak Abjad's source code in order for it to perform functions that I desire, but more often than this, the composer will find the need to build tools to simplify the process of engraving.

Another important aspect of these libraries is their public nature. Since both Abjad and Auxjad are open-source software whose source code is freely available to everyone, they naturally invite their users to build upon their codebase. As a matter of fact, the Abjad library is itself made out of more than the main package named `abjad`: it also consists of two other extension packages named `abjad-ext-rmakers` and `abjad-ext-nauert`. These are not required for Abjad's basic functionality, and thus users can decide whether or not to install them. Each of them caters to specific techniques for rhythm generation and rhythmic quantisation, respectively. Extensibility is thus a core ideal of Abjad: by allowing as much technological transparency as possible, Abjad ensures a very high potential for being expanded by its users (Treviño, 2013, pp. 30–35).

Besides these extension packages provided by Abjad, many composers working with this library have written and released their own collections of functions and classes. As is the case with Auxjad, these collections seem to have all been created by composers primarily for their own musical needs, a fact that is reflected in their design choices. A notable example of this type of extension is the Consort library written by Josiah Wolf Oberholtzer (2018). This library allows composers to populate complex structures of segmented time spans with arbitrary musical material, among many other included tools (Oberholtzer, 2015, pp. 158–217). Other examples include the Evans library, written by Gregory R. Evans (Evans, 2021; 2019, pp. 54–57), the Calliope library, written by Randall West (West, 2016; Evans, 2019, p. 54), the Tsmakers and Mtools libraries, both written by Ivan Alexander Moscotta (Moscotta, 2019; Evans, 2019, p. 54),[8] and muda, written by Davi Raubach Tuchtenhagen (2021).

Due to its open-source nature and object-oriented design, Abjad enables composers to gradually build their own toolboxes based on the code written by its developers as well as its users. Composers can thus develop a highly customised version of the Abjad package by adding extensions and making modifications to its source code. This flexibility of adjusting the tools to better suit one's musical needs is of great importance to algorithmic composers, as these malleable tools can also suggest specific routes of musical experimentation.

## 4.6 Auxjad: Structure and Members

This section will discuss how the Auxjad library is structured and demonstrate some of its main functionality. This text is not intended as a comprehensive description of the Auxjad API[9] but rather as an overview of its key components; for an in-depth description of each of its members, see the Auxjad documentation page available at `https://gilbertohasnofb.github.io/auxjad-docs/`. Auxjad's code is published under the MIT License and is publicly available at the repository `https://github.com/gilbertohasnofb/auxjad/`. The MIT License is a free software license that grants permission for anyone to adapt, reuse, and distribute code published under it, aligning Auxjad with the core values of open-source software development.

Auxjad is written for Python 3.9 or greater and should be used in conjunction with the latest Abjad release, which is version 3.4.[10] Abjad has experienced an immense amount of development over the past couple of years. Unfortunately, it has also undergone multiple structural and design changes during this period,

[8]Mtools is unfortunately no longer available in the repository cited by Evans (2019, p. 54).

[9]API stands for 'application programming interface'. The API of an object-oriented library describes all its public members as well as the attributes and methods used to interact with them.

[10]As of December 2021.

which invariably led to older user code becoming incompatible with its most recent versions. Given that Auxjad is entirely dependent on Abjad in order to function, I intend to continually maintain Auxjad as to keep it always up-to-date with future releases of Abjad, although this may lead to backward compatibility issues for all user code that employs it (including my own).[11]

As previously demonstrated in Section 4.3, Abjad can be used not only to manually and interactively construct music scores but also to algorithmically generate them. My library encourages composers to focus on the latter, as its members are designed with automatic and generative strategies in mind. This closely reflects my compositional approach, as Auxjad is primarily written as a means of exploring and realising algorithmic strategies for this practice-based research.

Given this practical origin, Auxjad displays a somewhat idiosyncratic nature at times. The majority of this library's core members were written for specific compositions of mine, or at least with certain musical experiments in mind. Despite this, Auxjad has a broad range of applications that are general enough to be useful for many other composers. Therefore, even though my compositional practice is embedded in Auxjad, it can produce open-ended results with significant potential for wider applications beyond my own personal practice.

### 4.6.1 Package Structure

Auxjad is divided into eight subpackages that closely relate to Abjad's own structure. These subpackages are `core`, `get`, `indicators`, `makers`, `mutate`, `score`, `spanners`, and `utilities`. This division in subpackages serves the sole purpose of structuring the library's code since, from the user's point of view, all members of Auxjad other than those in `get` and `mutate` also reside in the `auxjad` namespace and can, thus, be directly accessed using the syntax `auxjad.member`. This type of project structure is similar to how Abjad itself is designed: all of its members other than those in `get` and `mutate` reside in the `abjad` namespace, even though their source code is organised in multiple subpackages. In practical terms, this means that when using a member of Auxjad, such as `auxjad.LeafDynMaker`, the user does not have to remember

[11]Since all previous releases of both Abjad and Auxjad are available online, any older score can still be compiled using the specific versions of these packages that they were written for, as long as these versions are known (one of the reasons as to why including a `requirements.txt` file or a `setup.py` file in all Python projects is considered good practice). The main issue is thus reusing code written for older versions of Abjad, as it will likely not be compatible with newer versions without some rewriting.

what subpackage it belongs to and can simply use the very straightforward syntax:

```
>>> import auxjad
>>>
>>>
>>> maker = auxjad.LeafDynMaker()
```

The naming convention used in these subpackages closely resembles the one used by Abjad itself. Abjad's score component classes, which include leaf classes (such as `abjad.Note` and `abjad.Chord`) as well container classes (such as `abjad.Staff` and `abjad.Score`), reside in the module `score`. Indicator classes, which include classes such as `abjad.Clef`, `abjad.TimeSignature`, and `abjad.Articulation`, are located in the subpackage `indicators`.[12] Spanner functions, such as `abjad.ottava()` and `abjad.piano_pedal()`, reside in the `spanner` module. Classes that create leaves, such as `abjad.LeafMaker`, are located in the `makers` module. Inspection and mutation functions, which are used to inspect and mutate the state of score component objects, reside in two modules named `get` and `mutate`, respectively; these are two cases in which the members of a module or subpackage are not imported directly into the `abjad` namespace and therefore require the syntax `abjad.get.function()` and `abjad.mutate.function()`.

With the above structure in mind, the naming of the subpackages of Auxjad closely reflects the naming choices used in the Abjad project. `indicators`, `makers`, `score`, and `spanners` mostly contain classes that inherit from Abjad's own classes and extend their functionality. All inspection and mutation functions from Auxjad are grouped into subpackages named `get` and `mutate`, respectively, and which require the syntax `auxjad.get.function()` and `auxjad.mutate.function()`. The only two Auxjad subpackages whose names cannot be found in Abjad are `core` and `utilities`. These use names of former subpackages that Abjad used to have and which I decided to keep. Although these subpackages no longer exist in Abjad (and their members have since been moved to different modules), their naming convention still proves

[12]In recent versions, Abjad has shifted from structuring using subpackages in favour of modules that group classes and functions. `indicators` remains one of the few subpackages still in use as of December 2021 but will likely be refactored as a module in a future release. I have decided to continue using only subpackages in Auxjad as they allow functions and classes to each reside in their own files, which are then organised in different directories for each subpackage. In my opinion, this makes the code vastly more readable and the structure of the library substantially clearer.

useful in my library's case. Auxjad's `core` subpackage contains its main classes, i.e. those that implement algorithmic manipulations of containers and which constitute fundamental mental models for some of my compositions; these provide the core functionality of this library. Auxjad's `utilities` contain functions that are neither mutations nor inspections.

When naming specific members of my library, my primary concern was to remain consistent with Abjad's conventions while using names that hint at the member's functionality. In other words, I chose names that would allow users to have an intuitive understanding of what to expect from these classes and functions only by reading them; examples include `auxjad.PitchRandomiser`, `auxjad.Phaser`, and `auxjad.mutate.enforce_time_signature()`, all of which should hint at the type of algorithms they implement. With that being said, some of Auxjad members—most of which are core classes—have somewhat poetic names; these include `auxjad.CartographySelector`, which is named after my series of compositions called *Cartographies*, `auxjad.Fader`, which implements a non-continuous process of masking or unmasking leaves of a container and can be understood through the metaphor of a 'discrete fader', and `auxjad.CrossFader`, which is composed of two `auxjad.Fader`'s operating simultaneously but in different modes, thus creating a 'discrete cross fade' from a material $A$ to a material $B$.

Although these names can initially make the behaviour of these classes less evident to the user, they were all created with these specific compositions in mind and are idiosyncratic enough to make conventional naming too cumbersome. Using the title of a composition of mine and other non-evident composition-oriented metaphors in the name of a class could be criticised as somewhat inadequate from a design point of view. However, from my perspective as a composer, I would argue that this creates an environment where these names correspond to how I conceptualise my own music and its processes.

The following subsections will focus on some of the individual members that make up Auxjad and demonstrate some of their functionalities. It is important to note that all examples in the following subsections assume that both Abjad and Auxjad have been imported into Python 3.9 using the lines below.

```
>>> import abjad
>>> import auxjad
```

### 4.6.2 The `core` Subpackage

The `core` subpackage contains the main classes of Auxjad. These are implementations of algorithmic processes that transform containers using composition-oriented processes. Many of these classes constitute fundamental mental models used throughout the accompanying portfolio. By including classes consisting of high-level encapsulations of algorithmic manipulations that can be applied to any music material, this library invites users to interact with its members as building blocks for algorithmic music processes.

The classes in this subpackage can be further divided into two categories: there are selectors, which are used to randomly select elements from lists, and there are classes that algorithmically manipulate an `abjad.Container` (or child class), returning an `abjad.Selection` for each iteration of the processes they implement.

#### 4.6.2.1 Selectors

Selectors form a category of classes that can randomly select elements from input lists. Auxjad currently has three selector classes, `auxjad.CartographySelector`, `auxjad.TenneySelector`, and `auxjad.GeneticAlgorithm`. The first two can be instantiated using only the input list from which the selection takes place, while the latter requires further arguments (as shown further below). For instance, `auxjad.CartographySelector` can be instantiated as:

```
>>> selector = auxjad.CartographySelector(['A', 'B', 'C', 'D', 'E', 'F'])
>>> selector.contents
['A', 'B', 'C', 'D', 'E', 'F']
```

Calling a selector will return a randomly chosen element from that list. These elements can be any Python object, including Abjad's native types.

```
>>> selector()
'A'
>>> selector()
'D'
```

`auxjad.CartographySelector` applies different weights to each of its elements using a fixed decay rate associated with their indices. The decay rate represents the ratio of probabilities of an index to its preceding one. For instance, if the decay rate is set to 0.75 (which is its default value as well as the value used in my *Cartographies* series of compositions), the probability of the element in index 1 of the input list being selected is 75% the probability of the element in index 0, and the probability of the element in index 2 is 56.25% (i.e. $0.75^2$) the probability of the element in index 0. The probability $P(n)$ of the $n$-th element can thus be expressed as a relation to the probability of another element $k$ indexes apart using the formula:

$$P(n) = (3/4)^k \times P(n-k)$$

Individual weights are thus associated with specific indices of the input list but not with its content itself.

```
>>> selector.weights
[1.0, 0.75, 0.5625, 0.421875, 0.31640625, 0.2373046875]
```

Manipulations of the positions of certain elements will thus result in a change of their effective weights, even though the probability value for a given index will remain the same. `auxjad.CartographySelector` comes with many of such transformation methods. For instance, the method `drop_first_and_append()` discards the first element of the contents, shifts all others leftwards, and then appends the new element to the last index. Some other methods can rotate, shuffle, and mirror elements. Applications of these will be further explored in Section 5.1.2 which discusses the *Cartographies* series in detail.

`auxjad.TenneySelector` is an implementation of the 'dissonant counterpoint' algorithm by James Tenney, based on the description of this algorithm by Polansky, Barnett, & Winter (2011). This class can be used to randomly select elements from an input list, giving weight to elements proportionally to how long they have not been selected in previous iterations. In other words, Tenney's algorithm uses feedback to lower the weight of recently selected elements, prioritising those that have not recently occurred. One of the resulting properties of this algorithm is that the same element cannot be chosen twice in a row.

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'])
>>> selector.probabilities
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> result = ''
>>> for _ in range(30):
...     result += selector()
>>> result
DAFDEFEBCECDAFBDECFDEACDFBCEDA
>>> selector.probabilities
[0.0, 4.0, 3.0, 1.0, 2.0, 5.0]
```

The probability associated with each element is thus dynamic, as shown in the code above. Each time an element is not selected, its probability of being selected next increases. Each time it is selected, its probability is set to 0.0. However, this increase of chance when an element is not selected does not need to be linear: a property named `curvature` (with a default value of 1.0) can be used to adjust how fast or slow the probabilities of non-selected elements grow over each iteration. When setting it to a value between 0.0 and 1.0 (non-inclusive on both extremes), the growth is said to be 'concave'; as such, the chance of an element that has not been selected in consecutive iterations will grow at increasingly lower rates as the number of iterations increase. Setting it to values larger than 1.0 will make the growth 'convex', and the chance of non-selected elements will grow at increasingly higher rates the longer they are not selected.

The final selector, `auxjad.GeneticAlgorithm`, is a simple implementation of a genetic algorithm, a type of stochastic algorithm inspired by Darwinian evolution (Mirjalili, 2019). Genetic algorithms generate populations of data structures through a continuous process of random generation and evaluation. Individuals are evaluated by a 'fitness function' (the analogue to the Darwinian environment) that gives a score to each individual in a population. High-scoring individuals are then chosen to be the next generation's parents: their data structures (representing their genes) are then combined through the so-called 'crossover process', in which half of one parent's data is combined with the other half of the second parent. At this point, each individual can also be affected by random mutations.

This process is repeated any specified number of times. Despite genetic algorithms typically starting with a completely random population, the selection of the fittest parents coupled with the introduction of random mutations will, on average, result in new populations of individuals that are increasingly better fit for the given environment.

My implementation of the genetic algorithm takes a list of genes, which can be any Python object, and a 'target' list that represents the best possible combination of genes for the given environment. This target is used when evaluating individuals: each gene of an individual is compared with the corresponding gene of the target. The evaluation considers how close these two genes are in the gene list, with identical genes are scored the highest.

This is a somewhat idiosyncratic implementation of a genetic algorithm. In particular, this type of algorithm tends to be used with more open-ended evaluation functions than a simple comparison to a target. This is because genetic algorithms are most often used to heuristically find near-optimal solutions for a given problem. However, as a composer, my interest in them lies in the process itself rather than the results it outputs; as such, defining the environment through a list of ideal genes allows me to control the overall direction the algorithm will take.[13] This algorithm is the primary mental model employed in my composition *methinks it is like a weasel* (2021), for violin, cello, and piano, which explores how a genetic algorithm can slowly transform a population of musical measures within a given environment.

The listing below demonstrates how this class can be used:

```
>>> ga = auxjad.GeneticAlgorithm(
...     target=['A', 'B', 'C', 'D', 'E'],
...     genes=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
...     population_size=4,
...     select_n_parents=2,
... )
>>> ga()
>>> ga.population
[['A', 'J', 'J', 'B', 'H'],
 ['D', 'A', 'E', 'A', 'F'],
 ['F', 'F', 'A', 'F', 'F'],
 ['F', 'F', 'E', 'J', 'C'],
 ]
>>> ga.scores
[0.209603072,
 0.0912,
 0.05638400000000001,
 0.016396800000000003,
 ]
 >>> ga.fittest_individual
 ['A', 'J', 'J', 'B', 'H']
```

At each call to an instance of this class, the processes of selection, crossover, and mutation are carried out for one further iteration. As expected, each new generation will become increasingly fitter in relation to the target. The listing below prints out the fittest individual from ten consecutive generations (each containing 50 individuals). In this particular case, both the target and the number of genes are relatively small, and thus the genetic algorithm is able to

[13]My implementation could be considered as the combination of a genetic algorithm with the 'Goal Seeker' design pattern, as defined by Woodbury (2010, pp. 269–274). A Goal Seeker is a feedback model in which the output serves as the input for the next iteration of its process; the output is evaluated against a target value, with the process ending once a threshold of similarity is reached. My implementation works similarly to this but borrows the crossover, mutation, and selection processes from a more traditional genetic algorithm implementation.

quickly generate a near-optimal individual at its sixth iteration, with a single gene differing from the target.

```
>>> ga = auxjad.GeneticAlgorithm(
...     target=['A', 'B', 'C', 'D', 'E'],
...     genes=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
...     population_size=50,
... )
>>> for _ in range(10):
...     ga()
...     print(ga.fittest_individual, ga.fittest_individual_score)
['A', 'J', 'B', 'E', 'E'] 0.480000512
['A', 'H', 'C', 'D', 'J'] 0.6000768
['A', 'C', 'D', 'D', 'E'] 0.6799999999999999
['A', 'C', 'D', 'D', 'E'] 0.6799999999999999
['A', 'C', 'D', 'D', 'E'] 0.6799999999999999
['A', 'C', 'C', 'D', 'E'] 0.8400000000000001
['A', 'C', 'C', 'D', 'E'] 0.8400000000000001
['A', 'C', 'C', 'D', 'E'] 0.8400000000000001
['A', 'C', 'C', 'D', 'E'] 0.8400000000000001
['A', 'C', 'C', 'D', 'E'] 0.8400000000000001
```

#### 4.6.2.2 Loopers and Phaser

Auxjad has four classes that can shift musical material horizontally. They include three loopers and one phaser. My accompanying portfolio makes extensive use of these types of loopers; `auxjad.WindowLooper` and `auxjad.LeafLooper` are used in many of my *Cartographies*, while `auxjad.ListLooper` has been used in *and thereafter they shape us* (2019).

`auxjad.WindowLooper` implements the mental model of a looping window: this algorithm slices and outputs a window of arbitrary duration from an input musical material. After outputting this slice of material, the algorithm moves the window forwards by a certain duration, which, in my work, is usually set to a fractional value of the total window size. As a result, the process moves forwards relatively slowly and often outputs notes and chords that were already present in previous iterations. Depending on these parameters, the algorithm can produce music that sounds nearly identical on a microscale but which has forward momentum on a macroscale.

`auxjad.WindowLooper` takes an `abjad.Container` as well as window and step sizes as input. At each call, the window is moved forward by the given step size and an `abjad.Selection` of the current window is returned. Consider the material shown in Figure 4.11, which will be used as input for the looping window process.

```
>>> container = abjad.Staff(
...     r"c'4\p( d'16-.) r8. e'4\mf( ~ e'8 f'8 g'2 a'8) r4."
... )
>>> abjad.show(container)
```

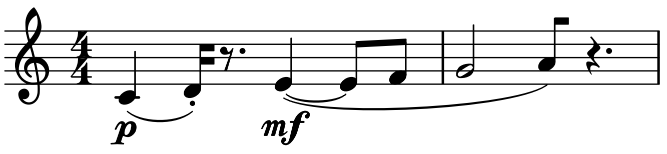

**Figure 4.11:** Material used as input for `auxjad.WindowLooper` and `auxjad.LeafLooper`

The following example uses the material above to initialise `auxjad.WindowLooper` together with a window size of three crochets and a step size of a semiquaver. The method `output_n()` will be used to output four iterations of the looping process. The result of the code below is shown in Figure 4.12.

```
>>> looper = auxjad.WindowLooper(container,
...                              window_size=(3, 4),
...                              step_size=(1, 16),
...                              )
>>> notes = looper.output_n(4)
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

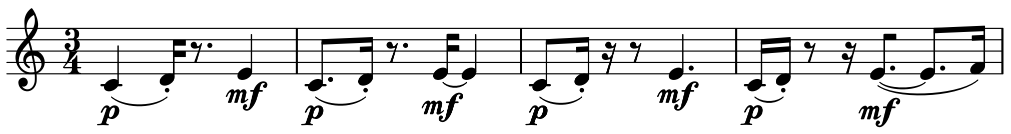

**Figure 4.12:** Output of `auxjad.WindowLooper`

Since all its properties can be overwritten during the program's execution, both window and step sizes can be set to different values after instantiation. This class also has probabilistic options: the number of steps taken at each iteration can be set to a range instead of a fixed value, and the direction of movement can be set to a probabilistic value using a property that controls the overall forward bias of the window.

`auxjad.LeafLooper` is a similar class to `auxjad.WindowLooper` in which it takes an `abjad.Container` as input and outputs slices of it at each call. However, instead of using a duration for the window size, its window is given by a certain number of musical elements (known as ‘logical ties’ in Abjad).[14] Thus its window length can vary in duration at each iteration since it depends on the sum of the individual durations of its elements.

Initialising this class using the container previously shown in Figure 4.11 and a window size of three logical ties will result in the following output after four iterations:

```
>>> looper = auxjad.LeafLooper(container,
...                            window_size=3,
...                            )
>>> notes = looper.output_n(4)
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

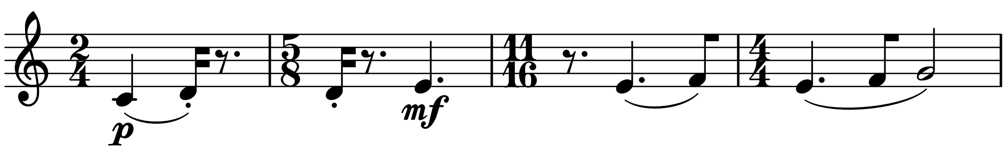

**Figure 4.13:** Output of `auxjad.LeafLooper`

The final looper class is `auxjad.ListLooper`. Similarly to `auxjad.LeafLooper`, an integer number determines its window size. However, instead of taking an `abjad.Container` as input as the two previous classes did, it simply takes a list of any Python objects. For instance, if the initial list consists of $[A, B, C, D, E, F]$ (where each letter represents an element of an arbitrary type) and the looping window is set to length 3, its output would be:

```
A B C B C D C D E D E F E F F
```

This can be better visualised by displaying each iteration of the process in a separate line:

```
A B C
  B C D
    C D E
      D E F
        E F
          F
```

Each call to an instance of `auxjad.ListLooper` will output a slice of the input list containing the number of elements set by `window_size`.

[14]In Abjad, a logical tie is a group of leaves that belong to the same note, chord, or rest. It can consist of a single leaf (a single note, chord, or rest), or multiple tied notes or chords.

```
>>> input_list = ['A', 'B', 'C', 'D', 'E', 'F']
>>> looper = auxjad.ListLooper(input_list, window_size=3)
>>> looper()
['A', 'B', 'C']
>>> looper()
['B', 'C', 'D']
>>> looper()
['C', 'D', 'E']
```

Since the input list for `auxjad.ListLooper` can have any type of element, it can also use any Abjad objects.[15] Thus the looping window mental model can be applied to arbitrary collections of complex musical cells.

Similarly to the previous two classes, both the window length and its step size can be dynamically modified at any point after their instantiation using the `window_size` and `step_size` properties. Likewise, this class also share other properties for controlling parameters such as forward bias, initial head position, chance of output repetition, and the maximum number of steps allowed per iteration, among others.

`auxjad.Phaser` implements a simple phasing algorithm. An `abjad.Container` is phased by a `step_size` at each iteration, with the result returned as an `abjad.Selection`. Consider the container shown in Figure 4.14, which will be used as input for a phaser further below.

```
>>> container = abjad.Staff(
...     r"c'2(\p\< d'4. e'8\f f'4\p\> g'2 a'4\pp)"
... )
>>> abjad.show(container)
```

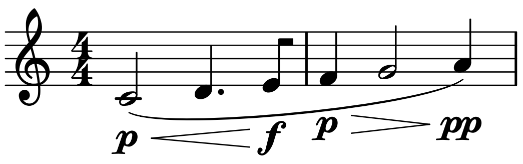

**Figure 4.14:** Material used as input for `auxjad.Phaser`

Applying it to a phaser with a step size of a semiquaver will result in the output of Figure 4.15 after five iterations.

```
>>> phaser = auxjad.Phaser(container, step_size=(1, 16))
>>> notes = phaser.output_n(5)
```

[15]All of the looper classes employ `copy.deepcopy()` when returning windows, so the output of repeated elements will not conflict with Abjad's exclusive membership requirement.

```
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

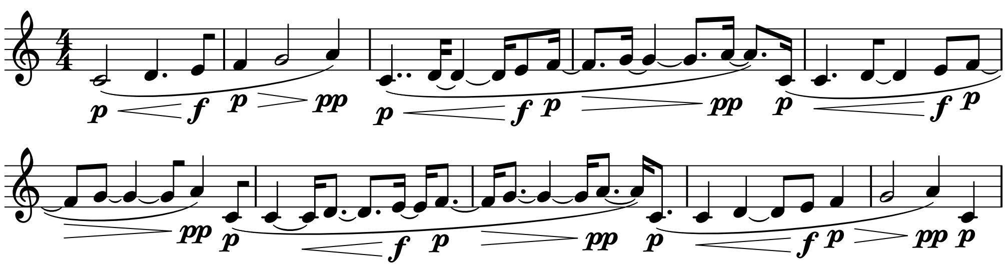

**Figure 4.15:** Output of `auxjad.Phaser`

### 4.6.2.3 Randomisers

Auxjad has three classes that can output randomised versions of an input container. The simplest one is `auxjad.PitchRandomiser`, which takes an `abjad.Container` and a list of pitches (or, alternatively, a single `abjad.PitchSegment`) and substitutes the pitches in the container with random ones from the pitch list. By default, it uses a random distribution with equal weights when selecting pitches, but the property `weights` can be used to give more probabilistic weight to certain pitches over others. If the property `use_tenney_selector` is set to `True`, this class will use an instance of `auxjad.TenneySelector` for the pitch selection instead of a regular random selector with fixed weights.

Consider the material shown in Figure 4.16, which will be used as input in the next example.

```
>>> container = abjad.Container(
...     r"<c' e' g'>8.(\p d'4) r8 r8. e'16-. <f' a'>8.--"
... )
>>> abjad.show(container)
```

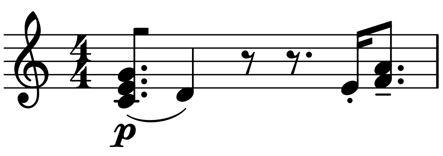

**Figure 4.16:** Material used as input for `auxjad.PitchRandomiser` and `auxjad.Shuffler`

Using the container above as input for the randomiser together with the pitch segment `g af cs' d' e' fs' c'' df'' a'' bf''` resulted in the output of Figure 4.17 after three iterations.

```
>>> pitches = abjad.PitchSegment(r"g af cs' d' e' fs' c'' df'' a'' bf''")
>>> randomiser = auxjad.PitchRandomiser(container,
...                                      pitches,
...                                      )
>>> notes = randomiser.output_n(3)
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

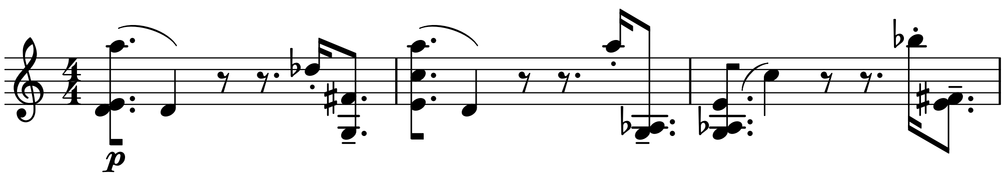

**Figure 4.17:** Output of `auxjad.PitchRandomiser`

`auxjad.Shuffler` can be used to output copies of an input container with shuffled logical ties. It also has a second mode in which only pitches are shuffled while the original rhythmic structure remains intact.

Below is an example of the result obtained after three iterations of this shuffler when initialised using the same container, as shown in Figure 4.16.

```
>>> shuffler = auxjad.Shuffler(container)
>>> notes = shuffler.shuffle_n(3)
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

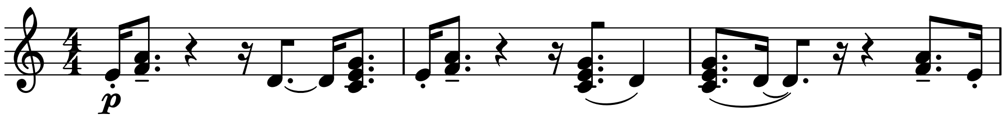

**Figure 4.18:** Output of `auxjad.Shuffler`

The last randomiser, `auxjad.Hocketer`, can be used to create hockets; that is, it will randomly distribute logical ties of an input container into a given number of staves. It has many properties to control the final result, such as the number of voices (given by `n_voices`), the number of times each logical tie is processed (given by the property `k`), and a property that ensures that each logical tie is present in $k$ number of voices (given by the property `force_k_voices`). Unlike other classes which return `abjad.Selection`'s, `auxjad.Hocketer` will return a `tuple` of `abjad.Selection`'s, making the process of assigning them to an `abjad.Score` or multiple `abjad.Staff`'s very straightforward. It is also possible to retrieve individual selections for each voice via indexing.

The container shown in Figure 4.19 will be used as input for the hocketer in the next example.

```
>>> container = abjad.Container(
...     r"c'2-.\p\< d'2-.\f\> e'1 f'2.\pp\< g'4--\p "
...     r"a'2\ff\> b'2\p\> ~ b'2 c''2\!"
... )
>>> abjad.show(container)
```

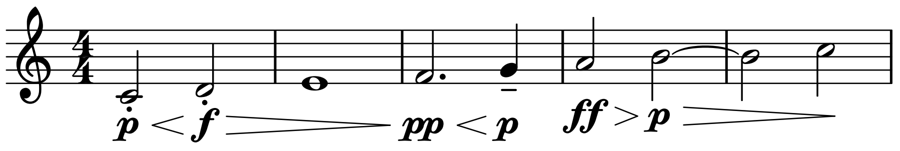

**Figure 4.19:** Material used as input for `auxjad.Hocketer`

The hocketer below is initialised with the container shown above as well as with three voices. Each logical tie will be processed twice (using $k = 2$), and setting `force_k_voices` to `True` will ensure that each logical tie will appear in exactly two voices. The results are shown in Figure 4.20.

```
>>> hocketer = auxjad.Hocketer(container,
...                            n_voices=3,
...                            k=2,
...                            force_k_voices=True,
...                            )
>>> music = hocketer()
>>> score = abjad.Score()
>>> for selection in music:
...     score.append(abjad.Staff(selection))
>>> abjad.show(score)
```

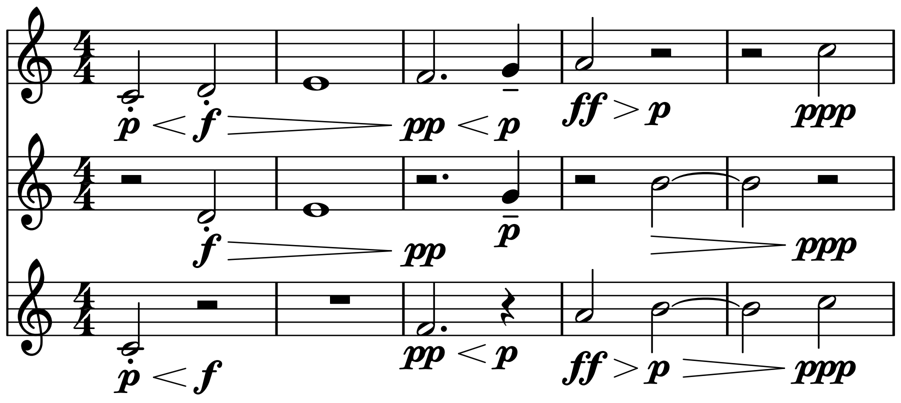

**Figure 4.20:** Output of `auxjad.Hocketer`

#### 4.6.2.4 Faders

`auxjad.Fader` is an implementation of the 'fader' mental model, which applies a process of gradually masking or unmasking notes of an input container so that its notes 'fade in' from or 'fade out' into silence. It can be framed by the metaphor of a 'discrete fader' that controls the density of notes of a musical cell. The mode of the fader (either 'in' or 'out') is given by the property `mode`, and each call to an instance of this class will alter one element of the mask before outputting the results. The mask can be overridden at any point using the property `mask`. In the case of chords, individual notes are added or removed independently.

For the following two examples, consider the container shown in Figure 4.21.

```
>>> container = abjad.Container(
...     r"<c' e'>4 ~ <c' e'>16 d'8. <gs e'>8 <bf f' a'>8 ~ <bf f' a'>4"
... )
>>> abjad.show(container)
```

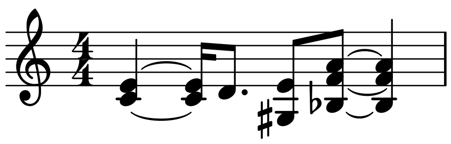

**Figure 4.21:** Material used as input for `auxjad.Fader`

The code below instantiates `abjad.Fader` with `mode` set to `'out'` and uses the method `output_all()` to execute and output the entire fading process. It results in the score shown in Figure 4.22.

```
>>> fader = auxjad.Fader(container, mode='out')
>>> staff = abjad.Staff(fader.output_all())
>>> abjad.show(staff)
```

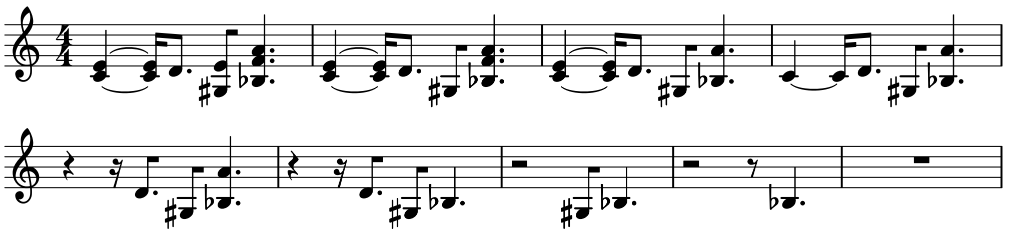

**Figure 4.22:** Output of `auxjad.Fader` with `mode='out'`

Using the same material from the preceding example but setting `mode` to `'in'` will result in the score shown in Figure 4.23.

```
>>> fader = auxjad.Fader(container, mode='in')
>>> staff = abjad.Staff(fader.output_all())
>>> abjad.show(staff)
```

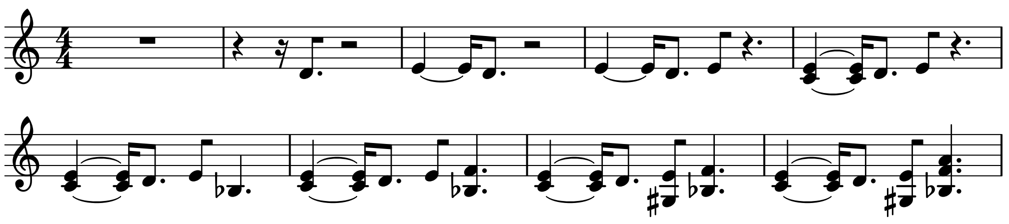

**Figure 4.23:** Output of `auxjad.Fader` with `mode='in'`

`auxjad.CrossFader` is a good example of the object-oriented principle of composition, previously discussed in Section 4.5, as this class is composed of two `auxjad.Fader`'s of opposing modes. My composition *adrift* (2020), for two pianos, is based primarily on this mental model. In the context of that piece, a crossfader employs two of the previously discussed discrete faders, each on different musical layers. These layers are processed in opposite directions, i.e. one is fading out while the other fades in. At each iteration of this process, one of the two faders is selected to be called while the other remains unchanged. Setting the property `fade_in_first` to `True` ensures that a note from the layer starting with silence will necessarily fade in before any note from the other layer fades out. Similarly, `fade_out_last` ensures that the last step of the process will necessarily remove a note from the layer fading out.

The following example shows `auxjad.CrossFader` in use. Figure 4.24 shows the material that will be faded out, while Figure 4.25 shows the material that will fade in.

```
>>> fade_out_container = abjad.Container(r"c'4.\p( e'8--\f ~ e'2)")
>>> abjad.show(fade_out_container)
```

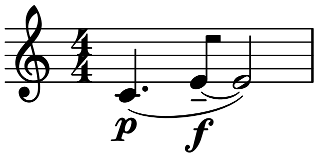

**Figure 4.24:** Material to be faded out by `auxjad.CrossFader`

```
>>> fade_in_container = abjad.Container(
...     r"\times 2/3 {f'4-.\pp r4 d'4->\f ~ } d'2"
... )
>>> abjad.show(fade_in_container)
```

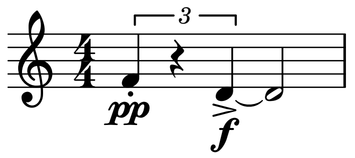

**Figure 4.25:** Material to be faded in by `auxjad.CrossFader`

Using the two containers above, the result of the entire crossfading process is shown in Figure 4.26.

```
>>> fader = auxjad.CrossFader(fade_out_container,
...                           fade_in_container,
...                           fade_in_first=True,
...                           fade_out_last=True,
...                           )
>>> selection_a, selection_b = fader.output_all()
>>> score = abjad.Score([
...     abjad.Staff(selection_a),
...     abjad.Staff(selection_b),
... ])
>>> abjad.show(score)
```

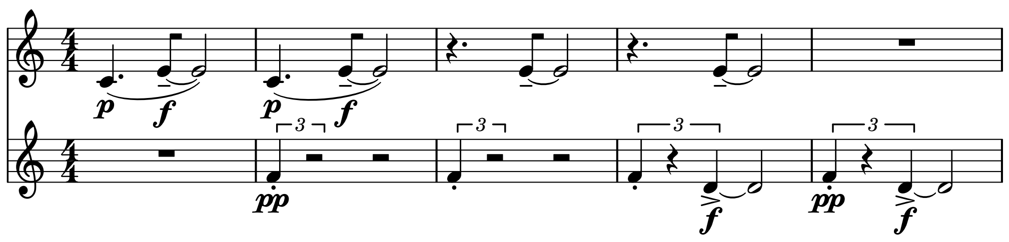

**Figure 4.26:** Output of `auxjad.CrossFader`

### 4.6.3 The get Subpackage

The `get` subpackage contains several inspection functions that can retrieve information about an object's current state. Similar to Abjad's own inspections, these should be used with the syntax `auxjad.get.function()`.

When a user imports `auxjad`, all inspection and mutation functions of Auxjad are automatically added as extension functions to `abjad.get` and `abjad.mutate`, so it is possible to invoke them from the `abjad` namespace as well as from `auxjad`. For instance, the function `underfull_duration()` is an inspection from Auxjad that can be invoked from either `auxjad` and `abjad` namespaces once Auxjad is imported, as shown by the equivalent lines 5 and 8 in the listing below:

```
1 >>> import abjad
```

```
>>> import auxjad
>>>
>>>
>>> container = abjad.Container(r"c'4 d'4 e'4")
>>> # invoking underfull_duration() from auxjad's namespace
>>> auxjad.get.underfull_duration(container[:])
1/4
>>> # invoking underfull_duration() from abjad's namespace
>>> abjad.get.underfull_duration(container[:])
1/4
```

The main reason for this decision is to ensure that end-users do not have to memorise which inspection and mutations were introduced by Auxjad and which already belonged to Abjad.

The function `auxjad.get.selections_are_identical()` compares the contents of two selections and evaluates whether they are identical. By default, it compares all the components of these selections, including not only leaves but also any subcontainers they may have, such as tuplets and grace note containers.

```
>>> container1 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> container2 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> container3 = abjad.Staff(r"r1 c'2 d'2")
>>> selections = [container1[:], container2[:]]
>>> auxjad.get.selections_are_identical(selections)
True
>>> selections = [container1[:], container3[:]]
>>> auxjad.get.selections_are_identical(selections)
False
```

Indicators can also be included by setting the argument `include_indicators` to `True`.

```
>>> container1 = abjad.Staff(r"c'4\pp d'4 e'4-. f'4 <g' a'>2-> r2")
>>> container2 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> selections = [container1[:], container2[:]]
>>> auxjad.get.selections_are_identical(
...     selections,
...     include_indicators=True,
... )
False
```

The function `auxjad.get.leaves_are_tieable()` compares the pitch content of two leaves in order to evaluate whether they can be tied or not.

```
>>> leaf1 = abjad.Note(r"c'2.")
>>> leaf2 = abjad.Note(r"c'16")
>>> leaf3 = abjad.Note(r"f'''16")
>>> auxjad.get.leaves_are_tieable([leaf1, leaf2])
True
>>> auxjad.get.leaves_are_tieable([leaf1, leaf3])
False
```

By default, the function will return `True` if any single pitch from the first leaf can be tied to the second leaf. If the argument `only_identical_pitches` is set to `True`, the function will now only consider identical pitch content as tieable, and leaves that only partially match will now return `False`.

```
>>> chord1 = abjad.Chord(r"<c' e' g'>4")
>>> chord2 = abjad.Chord(r"<c' e' g' bf'>16")
>>> auxjad.get.leaves_are_tieable([chord1, chord2])
True
>>> auxjad.get.leaves_are_tieable([chord1, chord2],
...                               only_identical_pitches=True,
...                               )
False
```

`auxjad.get.selection_is_full()` simply evaluates whether the last bar of a container is completely filled or not.

```
>>> container1 = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"c'4 d'4 e'4")
>>> container3 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4")
>>> container4 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4 d'4 e'4 f'4")
>>> auxjad.get.selection_is_full(container1[:])
True
>>> auxjad.get.selection_is_full(container2[:])
False
>>> auxjad.get.selection_is_full(container3[:])
False
>>> auxjad.get.selection_is_full(container4[:])
True
```

Not all inspections return boolean values like the ones above. `auxjad.get.underfull_duration()`, which is closely related to the function `auxjad.get.selection_is_full()` shown above, will return an `abjad.Duration` with the precise missing duration of the last bar of a container.

```
>>> container1 = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"c'4 d'4 e'4")
>>> container3 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4")
>>> container4 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4 d'4 e'4 f'4")
>>> auxjad.get.underfull_duration(container1[:])
0
>>> auxjad.get.underfull_duration(container2[:])
1/4
>>> auxjad.get.underfull_duration(container3[:])
3/4
>>> auxjad.get.underfull_duration(container4[:])
0
```

`auxjad.get.time_signature_list()` will return a list with all time signatures of a container, one per measure. When an initial time signature is not present, it assumes a default value of $\frac{4}{4}$, since this is the fallback time signature that LilyPond uses. This behaviour can be disabled by setting the `implicit_common_time` argument to `False`.

```
>>> container = abjad.Container(r"\time 3/4 c'2. \time 4/4 d'1 e'1")
>>> time_signatures = auxjad.get.time_signature_list(container)
>>> time_signatures
[TimeSignature((3, 4)), TimeSignature((4, 4)), None]
```

`auxjad.get.virtual_fundamental()` is inspired by the so-called 'virtual pitch algorithm' formulated by Terhardt (1979), although the former implements a far simpler algorithm. While Terhardt's algorithm can take complex tone signals as input and uses their absolute frequencies in its calculations, Auxjad's virtual fundamental algorithm uses twelve-tone equal temperament to approximate the overtone series. It returns the highest common fundamental shared by all pitches in an `abjad.PitchSegment` or `abjad.Chord`.

```
>>> pitches = abjad.PitchSegment(r"c'' cs'' d'' ef'' e'' fs''")
>>> auxjad.get.virtual_fundamental(pitches)
d,,
>>> chord = abjad.Chord(r"<c'' cs'' d'' ef'' e'' fs''>4")
>>> auxjad.get.virtual_fundamental(chord)
d,,
```

### 4.6.4 The `mutate` Subpackage

The `mutate` subpackage contains functions that mutate the state of an object (in this case, either an `auxjad.Selection` or an `auxjad.Container`) in place and have no return value. These are the most numerous functions in this library, and so this overview will focus on a small selection of them. Similarly to `auxjad.get`, all functions in `auxjad.mutate` are added as extension functions to `abjad.mutate`.

Three of these functions can be used to move indicators and spanners of a selection into better positions. `auxjad.mutate.reposition_dynamics()` will handle dynamics as well as hairpins, adjusting them when rests are present. `auxjad.mutate.reposition_slurs()` will handle unterminated slurs as well as any slurs that cross over rests. Lastly, `auxjad.mutate.reposition_clefs()` will shift clefs from rests to the next pitched leaf. These are extensively used internally in Auxjad's core classes since many of these convert notes into rests or shift the start or endpoints of spanners. Figure 4.27 shows the material before these three functions are applied, and Figure 4.28 shows the resulting output.

```
>>> staff = abjad.Staff(
...     r"c'4.\p( d'8 r2\f\> \clef bass r8 e8 ~ e4 f4) g8-.\pp a8-."
... )
>>> abjad.show(staff)
```

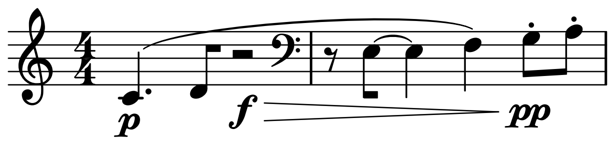
**Figure 4.27:** Material used for the reposition mutation functions

```
>>> auxjad.mutate.reposition_dynamics(staff[:])
>>> auxjad.mutate.reposition_slurs(staff[:])
>>> auxjad.mutate.reposition_clefs(staff[:])
>>> abjad.show(staff)
```

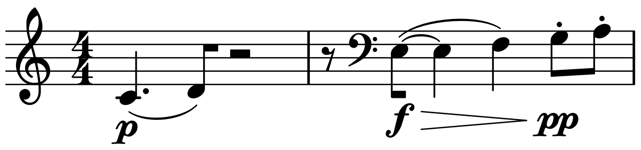
**Figure 4.28:** Container after the reposition mutation functions

Other related functions are `auxjad.mutate.remove_repeated_dynamics()` and `auxjad.mutate.remove_repeated_time_signatures()`, which respectively remove dynamics and time signatures when they are equal to the effective ones of the preceding leaf.

The mutation `auxjad.mutate.enforce_time_signature()` takes either a single time signature or a list of time signatures and enforces it into a container, splitting leaves accordingly and rewriting the rhythmic notation. This is another extremely useful mutation for improving the readability of a score and was utilised extensively throughout the compositions of the accompanying portfolio.

```
>>> staff = abjad.Staff(r"c'1 d'1 e'1 f'1")
>>> abjad.show(staff)
```

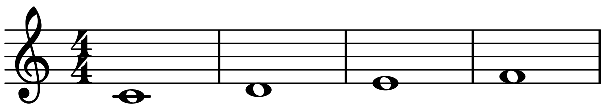

**Figure 4.29:** Container before applying `auxjad.mutate.enforce_time_signature()`

```
>>> time_signatures = [(2, 4), None, None, (3, 4), None, (4, 4)]
>>> auxjad.mutate.enforce_time_signature(staff, time_signatures)
>>> abjad.show(staff)
```

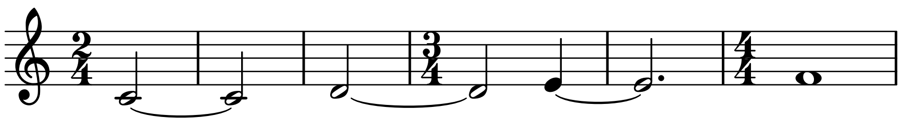

**Figure 4.30:** Container after applying `auxjad.mutate.enforce_time_signature()`

`auxjad.mutate.respell_augmented_unisons()` improves pitch spellings of chords. By default, Abjad spells notes with a fixed pattern of accidentals, using the following list: C♯, E♭, F♯, A♭, and B♭. This works well for most non-tonal music such as my own but can become problematic with some chords, particularly when minor seconds are present. With this fixed spelling, some minor seconds are spelt as augmented unisons. This function looks for those cases and adjusts them. Figure 4.31 shows a comparison between Abjad's default spelling when using numbered pitches (upper staff) and the same staff processed by `auxjad.mutate.respell_augmented_unisons()` (lower staff).

```
>>> pitches = [(0, 1), (8, 9, 12), (0, 4, 5, 6), (-1, 10)]
>>> durations = [(1, 8), (3, 8), (7, 16), (1, 16)]
>>> maker = abjad.LeafMaker()
>>> chords = maker(pitches, durations)
>>> staff1 = abjad.Staff(chords)
>>> staff2 = abjad.mutate.copy(staff1)
>>> auxjad.mutate.respell_augmented_unisons(staff2[:],
...                                         include_multiples=True,
...                                         )
>>> literal = abjad.LilyPondLiteral(r'\accidentalStyle dodecaphonic')
>>> abjad.attach(literal, staff1)
>>> abjad.attach(literal, staff2)
>>> score = abjad.Score([staff1, staff2])
>>> abjad.show(score)
```

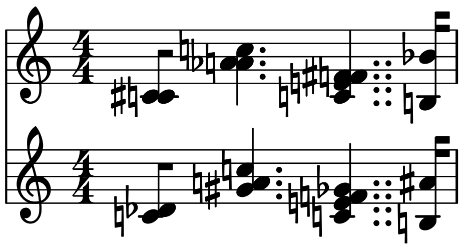

**Figure 4.31:** Comparison between Abjad's default pitch spelling and the output of `auxjad.mutate.respell_augmented_unisons()`

Auxjad has two small mutations that can extend notes; these are `auxjad.mutate.sustain_notes()` and `auxjad.mutate.extend_notes()`. The first extends the duration of notes and chords followed by rests until another note or chord is reached. The second works similarly but takes a specific maximum duration for each note. They are both very useful in conjunction with procedures that generate only attack points, which they can then convert into longer durations. I used the former in *adrift*, which makes extensive use of crossfaders that generates a multitude of rests. Removing these rests made the rhythms easier to read and my musical intentions clearer. The latter function was used when writing *follow the well-worn path in the grass* (2021). This piece makes use of a hocketer that distributes notes among several staves of a six-instrument chamber ensemble. `auxjad.mutate.extend_notes()` was then used to elongate those distributed notes up to a specific duration, which blurred the texture further while still allowing for breathing points for the wind instruments.

Figure 4.32 shows a measure with short notes. Applying `auxjad.mutate.extend_notes()` with the duration of a crotchet results in the measure shown in Figure 4.33 while applying `auxjad.mutate.sustain_notes()` results in the measure shown in Figure 4.34.

```
>>> staff = abjad.Staff(r"c'16 r2... d'8 r2.. e'8. r16 r2. f'4 r2.")
>>> abjad.show(staff)
```

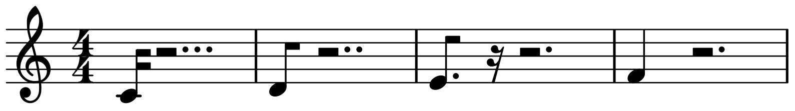

**Figure 4.32:** Container before applying `auxjad.mutate.extend_notes()`

```
>>> auxjad.mutate.extend_notes(staff, abjad.Duration((1, 4)))
>>> abjad.show(staff)
```

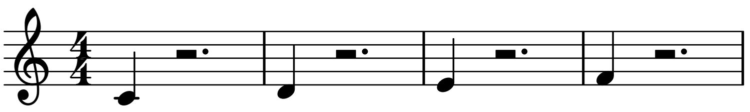

**Figure 4.33:** Container after applying `auxjad.mutate.extend_notes()`

```
>>> auxjad.mutate.sustain_notes(staff)
>>> abjad.show(staff)
```

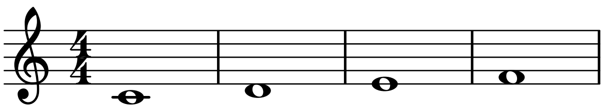

**Figure 4.34:** Container after applying `auxjad.mutate.sustain_notes()`

The function `auxjad.mutate.prettify_rewrite_meter()` is perhaps one of the most peculiar mutations in this library. Its purpose is to fuse pitched leaves according to specific hard-coded rules, improving the results of `abjad.Meter.rewrite_meter()`. The latter is an extremely useful and powerful function that can improve the rhythmic notation of a selection—which is particularly helpful when generating notes and chords algorithmically. It does, however, output less-than-ideal rhythmic notation in a couple of cases. See Figures 4.9 and 4.10 in Section 4.4 for an example of the typical usage of this mutation.

Although `abjad.Meter.rewrite_meter()` can improve notation in most cases, it has a tendency to split and tie leaves that are offbeat but whose total duration is still completely contained within a single beat.[16] To illustrate this, consider the container shown in Figure 4.35. The rhythm is not ideally notated, with the note G4 crossing the third beat of the measure.

```
>>> staff = abjad.Staff(
...     r"\time 3/4 c'16 d'8 e'16 f'16 g'4 a'8."
... )
>>> abjad.show(staff)
```

[16] `abjad.Meter.rewrite_meter()` has a property called `boundary_depth` which can set different depths for fusing and splitting leaves; unfortunately, there are still some specific cases (some of which appear often in my music) in which `rewrite_meter()` fails to output the best possible notation.

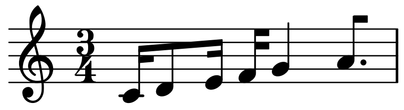

**Figure 4.35:** Container before applying `abjad.Meter.rewrite_meter()`

`abjad.Meter.rewrite_meter()` takes care of that awkward rhythmic notation, as shown in its output in Figure 4.36 below.

```
>>> meter = abjad.Meter((3, 4))
>>> abjad.Meter.rewrite_meter(staff[:], meter)
>>> abjad.show(staff)
```

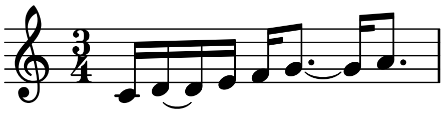

**Figure 4.36:** Container after applying `abjad.Meter.rewrite_meter()`

This makes the rhythm of the second and third beats substantially clearer and easier to read. But this example also shows the side effect previously mentioned in which a leaf is split in two and tied together within the same beat, as is the case with the D4 in the first beat of this example.

Due to the common occurrence of these rhythms in my music, I wrote a dedicated function, `auxjad.mutate.prettify_rewrite_meter()`, to remedy this issue. This function fuses those notes back together and takes care of some other edge cases of rhythmic notation. Applying it to the container above results in the notation shown in Figure 4.37.

```
>>> auxjad.mutate.prettify_rewrite_meter(staff[:], meter)
>>> abjad.show(staff)
```

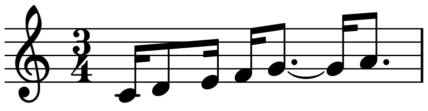

**Figure 4.37:** Container after applying `auxjad.mutate.prettify_rewrite_meter()`

Finally, `auxjad.mutate.auto_rewrite_meter()` automates the process of applying `abjad.Meter.rewrite_meter()` and `auxjad.mutate.prettify_rewrite_meter()` to a container. It is particularly useful when containers span several measures and have time signature changes. `abjad.Meter.rewrite_meter()` requires both a selection of a single measure as input and a meter value corresponding to the effective one in the input measure. This means that the user needs first to extract a list of meters from the time signatures used in the score, then select and group its leaves by individual measures, and finally match each meter with its corresponding measure selection. The example below shows how this process works in practice:

```
>>> staff = abjad.Staff(
...     r"\time 3/4 c'8 d'2 ~ d'8 "
...     r"\time 4/4 e'8 f'2 ~ f'8 g'4 "
...     r"\time 2/4 a'16 b'4 c''8."
... )
>>> abjad.show(staff)
```

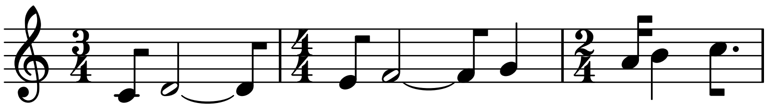
**Figure 4.38:** Container with multiple measures before applying `abjad.Meter.rewrite_meter()`

```
>>> meters = [abjad.Meter((3, 4)),
...           abjad.Meter((4, 4)),
...           abjad.Meter((2, 4)),
...           ]
>>> measures = abjad.select(staff[:]).group_by_measure()
>>> for measure, meter in zip(measures, meters):
...     abjad.Meter.rewrite_meter(measure, meter)
>>> abjad.show(staff)
```

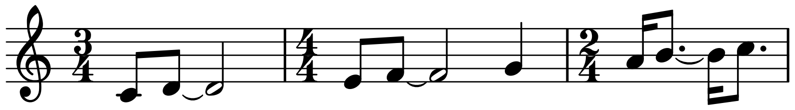
**Figure 4.39:** Container with multiple measures after applying `abjad.Meter.rewrite_meter()`

This is the type of scenario where `auxjad.mutate.auto_rewrite_meter()` is extremely useful. It automatically detects the effective meter of each measure in a container, individually applies `abjad.Meter.rewrite_meter()` to each of them, and then invokes `auxjad.mutate.prettify_rewrite_meter()`. This whole process takes a single call to this mutation, as shown below.

```
>>> staff = abjad.Staff(
...     r"\time 3/4 c'8 d'2 ~ d'8 "
...     r"\time 4/4 e'8 f'2 ~ f'8 g'4 "
...     r"\time 2/4 a'16 b'4 c''8."
... )
>>> auxjad.mutate.auto_rewrite_meter(staff)
>>> abjad.show(staff)
```

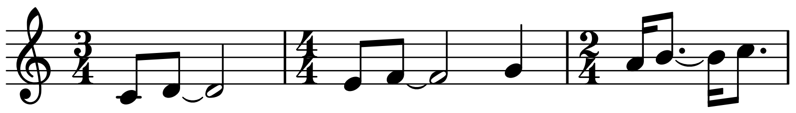

**Figure 4.40:** Container after applying `auxjad.mutate.auto_rewrite_meter()`

### 4.6.5 The `makers` Subpackage

The `makers` subpackage currently contains two makers. The first one, `auxjad.LeafDynMaker`, is a good example of the type of extension that the object-oriented paradigm allows and which was previously discussed in Section 4.5. It inherits from `abjad.LeafMaker` and extends its behaviour. The latter is a maker class that takes two lists as input, one for pitches and another for durations, and generates logical ties using those lists. As explored in Section 4.4, this class can automate the job of creating tied notes and chords when a duration value cannot be reduced to a single note value (e.g. durations such as `abjad.Duration((5, 8))`, `abjad.Duration((7, 16))`, etc.).[17]

```
>>> pitches = [0, 2, 4, 5, 7, 9]
>>> durations = [(1, 32), (2, 32), (3, 32), (4, 32), (5, 32), (6, 32)]
>>> maker = abjad.LeafMaker()
>>> notes = maker(pitches, durations)
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

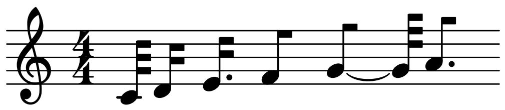

**Figure 4.41:** Output generated by `abjad.LeafMaker`

`auxjad.LeafDynMaker` extends `abjad.LeafMaker`'s functionality by adding support for optional lists of dynamics and articulations. The example below shows its basic usage, and the output is shown in Figure 4.42.

[17]These types of durations are known as non-assignable in Abjad. One cannot simply instantiate a note using `abjad.Note(0, (5, 8))` since no conventional duration values (even those with multiple augmentation dots) add up to five quavers, and thus it requires at least two tied leaves to be notated, such as a crotchet tied to a semiquaver. See Section 4.4 for further details on this topic.

```
>>> pitches = [0, 2, 4, 5, 7, 9]
>>> durations = [(1, 32), (2, 32), (3, 32), (4, 32), (5, 32), (6, 32)]
>>> dynamics = ['pp', 'p', 'mp', 'mf', 'f', 'ff']
>>> articulations = ['.', '>', '-', '_', '^', '+']
>>> maker = auxjad.LeafDynMaker()
>>> notes = maker(pitches, durations, dynamics, articulations)
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

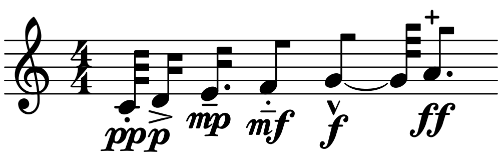

**Figure 4.42:** Output of `auxjad.LeafDynMaker`

`auxjad.LeafDynMaker` also accepts partial lists or even single dynamics and articulations, in which case they are applied only to the first elements. It can also apply them cyclically by calling it with the optional keyword arguments `cyclic_dynamics` and `cyclic_articulations` set to `True`.

The other maker, `auxjad.GeneticAlgorithmMusicMaker`, is composed of two instances of `auxjad.GeneticAlgorithm`. It is a very idiosyncratic class, which I used in my composition *methinks it is like a weasel* (2021). One of the genetic algorithms handles the evolution of pitch while the other handles attack points. `auxjad.GeneticAlgorithmMusicMaker` combines the results of each of those genetic algorithms, scoring the *combination* of pitches and attack points together, thus creating a single entity out of them.[18] Similar to `auxjad.GeneticAlgorithm`, each call to `auxjad.GeneticAlgorithmMusicMaker` will select pitch and attack point parents, create offspring using the crossover process, apply mutations to them, and rank the individuals created according to their fitness score. In the case of `auxjad.GeneticAlgorithmMusicMaker`, a call to an instance will return the fittest individual of a generation in the form of an `abjad.Selection`, which is then ready to be used in a score. The property `target_music` can be used to retrieve the environment's target (also in `abjad.Selection` form).

```
>>> maker = auxjad.GeneticAlgorithmMusicMaker(
...     pitch_target=["c'", "d'", "e'", "f'"],
...     pitch_genes=["c'", "d'", "e'", "f'", "g'", "a'", "b'", "c''"],
```

[18]For an overview of the basic functionality of my implementation of the genetic algorithm, see Section 4.6.2.1.

```
...     attack_point_target=[0, 4, 8, 12],
...     attack_point_genes=list(range(16)),
...     population_size=100,
... )
>>> notes = maker.target_music
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

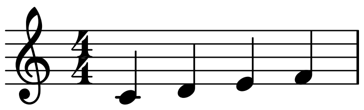

**Figure 4.43:** Target of `auxjad.GeneticAlgorithmMusicMaker`

The following example uses the genetic algorithm maker from the code above, invoking it for six iterations. It shows how the fittest individual is continuously progressing towards the defined environment target.

```
>>> notes = maker.output_n(6)
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

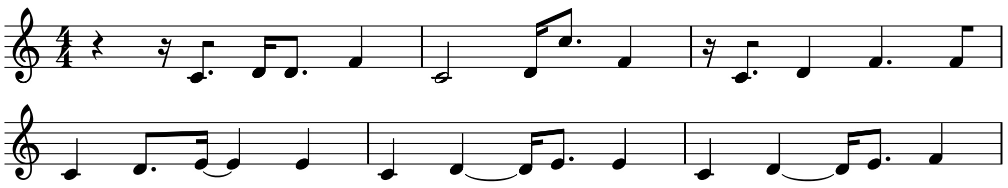

**Figure 4.44:** Output of `auxjad.GeneticAlgorithmMusicMaker` after six iterations

The values of parameters such as `population_size` and `select_n_parents` (which are also present in `auxjad.GeneticAlgorithm`) together with the number of genes in both the targets and the gene pool will affect how quickly this convergence takes.

### 4.6.6 The Other Four Subpackages: `indicators`, `score`, `spanners`, and `utilities`

The other subpackages of Auxjad mostly contain extensions for certain Abjad's own classes and functions. These include extension methods for `abjad.TimeSignature` and `abjad.Score` and an extended version of `abjad.piano_pedal()`. The subpackage `score` also contains two new types of leaves, `auxjad.ArtificialHarmonic` and `auxjad.NaturalHarmonic`, which are helpful when notating harmonics. They create notes with appropriate diamond note heads and allow for easy parenthesising of open strings and adding markup indicators for string numbers. The code below shows a staff with both types of harmonics.

```
>>> staff = abjad.Staff([
...     auxjad.ArtificialHarmonic("<a e'>4."),
...     auxjad.ArtificialHarmonic(r"<g c'>8", is_parenthesized=True),
...     auxjad.HarmonicNote(r"a''2", markup='I.', style="#'harmonic-mixed"),
... ])
>>> abjad.show(staff)
```

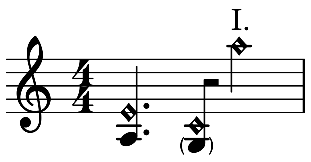

**Figure 4.45:** Staff containing leaves created with both `auxjad.ArtificialHarmonic` and `auxjad.NaturalHarmonic`

The core classes of Auxjad can make use of these harmonic classes too. For instance, Figure 4.46 shows the result of using the staff above as input to an `auxjad.Phaser`.

```
>>> phaser = auxjad.Phaser(staff, step_size=(1, 16), forward_bias=0.0)
>>> notes = phaser.output_n(3)
>>> staff = abjad.Staff(notes)
>>> abjad.show(staff)
```

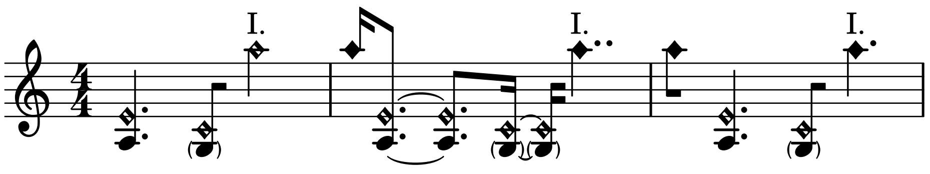

**Figure 4.46:** Output of `auxjad.Phaser` on a container with harmonics

I briefly debated whether or not to include these two classes in Auxjad, as they differ substantially from the other classes and functions in this library in terms of intent. On the one hand, Auxjad's members are primarily focused on transformations, be they algorithmic processes or simple extension methods for altering object states. On the other hand, my compositional approach often involves thinking of harmonic notes as single entities that can be used to make up larger containers, which in turn can be algorithmically manipulated (an example of this approach can be found in composition *and thereafter they shape us*). As such, the practical advantages of including these classes outweighs any concerns about the potential lack of coherence in the overall design of this library.

Auxjad currently has a single spanner function, `auxjad.piano_pedal()`, which extends the functionality of Abjad's own `abjad.piano_pedal()`. When Auxjad is imported, it automatically replaces Abjad's native function with its extended version. Using it without the new arguments will generate the same output as the original `abjad.piano_pedal()`. The new argument `half_pedal` controls half-pedalling notation, replacing the original pedal mark with '1/2*Ped.*'; `until_the_end` can be used to add an additional '→' mark after the pedal glyph, commonly used when the piano pedal is to be depressed throughout a whole piece or over substantial periods of time; `omit_raise_pedal_glyph` removes the symbol for raising the pedal from the last leaf. This is utilised extensively throughout the portfolio as the textures I work with often require half or full pedalling throughout an entire composition.

```
>>> staff = abjad.Staff(r"c'4 d'4 e'4 f'4")
>>> abjad.piano_pedal(staff[:],
...                   half_pedal=True,
...                   until_the_end=True,
...                   omit_raise_pedal_glyph=True,
...                   )
>>> abjad.show(staff)
```

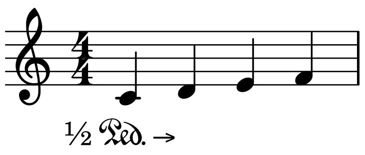

**Figure 4.47:** Output of the extended `abjad.piano_pedal()` on a container

The subpackage `utilities` currently contains a single member, the function `auxjad.staff_splitter()`, which is neither an inspection nor a mutation. This function takes an `abjad.Staff` and splits it into two staves using a specified threshold pitch. This splitting is extremely useful when working with piano music.

```
>>> staff = abjad.Staff(
...     r"\time 2/4 a8( b c' d') \times 2/3 {<g b d'>2 <e' f'>4}"
...     r"\time 3/4 <d a c' g'>4--  r8 <f a bf>4."
... )
>>> abjad.show(staff)
```

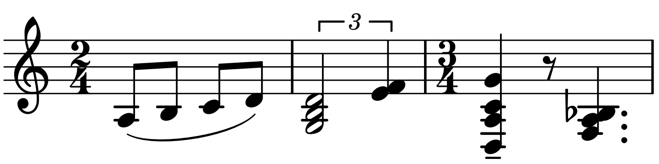

**Figure 4.48:** Material used as input for `auxjad.staff_splitter()`

```
>>> staves = auxjad.staff_splitter(staff)
>>> score = abjad.Score(staves)
>>> abjad.show(score)
```

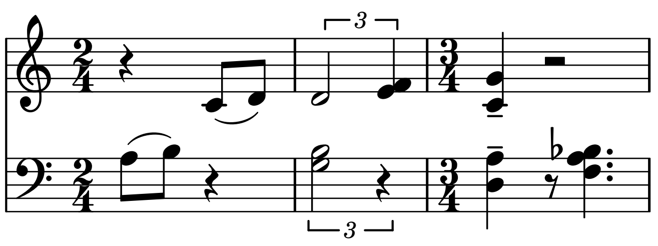

**Figure 4.49:** Output of `auxjad.staff_splitter()`

This function can thus help separate the generation of the music for piano from its notation. I used this function when composing *adrift*; the two fading layers are generated linearly and are later processed by `auxjad.staff_splitter()` for a much-improved notation.

## 4.7 Conclusion

This chapter focused on overviewing Auxjad's structure and the design choices that were taken while developing it. Many of Auxjad's classes and functions consist of direct implementations of the mental models that I employ in my current work. Most of these mental models—and, therefore, the Auxjad classes that implement those—are related to the aesthetic concepts that I explore in my music and which were previously introduced in Chapter 3, namely slippage, fragility, emergence, and liminality.

Since all loopers and faders are based on repetition techniques, they can be used to create perceptual instability in a composition. This is particularly true when working with fragile input materials, such as those that are quiet and slow, and with operating time frames that emphasise perceptual disorientation, such as using looping windows that are several seconds long but with short step lengths. In such cases, loopers and faders become essential contributors to the overall sense of slippage of a composition. The slow and repetitive unfolding that is created will, in turn, contribute to the appearance of emergent structures as well as to an overall sense of liminality in these pieces.

Emergence is another central concept explored in my music. Classes such as my selectors can create the right conditions from which emergent structures can appear in the resulting music. Although the selection process from these classes is randomised, using a heavily weighted distribution and a small pool of elements can emphasise certain random combinations over others, combinations which are often grouped together by our ears as single linkages. As discussed in Section 3.2, both loopers and faders are also able to create new emergent relationships of materials, particularly at the threshold of consecutive looping windows.

Applications of this library will be discussed in the next chapter, which will analyse some of the specific compositions that make up my accompanying portfolio. It will discuss how these mental models and aesthetic concepts can be realised in practice using Auxjad.

# Chapter 5

# Commentary on My Music

## 5.1 *Cartographies* series

*Cartographies* is a series of chamber works written between 2017 and 2020. The initial concept for this work was to create a series of automatic algorithmic compositions exploring the metaphor of transformable containers. In the context of these pieces, a container is an entity that can store any type of musical object—from simple parameters such as pitch, duration, and dynamics, to higher-level objects such as musical cells—which can then be randomly selected by an algorithm. These containers have fixed probability values associated with each of their indices, enabling particular types of operations for changing an element's effective probability. This happens because shifting an element to another position in the same container will also shift its chance of being selected. The types of transformations available to a container—such as rotation of elements or discarding an element and shifting all others—form an intrinsic part of them; that is, the available operations are embedded in the container metaphor itself. Containers thus bear a strong relationship with the notion of classes in object-oriented programming since both encapsulate a state and a specific set of transformations. This composer-oriented metaphor allowed me to approach these compositions differently than using more traditional stochastic methods, even though the mechanics of these containers are not aurally perceivable by a listener. Nevertheless, working within this framework allowed me to conceptualise this series of compositions in a unique manner, resulting in music that would not have been written had I utilised a traditional stochastic approach. Therefore, all pieces in this series share this notion of a container and are built using it as their primary compositional technique in a process that will be detailed in the following subsections.

When sketching the very first work of *Cartographies*, I decided to limit myself to defining the whole piece within a single A4 page, which should thus describe the individual containers of each parameter, the rules for their transformations, and all values used to initialise the piece.[1] The main reason for this self-imposed limitation was to shift my focus from designing more complex algorithms to the creation and selection of materials and constraints for shaping these pieces. I was immediately surprised by how simple tweaks of procedures could lead to vastly different results, which supported my initial aim of creating a series of pieces sharing a common compositional framework. This led to a substantial reformulation of my compositional approach as only a few rules could be set for each piece, and, as such, experimentation became an even more critical step in my compositional practice. The design of each piece's algorithm was done solely via the definition of three elements: the constraints of the algorithm (typically containing the total length of the piece in question, the instrumentation, the ranges of individual parameters), the transformation operations that are to be applied to containers, and the initial states of these containers.

These are thus highly parametrised compositions that employed simultaneous containers for separately manipulating different parameters. Parametric approaches to composition have a long history, dating back to before the advent of computer music. For instance, since the late 1940s and early 1950s, composers such as John Cage, Pierre Boulez, and Karlheinz Stockhausen have generalised the ideas of the twelve-tone composers of the earlier generation and created compositional frameworks that were highly parametrised (Griffiths, 2011, pp. 34–56, Boulez & Cage, 1995). In computer music, this approach of splitting sound events into separate parameters to be individually manipulated has remained common since the earliest experiments in this field (L. A. Hiller & Baker, 1964; Koenig, 1983; Barlow, 1990). Concerning working with independent parameters, Kramer (1996, pp. 24–25) states:

> Once listeners understand loudness and textural density, for example, as independent, they can comprehend each of these parameters as providing its own sense of direction. [...] these ideas [are] essentially modernist: a structuralist attempt to redefine musical temporality by creating independent structures in different parameters. But there are undercurrents of postmodernist thinking evident as well, because what the parametric

[1]This type of artistic restriction bears a connection to those employed by the members of the group commonly known as *Oulipo*, a French group of writers and mathematicians interested in exploring how writing constraints could be applied to literature. These authors worked with extreme self-imposed restrictions, such as writing a whole novel without using the letter 'e' or replacing all nouns in a text by the seventh entry that follows them in a dictionary (Baetens, 2012, pp. 117–118; Despeaux, 2015, p. 239).

> concept actually does is deconstruct the previously holistic idea of musical structure.

This deconstruction of the musical structure into independent layers is an essential aspect of this series. Each algorithmically manipulated parameter in these pieces is conceptualised as an individual container of elements undergoing specific transformations. These containers can have different lengths from one another as well as unique transformations that can be applied at independent points in time. The resulting pieces are the sum of these independent processes, each conceptualised as their own abstraction.

Besides sharing the container mental model, this series of works is also unified by using a single probability distribution shared by all pieces. By working with this fixed distribution, individual pieces are shaped solely through the use of unique mappings and constraints together with the specific characteristics of each of their containers. When I started working with this particular notion of a container, I was largely influenced by Lakoff & Núñez's ideas on cognitive metaphors. They write:

> Each such conceptual metaphor has the same structure. Each is a unidirectional mapping from entities in one conceptual domain to corresponding entities in another conceptual domain. As such, conceptual metaphors are part of our system of thought. Their primary function is to allow us to reason about relatively abstract domains using the inferential structure of relatively concrete domains. (Lakoff & Núñez, 2000, p. 42)

I based the idea of a container in *Cartographies* on the fundamental conceptual metaphor of *container of objects* $\rightarrow$ *map of musical entities* (as shown in Table 5.1). In the context of my pieces, a container is an entity made out of $n$ ordered partitions (i.e. the indices of the container) in which musical elements can be placed, and which is then used as the source pool for random selection of material. In other words, when the algorithm in any of the pieces in this series must select and output a musical element into its score, it randomly selects it from the population of its containers at that given point. Changes in the composition are thus realised by altering this population over time.

Each container index has a fixed probability distribution independent of what element is currently residing in it: that is, it is not the elements themselves that have a specific chance of being selected but rather the container's individual indices. With this metaphor's property of *shift in position* $\rightarrow$ *shift in probability*, the mapping of parameters becomes fluid, inviting algorithmic manipulations. The contents of these containers can be transformed in any number of ways, such as by rotation, shifts of positions in either direction, and replacement of

| *Source Domain* CONTAINER OF OBJECTS | | *Target Domain* CARTOGRAPHY CONTAINER |
|---|---|---|
| container | → | map |
| ordered partitions | → | index |
| physical objects | → | musical elements |
| shift in position | → | shift in probability |
| contents | → | algorithmic state |
| operations of movement of objects | → | methods of transformation of elements' indices |

**Table 5.1:** Metaphors used to formalise the container of *Cartographies*

specific elements. Any transformation that shifts elements between indices will thus alter their effective probability at that given point in the composition. These particular transformations are an integral part of the container metaphor, as *operations of movement of objects → methods of transformation of elements' indices*. In these pieces, these transformations happen at fixed cycles for a given container, although multiple containers may have different cycle lengths in a single composition. These cycles range from just a few beats to many measures and are used to slowly alter the available pool of elements for any given moment in a composition.

Although all definitions above (including the transformations available in specific containers) are deterministic in nature, stochastic processes are used to create the resulting piece of music. A container, therefore, will dictate the possible elements available at a given point in the piece and their associated probability weights, but the selection of elements that end up in the score is realised using probabilistic methods.

It is important to note that although I refer to this mental model as a 'container' throughout this chapter, its implementation in my Auxjad library uses the term 'selector' for it, as in `auxjad.CartographySelector`[2] (see Subsection 4.6.2.1). The reason for this somewhat discrepant nomenclature is that Abjad (the Python library upon which my Auxjad library depends) already had an important class named `abjad.Container` that serves a very distinct purpose.[3] Since Auxjad is dependent on and must be used with Abjad, Auxjad's

[2]The same is true for its related class `auxjad.TenneySelector`.

[3]In Abjad, an `abjad.Container` is a class whose data structure is made out of a tree of components known as leaves. This class serves as the parent for other classes such as `abjad.Voice`, `abjad.Staff`, and `abjad.Score`.

API must respect the latter's naming conventions. However, for the purpose of this dissertation, I believe it is important to use the term 'container' as it helps the visualisation of the metaphor behind this mental model, i.e. the metaphor of physical objects being added to or removed from slots of a physical container, each slot with specific characteristics, whereas the term 'selector' does not imply any such manipulations.

### 5.1.1 Probability Distribution

All works in *Cartographies* share the same probability distribution function shown in Figure 5.1, which assigns a probability value to any given index in relation to its surrounding indices. The use of a single probability distribution for all twelve works in this series not only helps to unify them under a shared fundamental principle (and, in the process, bringing an extra layer of conceptual coherence to the whole series) but also shifts the focus of the composition process into the design of containers. It also allows for easier comparisons across different pieces, with the whole cycle then showcasing twelve unique approaches to a common compositional framework.

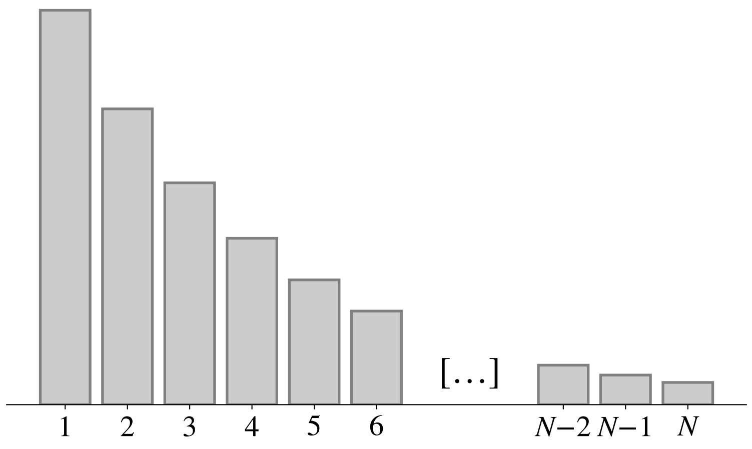

**Figure 5.1:** Probability distribution given by $P(n) = (3/4)^k \times P(n - k)$

This probability distribution is given by the formula shown below, where $P(n)$ is the probability associated with the index $n$ and $P(n - k)$ is the probability of the index $n - k$, which is $k$ steps away from $n$. In other words, the probability of any given index is 3/4 that of the index that just preceded it.

$$P(n) = (3/4)^k \times P(n - k)$$

For every random selection in a piece, the algorithm uses the distribution above to weight the elements of all containers according to their indices. Although the probabilities of all individual indices are fixed, those of individual elements can be changed by manipulating their position in a container using several types of transformations. Figure 5.1 shows that the lower an index is, the higher is the chance of it being selected and, with it, the element that currently occupies it. Coupled with shifts of elements sideways (passing an element to the previous or next index), this property of the probability distribution can be used to create trends in the resulting music, since continuously shifting an element sideways will, at each step, change its chance accordingly in the same direction. This property will be further discussed in Subsection 5.1.2.

Since the probability of any given index is defined in terms of other indices,[4] this distribution can be applied to containers of arbitrary sizes. For any length of value $N$, the algorithm calculates the individual probabilities by initialising itself with $P'(1) = 1.0$, generating the probabilities $P'(n)$ for all values of $n$ between 2 and $N$, and then calculating the normalised probability $P(i)$ for all indices between 1 and $N$ using:

$$P(i) \leftarrow \frac{P'(i)}{\sum_{n=1}^{N} P'(n)}, \forall i \in \{1, 2, 3, \ldots, N-1, N\}$$

The code below shows an implementation of this probability distribution written in Python using a function that takes an arbitrary length as input and returns a list with the same length containing the normalised probability values for each of its indices. The resulting individual probabilities returned by this function for containers with lengths ranging from 2 to 6 are displayed in Table 5.2 below.

```python
def probability_generator(length: int) -> list:
    """Returns a list with the probability values for each index of a
    container of any given length.
    """
    probabilities = [1.0]
    normalisation_factor = 1.0
    for _ in range(length - 1):
        next_probability = probabilities[-1] * (3 / 4)
        probabilities.append(next_probability)
        normalisation_factor += next_probability
    probabilities = [value / normalisation_factor for value in probabilities]
    return probabilities
```

[4]That is, the probability values are generated by a recursive function.

| | Index | | | | | |
|---|---|---|---|---|---|---|
| Length of container | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 57.1% | 42.9% | | | | |
| 3 | 43.2% | 32.4% | 24.3% | | | |
| 4 | 36.6% | 27.4% | 20.6% | 15.4% | | |
| 5 | 32.8% | 24.6% | 18.4% | 13.8% | 10.4% | |
| 6 | 30.4% | 22.8% | 17.1% | 12.8% | 9.62% | 7.22% |

**Table 5.2:** Absolute probabilities per index

### 5.1.2 Containers

The fundamental concept behind the containers of *Cartographies* is that the mapping of musical elements is malleable since these elements point only temporarily to specific indices of a given container. Consider the example shown in Figure 5.2; seven arbitrary pitches (C♯4, D4, F4, F♯4, A♭4, B♭4, and B4) are mapped in three different ways into a container with five indices. Each of these will result in a different pool of pitches, shown on the right-hand side of the figure. It is then from those pool of five pitches that an algorithm would select pitches for notes in the composition, and each of those pitches would have different weights given to them according to their position.

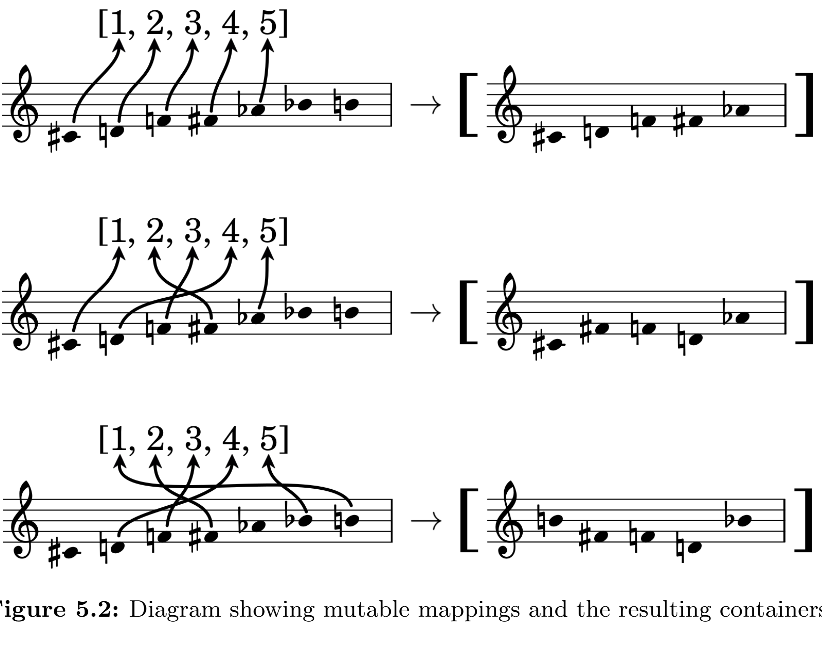

**Figure 5.2:** Diagram showing mutable mappings and the resulting containers

Some of the basic types of mapping transformations that I have explored in this series of works (but by no means an exhaustive list of the possibilities of the container metaphor) are shown below. These transformations can also be combined together to form other more complex ones.

1. rotation: e.g. $[a, b, c, \ldots, i, j] \rightarrow [b, c, \ldots, i, j, a]$.
2. replacement: e.g. $[a, b, c, \ldots, i, j] \rightarrow [a, b, c, \ldots, m, x]$, with $x$ being selected according to some arbitrary process.
3. drop first element and append new one: e.g. $[a, b, c, \ldots, i, j] \rightarrow [b, c, \ldots, i, j, x]$, with $x$ being selected according to some arbitrary process.
4. numeric operation on parameter values: e.g. $[a, b, c, \ldots, i, j] \rightarrow [f(a), f(b), f(c), \ldots, f(i), f(j)]$, with $f(x)$ being an arbitrary function.

The main characteristic of the probability distribution used in this work is that the chance of an element being selected increases if it shifts leftwards but decreases if it shifts rightwards. This property can be used to generate seemingly continuous musical transitions. For example, by using a transformation of the type 'drop first element and append new one', such as $[a, b, c, \ldots, i, j] \rightarrow [b, c, \ldots, i, j, x]$, the element $x$ is introduced at an index that has very little chance of being selected in comparison to the others. All other remaining elements (in this case $b$ to $j$) were already present before the transformation was applied and, thus, the resulting pool of available elements closely resembles the previous one with the exceptions that 1) the dominant element $a$ disappeared, 2) a new element $x$ appeared but is rarely selected, and 3) all other elements have a slightly increased probability of being selected. If such changes are applied at relatively long time intervals (e.g. once every several measures), the result is a relatively smooth but constant change of values for this parameter. This is aided by the fact that selections are made stochastically, which helps to mask these non-continuous changes in probability. The higher the number of indices, the smoother this transition will seem.

I have chosen to apply these transformations at regular intervals (although different containers can have different intervals from one another). It is important to emphasise that this is purely a compositional choice and not an inherent property of these containers. My predilection for constant rates of change is connected with my interest in exploring how linear algorithmic processes can generate non-linear listening experiences.[5] As such, I am often drawn to simple gradual processes in the form $A \rightarrow B$, where only the start and end points are defined, and all points in between are a direct result of the process itself—a

[5]These constant rates of change and non-linear sonic results can also be observed in the looping windows used in the later pieces of *Cartographies* (see Subsection 5.1.4).

decision that further aligns my music with my aesthetic of low intervention. In *Cartographies*, these gradual processes are not continuous due to the non-continuous nature of the containers. By applying transformation at equally distant points, the process results in a symmetrical staircase function that 'approaches' a gradual linear process, as shown in Figure 5.3.

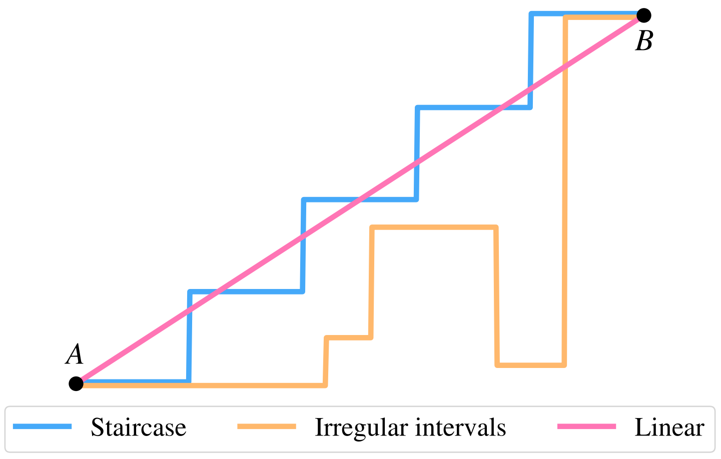

**Figure 5.3:** Comparison between linear and a staircase functions for the trajectory $A \to B$

Using transformations that remove and add only two elements at a time (such as 'drop first element and append new one'), it is, therefore, possible to change the contents of a container into an entirely different set of elements by continuously applying this same transformation. For example, consider the container of pitches [E4, F4, F♯4] undergoing the following transformations:

[E4, F4, F♯4] → [F4, F♯4, G4] → [F♯4, G4, A♭4] → [G4, A♭4, A4]

In this example, the container's initial pool of pitches is given by [E4, F4, F♯4]. After a transformation, the contents of the container will be transformed using the rule $[a, b, c] \to [b, c, d]$, with the new element $d$ being the pitch one semitone above $c$. This rule is applied two more times to generate this container's third and fourth iterations. Figure 5.4 shows a simple musical realisation of this process using notes with a constant duration of a quaver, and with transformations taking place at every measure. Since every two consecutive iterations of this container have two elements in common, the resulting music shows a smooth transition of pitch content.

### 5.1.3 Constraints and Initial States

A fundamental step in the design of these algorithms consists of choosing their constraints and initial states. These are essentially tied to the design of the

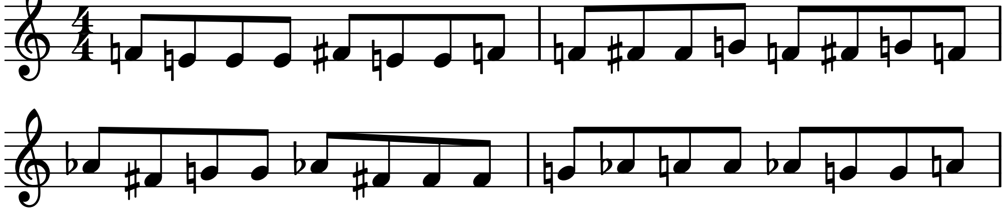
**Figure 5.4:** Example of a transformation of type 'shift with replacement'

containers themselves: it is not only their length and transformation rules that affect the final output but also how they are initialised. The initial state of a container is given by their initial set of elements, and its progression will be dictated by both the transformation rules used and any constraints that these transformations must obey. These constraints can be used to limit some of the transformations, such as by restricting a parameter's range or disallowing some specific values. Other constraints that affect material progression include the rate by which transformations are applied. Therefore, changing either the initial state or the algorithmic constraints of a container will fundamentally alter the resulting music. As such, composing becomes a heuristic act of balancing all these decisions together since finding effective values will often involve much trial and error.

An important constraint that was individually set for each of these pieces consists of fixing the initial seed of the pseudorandom number generator (commonly abbreviated as PRNG) to a specific value. PRNGs are deterministic functions that output seemingly random numbers (i.e. numbers that obey specific statistical requirements) given a start value known as initial seed (Park & Miller, 1988). PRNGs are usually capable of self-initialisation by setting these initial seeds to arbitrary values (e.g. using the computer's internal clock as a source of a random value that changes at every execution). To fix the PRNG to a specific reproducible (and, thus, repeatable) series of outputs, the initial seed can be manually set to an arbitrary value, which effectively 'locks' the resulting music as the single specific version of the score. Even though these works use a fixed initial seed, I consider that these systems exist beyond the realm of a single fixed score, while the latter is only one of the infinitely many possibilities that could have been realised by the system.

I made use of somewhat idiosyncratic constraints in most pieces of this series. For instance, in *Cartography #9* I employed a narrow pitch range of only two and a half octaves shared by all instruments,[6] while both *Cartographies #7* and *#8*

[6]The effective range of the composition as observed in the final score is actually B3–B♭5.

use only the higher register of all instruments. In most of these pieces, dynamics and articulations are used in a way as to produce music that is characteristically quiet (e.g. fixing stronger articulations to the indices with the lowest probability of being selected or by not using loud dynamics at all). These aesthetic choices limit the palette of materials and contribute to the overall fragility of these pieces. Accordingly, each of the resulting pieces becomes more distinctive, which further emphasises the aim of this series of exploring different sonic worlds through a shared algorithmic principle. The works in this series can be understood as the application of the container metaphor to unique palettes of musical material.

Some of the typical constraints used in these pieces include the range of the main parameters (pitch, duration, dynamics), set of available instruments and instrumental timbres, total duration of the piece, length of the cycles of transformation of containers, to name a few. In some cases, two containers of the same length are linked together so that whenever the $n$-th element of the first is selected, the $n$-th element of the second also is selected. I have used this technique in multiple pieces of this series, such as the linked pitch and articulations containers found in *Cartography #5* and the linked dynamics and duration containers in *Cartography #10.* However, this does not mean that elements themselves are tied together for the whole piece, but rather that the same *indices* of two containers are linked, allowing for elements to be shifted or manipulated. To illustrate this, consider the linked containers [G4, F♯5, B♭5] and [***pp***, ***mp***, ***f***]; resulting random selections will pick among three options of pitch-dynamic pairs: (G4, ***pp***), (F♯5, ***mp***), and (B♭5, ***f***). Figure 5.5 shows an example of six random selections made with this process.

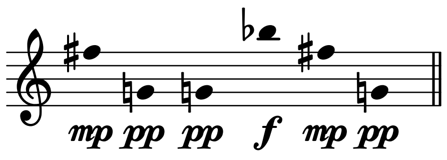

**Figure 5.5:** Example of six random selections from the paired containers

Although their indices are linked, the mapping of these containers can be changed individually by applying transformations to either of them. Suppose the pitch container undergoes the following transformation [G4, F♯5, B♭5] → [F♯5, B♭5, E♭4], while the dynamic container remains the same. After the transformation is applied, the resulting pitch-dynamics pairs thus become (F♯5, ***pp***), (B♭5, ***mp***), and (E♭5, ***f***). Figure 5.6 shows twelve random selections made from these

This happened by chance due to the stochastic procedures not selecting notes in the lowest part of the range (F3–B♭3).

containers, with the first six being chosen before the transformation is applied and the subsequent six after it.

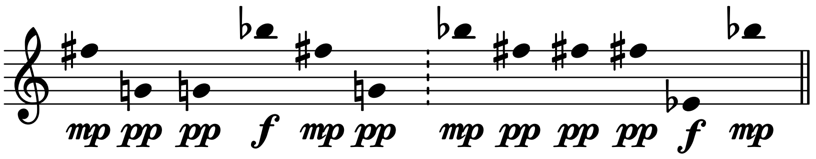

**Figure 5.6:** Example of twelve random selections from the paired containers

The choice of these initial states is fundamentally associated with both the design of the container's transformations and the total number of operations that a container will undergo throughout the whole piece. As an example, consider the numeric operation $f(x) = \max(x - 1, 1)$.[7] When applied to each of the elements of the container $[5, 4, 3, 2]$, it yields $[4, 3, 2, 1]$, maintaining the same relationship between consecutive elements (i.e. the difference between consecutive elements is still 1). However, if applied once more, it yields $[3, 2, 1, 1]$, thus modifying the relationship between the last two elements. After two more operations, the contents reach the set $[1, 1, 1, 1]$ and will not change any further by using that numeric operation. Therefore, in this particular case, it is crucial to consider this behaviour when selecting an initial state as well as the total number of operations applied throughout a piece. Using the same example, if a programmer would not use the max() function but instead define the transformation function simply as $f(x) = x - 1$, the output values could end up becoming negative, which may be undesirable if they are mapped to parameters that are expected to always be positive (such as durations). It is thus crucial to consider the number of times an operation will take place when designing their behaviour and selecting the container's initial state.

In terms of the pitch sets employed, most pieces in this series use straightforward sets made out of chromatic pitch fields that are sequentially separated by one semitone. Table 5.3 shows the initial states of all pitch containers in *Cartographies*. Other than Cartographies *#1* and *#3*, all other pieces employ chromatic ascending or descending initial pitch sets. Most of the pieces make use of transformations that keep pitch-class sets chromatic; these employ the already-mentioned 'drop first element and append new one' transformation, in which the new appended element is a semitone above or below the preceding one (depending on whether the set is ascending or descending chromatically, respectively). In

[7]This operation takes an input number, subtracts 1 from it and then outputs either the result of this subtraction or the numeric value 1, whichever is the greatest; e.g. $f(6) = 5$, $f(4) = 3$, $f(1) = 1$, $f(0) = 1$, $f(-3) = 1$, etc.

other words, the sets follow the transformation $[a, b, c, \ldots, i, j] \rightarrow [b, c, \ldots, i, j, k]$, with $k \bmod 12 = (j \bmod 12) \pm 1$, and $k$ at a uniformly randomly selected octave transposition selected from within the pitch range constraint for the given instrument.

As can be seen in Table 5.3, there are exceptions to the chromatic approach previously described. *Cartography #3* is not formalised using pitches per se, but instead uses a system that generates possible guitar fingerings from which pitches are then derived. *Cartography #5* starts chromatically, but the G4 pitch is fixed at the first index (i.e. it is never shifted under any transformation), a decision that emphasises the open G-string of the violin throughout this work. Given my predilection for unique pitch classes in my sets, chromatic transformations would not be possible while keeping the constant G4 since, at some point, the new pitch class in the last element would shift from F♯ to A♭, an interval of 2 semitones. This is the reason behind the decision of starting the piece chromatically but then selecting new pitch classes randomly, with the only constraint that pitch classes must be unique within the same container. This approach is similar to that found in *Cartography #1*, which was written before I started working primarily with chromatic fields.

Most of the pieces that make up *Cartographies* display my clear preference for chromatically saturated pitch fields. This comes in part due to the overall 'flatness' that such constant fields evoke during the listening experience, which is also aided by the flatness of the other musical structures in this music (e.g. non-hierarchical form, use of constant steps in looping windows, soft and quiet textures). The flatness of the harmonic experience in these pieces comes mostly from a non-hierarchical approach to pitches: all pitch classes are effectively equal as they are derived from a single intervallic procedure of shifting one semitone. A pitch-class set such as these will always cycle back to itself after 12 iterations (even though the container's effective pitches might be located in different octave transpositions). The aural result is that of a long-scale drift (either upwards or downwards), akin to a stepwise Shepard tone: the process gives the illusion of constant rise or falls when, in fact, the pitch classes change cyclically. The periodical nature of these transformations can also be related to another characteristic of these pieces, namely the predominant use of musical repetition as a structuring device (a topic which will be discussed in the next section). The pitch-class sets are thus cyclic and the process of shifting them is realised by a constant operation of either $f(a_n) = a_{n-1} + 1$ or $f(a_n) = a_{n-1} - 1$ repeated over and over again. This 'flatness' and large-scale drift will further contribute to a piece's sense of slippage and liminality, two notions previously

| *Cartography #* | Initial Pitch Container(s) | Initially Chromatic | Chromatic Transformations |
|---|---|---|---|
| *#1* | [G4, D4, E♭4, A4, F4, B4] | **no** | **no** |
| *#2* | [G4, F♯4, F4, E4, E♭4, D4, C♯4, C4] | yes | yes |
| *#3* | selected through randomised fingerings | **no** | **no** |
| *#4* | { [C6, B5, B♭5, A5, A♭5, G5, F♯5, F5]<br>[G♭3, A3, B♭3, B3, C4, C♯4, D4, E♭4]<br>[G6, F♯6, F6, E6, E♭6, ∅], ∅ = rest | yes | yes |
| *#5* | [G3, A♭3, A3, B♭3, B3, C4, C♯4] | yes | **no** |
| *#6* | { [F4, F♯4, G4, A♭4, A4, B♭4]<br>[E4, E♭4, D4, C♯4, C4, B3] | yes | yes |
| *#7* | [C6, B5, B♭5, A5, A♭5, G5, F♯5] | yes | yes |
| *#8* | [A5, B♭5, B5, C6, C♯6, D6] | yes | yes |
| *#9* | [C4, C♯4, D4, E♭4, E4, F4] | yes | yes |
| *#10* | [C6, B5, B♭5, A5, A♭5, G5] | yes | yes |
| *#11* | [C4, C♯4, D4, E♭4, E4, F4] | yes | yes |
| *#12* | [F4, E4, E♭4, D4, C♯4, C4] | yes | yes |

**Table 5.3:** Initial pitch containers for all *Cartographies*

discussed in Chapter 3.

Writing about non-hierarchical approaches to pitch will invariably evoke connections to serial techniques. It could indeed be said that there are shared aims between my techniques for pitch manipulation and traditional serial ones; but as Xenakis famously pointed out, the complexity of the sonic result of serialism—with pieces often being the polyphonic result of multiple parallel instances of twelve-pitch series used at different transpositions and modes —has the effect, at the perceptual level, of a single mass of sounds distributed statistically around the whole chromatic field (1994). By narrowing the lengths of my containers to values smaller than twelve (as shown in Table 5.3) and by using simple ordered chromatic pitch classes that are shared among all instruments in a given piece, the resulting music displays aurally recognisable harmonic fields that change over time, with the transformations of harmony becoming transparent in the process.

### 5.1.4 Use of Repetition

Repetition plays a crucial perceptual role in my recent work. As previously discussed in Chapter 3, repetition is an important technique that I use in my music to increase its sense of slippage as well as generate emergent musical structures. My repetition-based procedures (such as looping windows) create music that, on a local level, gives rise to micro-variations of material. These are often subtle enough that changes become imperceptible, particularly when the materials used are fragile. On this local scale, the ungraspability of these repetition processes creates a sense of disorientation for the listener, which I refer to as slippage. On a larger scale, these types of repetition processes give rise to a sense of liminality: they create music that is in constant motion, leading to a sense of transience and detachment on a larger scale. These are notions that emerge from the repetition processes I employ and have become the main focus of my recent work.

Although I only started working with looping processes from *Cartography #8* onwards, repetition also plays a role in earlier pieces of this series. By using idiosyncratic constraints that narrow the available range of a piece's parameters, stochastic processes alone can yield very similar or even identical structures from time to time. This stochastically created similarity is very compelling for the listener: structures that emerge from randomness are picked by the ear, which groups them together due to their temporal and pitch similarities, such as when consecutive notes are placed on the same register (Bregman, 1990, pp. 455–528; Tenney & Polansky, 1980). Thus, locally, these emergent patterns are often perceived as having interconnected musical relationships, even though, in fact, they are the resulting shimmering in the surface of randomness. Such emergent relationships are often present in the 'input music' that my algorithms first generate, before any looping process is applied. Nevertheless, repetition processes also play an important role in promoting emergent structures, as these can often form at the borders of consecutive looping windows. These are moments of disjunction in relation to the input music, which happen when the algorithm jumps back to revisit a past point in time. At these borders, elements that are not consecutive in the input music find themselves side by side, and can end up linked together by our ears depending on their characteristics.

The nature of the containers used in *Cartographies* makes them particularly favourable for generating such emergent structures. Since the probabilities of a container's indices are not equally distributed, some indices (and, therefore, whatever element that happens to be residing in it) have a much higher likelihood of being selected than others. Particular constellations of different musical parameters can appear multiple times in short succession, further contributing to creating structural links. Figure 5.7 shows one such example from the first movement of *Cartography #4*. In measures 65–68, a small identifiable unit made out of a semitone slide between D2 and D♯2 is played multiple times in the harp. This very same structure can also be heard once in measures 54–55 and then once more in measures 59–60, but in measures 65–68, it becomes a dominant presence. This little pattern is very recognisable due to its unique register in the texture, as these happen to be the two lowest pitches in the whole movement. In reality, the algorithm did not 'prepare' this pattern in the earlier two occurrences in order to reintroduce it multiple times in short succession, but instead, these two pitches were randomly chosen multiple times in a short span, leading our ears to recognise them as a single structural group.

Starting with *Cartography #8*, I began working with the mental model of a moving looping window. I have since explored two types of looping windows in my work: one whose end points move note-wise (i.e. it always has $n$ notes, despite the sum of their durations) and another whose end points move according to a given time unit (i.e. the looping window has a specified total duration).[8] Practical applications of these two types of looping window will be discussed in more detail in Sections 5.1.8 and 5.1.9, respectively, which focus on the pieces in

[8]These are implemented in the Auxjad library as `auxjad.LeafLooper` and `auxjad.WindowLooper`, respectively. Auxjad also has a third looper class named `auxjad.ListLooper`, which is similar to the first one but handles lists with any type of elements as opposed to an `abjad.Container` with multiple leaves.

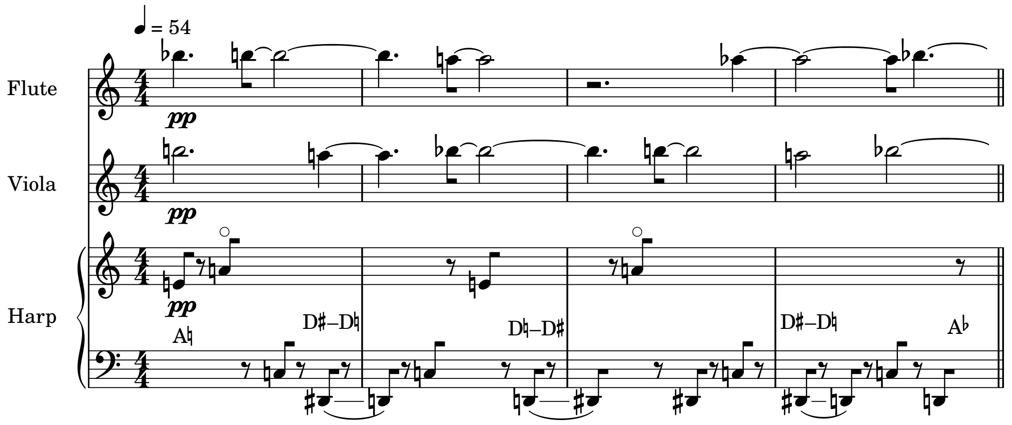

**Figure 5.7:** Excerpt of *Cartography #4*, first movement, measures 65–68

which I introduced these ideas.

Figure 5.8 shows an example of a looping window of the first type with a length of four notes, with the resulting music shown at the bottom. Figure 5.9 illustrates the second type of looping window, which has a fixed size and moves forward by one temporal unit (in this case, a semiquaver), with the resulting music shown at the bottom.

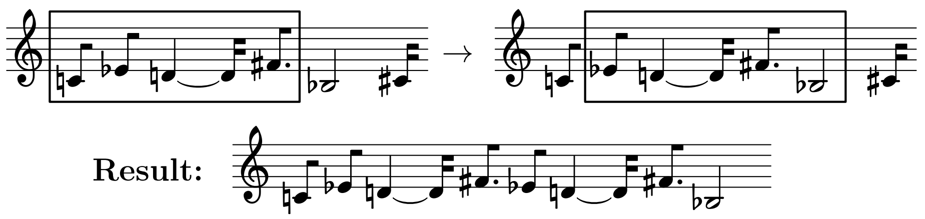

**Figure 5.8:** Example of a looping window moving forward by one note

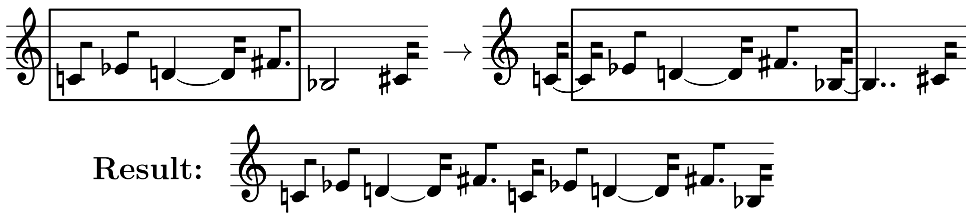

**Figure 5.9:** Example of a looping window moving forward by one temporal unit

In both cases, the algorithm first creates the input music, a complete piece of music created using the previously defined initial states, constraints, and container transformations for the sole purpose of serving as input material for the looping process. After generating this input music, the algorithm applies the looping window process to it in order to generate the final work. In these compositions, loops are thus used ‘as a means of deconstruction’ (Lang, 2003) of this input music. My use of repetition thus consists of one more procedure on top of a long chain of algorithmic processes used to create and deconstruct this never-to-be-heard input music that is generated from within the algorithm itself.

These are thus process-based works that use linear musical processes to create repetitive, disorienting, and non-linear listening experiences. As previously explored in Chapter 3, the repetition processes employed in my pieces give rise to both local and global disorientating phenomena, which I frame using the concepts of ‘slippage’ and ‘liminality’, respectively. The mental model of looping windows has been of crucial importance for investigating these two notions. With the right set of materials, they can produce straightforward algorithmic processes that generate perceptually ambiguous music.

### 5.1.5 *Cartography #1*

The first piece in this series, *Cartography #1* (2017) for piano and vibraphone, is also the first work of mine composed using the container mental model. After coming up with the general idea of how containers could operate (as described in the previous sections), I wanted to write a short piece to put them into musical practice. As such, this series was not initially conceived as a large scale project containing multiple individual compositions, but instead, it naturally evolved towards that.

As a departing point for this first piece, I aimed at creating a gradual and continuous ‘instrumental crossfade’, which was to be achieved solely by shifting the mappings of multiple parametric containers. This idea came about from a simple line of questioning: would these shifts in mappings be able to create a convincing crossfade structure? Would the exact moments of shift be perceivable, or would the listening experience be smooth as I aimed? Would these containers be able to effectively handle parameters not involved in the crossfade process? Would the idea of containers promote the development of emergent structures?

Gradual crossfades can be easily achieved using traditional stochastic techniques: given two instruments $A$ and $B$, the probability for the first can continuously[9] change from the maximum to the minimum value, i.e. $P_A = 1.0 \to 0.0$,

[9]Computers are actually not capable of producing true continuous changes of numeric quantities. The data type commonly used to emulate this sort of operation are floating-point numbers (commonly referred to as ‘floats’), which have a fixed precision value (i.e. the maximum number of significant digits they can store). Since computers can handle floats with large numbers of decimal places, they are considered an effective approximation of actual real numbers ($\mathbb{R}$). As such, small non-continuous increments of values can thus give the illusion of actual continuity.

while, at the same time, the probability of the second continuously change in the opposite direction, i.e. $P_B = 0.0 \rightarrow 1.0$. In this work, I wanted to investigate whether containers could also produce similar results with continuously sounding changes despite their fixed probabilities. While the stochastic technique mentioned above require manipulation of the probability distribution of individual parameters, containers have a fixed probability value assigned only to their indices. However, shifting an element leftwards will increase its chance of being selected, while the opposite holds for a rightwards shift. In the right circumstances, the changes in the musical result created by this technique will be perceived as smooth and gradual, despite its discontinuous realisation.

In order to achieve this illusion of continuity, this work uses two containers of eight indices each, one for the durations of the piano and the other for those of the vibraphone. These durations actually represent the distance between consecutive attack points, with all notes notated as semiquavers. In this composition, I used constant pedalling for both instruments as a blurring element, and therefore these semiquavers will effectively last for a substantially longer time. At every moment in this piece, there are eight values in the pool of available durations for any given note of each instrument. The vibraphone starts with a container filled with longer durations (resulting in less density of attack points); the contents of this container will then gradually shorten until it is filled with attack points at every single semiquaver. The piano follows the opposite procedure, starting with a container filled with the minimum duration, which slowly moves towards the initial density set for the vibraphone. These two procedures are formalised using Transformations 5.1 and 5.2, which are applied to the vibraphone and piano, respectively.

$$\begin{aligned}[a, b, c, \ldots, g, h] \rightarrow [&\max(a+1, 1), \max(a, 1), \max(a-1, 1), \\ &\ldots, \max(a-5, 1), \max(a-6, 1)]\end{aligned} \tag{5.1}$$

$$\begin{aligned}[a, b, c, \ldots, g, h] \rightarrow [&\max(a-1, 1), \max(a-2, 1), \max(a-3, 1), \\ &\ldots, \max(a-7, 1), \max(a-8, 1)]\end{aligned} \tag{5.2}$$

Using the initial states $[10, 9, 8, 7, 6, 5, 4, 3]$ for the vibraphone, and $[1, 1, 1, 1, 1, 1, 1, 1]$ for the piano, the application of these two Transformations above will result in the changes of durations shown in Table 5.4. Each transformation step takes place every 16 measures; this means that, at every 16 measures, a new duration enters the container (although with a very low probability of being selected), and the longest duration disappears, with all others receiving a small increment in their probability. The key aspect for creating this experience of smooth changes is that two consecutive containers have almost the same content, so the overall density of notes will change only ever so slightly. The musical results of this process can be observed in Figure 5.10, which contains six excerpts of a single measure, taken from distinct points in the composition's trajectory.

| Transformation # | Vibraphone | Piano |
|---|---|---|
| 1 | $[10, 9, 8, 7, 6, 5, 4, 3]$ | $[1, 1, 1, 1, 1, 1, 1, 1]$ |
| 2 | $[9, 8, 7, 6, 5, 4, 3, 2]$ | $[2, 1, 1, 1, 1, 1, 1, 1]$ |
| 3 | $[8, 7, 6, 5, 4, 3, 2, 1]$ | $[3, 2, 1, 1, 1, 1, 1, 1]$ |
| 4 | $[7, 6, 5, 4, 3, 2, 1, 1]$ | $[4, 3, 2, 1, 1, 1, 1, 1]$ |
| 5 | $[6, 5, 4, 3, 2, 1, 1, 1]$ | $[5, 4, 3, 2, 1, 1, 1, 1]$ |
| 6 | $[5, 4, 3, 2, 1, 1, 1, 1]$ | $[6, 5, 4, 3, 2, 1, 1, 1]$ |
| 7 | $[4, 3, 2, 1, 1, 1, 1, 1]$ | $[7, 6, 5, 4, 3, 2, 1, 1]$ |
| 8 | $[3, 2, 1, 1, 1, 1, 1, 1]$ | $[8, 7, 6, 5, 4, 3, 2, 1]$ |
| 9 | $[2, 1, 1, 1, 1, 1, 1, 1]$ | $[9, 8, 7, 6, 5, 4, 3, 2]$ |
| 10 | $[1, 1, 1, 1, 1, 1, 1, 1]$ | $[10, 9, 8, 7, 6, 5, 4, 3]$ |

**Table 5.4:** Duration container progression in *Cartography #1*

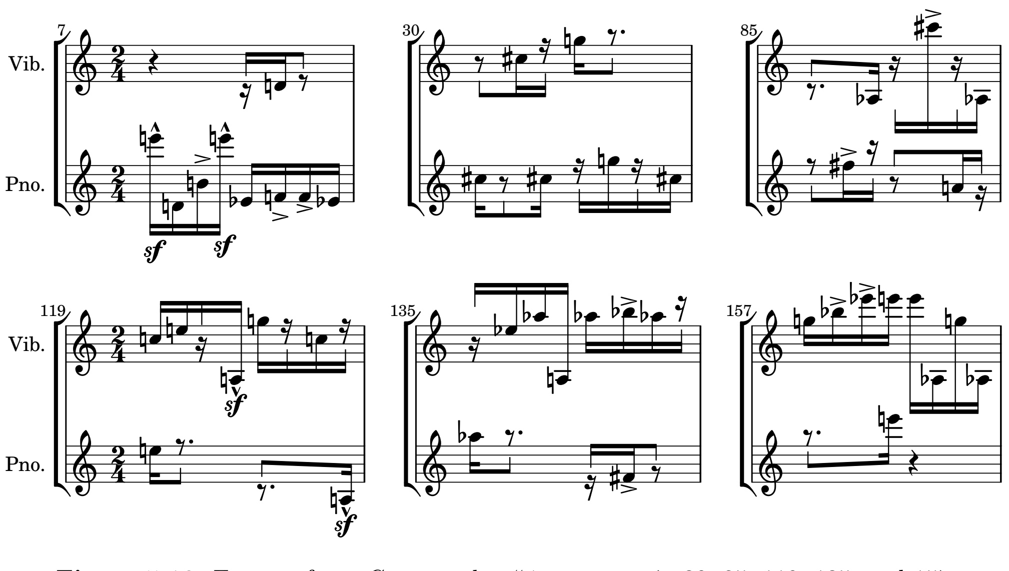

**Figure 5.10:** Excerpt from *Cartography #1*, measures 7, 30, 85, 119, 135, and 157

The tempo of this work and length of the transformation cycles are both crucial for generating this impression of gradual transition. With the constraint of composing a relatively short piece with less than five minutes in total duration, I used trial and error to adjust those other parameters. I set on a tempo of ♩ = *ca.*

84 and a total of ten transformations of sixteen $\frac{2}{4}$ measures each, a combination that generated the density of notes that I looked for and created a very smooth impression of the probability changes. This type of heuristic approach for setting some specific constraints is typical for all *Cartographies*.

While each instrument has its own duration container (necessary for achieving the crossfade), containers with pitches and articulations are shared by both of them. The pitch range used for the whole piece is F3–E5 so that the same container can be used for both the vibraphone and the piano. The pitch container is initialised as [G4, D4, E♭4, A4, F4, B4] and is transformed using the procedure $[a, b, c, d, e, f] \rightarrow [b, c, d, e, f, g]$, with $g$ being a pitch with a new pitch class (i.e. not in the container) and uniformly chosen within the shared pitch range.

The loudness of notes is notated solely using articulations; softer notes are notated with no articulation at all, medium dynamic notes are notated with marcato articulations, and louder ones are notated with both a martellato articulation and a ***sf*** dynamic indication.[10] Their container has length six and is fixed throughout the work with the values [∅, ∅, ∅, >, >, ^], where ∅ represents no articulation.[11] This container is linked with the pitch container, and so the rarest of the pitches will receive a martellato articulation while the pitches in the fourth and fifth indices will receive a marcato articulation. This is a technique that I used throughout this whole series: by linking louder dynamics to the least likely pitches or durations, any newly introduced element in those containers becomes more salient while misdirecting the attention of the listener from the disappearing element that was just removed.

This piece also showcases an important aspect of working with containers: they can promote the development of emergent structures. Due to the limited pool of elements available in each container at any given time and the different weights assigned to each of them, specific combinations of notes may happen

[10]Although I initially used only the martellato articulations, performers of this piece found it easier to read the louder attacks when they also had a ***sf*** dynamic. These attacks seldom appear in this composition due to their position in the articulation container, so a ***sf*** indication makes it easier to recognise them.

[11]Most mapping functions I used in the containers of *Cartographies* are classified as bijective, in which each index of a container is mapped onto a unique value of a parameter. In other words, there are no repeated parameters in a container and, by definition, no repeated indices either, creating a one-to-one correspondence. Occasionally, I have also employed surjective functions, as is the case of the container [∅, ∅, ∅, >, >, ^]. In this case, different indices may output the same value for a parameter as their mappings are not all unique. This technique was particularly useful when I wanted to make an element even rarer in relation to others. In this example, with three options for articulations, I found that using a smaller container such as [∅, >, ^] was not resulting in the dynamic range I aimed for. Using surjective containers allowed me to further adjust these relationships of probability while continuing to work with this mental model for all parameters.

repeatedly in short succession. Depending on their characteristics, such as sharing the same register or having louder articulations, these notes can form structures that are perceived as single groups by the listener (Bregman, 1990, pp. 455–528; Tenney & Polansky, 1980), which are then emphasised and validated by the near-repetitions that may occur. Although the strong links between specific elements form relationships that are hearable by the listeners, they cannot be accounted for in the notion of the container itself, which is blindly selecting parameters on a note-by-note case. These relationships, thus, emerge from the algorithmic process itself.

### 5.1.6 *Cartography #4*

*Cartography #4* (2017, revised 2020), for flute, viola, and harp, is the only composition written in multiple movements in this series. This work was originally written as a single movement (which is the second one in the final version), but during a revision in 2020, I decided to include two others. Through the use of a multi-movement structure, *Cartography #4* functions as a triptych: three 'panels' exploring the same thematic, which include similar textures, the use of containers, and approach to pitch range.

The initial compositional idea for this work was to use containers for exploring particular uses of texture, pitch range, and timbre. The main textural characteristics of each of these three movements arise from their individual constraints: the viola's use of sul ponticello tremolo in the second movement and sul tasto in the last, the specific uses of harmonics throughout the piece, and the contrasting instrumental colours of the different pitch ranges. Tempo also plays an important role in this piece, with the first movement using it as a structural element selected from a container of several fixed tempi. Figure 5.7 (on page 143) shows an excerpt of the first movement, while Figures 5.11 and 5.12 show the first bars of the second and third movements, respectively.

Unlike most of the other non-solo *Cartographies*, the instruments in *#4* do not share a common pitch range. The harp serves as the central unifying element, being able to play notes within its full range on all movements. However, the flute and viola are always confined to a single octave each, which is shared with one another for most of the time. Table 5.5 shows the pitch range progression for all movements of this work. Note in the table below that, in the third movement, the pitch range for the viola and flute changes halfway through, creating a movement structured in two clear sections.

For the first two movements, this pitch container has length eight, while for the last movement, it has length six (with the rightmost index set to a fixed

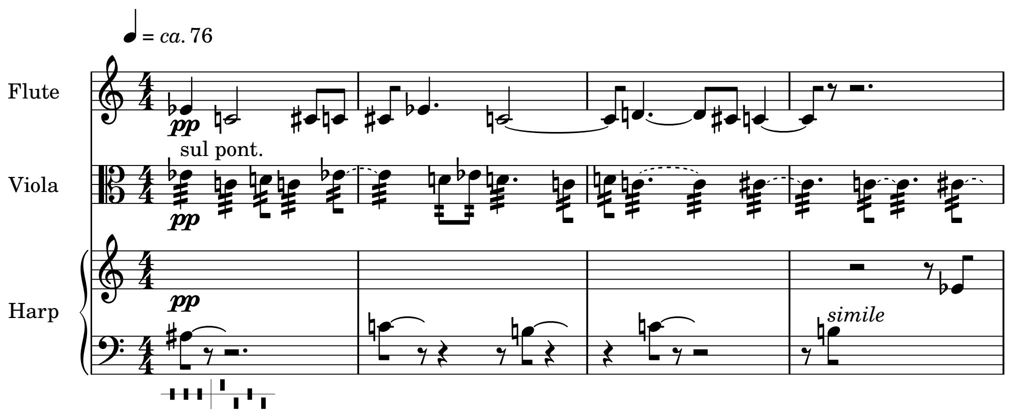

**Figure 5.11:** Excerpt of *Cartography #4*, second movement, measures 1–4

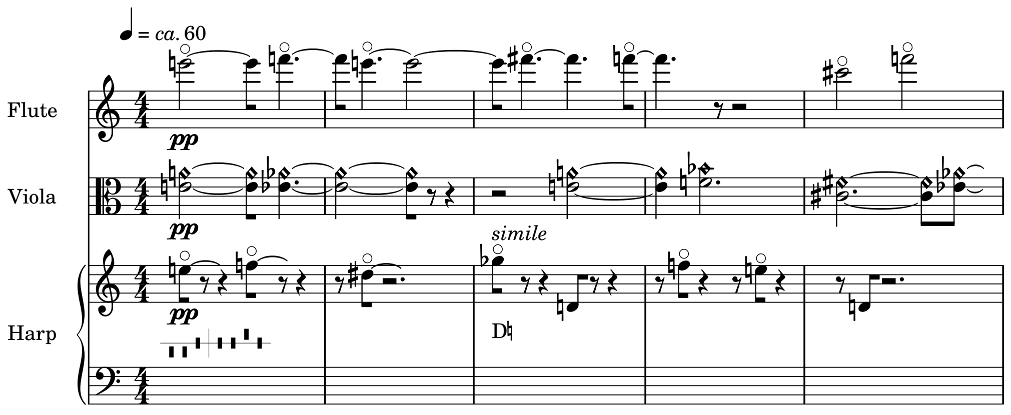

**Figure 5.12:** Excerpt of *Cartography #4*, third movement, measures 1–5

value of 'no pitch', i.e. silence). Despite not employing a common pitch range in this composition, all instruments share the same pitch container. Thus, when choosing a pitch for a specific instrument, the algorithm uses a select and test approach: it selects a random pitch and tests it against the allowed pitch range for that particular instrument. If the pitch is allowed, it is then applied to the note; otherwise, it is discarded, and a new pitch is selected and tested. Another related constraint applied to this algorithm relates to adding a new pitch to this container: the algorithm must always ensure that at least two pitches are available for each instrument at any given time. This means that, while the harp will likely be able to play any pitches in this container at all times, the flute and viola will only have a subset of those available for them. When the flute and the viola share the same pitch range (movements I and II, and the first section of movement III), this smaller pool of available pitches creates another emergent effect: these two instruments will often give the impression of following and imitating one another, as there are fewer options for each note (see the opening

|  | Movement |  |  |  |
|---|---|---|---|---|
|  | I | II | III |  |
| Instrument |  |  | mm 1–31 | mm 32–60 |
| Flute | C5–C6 | C4–C5 | B♭6–B♭7 | C4–C5 |
| Viola | C5–C6 | C4–C5 | B♭6–B♭7 | C3–C4 |
| Harp | A♭1–A6 | A♭1–A6 | A♭1–A6 | A♭1–A6 |

**Table 5.5:** Absolute probabilities per index

measures of the second movement in Figure 5.11 for an example of this). This is an emergent musical characteristic that arises solely from the simple rules of how containers operate coupled with the small number of available pitches.

### 5.1.7 *Cartography #7*

In *Cartography #7* (2018), for four electric guitars, I wanted to explore how containers could be used in conjunction with higher-level musical entities as opposed to single parameters. Unlike the previous works in this series, all of which employed a purely parametric approach to music generation, *Cartography #7* is created using pre-defined musical cells upon which pitch is then later imposed. Using container transformations like those previously explored in this chapter, the probability values of these cells can become variable, creating a slowly changing musical texture. Cells at the leftmost indices of the container are given priority, while cells at the rightmost index have less chance of being selected. In this piece, form thus becomes an emergent property of this process, i.e. the result of the list of available cells and their container's transformations.

In this work, the primary type of container transformation used for musical cells is $[a, b, c, d, e] \rightarrow [b, c, d, e, f]$. This choice reflects a similar musical intention as the crossfade effect in *Cartography #1* (previously discussed in Subsection 5.1.5): to create the illusion of smooth continuous transitions despite working with stepwise operations. By introducing new musical cells at the index with the lowest probability and keeping most of the contents identical between operations, these changes are perceived as smooth by the ears. The size of the container is of fundamental importance if these smooth changes are to be perceived; Table 5.6 shows a list of ratios between the last and first indices for a container of length ranging from two to six. The lower this ratio, the 'smoother' the changes will sound since the difference between the minimum and maximum probability values is lowered, resulting in less abrupt changes. Also, the length of a container dictates the number of steps between these minimum and maximum probabilities, so a higher length allows for smaller incremental changes in probability, aiding the perception of smooth transitions.

| Length of container | Ratio between last and first indices |
|---|---|
| 2 | 0.75 |
| 3 | 0.563 |
| 4 | 0.422 |
| 5 | 0.316 |
| 6 | 0.237 |

**Table 5.6:** Ratio between the last and the first indices of a container of a given length

For this work, I set on a container of length five for the musical cells. For a container of length five, the last index is about three times less likely to be selected than the first one. In this case, an element moving incrementally from the last to the first index will, as given by Table 5.2, follow a trajectory of absolute probabilities in the range $10.4\% \rightarrow 32.8\%$.

The musical cells used in this work are listed below. Except for the first cell, which has a *cresc.* from *al niente*, the dynamics of all other cells remain at a soft level of ***p***.

1. single note with *cresc.* from *al niente*
2. single tap harmonic
3. group of $n$ consecutive semiquavers, with $n$ selected from $[2, 3, 4, 5, 6]$[12]
4. single note with vibrato
5. single regular note
6. mixed group of $m$ consecutive semiquavers, the last one with a vibrato, with $m$ selected from $[4, 5, 6, 7, 8]$[13]

Each cell type above is mapped to a number ranging from 1 to 6. These are not merely arbitrary labels but also serve to manipulate the transformations between cell types. In other words, the transformations of this container employ numeric manipulations for deriving the new contents. The container used to select the cell type is initialised with $[1, 1, 1, 1, 1]$ (that is, the composition starts only with cells of type 1). There are four stages for the transformation

[12]Note that the selection of the number of consecutive semiquavers for both groups 3 and 6 uses a container with the same distribution described on Figure 5.1. This means that shorter groups are more likely to be selected.

[13]See footnote 12.

mechanism of the cells. The first four transformations are given by $[a, b, c, d, e] \rightarrow [b, c, d, e, e+1]$. The next four are given by $[a, b, c, d, e] \rightarrow [b, c, d, e, a]$. The next five are given by $[a, b, c, d, e] \rightarrow [b, c, d, e, 6]$. The final five transformations are given by $[a, b, c, d, e] \rightarrow [b, c, d, e, f]$, where $f = (e \bmod 6) + 1$. This results in the following states for this container (with a different type of transformation per line):

$$
\begin{aligned}
&[1,1,1,1,1] \rightarrow [1,1,1,1,2] \rightarrow [1,1,1,2,3] \rightarrow [1,1,2,3,4] \rightarrow [1,2,3,4,5] \rightarrow \\
\rightarrow&[2,3,4,5,1] \rightarrow [3,4,5,1,2] \rightarrow [4,5,1,2,3] \rightarrow [5,1,2,3,4] \rightarrow \\
\rightarrow&[1,2,3,4,6] \rightarrow [2,3,4,6,6] \rightarrow [3,4,6,6,6] \rightarrow [4,6,6,6,6] \rightarrow [6,6,6,6,6] \rightarrow \\
\rightarrow&[6,6,6,6,1] \rightarrow [6,6,6,1,2] \rightarrow [6,6,1,2,3] \rightarrow [6,1,2,3,4] \rightarrow [1,2,3,4,5]
\end{aligned}
$$

From the trajectory above, it can be seen that it is possible to use simple container transformations to generate a complex evolution of the musical material. The piece starts only with the first type of cell and slowly gets more diversified (although, for the initial transformations, cell 1 remains the dominant one). Once the contents reach $[1, 2, 3, 4, 5]$, they are rotated leftwards, and so each cell takes a dominant role once before being moved to the last index. When this rotation is about to result in $[1, 2, 3, 4, 5]$, the algorithm changes again and introduces a new cell of type 6, albeit in the index with the lowest probability. This cell slowly starts to dominate the container until the contents become $[6, 6, 6, 6, 6]$. Finally, the initial five cells start to appear again, slowly taking over from the cell of type 6, which is then not present any longer at the very end of the composition.

Figures 5.13–5.17 show five different excerpts during the first stage of the transformations, with each cell type being highlight by a different colour: blue for cell 1, red for cell 2, green for cell 3, yellow for cell 4, and purple for cell 5.

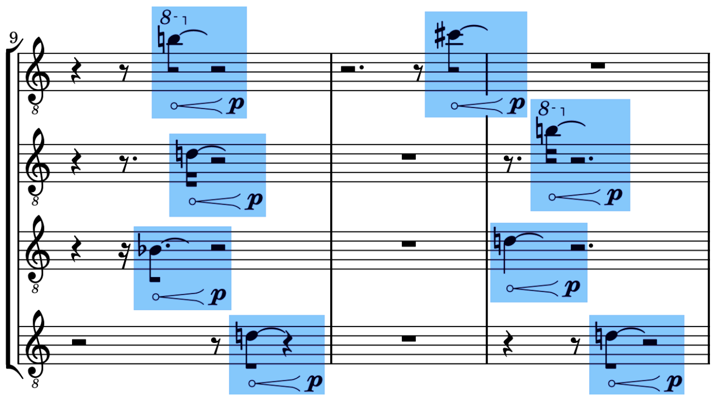

**Figure 5.13:** Excerpt of *Cartography #7*, measure 9, with highlighted cell types

Other than the container controlling these musical cells, only two other

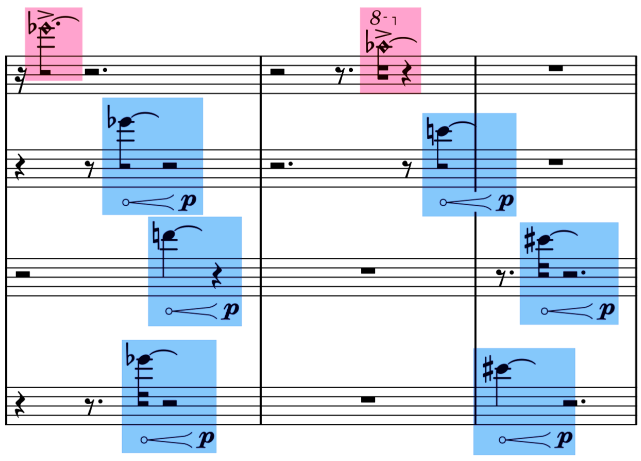

**Figure 5.14:** Excerpt of *Cartography #7*, measure 22, with highlighted cell types

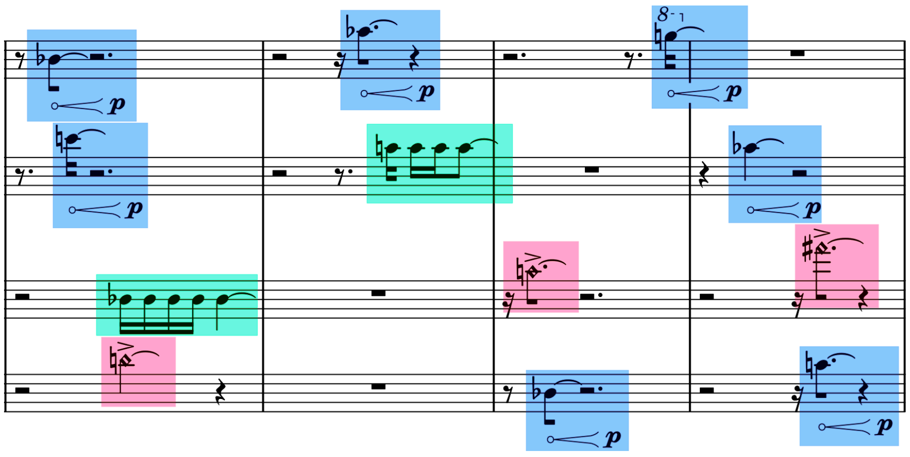

**Figure 5.15:** Excerpt of *Cartography #7*, measure 37, with highlighted cell types

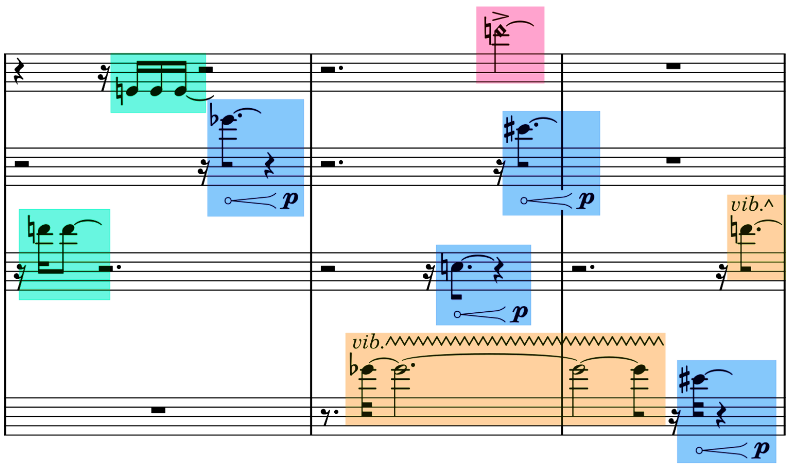

**Figure 5.16:** Excerpt of *Cartography #7*, measure 59, with highlighted cell types

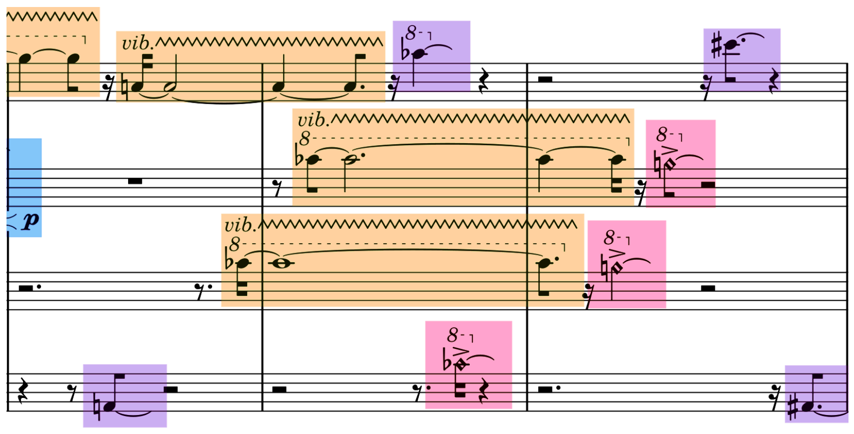

**Figure 5.17:** Excerpt of *Cartography #7*, measure 88, with highlighted cell types

containers control the pitch and duration parameters. The pitch container is initialised with [C6, B5, B♭5, A5, A♭5, G5, F♯5] and is affected by two processes: first, pitch classes are manipulated using the transformation $[a, b, c, d, e, f, g] \rightarrow [b, c, d, e, f, g, h]$, where $h = (g - 1) \bmod 12$. An octave transposition value is then uniformly randomly selected, respecting the range constraint of E3–E6. The pitch container is in constant transformation, with the process described above happening at each and every measure. Selected pitches are then imposed to whatever cell is selected at a given moment in the composition.

Durations are given in number of semiquavers and dictate the distance between consecutive starting points of cells. The duration container is initialised with $[28, 27, 26, 25, 24]$ and has three stages of transformations. The first thirteen transformations are given by $[a, b, c, d, e] \rightarrow [a-1, b-1, c-1, d-1, e-1]$ so that the distance between attack points becomes smaller and the texture denser. The second stage starts with the fourteenth transformation and lasts until the coda and is given by $[a, b, c, d, e] \rightarrow [a+1, b+1, c+1, d+1, e+1]$, which will increase the distance between attack points. Finally, the short coda starts one measure after the last active note ends and uses a fixed set $[24, 24, 24, 24, 24]$ (that is, all attacks occur at every one and half $\frac{4}{4}$ measures). The duration container changes every six measures.

Besides the initial states of the already-mentioned containers, this piece makes use of the following other constraints:

1. Number of pitch transformations: 217 ($216 + 1$ for the coda).
2. Number of duration transformations: 19 ($18 + 1$ for the coda).
3. Pitch range: E3–E6.
4. If a pitch below D4 is selected, the harmonic cell cannot be selected.
5. A semiquaver rest is added after a vibrato note.

### 5.1.8 *Cartography #8*

*Cartography #8* (2018), for flute, soprano sax, violin, violoncello, and accordion, was the first piece in which I made use of the looping window mental model. In the case of this piece, the looping window consists of a straightforward additive method for generating windows of repeated measures, which can be understood as a looping window moving forwards. First, the algorithm generates the input music lasting for 40 measures, created without any repetition process. Each of these measures contains a single note or rest per instrument, so the whole composition is written in a homophonic texture. Once this input music is created, the algorithm applies a three-stage process to generate the final score. The first stage starts with a looping window whose only element is measure one; while this window consists of less than 10 measures, it will append the next measure at each iteration. Once the window is 10 measures long, the second stage starts, and the window moves forwards by one measure in relation to the input music. Once the window reaches the last measure (number 40), it will enter the third stage, which is the opposite of the first one: starting with a looping window of 10 elements, the algorithm discards the first measure of the window at each iteration until only a single element is left. These three stages can be visualised as:

$$
\begin{aligned}
&[1] \to [1,2] \to [1,2,3] \to \cdots \to [1,2,3,\ldots,9,10] \to \\
\to &[2,3,4,\ldots,10,11] \to [3,4,5,\ldots,11,12] \to \cdots \to [31,32,33,\ldots,39,40] \to \\
\to &[32,33,34,\ldots,39,40] \to [33,34,35,\ldots,39,40] \to \cdots \to [39,40] \to [40]
\end{aligned}
$$

This process results in the following sequence of measure numbers:[14]

---

1   1 2   1 2 3   1 2 3 4   1 2 3 4 5   1 2 3 4 5 6   1 2 3 4 5 6 7   1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9   1 2 3 4 5 6 7 8 9 10   2 3 4 5 6 7 8 9 10 11   3 4 5 6 7 8 9 10
11 12   4 5 6 7 8 9 10 11 12 13   5 6 7 8 9 10 11 12 13 14   6 7 8 9 10 11 12 13
14 15   7 8 9 10 11 12 13 14 15 16   8 9 10 11 12 13 14 15 16 17   9 10 11 12 13
14 15 16 17 18   10 11 12 13 14 15 16 17 18 19   11 12 13 14 15 16 17 18 19 20
12 13 14 15 16 17 18 19 20 21   13 14 15 16 17 18 19 20 21 22   14 15 16 17 18
19 20 21 22 23   15 16 17 18 19 20 21 22 23 24   16 17 18 19 20 21 22 23 24 25
17 18 19 20 21 22 23 24 25 26   18 19 20 21 22 23 24 25 26 27   19 20 21 22 23
24 25 26 27 28   20 21 22 23 24 25 26 27 28 29   21 22 23 24 25 26 27 28 29 30

[14] Extra-wide spaces were added between each iteration for a better visualisation of the process.

22 23 24 25 26 27 28 29 30 31   23 24 25 26 27 28 29 30 31 32   24 25 26 27 28 29 30 31 32 33   25 26 27 28 29 30 31 32 33 34   26 27 28 29 30 31 32 33 34 35 27 28 29 30 31 32 33 34 35 36   28 29 30 31 32 33 34 35 36 37   29 30 31 32 33 34 35 36 37 38   30 31 32 33 34 35 36 37 38 39   31 32 33 34 35 36 37 38 39 40 32 33 34 35 36 37 38 39 40   33 34 35 36 37 38 39 40   34 35 36 37 38 39 40   35 36 37 38 39 40   36 37 38 39 40   37 38 39 40   38 39 40   39 40   40

---

The length of this looping window poses a challenge to the listener's memory as it can consist of up to 10 measures—totalling up to 20 seconds, longer than the commonly accepted upper range of our short-term memory, which is around 10–12 seconds (Snyder, 2000, pp. 49–51; Huron, 2006, p. 228). Since, at each step, only one new element is added, and only one element is removed, the changes between each iteration of the window happen very gradually. The monophonic texture helps amplify the disorientation caused by this process, creating a constant sense of *déjà vu*. Emergent structures arise from the sequence of measures, particularly at the borders between windows where material that is not consecutive in the input music is placed side by side. The simplicity of the material used, together with the straightforward texture and soft dynamics, help to obfuscate these points further and, in turn, increase the sense of slippage of this work. This, coupled with the long looping window, makes this otherwise easy-to-grasp process substantially less evident to the listeners. Figure 5.18 shows an excerpt of the first fifteen measures of this composition, with double measure lines added between the transformations described above for extra clarity.

This piece also uses containers to control pitch, dynamics, hairpins, timbre, measure length, and the number of instruments playing in a given measure. Pitches, measure lengths, and dynamics use transformations of the type 'drop first element and append new one', while the containers of dynamics, hairpins, and the number of active instruments are fixed throughout the piece.

Some of the most relevant constraints are:

1. Total number of measures (pre-loop): 40
2. Note durations: all notes last for a whole measure
3. Pitch range: A♭4–E7
4. Condition: at least one pitch in the set should be equal to or below C6 (to ensure that all instruments have at least one available pitch)
5. Highest pitches per instrument: flute C7, soprano saxophone E♭6, violin E7 (harmonics from E6 and above), violoncello E7 (harmonics from G5 and above) and accordion C7
6. Hairpin range: hairpins always lead to a dynamic one step above/below

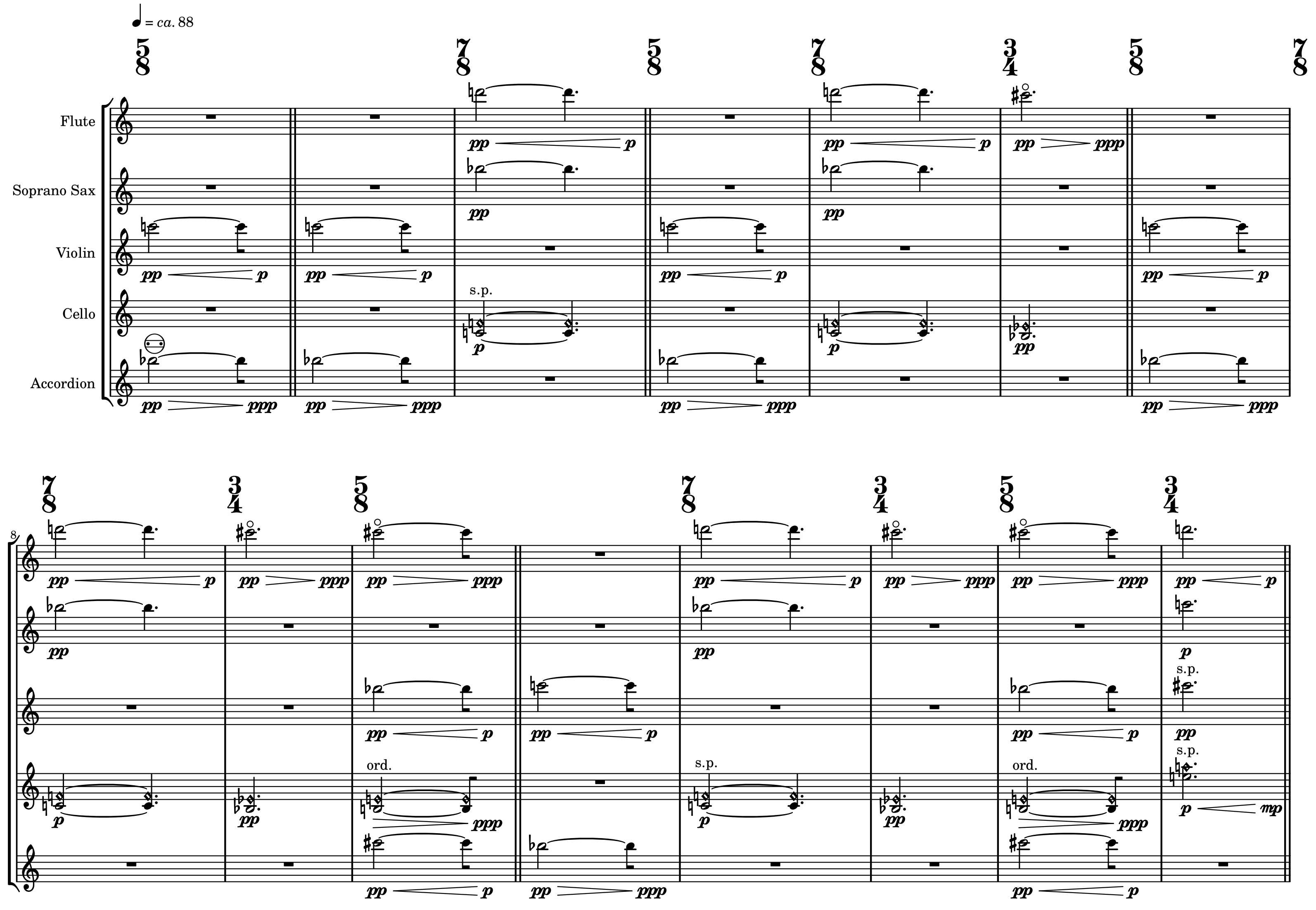

**Figure 5.18:** Excerpt of *Cartography #8* with double bar lines between looping window instances

the initial one

The algorithm uses a fixed container of length two containing two options for timbre, which are instrument-specific:

| | Timbres | |
|---|---|---|
| Instrument | Option 1 | Option 2 |
| Flute | ord. | flageolet |
| Soprano saxophone | ord. | bisbigliando |
| Violin | ord. | sul ponticello |
| Cello | ord. | sul ponticello |
| Accordion | 8'+8' | 8'+8'+4' |

**Table 5.7:** List of timbral options used in *Cartography #8*

The number of active instruments in a measure and the hairpin parameter both use fixed containers, $[5, 4, 3, 2, 1, 0]$ and [none, *cresc.*, *dim.*], respectively.

Other parameters shift continuously throughout the piece. Measure lengths continuously shift from $[\frac{7}{8}, \frac{6}{8}, \frac{5}{8}]$ to $[\frac{3}{8}, \frac{2}{8}, \frac{1}{8}]$.[15] Dynamics shift from [***pp***, ***p***] to [***mp***, ***mf***].

The contents of the pitch container are affected by two processes: first, pitch classes are transformed using $[a, b, c, d, e, f] \to [b, c, d, e, f, g]$, where $g = (f + 1) \bmod 12$. An octave transposition is then uniformly randomly selected, respecting the range constraint of A♭4–E7 as well as the conditional constraint that at least one pitch in the container must be equal to or below C6.

### 5.1.9 *Cartography #9*

The subsequent work in this series, *Cartography #9* (2018), for clarinet, viola, vibraphone, and piano, builds upon the ideas explored in *Cartography #8*. In this work, I once again employed a looping window, but this time the window has a fixed duration (sixteenth semiquavers) and shifts by a fixed unit (one semiquaver), as opposed to the measures with arbitrary sizes used as windows in *Cartography #8*. Similar to the previous work, this composition is also implemented by first generating its input music that is then processed by the looping window procedure. Despite sharing a similar technique, the musical result of *Cartography #9* is substantially different than that of the preceding work: not only there are

[15]In the final score, these ratios are simplified to a crotchet denominator whenever possible. For instance, a $\frac{6}{8}$ time signature is notated as $\frac{3}{4}$.

no literal repetitions of any measures, but also the looping window itself has now a fixed length of approximately 3 seconds, which happens to be within the typical range for our short-term memory.[16] Therefore, the slippage in this piece comes mostly from the disorienting near-repetitions of fragile materials and not by burdening our short-term memory as in *Cartography #8*'s 20-second long windows.

In this work, the looping window is moved forward by a unit and slices all notes on its borders, with the resulting window becoming a new measure in the final score. This means that notes whose attack points have already left the looping window but are still sustained within it will receive a new attack on the first beat of the output measure (see Figure 5.19 for a concrete example of this process). This results in the effect that notes seem to shrink towards the left border of the window, and new notes expand from the right border. This, in turn, creates new relationships of material at these borders since two notes can be far apart in the input music but happen to sound adjacent to one another due to this looping process. This, in turn, can result in emergent structures, particularly when these notes are grouped together by our ears due to their temporal and pitch proximities.

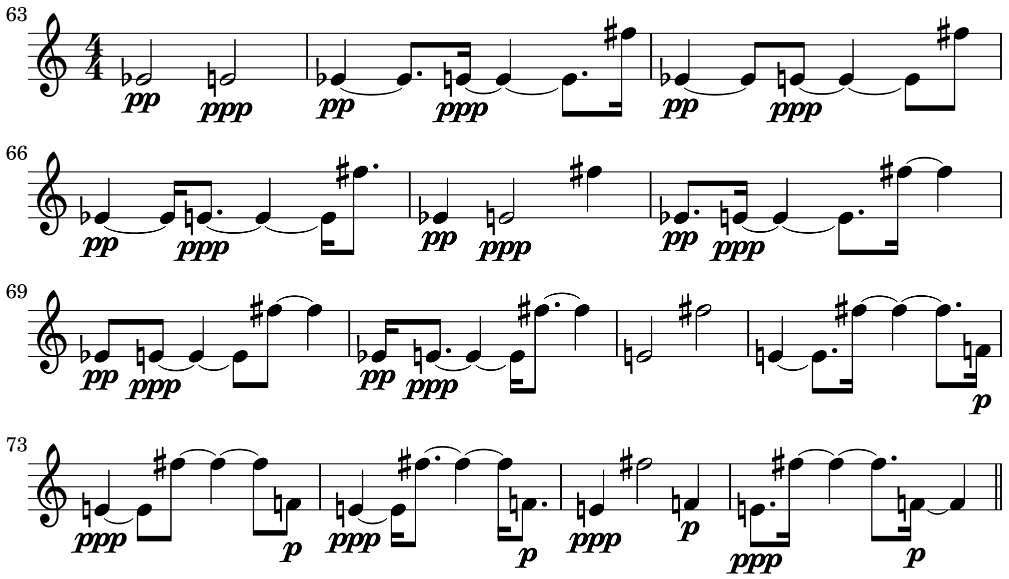

**Figure 5.19:** Excerpt from the piano part of *Cartography #9*, measures 63–76

[16]Lang argues that composing with loops lasting between 0.2 to 7 seconds is particularly effective since this range matches our own short-term memory (2002, p. 4). Our short-term memory's precise upper range limit is context-dependent and will vary substantially according to it (Snyder, 2000, pp. 49–51; Huron, 2006, p. 228). Snyder states that the average upper limit is between 3 to 5 seconds but may occasionally reach up to 12 seconds in certain conditions (2000, p. 50).

Figure 5.20 illustrates these emergent relationships that appear at the border of consecutive looping windows. At the top, there are three snapshots of a looping window moving forwards by a semiquaver in relation to an arbitrary input material, with the resulting music shown at the bottom. The note relationships at the bar lines (marked with asterisks) are not present in the original non-looped source.[17]

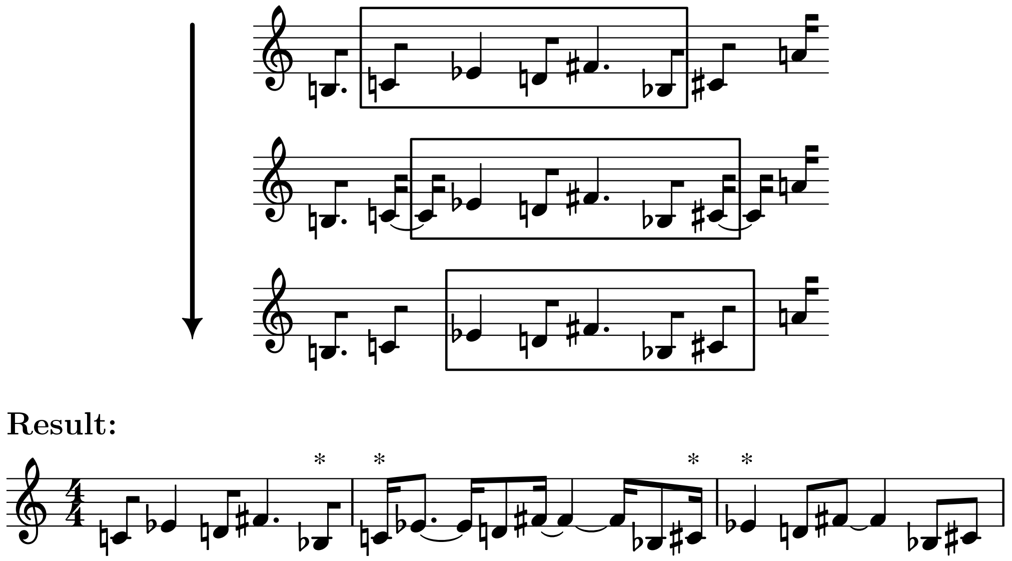

**Figure 5.20:** Example of emergent relationships formed at the looping window borders

The relatively short steps of one semiquaver taken by the looping window generate almost literal repetitions of material since each measure has nearly the same content as the preceding one. In fact, two consecutive measures will differ, at most, by two semiquavers: the one that left the looping window and the one that entered it. Often, pitches leaving and entering the window are the same, given the small pool of possibilities available in the container. The asymmetrical probability distribution of different indices will also affect this since some pitches are more likely to be selected than others. The result for the listener is often a state of *déjà vu*: one is constantly recognising structures that were previously heard but, slowly, these structures will slip away as the process is in constant flow. That is, despite its local static nature with very similar consecutive measures, the resulting music is always moving forwards, and any material will be gone

[17]In this example, the measure length happens to coincide with the looping window size (i.e. the $\frac{4}{4}$ measure and the window of 16 semiquavers have the same size). In my experience, this type of notation helps performers better visualise the musical process taking place, but it is by no means a requirement as the final music could easily be renotated with any combinations of time signatures.

after 16 iterations of the process (or about 48 seconds).

The instrumental constraints I used in this piece are crucial for creating a sense of disorientation. All instruments share very narrow ranges, with all pitches selected in the range F3–B♭5 and dynamics between ***ppp*** and ***mf***. This will create a somewhat uniform texture with the music of all four instruments being generated from the same algorithmic process. This piece also requires that both the piano and the vibraphone players hold down the sustain pedals of their instruments for the whole piece, creating a blurred landscape in which precise identification of patterns becomes somewhat challenging.

*Cartography #9* is one of the most straightforward pieces in this series when it comes to the handling of parameters. Pitches, dynamics, and durations all follow very similar processes as the ones previously discussed in this chapter. The pitch container is initialised with [C4, C♯4, D4, E♭4, E4, F4] and is affected by two processes: first, pitch classes are transformed using the transformation $[a, b, c, d, e, f] \rightarrow [b, c, d, e, f, g]$, where $g = (f + 1) \bmod 12$. An octave transposition is then uniformly randomly selected, respecting the range constraint of F3–B♭5. The pitch container is transformed once per measure of the input music. The duration container changes more slowly, once every two measures. This container is initialised with $[10, 9, 8, 7, 6]$ and goes through the transformation $[a, b, c, d, e] \rightarrow [b, c, d, e, f]$, with $f = e - 1$. The dynamic container is fixed as [***ppp***, ***pp***, ***p***, ***mp***, ***mf***] and is linked to the duration container: if the duration element on an index $n$ is selected for a given note, then the dynamic element at the index $n$ of the dynamic container will also be applied to it. These two containers thus give a statistical preference to soft and long notes, and the link between them ensure that only the shortest available durations will have a slightly louder dynamic value.

### 5.1.10 *Cartography #11*

The principles behind *Cartography #11* (2018) for solo piano are, in many ways, quite similar to those of *Cartography #9*. Most noticeably, both compositions use a looping window of constant length that always moves forwards by a single unit, and the container transformations and parameter constraints employed are also quite similar. The primary methodological difference between these two works is rather subtle: instead of using notes that last for several 'shifting units' (i.e. the duration by which the looping window shifts at each iteration, which is a semiquaver for both pieces), all notes are notated as single semiquavers. On the surface, this difference may appear to be solely notational, particularly as the piano plays with the sustain pedal held down throughout the piece and, thus, will invariably extend those short notes into long decaying sounds. However, the consequence of this decision is that once an attack point has left the shifting window, that note will not receive a renewed attack as in *Cartography #9* since, by that point, it has no duration left inside the next looping window. Figure 5.21 illustrates the difference between these two processes using an arbitrary sequence of notes, with the *Cartography #9* approach notated on the top staff and the *#11* at the bottom.

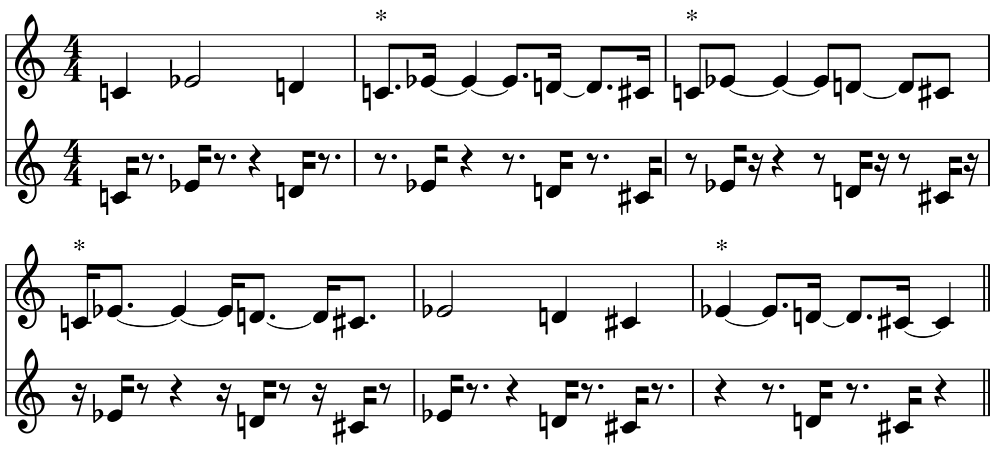

**Figure 5.21:** Comparison between the processes of *Cartographies #9* and *#11*, with the differences in attack points marked with an asterisk

The result is a substantially thinner texture than the previous compositions in this series. However, it is precisely for this lower density of notes that the boundaries of the looping windows become more blurred and, thus, more difficult to pinpoint. In the previous piece, *Cartography #9*, the renewed note attacks after bar lines made its window borders slightly more straightforward to be aurally located. This is contrasted with *Cartography #11*, in which looping windows can start with rests too. It is perhaps for this reason that this work also has a stronger tendency to display emergent structures that are aurally recognisable, more so than in any other piece addressed so far.

Other than increasing the number of emergent structures, this lack of renewed attack points after a looping window border results in the perception of literal repetition. In fact, while *Cartography #9* could not be notated using explicit loops with repetition bar lines, *Cartography #11* can be rewritten as a series of $\frac{15}{16}$ measures (i.e, $\frac{4}{4}$ minus a semiquaver), some of which are identical to others. Compare the original score shown in Figure 5.22 with Figure 5.23, which illustrates how the first ten measures could be renotated using explicit loops.

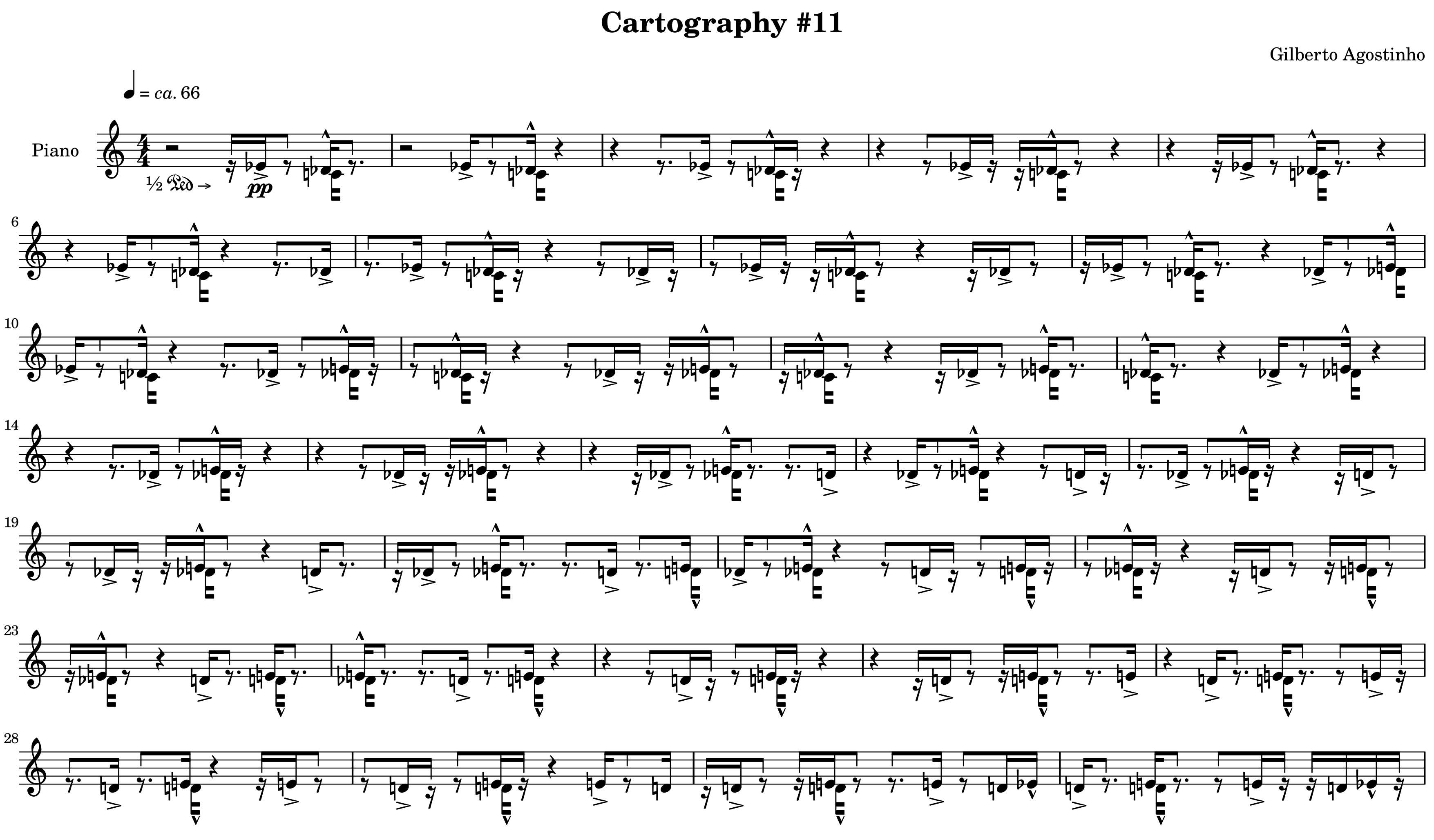

Figure 5.22: First page of *Cartography #11*

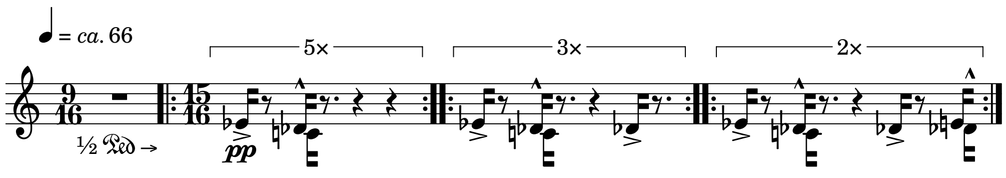

**Figure 5.23:** First ten measures of *Cartography #11* rewritten with explicit loops

The differences between these two ways of notating the same music demonstrate the contrast between the strict linear looping process and how the resulting music is effectively perceived by the listener. In other words, the music is created using a looping window that moves forward every bar, which can be clearly observed in the score shown in Figure 5.22; however, our ears will group identical structures together and interpret them as literal repetitions, resulting in a far less-linear experience, as shown Figure 5.23. This discrepancy between process and perception is vital for creating a disorienting listening experience that leads to high levels of slippage. This discrepancy is also a crucial contributor for the appearance of emergent structures. We perceive this music as being made of blocks with literal repetition, which helps mask the looping window's border that is in constant movement. It is often at these borders that new emergent structures appear, as these are the locations in which distinct moments of the input music find themselves side by side, creating new relationships.

The containers, transformations, and constraints in this composition are, once again, relatively straightforward. The length of the input music that is first generated by the algorithm is eight measures, the pitch range is set to C3–C7, and the dynamic level is set to a constant ***pp***, with variations in loudness being notated using articulations. The pitch container is initialised with a chromatic pitch field of [C4, C♯4, D4, E♭4, E4, F4] and follows an identical transformation process as in *Cartography #9* (described in Subsection 5.1.9). The same is true for the duration container, which only differs for its initialisation values of $[12, 11, 10, 9, 8]$.

While this piece is notated using a single dynamic level of ***pp***, it uses articulations to notate changes in loudness. Articulations are not controlled by a container but are actually the result of the superimposition of three voices, each with their own dynamic level (represented by no articulation, marcato articulation, and martellato articulation). The algorithm thus formalises this work as a trio of highly rarefied densities, which are later merged into a single staff. Despite using three levels of loudness, the overall dynamic level of the piece is very quiet throughout, creating a very similar aesthetic world to that of *Cartography #9*.

## 5.2 *and thereafter they shape us*

*and thereafter they shape us* (2019), for solo violoncello, is the first instrumental work I composed after my *Cartographies* series. *Cartographies* heavily influenced this new piece in terms of composition technique as well as informed its aesthetic world. The title of this work comes from an article by Culkin on the topic of media theory and the work of Marshall McLuhan. Culkin (1967, p. 70) writes, '*Life imitates art.* We shape our tools and thereafter they shape us.' This notion that the tools we choose will also shape our artistic approach is crucial for algorithmic composers such as myself. According to Hamman, technology becomes an integral part of the final artwork, embedding itself in the result and informing its aesthetics (2000c, pp. 7–8). Our tools also influence how we *conceptualise* the act of composing; that is, they are a fundamental part of the framework within which we operate, and they can suggest not only technical solutions to problems but also shape our compositional ideas themselves.

In this work, I set out to further explore some of the themes that have been the focus of the *Cartography* series, particularly in relation to musical repetition and looping. Due to its own nature, repetition is highly suitable for algorithmic investigation, as it can be easily formalised with strict processes. This work uses a similar technique to the looping window of *Cartographies*, but instead, it applies these techniques to high-level musical objects, as opposed to single notes generated with a parametric approach. Similarly to the later pieces in *Cartographies*, new relationships of materials can emerge at the intersection of consecutive instances of a looping window, challenging our ability to exactly follow the process taking place. This is particularly pronounced in this work due to its slow pace and extremely quiet dynamics.

On the surface, this is not a very challenging work for the cellist, as it is constituted almost solely of long soft non-vibrato notes with occasional artificial and natural harmonics. However, it is precisely for these characteristics that this piece requires a high level of concentration and commitment from the performer, as due to its transparency, it leaves them with 'nowhere to hide'. *and thereafter they shape us* is about tension in quietness, in which the fragile sounds are close to their breaking point at every moment. The listening experience also demands a high degree of concentration from the audience, as the soft repetitive structures constantly challenge our memory. While pieces such as *Cartography #8* test our memory with its long patterns, the process here is reversed: the patterns are moderately short in terms of the total number of elements (four measures long for the looping process and six measures long for the shuffling process), but the cells used do not stand out from their surroundings. When listening to this work, we are often in doubt if we hear new materials altogether or identical repetitions of previously heard cells.

Most of the techniques in this piece are derived from those used in *Cartographies*. Its use of musical cells as building blocks is reminiscent of *Cartography #7* (as well as the use of a short coda section); the looping window mental model is applied similarly to *#8*; the use of multiple sections (looping–shuffling–coda) can be related to the multi-movement structure of *#4*; the fragile, thin texture is evocative of the sound world of *#11*. Despite all these observations, a significant technical distinction between this piece and those in *Cartographies* is that it does not employ the container mental model but rather a simple uniform distribution. With this, there are no hierarchies of musical cells as those found in *Cartography #7*, and the shape of the composition is dictated by its three sections—each with its own algorithmic process.

The musical cells that make up this work are defined through generalised instructions, that is, recipes for creating measures. Some use a single specific duration, while others use durations chosen from within a given range. Patterns of rests, notes, and grace notes can be pre-defined for some cells, as well as the tuplet ratio that forms these structures. Some cells require specific techniques (natural harmonics, artificial harmonics, use of chords), while others are open to any technique from the list of all possibilities. Table 5.8 lists all of these cells and their characteristics. In this table, parenthesised items represent sequences of elements while square brackets contain options for uniformly distributed choices.

Structurally, the piece is written in three seamless sections that very gently merge into one another. The first and longest section uses a looping window as its primary algorithmic mental model, while the second applies a shuffling method to musical cells. The piece finishes with a short coda made out only of a single type of cell.

Although the looping section uses a shorter looping window in terms of the number of elements, the total length in time (about 15 seconds on average) is similar to that of *Cartography #8*. Perhaps somewhat surprisingly, the borders created by the looping window are substantially more challenging to follow aurally in this work. The similarity between the cells coupled with the small number of elements (only four per window) makes the relationships between cells somewhat unpredictable. To illustrate this point, consider the following sequence of elements:

$$A \quad B \quad C \quad D \quad E \quad F \quad G$$

Applying a looping window of four elements and with a step size of one

| Cell Label | Time Signature | Tuplet Ratio | Structure | Rhythm Combinations | Playing Technique |
|---|---|---|---|---|---|
| A | $\frac{4}{4}$ | 5:4 | (rest, note) | [(♩ 𝅝), (𝅗𝅥 𝅗𝅥.), (𝅗𝅥. 𝅗𝅥)] | any |
| B | $\frac{3}{4}$ | 4:3 | (rest, note) | [(♩ 𝅗𝅥.), (𝅗𝅥 𝅗𝅥)] | any |
| C | $\frac{5}{8}$ | 1:1 | (note, note) | [(♩ ♩.), (♩. ♩)] | any |
| D | [$\frac{3}{4}$, $\frac{4}{4}$, $\frac{5}{4}$, $\frac{6}{4}$] | 1:1 | [(acciaccatura note), (acciaccatura note rest)] | if rest then last beat 𝄽, else note takes full measure | ordinario |
| E | $\frac{7}{8}$ | 1:1 | (note, note) | [(𝅗𝅥 ♩.), (♩. 𝅗𝅥)] | any |
| F | [$\frac{3}{4}$, $\frac{4}{4}$, $\frac{5}{4}$, $\frac{6}{4}$] | 1:1 | (note, note) | equal halves | natural harmonics |
| G | $\frac{4}{4}$ | 3:2 | ([note, rest], [note, rest], [note, rest]), with at least one note selected | (𝅗𝅥 𝅗𝅥 𝅗𝅥) | any |
| H | [$\frac{4}{4}$, $\frac{5}{4}$, $\frac{6}{4}$, $\frac{7}{4}$] | 1:1 | (rest, note, rest) | (𝄽, note, 𝄽) | artificial harmonics *cresc.* from *al niente* |
| Coda Cell | [$\frac{4}{4}$, $\frac{5}{4}$, $\frac{6}{4}$] | 1:1 | (note) or (note, rest), with rest only used in $\frac{6}{4}$ | if rest then last beat 𝄽, else note takes full measure | ordinario, but only using intervals: 2m, 7M, or 9m |

**Table 5.8:** List of cells used in *and thereafter they shape us*; parenthesis represent sequences and square brackets represent uniformly distributed choices

results in the following sequence:

$$\begin{array}{ccccccc} A & B & C & D & & & \\ & B & C & D & E & & \\ & & C & D & E & F & \\ & & & D & E & F & G \end{array}$$

This same pattern above can be rewritten as a linear sequence of elements:

$$A \quad B \quad C \quad D \quad B \quad C \quad D \quad E \quad C \quad D \quad E \quad F \quad D \quad E \quad F \quad G$$

In the example above, the element $D$ is present in all four iterations of this looping window (the highest number of times an element can appear in a looping window of length four). Yet, this element appears surrounded by different cells in three of its four appearances. These patterns of three elements are:

$$\begin{array}{ccc} C & D & B \\ C & D & E \\ C & D & E \\ F & D & E \end{array}$$

In other words, element $D$ appears in three distinct structural contexts. This factor applies to every single element in this looping window, and therefore it becomes no surprise that such a short number of elements coupled with a long looping window creates such demands to our memory and capacity to contextualise different elements.

Figure 5.24 shows this very process applied to the opening measures of this work. The looping window starts with a single element and grows at the rate of one more at every iteration, up to its maximum size of four, at which point it starts moving forwards. The result of this operation is shown in Figure 5.25, which contains an excerpt of the initial 18 measures of this composition. This figure has double bar lines added between consecutive instances of the looping window for clearer visualisation.

In the second section of this work, the looping window is replaced by a shuffling algorithm, which uses a fixed window of elements but, at each iteration, randomly swaps the positions of two of them. This section is made out of four subsections. At each of those, the algorithm first generates a window with six new cells, which is then shuffled four times. After four iterations of the shuffling process, the cells are discarded, and the next subsection starts with a new window of six random cells and a new round of shuffling. Figure 5.26 shows a diagram with the first four iterations of this process.

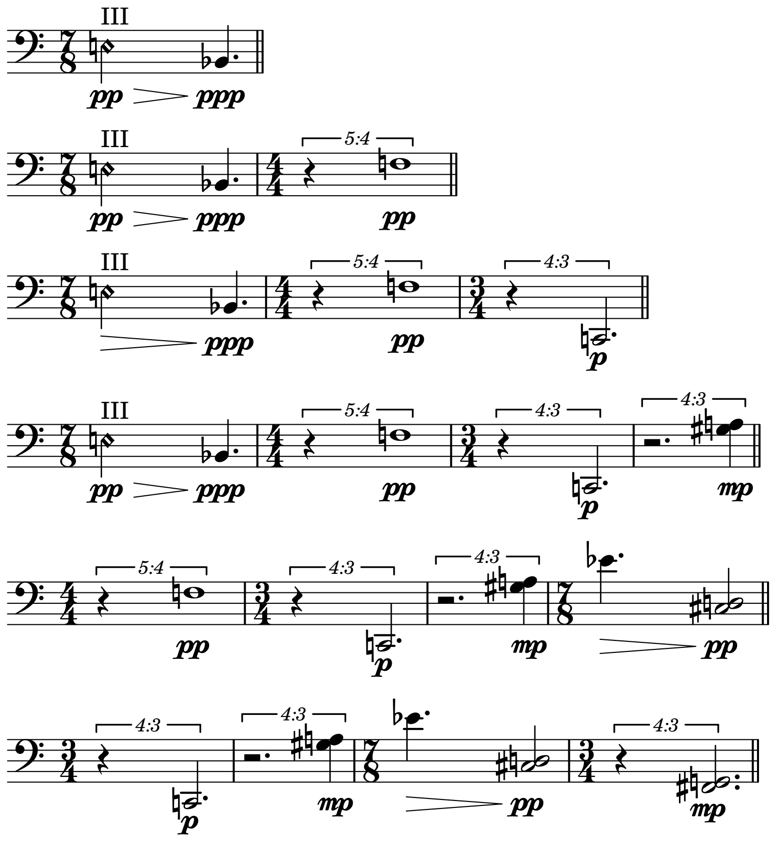

**Figure 5.24:** Looping process at the start of *and thereafter they shape us*

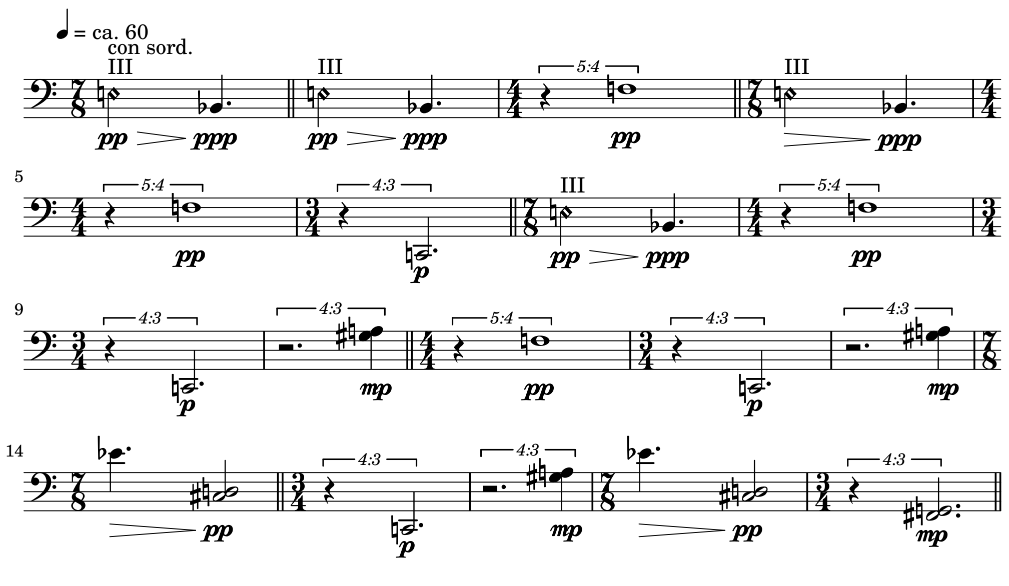

**Figure 5.25:** First eighteen measures of *and thereafter they shape us* with double bar lines between looping window instances

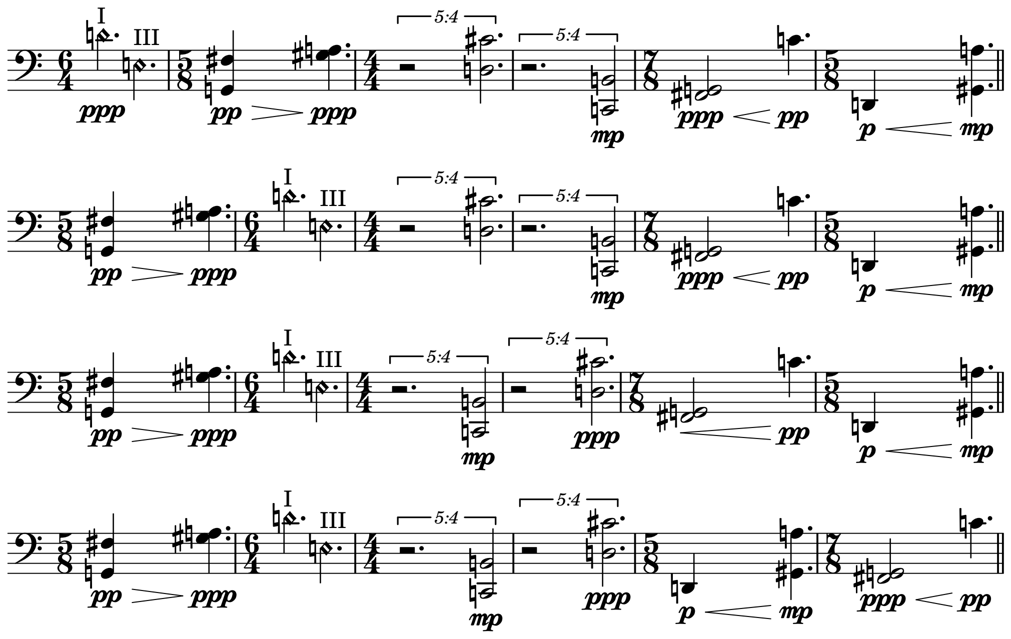

**Figure 5.26:** Shuffling process of *and thereafter they shape us*, measures 68–93

Similar to the looping window section, the shuffling section also poses a demanding perceptual challenge for the listener. At each subsection, the total material stays the same (i.e. the same six cells), but due to its partial position changes, some of its inner relations (though not all of them) change with every iteration. The impression is, again, of a *déjà vu*: the musical cells become familiar through repetition, but their intra-relationships are difficult to grasp.

The piece ends with a short quiet coda. This is the moment when the tension reaches a breaking point through an even more extreme use of fragility: long chords with open strings are played sul tasto at ***pppp*** and non-vibrato. This is a challenging section for the performer, as sound production may become unreliable at this soft level, making playing these chords uniformly very demanding.

## 5.3 *adrift*

After working with looping windows and containers in the later *Cartographies* as well as in *and thereafter they shape us*, I wanted to investigate what other mental models could be employed in the exploration of musical repetition. Using a metaphor that borrows from the electroacoustic studio, I wanted to explore how the notions of fading and, later, crossfading could be applied to loops of notated material. I aimed to morph one musical cell into another as smoothly as possible without employing dynamics as the fading parameter.

These ideas then gave origin to the mental model I refer to simply as ‘fader’. Given a musical cell $A$ with a sufficient number of notes, I realised that removing one single note at every iteration of the process would result in the gradual deconstruction of its contents (a parallel notion to fading out an electroacoustic material), a process that can be represented by $A \rightarrow \varnothing$. The higher the number of pitches in the input cell—and the longer its duration too—the ‘smoother’ the fading process will appear to be. This happens because removing a single note of a cell with many notes will produce a substantially similar new cell, to the point which our memory is not sure whether any change has occurred at all. Therefore, smoothness is achieved not by literal continuity—after all, this is a discrete process—but through similarity between iterations. For the listener, it ‘feels’ as if the material is continuously fading, slowly dissolving into nothing.

Similarly, notes can also be added one by one at each iteration, reversing the process above and resulting in a musical cell fading *in*. When starting from silence, this process can be represented by $\varnothing \rightarrow B$. Our perception of this process will once again become smoother the higher the number of pitches in the cell, as it becomes increasingly difficult for our ears to pick up the difference between consecutive iterations. In the case of fading in from silence, this means that the start of a process is easier to grasp. This is the opposite of the fading out process, which becomes more evident at the end when there are just a few notes left.

These two processes can then be alternately applied to two simultaneous layers of distinct cells, creating an interwoven effect similar to a crossfading:

$$\begin{array}{rl} 1: & A \rightarrow \varnothing \rightarrow C \rightarrow \varnothing \\ 2: & \varnothing \rightarrow B \rightarrow \varnothing \rightarrow D \\ \hline \text{combined:} & A \rightarrow B \rightarrow C \rightarrow D \end{array}$$

The resulting effect of such processes can be visualised in Figure 5.27, which contains a diagram of the crossfading principle applied to multiple cells, each represented by a unique colour, divided into two simultaneous layers. The combined aural result from these two layers alternating between fading in and out can then be visualised in Figure 5.28.

The starting point of the fading in process will tend to immediately attract the listener’s full attention since, out of silence, a new sound emerges. In fact, this property can actually be employed to create a type of auditory misdirection: when a new note suddenly appears, our attention can get focused on just one of the two layers, allowing the other to continue its fading out process unnoticed. Thus, the combined fading processes often lead to a new listening mode, one

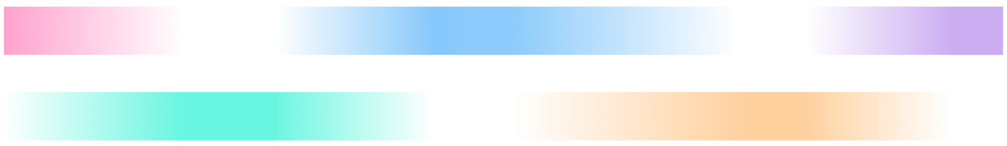

**Figure 5.27:** Diagram showing the crossfading principle applied to multiple cells in two simultaneous layers

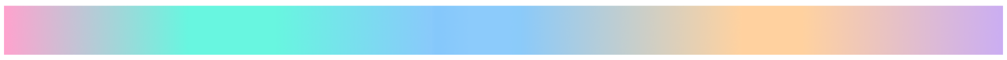

**Figure 5.28:** Diagram showing the effective result from two layers crossfading in alternation

that constantly shifts our attention between two opposing entities: once we focus on the layer that has notes appearing, the other layer's slow disappearance will go somewhat unnoticed; but if our attention shifts to the disappearing material, notes suddenly seem to have been added to the other process out of nowhere. It is thus the juxtaposition of these two contrary processes, pushing our attention and memory capacities back and forth, that creates the perceptual tension upon which this mental model relies.

These are the ideas that gave origin to *adrift* (2020) for two pianos. In this work, I use faders to gradually morph between several pre-composed musical cells.[18] Figure 5.29 shows an excerpt of this work (measures 256–268), where material played by the first piano gradually dissolves away while the second piano's material gains more and more notes. At each iteration of the process, either a note is removed from the cell fading out or added to the cell fading in. The process also has a very low chance of repeating a measure without altering either cell, further contributing to the disorientation and uncertainty caused by this music. The overall process thus consists of two simultaneous but contrary subprocesses; when a change between iterations is too subtle to be picked up by our ears, one is often left wondering what has changed from the previous iteration—if anything.

By using two pianos and assigning one layer to each of them, the physical space also becomes a property of the algorithmic process: given that consecutive cells alternate between the two pianos, they are not only fading in and out through time but also shifting across the physical performance space in alternation. Although the two pianos can be considered as sound sources located in two fixed

[18]The mental models used in this work are implemented in my Auxjad library as the classes `auxjad.Fader`, a single fader that either fades in or out, and `auxjad.CrossFader`, which is made up of two faders of opposite types.

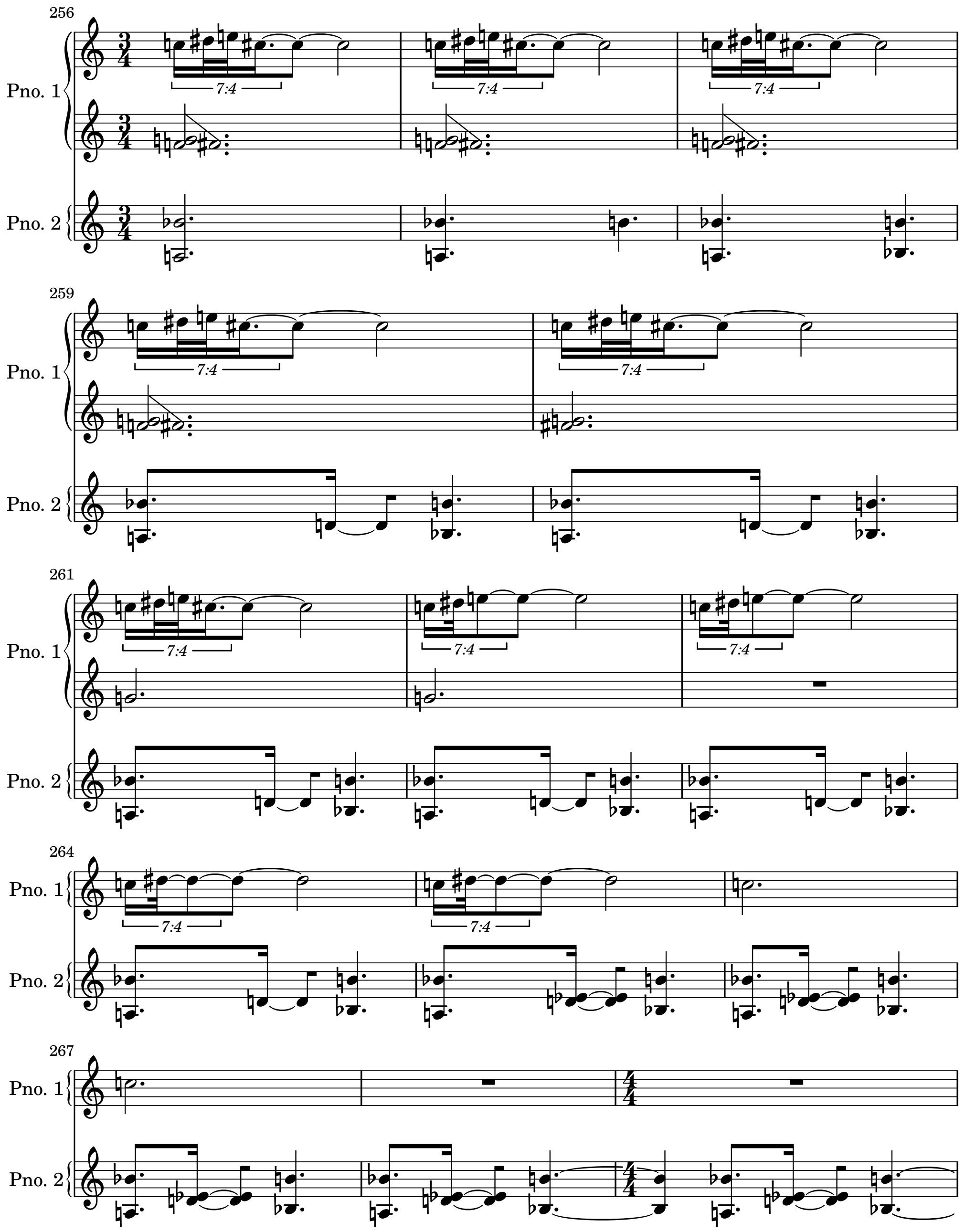

**Figure 5.29:** Excerpt of *adrift*, measures 256–268

(and separate) points in space, the density of notes played by each instrument is constantly and gradually changing through the algorithmic process. As a result, the 'spatial presence' of each layer is slowly changing over time, and so is their combined presence, which is constantly moving back and forth between two points. This effect resembles a continuous stereo panning process, borrowing another term from the electroacoustic studio.

As the title of the composition suggests, my goal was to create a listening experience of being adrift, with the listener being carried around aimlessly by the drift of the algorithmic process. Coupled with the fading and spatial processes described above, the type of material used as input for the algorithm is crucial for achieving the desired sonic result. Figure 5.30 shows all twenty cells used in this work. When composing these cells, I tried to emphasise highly chromatic material, with virtually all vertical moments being made out of cluster-like chords. This type of chord adds a further challenge to the listener's memory since adding or removing a single pitch in a dense and complex sound event is far more inconspicuous than adding a new pitch to a readily graspable material. Particularly when a pitch is removed or added at the middle of such dense structures, the change in the overall sound can be so subtle as to become imperceptible, leaving the listener questioning whether a change has occurred at all. The soft dynamics used in all of these cells (all played at ***ppp***) and the continual blending of sounds created by the piano pedalling (depressed throughout the piece) further amplify the perceptual challenge faced by the listener.

When composing this material, I attempted to emphasise the 'rhythmic dissonance' between consecutive cells by using tuplets of different ratios and different offsets (created by initial rests). This helped create a jarring effect between consecutive cells so that they become more clearly distinct from one another. In principle, it could be argued that such separation of the individual layers might cause the two simultaneous processes to become more distinct and, therefore, more graspable for the listener. In reality, this clear separation only adds to the cognitive demands imposed on the listener, who is tasked with a constant attempt of following two simultaneous but distinct processes moving in opposite directions. At certain points, the difference between consecutive measures is subtle enough that it is not perceivable. When this happens, the two processes fully blend together.

In order to achieve the psychoacoustic effect I was after, it was immediately evident to me that my material needed to be repeated multiple times before the fading out process started taking place. Familiarisation with the material is of crucial importance before the subtle modifications are applied by the process. These initial identical repetitions (generally of the order of around five loops) also have the role of creating a false sense of musical stasis, helping to blur the fading process even further: the starting point of the process becomes less noticeable and, when one realises, the material might have been already substantially modified. The process can thus sometimes seem to be temporarily suspended in

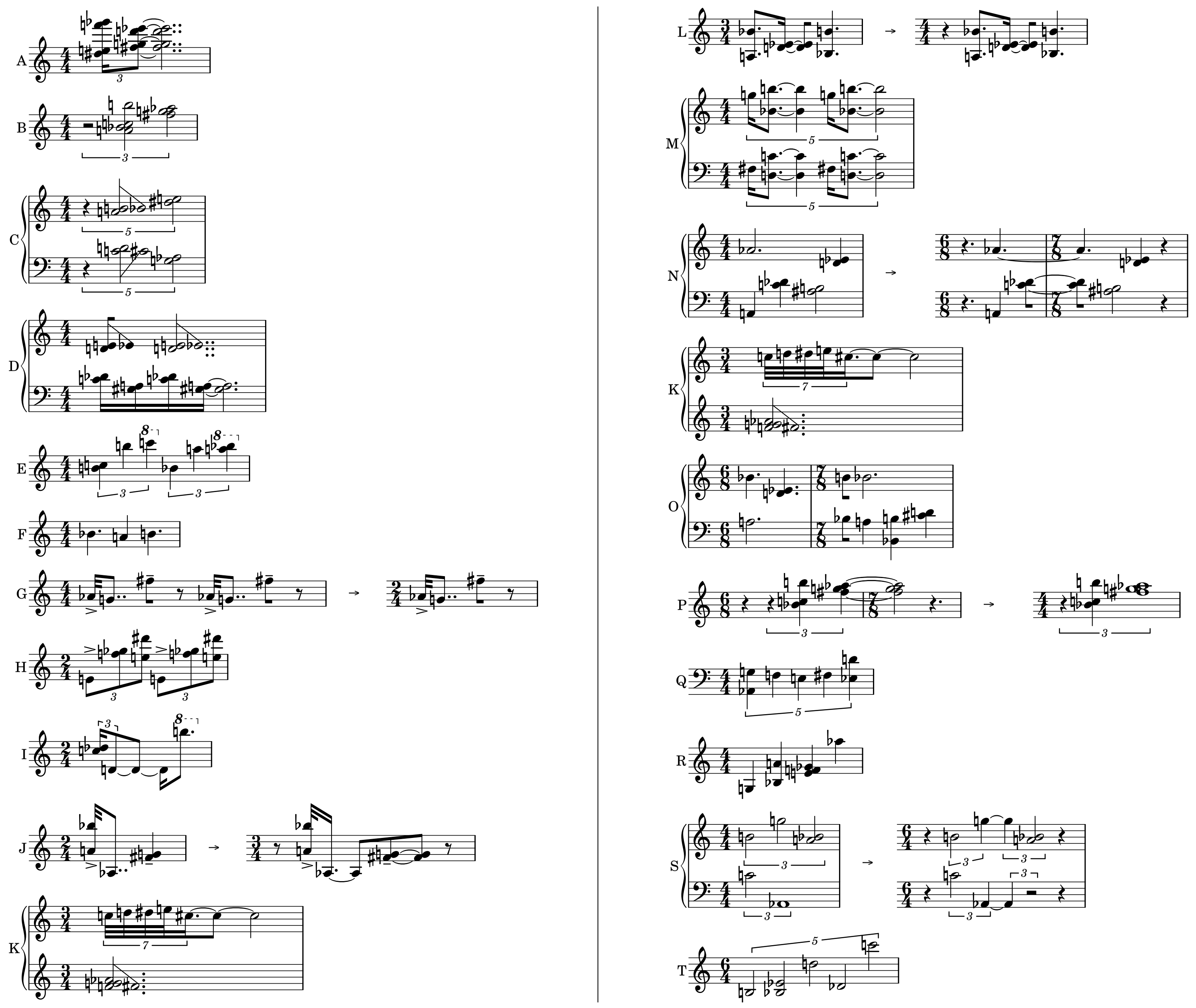

**Figure 5.30:** Material used in *adrift*

time, emphasising a mode of listening that could be described as ‘snapshots of memory’. This listening mode oscillates between the awareness of change and the illusion of stasis, oscillations that result in sudden shifts of perception. A visual representation of this notion is shown in Figure 5.31, which demonstrates how changes are perceived at fixed and discrete times, as opposed to the ‘absolute’ process shown in Figure 5.28. The resulting effect is that of a constantly rotating kaleidoscope whose changes are perceived suddenly but only at certain fixed moments in time.

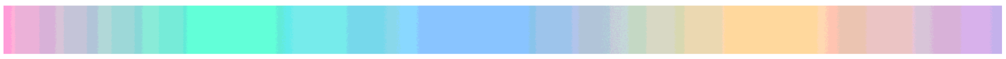

**Figure 5.31:** Diagram showing the perception of the crossfading process, where changes are perceived only once in a while

## 5.4 *what holds them together*

*what holds them together* (2020) is a piece written for portative organ, a small pipe organ that was particularly popular in the period ranging from the High Middle Ages until the Renaissance. The portative organ is played with a single hand on a keyboard while the other controls the bellows that supply the airflow for its operation. This piece is written for Ryszard Lubieniecki, who commissioned it as part of a project whose aim is to expand the new music repertoire of this instrument. I wrote this piece with Lubieniecki’s own portative organ in mind, as individual instruments can have unique characteristics, including pitch range, tuning, and bellows capacity.

The first question I wanted to address when approaching this composition was what type of musical material I would use. The portative organ I wrote for uses Pythagorean tuning, with A4 tuned at 494 Hz (about a whole tone above 440 Hz). The pitch range of this instrument is B3–A5, but it does not have a key for G♯5. At somewhat soft dynamics, the instrument can start to detune, and at extremely soft dynamics, the timbre becomes very unstable and the detuning very pronounced. The instrument must also be allowed to ‘breathe’ reasonably often so that the bellows can be refilled with air.[19] These unique characteristics of the instrument posed a substantial challenge to me, given the type of material I tend to work with.

[19]Unlike accordions whose bellows can produce airflow regardless of the direction of their movement, portative organs produce airflow in just one direction, thus needing short breathing points for refilling it.

My approach to this piece was to incorporate these instrumental idiosyncrasies as much as I could into the material itself. In terms of pitch, I decided to work solely with diatonic clusters, and thus the main parameters I worked with were cluster spread and keyboard position, as opposed to working with pitches per se. This allows for a very physical performance by the player, enabling them to use fists or parts of their hands to depress multiple keys at once. The diatonic nature of this material further emphasises this physicality since the 'white' keys[20] are of easier access for palm depression. Other than incorporating these performative aspects and this particular instrument's sonority in this piece, I also decided to only use diatonic notes as a way of creating a dialogue with the history of this instrument and the modal music often written for it. See Figure 5.32 for an excerpt of this work.

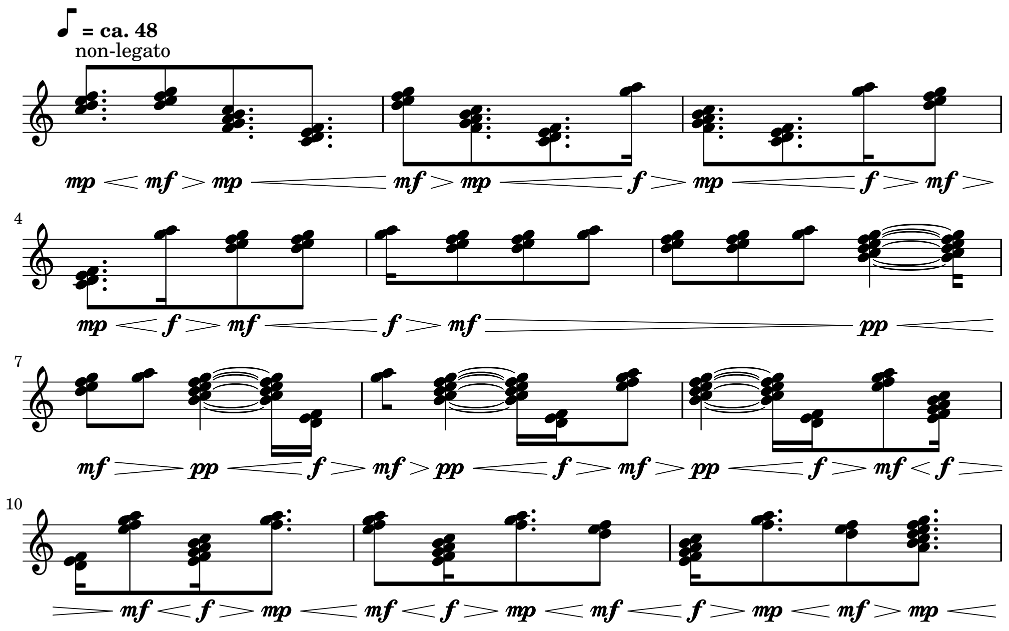

**Figure 5.32:** Excerpt of *what holds them together*, measures 1–12

The relatively frequent need for breathing demanded by the instrument also dictated the type of material I used. The number of simultaneously depressed keys is related to the amount of airflow used, which meant that a composition composed solely with clusters (each containing up to seven notes) would need to constantly deal with breathing points. My solution was to incorporate breathing as an intrinsic part of the material itself: most clusters are to be played non-legato, allowing the player to refill the bellows after each sonic event. I framed

[20]I.e. white keys as in the traditional piano keyboard, although the specific portative organ I wrote for had no difference in colouration between the natural and the sharp keys.

this need not merely as a practical requirement but as an essential element that informs the piece's aesthetics, so the performer *must* introduce these breaks between these clusters, even if the bellows still have some air left. Together with its large sways of dynamics (clusters are often connected by quick *crescendi* and *diminuendi* hairpins), the final result is an organic and idiomatic composition that allows the instrument to breathe naturally. I had initially written all four sections of the composition with non-legato clusters, but with Lubieniecki's advice, I worked out a reasonable phrasing solution for the soft sections (i.e. sections 2 and 4), allowing for more formal contrast.

Playing the portative organ at extremely soft dynamics will result in severe detuning, together with spectral changes in the instrumental timbre. For this piece, I notated the softest possible dynamic level in which such effects are not present as ***p***. The piece then goes on to use ***pp*** and ***ppp***, thus incorporating these effects. These are an intrinsic part of this instrument and, thus, an aspect that I wanted to incorporate directly into the work itself. At its softest moments, the sounds are incredibly fragile and barely held together, always on the verge of breaking apart.

In terms of structure, I used four distinct sections that transition very naturally from one into another. They all share similar principles and a common algorithm but differ concerning playing techniques, dynamic range, use of hairpins, and range used for the cluster parameters. In this work, I was particularly interested in the notion of using random seeds to control the compositional path. That is, I fixed individual random seeds for each section so that the pseudorandom number generator gets locked in a specific state for each of them. By working with individual seeds for each section, I was able to investigate each section separately and, thus, explore their unique possibilities independently from one another.

This work employs a similar looping window mechanism as used in *Cartography #8* (see Subsection 5.1.8). This can be observed in Figure 5.32, in which each measure is the result of an instance of the looping process. During the first section, this process is easy to be aurally followed due to its wide dynamic range and large sizes of skips between consecutive clusters. These qualities help create musical structures with strong characteristics, which, in turn, serve as easily identifiable guiding points for our ears. In subsequent sections, the algorithmic process becomes more chaotic: the looping window size becomes variable and its movement erratic, with consecutive iterations sometimes skipping over a random number of musical elements while, other times, repeating the same material without any change. In the sections with softer materials, the emergent structures, such as the ones present at the beginning of the piece, disappear. This is because the music becomes more uniform and, in turn, also more disorienting.

## 5.5 Conclusion

All pieces in the *Cartographies* series share the same mental model, which is based on the underlying metaphor of maps as containers. This provided a common framework for my algorithmic operations, leading to a decidedly simple but highly unique toolbox. Once these tools were defined, the act of composing became an exploration of what this toolbox could achieve. Experimentation and heuristics became of primary importance, as the interest lay in discovering what these tools could achieve instead of designing new complex ones for each new composition. This approach has become common in my works after this series, with all pieces borrowing some notions from *Cartographies*.

It can be argued that virtually all the pieces in this series could have been generated using traditional stochastic methods that mapped the frequency of an element to a probability value rather than the container index. However, in order to create such an implementation, one would need to conceive probability distributions that are very complex: first, they need to be unique for each type of parameter or raw musical elements (since containers can have different lengths, transformations, and transformation periods). Second, these probability functions would need to shift in a seemingly arbitrary way in order to account for the specific changes I used in these works. Such an approach would be highly cumbersome, to say the least, and it is thus safe to assume that these works would not have come to light in their current form if created with other techniques. The elegance of using a single probability distribution coupled with the simplicity of the container metaphor has led me to work with very specific and idiosyncratic higher-level abstractions. Furthermore, these metaphors and abstractions are responsible not only for shaping the work with their operations and suggested materials but also for how I *framed* these works in my mind while composing them. In other words, these tools provided me with a specific and unique compositional framework within which these pieces were created. This understanding of mental models as the basis for the compositional framework is of extreme relevance for algorithmic composers; as Hamman (1999, p. 102) writes, 'the computer is itself a tool for the construction of tools—tools with which one might generate epistemological frameworks for imagining and solving problems of compositional significance.' Problems are thus not only solved by these tools but also *imagined* through them.

The distribution chosen for these works has also been fundamental for the resulting character of these pieces as well as for suggesting musical processes that shaped them. Had the decay rate between the probabilities of consecutive indices been much steeper than the chosen 3/4, the rightmost indices would rarely be selected by the algorithms, thus concealing part of the containers. Some elements would be disproportionally present while others would seldom appear, making any type of 'smooth' operations impossible. But if this decay rate had been too shallow, the difference between the chance of consecutive indices would become too difficult to be perceived, rendering an almost uniform distribution at its limit. A possible route for future exploration could be using more complex relationships between the probability of these indices, i.e. using a non-fixed decay rate. In my opinion, such complex relationships will likely get lost in the final result, as we do not hear this function itself but the probabilistic results generated by it. This is precisely the reason why smooth procedures can be approximated with these discrete methods: we are not detecting the precise probability values of these elements, and thus changes can *feel* gradual. It is also interesting to realise that using an irregular probability distribution whose values for individual indices do not continuously decay or rise would, surprisingly, not necessarily result in different outputs. Effectively, such probability is simply a reordering of a continuously decaying one, with specific indices of the first mapped into different ones of the latter. Figure 5.33 illustrates this idea: by mapping the irregular indices into ordered decaying ones, the distribution once again resembles the one used in this series.

With the works of the accompanying portfolio, I hope to have demonstrated the importance of the decisions that the system designer must make, in particular when choosing constraints and transformation procedures. These ad hoc decisions are some of the most important aspects dictating the final aesthetic results. This is particularly evident in *Cartographies #9* and *#11*, both of which demonstrate that structurally rich music with particular sound worlds can be achieved with very economical algorithmic means. *Cartographies #1* and *#7*, on the other hand, exemplified how these containers can be used to control form and generate smooth changes, while *Cartographies #8*, with its idiosyncratic homophonic constraint and use of repetition, presents another possible approach for the use of loops. All these works, created with highly economical means, explore aesthetic concepts such as slippage, fragility, emergence, and liminality (see Chapter 3). By working within a very limited framework, the composer can more clearly explore the richness available in a particular musical space (Harper, 2011, p. 95). As Stravinsky (1970, p. 65) famously remarks,

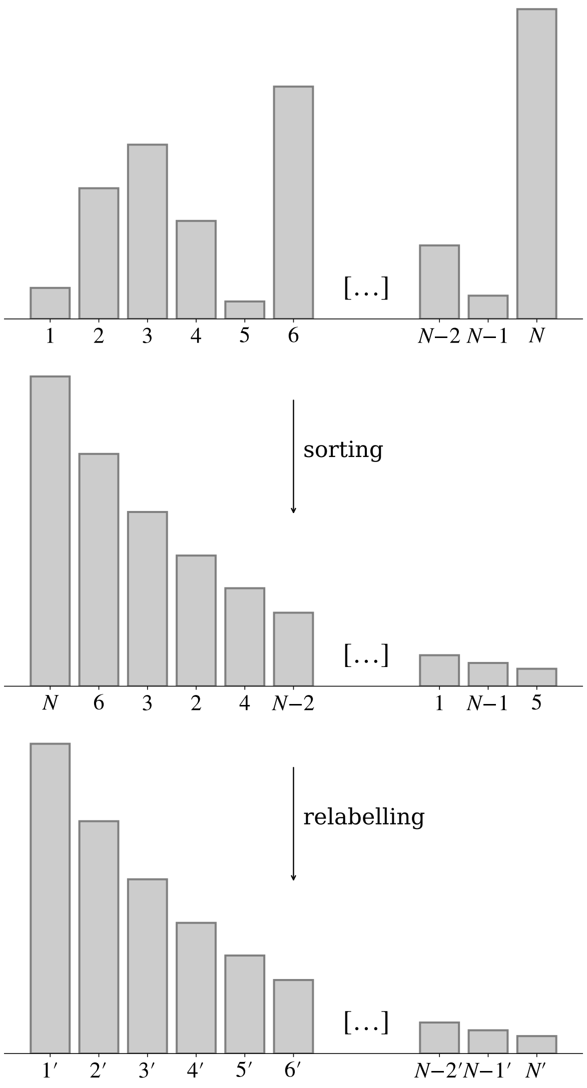

**Figure 5.33:** Process by which any irregular random distribution can be reordered into a decaying one

> my freedom will be so much the greater and more meaningful the more narrowly I limit my field of action and the more I surround myself with obstacles. Whatever diminishes constraint, diminishes strength. The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit.

The influence of this minimal algorithmic approach has remained with me in my works written after *Cartographies.* In particular, I continue to be interested in approaching new works with a few simple mental models that will guide both algorithmic and musical processes. Some of these are adapted and reused in new contexts, such as the variable looping windows of *what holds them together.* Others implement new mental models, such as the faders of *adrift* and the shufflers of *and thereafter they shape us*. Regardless of the framework used to compose these pieces, they all share the same aesthetic concern: the exploration of repetition and the perceptual disorientation caused by it, particularly when applied to fragile materials.

# Chapter 6

# Summary and Conclusion

> The serious artist is the only person able to encounter technology with impunity, just because he is an expert aware of the changes in sense perception.
>
> —Marshall McLuhan (1964, p. 33)

This dissertation explored some of the questions that composers must consider when working with algorithms and technology in general. Contrary to conventional wisdom, technology can become much more than just a tool: it not only embeds itself in the artwork it helps produce (Hamman, 2000c), but it also alters the way our minds conceive technological thinking in the context of artistic practice (Chowning, 1996, p. xii). Technology thus demands a symbiotic relationship with the artist, one in which the construction of tools is affected by the act of using these very tools.

A crucial cognitive mechanism used when engaging with technology is the development mental models through metaphors. These help us grasp abstract algorithmic ideas by grounding them with more concrete models and familiar concepts (Lakoff & Núñez, 2000), mediating our interaction with technology in the process. By conceptualising an abstract idea with a mental model, we are able to engage with substantially higher-level thinking since we can more readily grasp the complexity embedded in that model. This can lead to a substantially more malleable approach to algorithmic thinking, as the model can also suggest new ways for operating it. Such metaphors and mental models become of uttermost importance for the algorithmic composer: they not only allow the artist to better conceptualise their musical and algorithmic ideas but can also suggest specific epistemological paths to be explored.

Throughout my series of compositions entitled *Cartographies*, whose detailed commentary can be found in Chapter 5, I have demonstrated how some of these mental models can be put into practice and how they affect the final artwork. In all pieces of this series (totalling 12 individual works), I have employed a unifying mental model called 'container'. The idiosyncratic characteristics of the container mental model—e.g. the mapping of elements into indices, the association of probability values with indices and not elements, the encapsulation of both elements and behaviours—has had a profound effect on the way I approached these works from a compositional standpoint as well as how these works actually *sound*. As detailed in that chapter, these pieces bear an integral relationship with this mental model and could not have been thought of had different compositional methods been used. The same series of works (as well as later pieces also analysed in that same chapter) showcases other mental models, such as looping windows that constantly move forwards and the notion of 'input music'. The latter consists of the musical composition generated by the computer for the sole purpose of serving as input for the looping mechanism and, thus, is never heard in its unaltered form.

As detailed in Chapter 4, my research also led me to create the Auxjad programming library (Agostinho, 2021), which aims to extend the Abjad package for Python (Bača et al., 2021) for my own personal use. Auxjad contains classes and functions tailored to my own approach to algorithmic music, including mental models employed in my recent music. Examples of this include the container mental model explored in *Cartographies* (implemented as the class `auxjad.CartographySelector`), the 'looping window' mental model that is used in multiple works from *Cartographies* as well as compositions written after it (implemented as `auxjad.WindowLooper`, `auxjad.LeafLooper`, and `auxjad.ListLooper`), and the 'fader' mental model used to compose *adrift* (implemented as `auxjad.Fader` and `auxjad.CrossFader`).

Framing algorithmic composition as a form of exploration has also been a crucial attitude in my practice. Given that the algorithmic system does all the 'heavy lifting' for the artist (Pearson, 2011), experimentation becomes substantially more accessible than with more traditional compositional methods. However—and perhaps much more crucially than simply enabling experimentation—technology also opens up new routes of artistic endeavour that cannot be entirely pre-planned or foreseen beforehand. In other words, it allows us to create music using non-goal-oriented strategies and, in the process, to potentially transcend our personal artistic horizons (Eno, 1996; Nierhaus, 2010; Essl, 2007).

Virtually all of my recent music employs looping procedures or quasi-repetitions. Repetition is used not only as a way to structure my compositions but also for the way it can affect our perception of the resulting music. It can be used to create music that is ambiguous and disorienting, obfuscating the linear algorithmic processes that generated it. My work explores how algorithmic processes can create and maximise perceptual instabilities as well as the aesthetic properties that emerges from it.

The set of aesthetic concepts derived from these repetition techniques (first introduced in Chapter 3) became key elements of my recent music. These include the notions of slippage, fragility, emergence, and liminality. Slippage refers to the perceived disorientation created locally by near-repetition processes such as moving looping windows. When faced with consecutive windows in which very little changes, the listener is often unsure whether something is changing at all. This phenomenon is assisted by the fragile materials my music often employs: soft sounds that offer little contrast and emphasise the ambiguity of the musical texture. At the border of these windows, emergent structures can form when elements display pitch and temporal proximities, among other possible parameters (Bregman, 1990, pp. 455–528). As such, elements that were not initially side-by-side in the input music will suddenly be heard sequentially; if grouped together by our ear, this new emergent structure will obfuscate the looping point even further, reinforcing the overall sense of disorientation experienced by the listener. These elements can give rise to a large-scale liminal experience, one that is caused by the constant unfolding of the algorithmic process that never seems to reach a final destination. This music displays inner motion and yet lacks a more traditional feeling of musical progress. Together, this set of concepts leads to a schism between the strict process taking place in the piece and our perception of them; as such, simple linear algorithmic procedures will sound complex, disorienting, and non-linear.

The approach of working explicitly with mental models as compositional tools—that is, constructing them intentionally as opposed to making use of them unconsciously—is a powerful one. It has led to a substantial reformulation of my musical practice, allowing me to develop a new compositional approach and, more importantly, to compose music that sounds markedly different to my work written prior to this research project. What initially started as a technical endeavour has also led to a significant aesthetic shift in my music. The specific design of a mental model can have a direct influence on the resulting aesthetics of a composition, at the same time that specific aesthetic interests can inform the design of the mental models themselves. In other words, the creation of mental models, algorithmic systems, musical processes and materials, and aesthetics all emerge from a single compositional strategy and mutually influence one another.

Similarly to how my *Cartographies* series was based on exploring a single unifying mental model—which, in turn, suggested specific approaches to those pieces—other different mental models and metaphors can undoubtedly open up new paths for my future compositional research. Even the mental models explored during this research project are themselves far from being exhausted. In relation to looping windows alone, many other approaches could have been taken; not only the types of input materials can have an enormous effect on our perception of the looping process, but different behaviours of the windows can result in music with vastly different musical characteristics. These could include, for instance, looping windows that move erratically or that change step size, length, or direction of movement during its unfolding. Even a notion as simple as input music could be extended much further: a piece of music could use multiple 'input musics', which could then be combined and individually masked according to any arbitrary process. The very source of this input music could be explored, for instance, by using borrowed materials instead of internally generated pieces originating from the same algorithmic system. All these potential new ideas could become encapsulated into programming abstractions similar to those implemented in Auxjad, further extending this library. I very much hope to continue contributing to these topics as a composer, programmer, and researcher and, in the process, continue exploring the unique music that algorithmic methods enable me to create.

# Bibliography

Agostinho, G. (2019). *lilypondLibrary: A Library of Fortran Subroutines for Outputting Lilypond Code* [software]. Available at: `https://github.com/gilbertohasnofb/lilypondLibrary` [Accessed 5 June 2021].

——— (2021). *Auxjad: Auxiliary Classes and Functions for Abjad* [software]. Available at: `https://github.com/gilbertohasnofb/auxjad` [Accessed 5 June 2021].

Agostini, A. & Ghisi, D. (2015). 'A Max Library for Musical Notation and Computer-Aided Composition'. In: *Computer Music Journal* **39**(2). pp. 11–27.

Albiez, S. & Pattie, D. (2016). *Brian Eno: Oblique Music.* London: Bloomsbury Publishing.

Ames, C. (1990). 'Statistics and Compositional Balance'. In: *Perspectives of New Music* **28**(1). pp. 80–111.

Ames, P. K. (1996). 'Piano'. In: *The Music of Morton Feldman.* Ed. by DeLio, T. Westport, Connecticut: Greenwood Press. pp. 99–143.

Anders, T. (2018). 'Composing with Constraint Programming'. In: *The Oxford Handbook of Algorithmic Music.* Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 133–154.

Ariza, C. (2005). *An Open Design for Computer-Aided Algorithmic Music Composition: athenaCL.* Boca Raton, Florida: Dissertation.com.

Armstrong, N. (2021). *Fosc* [software]. Available at: `https://github.com/n-armstrong/fosc` [Accessed 5 June 2021].

Assayag, G., Rueda, C., Laurson, M., Agon, C., & Delerue, O. (1999). ‘Computer-Assisted Composition at Ircam: From PatchWork to OpenMusic’. In: *Computer Music Journal* **23**(3). pp. 59–72.

Bača, T., Oberholtzer, J. W., Treviño, J., & Adán, V. (2015). ‘Abjad: An Open-source Software System for Formalized Score Control’. In: *Proceedings of the First International Conference on Technologies for Music Notation and Representation*. Paris: Institut de Recherche en Musicologie. pp. 162–169.

——— (2021). *Abjad* [software]. Available at: `https://github.com/Abjad/abjad` [Accessed 5 June 2021].

Baetens, J. (2012). ‘OuLiPo and Proceduralism’. In: *The Routledge Companion to Experimental Literature*. Ed. by Gibbons, A., McHale, B., & Bray, J. Abingdon, United Kingdom: Routledge. pp. 115–127.

Baird, K. C. (2005). ‘No Clergy: Real-Time Generation and Modification of Music Notation’. In: *Spark Festival of Electronic Music and Art*. Minneapolis, Minnesota: University of Minnesota. pp. 40–41.

Barlow, C. (1990). ‘AUTOBUSK: An Algorithm Real-Time Pitch and Rhythm Improvisation Programme’. In: *International Computer Music Conference Proceedings*. Glasgow. pp. 166–168.

——— (2000). *AUTOBUSK 2000* [software]. Available at: `http://www.musikinformatik.uni-mainz.de/Autobusk/` [Accessed 12 March 2019].

Barros, B. (2018). *Fomus* [software]. Available at: `https://github.com/smoge/superfomus` [Accessed 5 June 2021].

——— (2019). *LilyCollider* [software]. Available at: `https://github.com/smoge/LilyCollider` [Accessed 5 June 2021].

Becher, B. & Becher, H. (1974). *Pitheads* [photographs]. Available at: `https://www.tate.org.uk/art/artworks/bernd-becher-and-hilla-becher-pitheads-t01922` [Accessed 28 September 2021].

——— (1988). *Water Towers* [photographs]. Available at: `https://www.moma.org/collection/works/49624` [Accessed 29 August 2020].

Bek, R. (2004). 'Conserving Junk and Movement: Machines by Jean Tinguely'. In: *Studies in Conservation* **49**(sup2). pp. 44–48.

Bense, M. (1971). 'The Projects of Generative Aesthetics'. In: *Cybernetics, Art and Ideas*. Ed. by Reichardt, J. London: Studio Vista. pp. 57–60.

Berendsen, W. (2020). *Frescobaldi* [software]. Available at: `https://frescobaldi.org/` [Accessed 5 June 2021].

Berg, P. (2009). 'Composing Sound Structures with Rules'. In: *Contemporary Music Review* **28**(1). pp. 75–87.

Bianchi, F. & Manzo, V. J. (2016). *Environmental Sound Artists: In Their Own Words*. Oxford: Oxford University Press.

Binu, A. (2010). *Problem Solving and Computer Programming Using C*. New Delhi, India: University Science Press.

Bohnacker, H., Gross, B., Laub, J., & Lazzeroni, C. (2012). *Generative Design: Visualize, Program, and Create with Processing*. New York: Princeton Architectural Press.

Boulez, P. & Cage, J. (1995). *The Boulez-Cage Correspondence*. Ed. by Nattiez, J.-J. Trans. by Samuels, R. Cambridge: Cambridge University Press.

Bown, O. & Martin, A. (2012). 'Autonomy in Music-Generating Systems'. In: *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*. Stanford, California: Stanford University. pp. 8–13.

Bregman, A. S. (1990). *Auditory Scene Analysis: The Perceptual Organization of Sound*. Cambridge, Massachusetts: The MIT Press.

Brindle, R. S. (1987). *The New Music: The Avant-Garde Since 1945*. Oxford: Oxford University Press.

Brown, A. (2018). 'Algorithms and Computation in Music Education'. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 582–601.

Brün, H. (1969). ‘Infraudibles’. In: *Music by Computers.* Ed. by Von Foerster, H. & Beauchamp, J. W. New York: John Wiley & Sons, Inc. pp. 117–121.

——— (2004). *When Music Resists Meaning: The Major Writings of Herbert Brün.* Ed. by Chandra, A. Middletown, Connecticut: Wesleyan University Press.

Buchanan, I. (2010a). ‘Liminality’. In: *Dictionary of Critical Theory* [online]. Oxford: Oxford University Press. Available at: `https://www.proquest.com/docview/2137958708/citation/1D0AC4C9AC5D40FFPQ/1` [Accessed 20 November 2021].

——— (2010b). ‘Oulipo’. In: *A Dictionary of Critical Theory* [online]. Oxford: Oxford University Press. Available at: `https://www.proquest.com/docview/2137942835/citation/CC21FE77FA6742E7PQ/1` [Accessed 20 September 2021].

Burnham, J. (2015). ‘Systems Aesthetics’. In: *Systems.* Ed. by Shanken, E. A. Documents of Contemporary Art. London: Whitechapel Gallery. pp. 112–115.

Cage, J. (2010). *Silence: Lectures and Writings.* Middletown, Connecticut: Wesleyan University Press.

Chau, C. (2014). ‘Movement and Time in the Nexus Between Technological Modes with Jean Tinguely’s Kineticism’. In: *Arts* **3**(4). pp. 394–406.

Childs, E. (2002). ‘Achorripsis: A Sonification of Probability Distributions’. In: *Proceedings of the 2002 International Conference on Auditory Display.* Kyoto, Japan. pp. 1–5.

Chilvers, I. & Glaves-Smith, J. (2009). ‘objet trouvé’. In: *Oxford Dictionary of Modern and Contemporary Art.* Oxford: Oxford University Press. p. 521.

Chowning, J. (1996). ‘Foreword: New Music and Science’. In: *The Computer Music Tutorial.* Cambridge, Massachusetts: The MIT Press. pp. ix–xii.

Clarke, M., Dufeu, F., & Manning, P. (2020). *Inside Computer Music.* New York: Oxford University Press.

Collins, N. (2018). ‘Origins of Algorithmic Thinking in Music’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 67–78.

Coolidge, C. (1988). ‘Regarding Morton Feldman’s Music and Wherever It All Now Goes’. In: *Sulfur* (22). pp. 123–129.

Cope, D. (1991). *Computers and Musical Style*. Oxford: Oxford University Press.

——— (2004). ‘A Musical Learning Algorithm’. In: *Computer Music Journal* **28**(3). pp. 12–27.

Cotton, C. (2009). *The Photograph as Contemporary Art*. London: Thames & Hudson.

Cox, G. & Riis, M. (2018). ‘(Micro)Politics of Algorithmic Music’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 603–626.

Crutchfield, J. P. (1994). ‘The Calculi of Emergence: Computation, Dynamics and Induction’. In: *Physica D: Nonlinear Phenomena* **75**(1–3). pp. 11–54.

Culkin, J. M. (1967). ‘A Schoolman’s Guide to Marshall McLuhan’. In: *The Saturday Review*. pp. 51–53, 70–72.

Dahlstedt, P. (2001). ‘A Mutasynth in Parameter Space: Interactive Composition Through Evolution’. In: *Organised Sound* **6**(2). pp. 121–124.

——— (2018). ‘Action and Perception’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 41–65.

Davancens, J. (2019). *Heave, Sway, Surge* [pdf]. PhD Thesis. Santa Cruz, California: University of California Santa Cruz. Available at: `https://github.com/jdavancens/dissertationpdf/blob/master/Heave%2C%20Sway%2C%20Surge%20-%20Essay.pdf` [Accessed 14 December 2021].

Davidson, J. (2014). ‘David Lang Wants to Be More Superficial’. *Red Poppy Music* [online]. Available at: `https://davidlangmusic.com/about/`

interviews/david-lang-wants-to-be-more-superficial [Accessed 26 August 2020].

Despeaux, S. E. (2015). ‘Oulipo: Applying Mathematical Constraints to Literature and the Arts in a Mathematics for the Liberal Arts Classroom’. In: *PRIMUS* **25**(3). pp. 238–247.

Deutsch, D., Henthorn, T., & Lapidis, R. (2011). ‘Illusory Transformation from Speech to Song’. In: *The Journal of the Acoustical Society of America* **129**(4). pp. 2245–2252.

Di Scipio, A. (2002). ‘Systems of Embers, Dust, and Clouds: Observations after Xenakis and Brün’. In: *Computer Music Journal* **26**(1). pp. 22–32.

——— (2003). ‘“Sound Is the Interface”: From Interactive to Ecosystemic Signal Processing’. In: *Organised Sound* **8**(3). pp. 269–277.

——— (2004). ‘Introduction’. In: *Journal of New Music Research* **33**(2). pp. 113–114.

Didkovsky, N. & Hajdu, G. (2008). ‘MaxScore: Music Notation in Max/MSP’. In: *Proceedings of the 2008 International Computer Music Conference* [online]. Belfast, Northern Ireland: Queen’s University: Michigan Publishing. n.pag. Available at: https://quod.lib.umich.edu/i/icmc/bbp2372.2008.034/--maxscore-music-notation-in-maxmsp?view=image [Accessed 14 December 2021].

Dijkstra, R. (1992). *De Panne, Belgium, August 7 1992* [photograph]. Available at: https://www.tate.org.uk/art/artworks/dijkstra-de-panne-belgium-august-7-1992-p78328 [Accessed 29 August 2020].

Doornbusch, P. (2005). ‘Pre-Composition and Algorithmic Composition: Reflections on Disappearing Lines in the Sand’. In: *Context* **29–30**. pp. 47–58.

Dysers, C. (2015). ‘Re-Writing History: Bernhard Lang’s Monadologie Series (2007–Present)’. In: *Tempo* **69**(271). pp. 36–47.

Dysers, C. (2019). *Loop Aesthetics: Repetition in the Work of Bernhard Lang* [pdf]. PhD Thesis. London: City, University of London. Available at: `https://openaccess.city.ac.uk/id/eprint/24762/` [Accessed 5 March 2020].

Eco, U. (1989). *The Open Work*. Cambridge, Massachusetts: Harvard University Press.

Edwards, M. (2011). 'Algorithmic Composition: Computational Thinking in Music'. In: *Communications of the ACM* **54**(7). pp. 58–67.

Ehn, P. (1988). *Work-Oriented Design of Computer Artifacts* [pdf]. PhD Thesis. Umeå, Sweden: Umeå University. Available at: `http://www.diva-portal.org/smash/get/diva2:580037/FULLTEXT02.pdf` [Accessed 23 February 2019].

Eldridge, A. & Brown, O. (2018). 'Biologically Inspired Algorithms for Music'. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 209–244.

Eno, B. (1996). *A Year with Swollen Appendices*. London: Faber & Faber.

——— (2004). 'The Studio as Compositional Tool'. In: *Audio Culture: Readings in Modern Music*. pp. 127–130.

Epstein, N. (2017). 'Musical Fragility: A Phenomenological Examination'. In: *Tempo* **71**(281). pp. 39–52.

Essl, K. (2007). 'Algorithmic Composition'. In: *Cambridge Companion to Electronic Music*. Ed. by Collins, N. & d'Escrivan, J. Cambridge: Cambridge University Press. pp. 107–125.

——— (2013). 'The Chances of Chance: Some Remarks on Artistic Practices Based on Chance and Collaboration'. In: *Handbook Of Research On Creativity*. Ed. by Thomas, K. & Chan, J. Cheltenham, United Kingdom: Edward Elgar Publishing. pp. 297–307.

——— (2018). 'Karlheinz Essl: Lexikon-Sonate – Algorithmic Music Generator (1992–2018)' [online]. Available at: `http://www.essl.at/works/Lexikon-Sonate.html` [Accessed 20 February 2019].

Evans, G. R. (2019). *An Introduction to Modeling Composition through Abjad's Model of Music Notation* [pdf]. MA thesis. Coral Gables, Florida: University of Miami. Available at: `https://github.com/GregoryREvans/thesis/raw/master/An_Introduction_to_Modeling_Composition_through_Abjad's_Model_of_Music_Notation.pdf` [Accessed 7 January 2019].

——— (2021). *Evans* [software]. Available at: `https://github.com/GregoryREvans/evans` [Accessed 5 June 2021].

Feldman, M. (2000). *Give My Regards to Eighth Street: Collected Writings of Morton Feldman*. Cambridge: Exact Change.

Fink, R. (2005). *Repeating Ourselves: American Minimal Music as Cultural Practice*. Berkeley, California: University of California Press.

Frey, J. (1996). 'Life Is Present'. In: *Edition Wandelweiser* [online]. Available at: `https://www.wandelweiser.de/_juerg-frey/texts-e.html#LIFE` [Accessed 14 December 2021].

——— (1998). 'The Architecture of Silence'. In: *Edition Wandelweiser* [online]. Available at: `https://www.wandelweiser.de/_juerg-frey/texts-e.html#THE` [Accessed 14 December 2021].

——— (2004). 'And on It Went'. In: *Edition Wandelweiser* [online]. Available at: `https://www.wandelweiser.de/_juerg-frey/texts-e.html#AND` [Accessed 14 December 2021].

Galanter, P. (2003). 'What Is Generative Art? Complexity Theory as a Context for Art Theory'. In: *Proceedings of the 6th Generative Art Conference* [pdf]. Milan. n.pag. Available at: `https://www.generativeart.com/on/cic/papersGA2003/a22.pdf` [Accessed 14 December 2021].

Gennep, A. van (1960). *The Rites of Passage*. Chicago: University of Chicago Press.

Griffiths, P. (2011). *Modern Music and After*. Oxford: Oxford University Press.

Groos, U. & Froitzheim, E.-M., (eds.) (2016). *[un]erwartet. Die Kunst des Zufalls* [Exhibition Catalogue, Kunstmuseum Stuttgart, 24 September 2016–19 February 2017]. Cologne, Germany: Wienand Verlag.

Hajdu, G. (2016). 'Resurrecting a Dinosaur—The Adaptation of Clarence Barlow's Legacy Software Autobusk'. In: *Proceedings of the International Conference on Technologies for Music Notation and Representation.* Cambridge: Anglia Ruskin University. pp. 181–186.

Hamman, M. (1999). 'From Symbol to Semiotic: Representation, Signification, and Composition of Music Interaction'. In: *Journal of New Music Research* **28**(2). pp. 90–104.

——— (2000a). 'From Technical to Technological: Interpreting Technology Through Composition'. In: *Proceedings of the 2000 Colloquium on Musical Informatics.* L'Aquila, Italy. pp. 91–94.

——— (2000b). 'Priming Computer-Assisted Music Composition Through Design of Human/Computer Interaction'. In: *Mathematics and Computers in Modern Science: Acoustics and Music, Biology and Chemistry.* Ed. by Mastorakis, N. E. New York: World Scientific & Engineering Society Press. pp. 75–82.

——— (2000c). 'The Technical as Aesthetic: Technology, Art-making, Interpretation'. In: *Proceedings of International Symposium on Music, Arts and Technology* [pdf]. Montpellier, France. pp. 1–10. Available at: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.550.6422&rep=rep1&type=pdf` [Accessed 30 October 2021].

——— (2004). 'On Technology and Art: Xenakis at Work'. In: *Journal of New Music Research* **33**(2). pp. 115–123.

Harley, J. (1995). 'Generative Processes in Algorithmic Composition: Chaos and Music'. In: *Leonardo* **28**(3). pp. 221–224.

Harper, A. (2011). *Infinite Music: Imagining the Next Millennium of Human Music-Making.* John Hunt Publishing.

Harrison, B. (2007). *Cyclical Structures and the Organisation of Time* [pdf]. PhD Thesis. Huddersfield, United Kingdom: University of Huddersfield. Available at: `http://www.brynharrison.com/BrynHarrisonThesis.pdf` [Accessed 2 October 2019].

Harrison, B. (2012). ‘Scanning the Temporal Surface: Aspects of Time, Memory and Repetition in My Recent Music’. In: *CeReNem journal* (3). pp. 59–71.

Harvey, D. (1991). *The Condition of Postmodernity: An Enquiry into the Origins of Cultural Change*. Cambridge, Massachusetts: Wiley-Blackwell.

Hattrick, A. & Mohr, M. (2012). ‘Interview with Manfred Mohr’. *The White Review* [online]. Available at: `http://www.thewhitereview.org/feature/interview-with-manfred-mohr/` [Accessed 23 May 2018].

Haworth, C. (2018). ‘Technology, Creativity, and the Social in Algorithmic Music’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 557–581.

Hertz, P. (2009). ‘Art, Code, and the Engine of Change’. In: *Art Journal* **68**(1). pp. 58–75.

Hiller, L. & Isaacson, L. M. (1959). *Experimental Music: Composition with an Electronic Computer*. New York: McGraw-Hill.

Hiller, L. A. (1981). ‘Composing with Computers: A Progress Report’. In: *Computer Music Journal* **5**(4). pp. 7–21.

Hiller, L. A. & Baker, R. A. (1964). ‘Computer Cantata: A Study in Compositional Method’. In: *Perspectives of New Music* **3**(1). pp. 62–90.

Hiller, L. A. & Isaacson, L. M. (1958). ‘Musical Composition with a High-Speed Digital Computer’. In: *Journal of the Audio Engineering Society* **6**(3). pp. 154–160.

Honour, H. & Fleming, J. (2010). *The Visual Arts: A History*. Upper Saddle River, New Jersey: Pearson Education.

Hope, C. & Vickery, L. (2011). ‘Visualising the Score: Screening Scores in Realtime Performance’. In: *Proceedings of Diegetic Life Forms II: Creative Arts Practice and New Media Scholarship*. Perth, Australia: Murdoch University: Interactive Media E-Journal. pp. 3–16.

——— (2015). ‘The DECIBEL Scoreplayer - A Digital Tool for Reading Graphic Notation’. In: *Proceedings of the First International Conference* *on Technologies for Music Notation and Representation*. Paris: Institut de Recherche en Musicologie. pp. 58–69.

Horvitz, S. (2010). *Automation Is My Salvation: Eight Studies for Automatic Piano* [pdf]. MA thesis. Oakland, California: Mills College. Available at: `http://www.context.fm/mills/thesis/HORVITZ_THESIS-FINAL_PRINT.pdf` [Accessed 2 October 2019].

Huang, W.-J., Xiao, H., & Wang, S. (2018). 'Airports as Liminal Space'. In: *Annals of Tourism Research* **70**. pp. 1–13.

Hunt, A., Wanderley, M. M., & Paradis, M. (2003). 'The Importance of Parameter Mapping in Electronic Instrument Design'. In: *Journal of New Music Research* **32**(4). pp. 429–440.

Huron, D. B. (2006). *Sweet Anticipation: Music and the Psychology of Expectation*. Cambridge, Massachusetts: The MIT Press.

Husarik, S. (1983). 'John Cage and Lejaren Hiller: HPSCHD, 1969'. In: *American Music* **1**(2). pp. 1–21.

I'Anson, M. (2018). 'Form, Chaos, and the Nuance of Beauty'. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 499–502.

Jakobovits, L. A. (1962). *Effects of Repeated Stimulation on Cognitive Aspects of Behavior: Some Experiments on the Phenomenon of Semantic Satiation* [pdf]. PhD Thesis. Montreal, Canada: McGill University. Available at: `https://escholarship.mcgill.ca/concern/theses/c821gp587` [Accessed 5 September 2021].

Johnson, T. (1998). 'Automatic Music'. In: *Actes des Journées d'Informatique Musicale 98*. Marseille, France: CNRS-LMA. pp. 1–4.

——— (2006). 'Self-Similar Structures in My Music: An Inventory'. In: *MaMuX Seminar*. Paris: Institut de Recherche en Musicologie. pp. 1–19.

——— (2011). 'Tiling in My Music'. In: *Perspectives of New Music* **49**(2). pp. 9–21.

Johnson, T. (2014). *Self-Similar Melodies.* Paris: Editions 75.

——— (n.d.). ‘I Want to Find the Music, Not to Compose It’ [pdf]. Available at: `http://editions75.com/texts/I_want_to_find_the_music.pdf` [Accessed 14 December 2021].

Keislar, D. (2009). ‘A Historical View of Computer Music Technology’. In: *The Oxford Handbook of Computer Music.* Ed. by Dean, R. T. Oxford: Oxford University Press. pp. 11–43.

Keller, D. & Ferneyhough, B. (2004). ‘Analysis by Modeling: Xenakis’s *ST/10-1 080262*’. In: *Journal of New Music Research* **33**(2). pp. 161–171.

Kelly, D. E. (2011). ‘Gemnotes: A Realtime Music Notation System for Pure Data’. In: *Pure Data Convention.* Weimar and Berlin. pp. 174–175.

Koenig, G. M. (1983). ‘Aesthetic Integration of Computer-Composed Scores’. In: *Computer Music Journal* **7**(4). pp. 27–32.

Kramer, J. D. (1981). ‘New Temporalities in Music’. In: *Critical Inquiry* **7**(3). pp. 539–556.

——— (1988). *The Time of Music: New Meanings, New Temporalities, New Listening Strategies.* New York and London: Schirmer Books.

——— (1996). ‘Postmodern Concepts of Musical Time’. In: *Indiana Theory Review* **17**(2). pp. 21–61.

——— (2016). *Postmodern Music, Postmodern Listening.* Ed. by Carl, R. London: Bloomsbury Publishing.

Lakoff, G. & Johnson, M. (2003). *Metaphors We Live By.* Chicago and London: University of Chicago Press.

Lakoff, G. & Núñez, R. E. (2000). *Where Mathematics Comes From: How the Embodied Mind Brings Mathematics Into Being.* New York: Basic Books.

Lamport, L. (2021). *LaTeX* [software]. Available at: `https://www.latex-project.org` [Accessed 5 June 2021].

Landy, L. (2008). ‘On the Paradigmatic Behaviour of Sound-Based Music’. In: *Proceedings of the Electroacoacoustic Music Studies Network International Conference*. Paris: INA-GRM and Université Paris-Sorbonne. n.pag.

Lang, B. (2002). ‘Loop Aesthetics’ [pdf]. Available at: `http://members.chello.at/bernhard.lang/publikationen/loop_aestet.pdf` [Accessed 2 October 2019].

——— (2003). ‘Bernhard Lang: Cycling ReCycling’ [online]. Available at: `http://members.chello.at/bernhard.lang/publikationen/cycling_recycl.htm` [Accessed 2 October 2019].

Latartara, J. (2010). ‘Laptop Composition at the Turn of the Millennium: Repetition and Noise in the Music of Oval, Merzbow, and Kid606’. In: *Twentieth-Century Music* **7**(1). pp. 91–115.

Levtov, Y. (2018). ‘Algorithmic Music for Mass Consumption’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 627–644.

Lewis, G. E. (2000). ‘Too Many Notes: Computers, Complexity and Culture in Voyager’. In: *Leonardo Music Journal* **10**. pp. 33–39.

——— (2009). ‘Interactivity and Improvisation’. In: *The Oxford Handbook of Computer Music*. Ed. by Dean, R. T. Oxford: Oxford University Press. pp. 457–466.

LeWitt, S. (1967a). ‘Paragraphs on Conceptual Art’ [pdf]. Available at: `http://arteducation.sfu-kras.ru/files/documents/lewitt-paragraphs-on-conceptual-art1.pdf` [Accessed 22 February 2019].

——— (1967b). ‘Serial Project #1, 1966’. In: *Aspen Magazine* (5+6). Ed. by O’Doherty, B. n.pag.

Longplayer (2019). *Longplayer* [online]. Available at: `https://longplayer.org` [Accessed 12 March 2019].

Magnusson, T. & McLean, A. (2018). ‘Performing with Patterns of Time’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 245–265.

Margulis, E. H. (2014). *On Repeat: How Music Plays the Mind*. Oxford: Oxford University Press.

Martin, A. (1994). *Untitled #5* [painting]. Available at: `https://www.tate.org.uk/art/artworks/martin-untitled-5-ar00177` [Accessed 31 August 2020].

McLean, A. & Dean, R. T. (2018a). 'Musical Algorithms as Tools, Languages, and Partners'. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 3–15.

——— (eds.) (2018b). *The Oxford Handbook of Algorithmic Music*. New York: Oxford University Press.

McLuhan, M. (1964). *Understanding Media: The Extensions of Man*. New York: New American Library.

Meadows, D. H. (2015). 'Dancing with Systems'. In: *Systems*. Ed. by Shanken, E. A. Documents of Contemporary Art. London: Whitechapel Gallery. pp. 57–61.

Mengozzi, S. (2008). '"Clefless" Notation, Counterpoint and the *Fa*-Degree'. In: *Early Music* **36**(1). pp. 51–66.

Milne, A. J. (2018). 'Linking Sonic Aesthetics with Mathematical Theories'. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 133–154.

Miranda, E. R. (2001). *Composing Music with Computers*. Oxford: Focal Press.

——— (2009). 'Preface: Aesthetic Decisions in Computer-Aided Composition'. In: *Contemporary Music Review* **28**(2). pp. 129–132.

Mirjalili, S. (2019). 'Genetic Algorithm'. In: *Evolutionary Algorithms and Neural Networks: Theory and Applications*. Studies in Computational Intelligence. Cham: Springer International Publishing. pp. 43–55.

Mohr, M. (2000). 'Artists' Statements'. In: *Leonardo*. Eighth New York Digital Salon **33**(5). pp. 437–446.

Moles, A. A. (1971). ‘Art and Cybernetics in the Supermarket’. In: *Cybernetics, Art and Ideas*. Ed. by Reichardt, J. London: Studio Vista. pp. 61–71.

Moscotta, I. A. (2019). *Tsmakers* [software]. Available at: `https://github.com/ivanalexandermoscotta/tsmakers` [Accessed 5 June 2021].

Nake, F. (2010). ‘Paragraphs on Computer Art, Past and Present’. In: *Proceedings of the Computer Art and Technocultures Conference*. London. pp. 55–63.

Nienhuys, H.-W. & Nieuwenhuizen, J. (2003). ‘Lilypond, a System for Automated Music Engraving’. In: *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*. Florence, Italy. pp. 167–172.

——— (2021). *LilyPond* [software]. Available at: `http://lilypond.org/` [Accessed 5 June 2021].

Nierhaus, G. (2010). *Algorithmic Composition: Paradigms of Automated Music Generation*. Vienna: Springer-Verlag.

Norman, D. A. (2013). *The Design of Everyday Things*. New York: Basic Books.

Oberholtzer, J. W. (2015). *A Computational Model of Music Composition* [pdf]. PhD Thesis. Cambridge, Massachusetts: Harvard University. Available at: `http://nrs.harvard.edu/urn-3:HUL.InstRepos:17463123` [Accessed 7 February 2019].

——— (2018). *Consort* [software]. Available at: `https://github.com/josiah-wolf-oberholtzer/consort` [Accessed 5 June 2021].

Oliver La Rosa, J. (2018). *notes* [software]. Available at: `https://github.com/lar0sa/notes_lib` [Accessed 5 June 2021].

Palisca, C. V. & Pesce, D. (2001). ‘Guido of Arezzo [Aretinus]’. In: *Grove Music Online* [online]. Available at: `https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000011968` [Accessed 13 October 2021].

Park, S. K. & Miller, K. W. (1988). ‘Random Number Generators: Good Ones Are Hard to Find’. In: *Communications of the ACM* **31**(10). pp. 1192–1201.

Pearson, M. (2011). *Generative Art: A Practical Guide Using Processing.* Shelter Island, New York: Manning.

Phillips, D. (2018). *Python 3 Object-Oriented Programming: Build Robust and Maintainable Software with Object-Oriented Design Patterns in Python 3.8.* Birmingham, United Kingdom: Packt Publishing.

Pickles, D. (2016). *Cybernetics in Music* [pdf]. PhD Thesis. Coventry, United Kingdom: Coventry University. Available at: `https://curve.coventry.ac.uk/open/items/6acfa32c-3113-4b11-9199-7eb5a418bb37/1/` [Accessed 26 August 2020].

Polansky, L., Barnett, A., & Winter, M. (2011). 'A Few More Words About James Tenney: Dissonant Counterpoint and Statistical Feedback'. In: *Journal of Mathematics and Music* **5**(2). pp. 63–82.

Pope, S. T. (1996). 'Object-Oriented Music Representation'. In: *Organised Sound* **1**(1). pp. 56–68.

Pritchett, J. (1996). *The Music of John Cage.* Vol. 5. Cambridge: Cambridge University Press.

Randall, J. K. (1969). 'Operations on Wave Forms'. In: *Music by Computers.* Ed. by Von Foerster, H. & Beauchamp, J. W. New York: John Wiley & Sons, Inc. pp. 122–128.

Rappengliick, M. A. (2006). 'The Whole World Put Between to Shells: The Cosmic Symbolism of Tortoises and Turtles'. In: *Mediterranean Archaeology and Archaeometry* **4**(3). pp. 223–230.

Raubach Tuchtenhagen, D. (2021). *muda* [software]. Available at: `https://github.com/DaviRaubach/muda` [Accessed 15 July 2021].

Reich, S. (2002a). 'Clapping Music (1972)'. In: *Writings on Music, 1965–2000.* New York: Oxford University Press. p. 68.

——— (2002b). 'Music As a Gradual Process'. In: *Writings on Music, 1965–2000.* New York: Oxford University Press. pp. 34–36.

Reichardt, J. (1971a). ‘Cybernetics, Art and Ideas’. In: *Cybernetics, Art and Ideas*. Ed. by Reichardt, J. London: Studio Vista. pp. 11–17.

——— (1971b). *The Computer in Art*. London: Studio Vista.

Roads, C. (1996). *The Computer Music Tutorial*. Cambridge, Massachusetts: The MIT Press.

Roberts, C. & Wakefield, G. (2018). ‘Tensions and Techniques in Live Coding Performance’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 293–317.

Rohrhuber, J., Campo, A. D., & Wieser, R. (2005). ‘Algorithms Today: Notes on Language Design for Just in Time Programming’. In: *International Computer Music Conference Proceedings*. Barcelona. n.pag.

Russeth, A. (2012). ‘Here Are the Instructions for Sol LeWitt’s 1971 Wall Drawing for the School of the MFA Boston’. *Observer* [online]. Available at: `https://observer.com/2012/10/here-are-the-instructions-for-sol-lewitts-1971-wall-drawing-for-the-school-of-the-mfa-boston/` [Accessed 30 August 2020].

Sabbe, H. (1996). ‘The Feldman Paradoxes’. In: *The Music of Morton Feldman*. Ed. by DeLio, T. Westport, Connecticut: Greenwood Press. pp. 9–15.

Scaletti, C. (2018). ‘Sonification $\neq$ Music’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 363–385.

Shanken, E. A. (2002). ‘Art in the Information Age: Technology and Conceptual Art’. In: *Leonardo* **35**(4). pp. 433–438.

——— (2014). ‘In Forming Software: Systems, Structuralism, Demythification’. In: *Revista ICONO14* **12**(2). pp. 9–28.

——— (ed.) (2015). *Systems*. Documents of Contemporary Art. London: Whitechapel Gallery.

Shvets, A. & de Paiva Santana, C. (2014). ‘Modelling Arvo Pärt’s Music with OpenMusic’. In: *Proceedings of the EVA London 2014 on Electronic Visualisation and the Arts*. London. pp. 9–16.

Simchy-Gross, R. & Margulis, E. H. (2018). ‘The Sound-to-Music Illusion: Repetition Can Musicalize Nonspeech Sounds’. In: *Music & Science* **1**. pp. 1–6.

Simoni, M. (2018). ‘The Audience Reception of Algorithmic Music’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 531–556.

Smalley, D. (1997). ‘Spectromorphology: Explaining Sound-Shapes’. In: *Organised sound* **2**(2). pp. 107–126.

Smith, L. A. (2013). *SCORE Music Publishing System* [software]. San Andreas Press. Available at: `https://web.archive.org/web/20190602035812/http://scoremus.com/` [Accessed 5 June 2021].

Snyder, B. (2000). *Music and Memory: An Introduction*. Cambridge, Massachusetts: The MIT Press.

Sontag, S. (2018). ‘One Culture and the New Sensibility’. In: *Notes on ‘Camp’*. London: Penguin Books. pp. 34–55.

Spiegel, L. (1981). ‘Manipulations of Musical Patterns’. In: *Proceedings of the Symposium on Small Computers in the Arts*. Philadelphia, Pennsylvania: IEEE Computer Society. pp. 19–22.

——— (2018). ‘Thoughts on Composing with Algorithms’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 105–111.

Steinbeck, P. (2018). ‘George Lewis’s Voyager’. In: *The Routledge Companion to Jazz Studies*. Ed. by Gebhardt, N., Rustin-Paschal, N., & Whyton, T. New York: Routledge. pp. 261–270.

Stravinsky, I. (1970). *Poetics of Music in the Form of Six Lessons*. Cambridge, Massachusetts: Harvard University Press.

Strogatz, S. H. (2000). *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering.* Boulder, Colorado: Westview Press.

Supper, M. (2001). ‘A Few Remarks on Algorithmic Composition’. In: *Computer Music Journal* **25**(1). pp. 48–53.

Tenney, J. & Polansky, L. (1980). ‘Temporal Gestalt Perception in Music’. In: *Journal of Music Theory* **24**(2). pp. 205–241.

Terhardt, E. (1979). ‘Calculating Virtual Pitch’. In: *Hearing Research* **1**(2). pp. 155–182.

Terry, P., (ed.) (2019). *The Penguin Book of Oulipo: Queneau, Perec, Calvino and the Adventure of Form.* London: Penguin Books.

Treviño, J. (2013). *Compositional and Analytic Applications of Automated Music Notation Via Object-Oriented Programming* [pdf]. PhD Thesis. San Diego, California: University of California San Diego. Available at: `https://escholarship.org/uc/item/3kk9b4rv` [Accessed 13 March 2019].

Uliasz, R. (2017). ‘Perverting Technological Correctness’. In: *arts.codes* [online] **1**. Available at: `http://arts.codes/articles/vol1/article0/article.html` [Accessed 16 July 2018].

Umland, A., Sudhalter, A., & Gerson, S., (eds.) (2008). *Dada in the Collection of the Museum of Modern Art.* New York: The Museum of Modern Art.

Varnedoe, K. (2006). *Pictures of Nothing: Abstract Art Since Pollock.* Princeton University Press.

Verbeeck, K. (2006). *Randomness as a Generative Principle in Art and Architecture* [pdf]. PhD Thesis. Cambridge, Massachusetts: Massachusetts Institute of Technology. Available at: `https://core.ac.uk/download/pdf/4400202.pdf` [Accessed 20 November 2021].

Vermeulen, T. & van den Akker, R. (2010). ‘Notes on Metamodernism’. In: *Journal of Aesthetics & Culture* **2**(1). n.pag.

Vigna, S. (2021). *LilyPond Snippet Repository* [online]. Available at: `http://lsr.di.unimi.it/LSR/html/whatsthis.html` [Accessed 1 August 2020].

Vitalis, A. (2016). *The Uncertain Digital Revolution.* Hoboken, New Jersey: John Wiley & Sons, Inc.

Wands, B. (2006). *Art of the Digital Age.* London: Thames & Hudson.

Wang, G. (2017). 'A History of Programming and Music'. In: *The Cambridge Companion to Electronic Music.* Ed. by Collins, N. & d'Escrivan, J. Cambridge: Cambridge University Press. pp. 58–74.

Weibel, P. (2004). 'Algorithmic Revolution: On the History of Interactive Art'. *ZKM* [online]. Available at: `http://zkm.de/en/media/audio/peter-weibel-algorithmic-revolution` [Accessed 16 July 2018].

——— (2007). 'It Is Forbidden Not to Touch: Some Remarks on the (Forgotten Parts of the) History of Interactivity and Virtuality'. In: *MediaArtHistories.* Ed. by Grau, O. Leonardo. Cambridge, Massachusetts: The MIT Press. pp. 21–41.

Weisfeld, M. (2004). *The Object-Oriented Thought Process.* Indianapolis, Indiana: Sams.

Weissenbrunner, K. (2017). *Experimental Turntablism Live Performances with Second Hand Technology* [pdf]. PhD Thesis. London: City, University of London. Available at: `https://openaccess.city.ac.uk/id/eprint/19919/` [Accessed 17 October 2020].

Wertheimer, M. (2012). 'Investigations on Gestalt Principles'. In: *On Perceived Motion and Figural Organization.* Ed. by Spillmann, L. Cambridge, Massachusetts: The MIT Press. pp. 127–182.

West, R. (2016). *Calliope* [software]. Available at: `https://github.com/mirrorecho/calliope` [Accessed 5 June 2021].

Wieser, R. (2018). 'Deautomatization of Breakfast Perceptions'. In: *The Oxford Handbook of Algorithmic Music.* Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 119–122.

Wiggins, G. & Forth, J. (2018). ‘Computational Creativity and Live Algorithms’. In: *The Oxford Handbook of Algorithmic Music*. Ed. by McLean, A. & Dean, R. T. New York: Oxford University Press. pp. 267–292.

Williams, A. (1997). *New Music and the Claims of Modernity*. Aldershot, United Kingdom: Ashgate.

Woodbury, R. (2010). *Elements of Parametric Design*. London: Routledge.

Wuorinen, C. (1994). *Simple Composition*. New York: C. F. Peters Corporation.

Xenakis, I. (1992). *Formalized Music: Thought and Mathematics in Composition*. Hillsdale, New York: Pendragon Press.

——— (1994). ‘La crise de la musique sérielle’. In: *Kéleütha: écrits*. Paris: L’Arche Éditeur. pp. 39–43.

York, W. (1996). ‘For John Cage’. In: *The Music of Morton Feldman*. Ed. by DeLio, T. Westport, Connecticut: Greenwood Press. pp. 147–195.

Zinkstok, R. & Zinkstok, J. (2020). ‘Zink Typography – LaTeX’. *Zink Typography* [online]. Available at: `http://www.rtznet.nl/zink/latex.php?lang=en` [Accessed 2 August 2020].

# Appendix A

# Code for Sol LeWitt's *Wall Drawing #118*

This appendix contains the code for an implementation of Sol LeWitt's *Wall Drawing #118* (1971) using Processing. The original instructions from the artist, as quoted in Russeth (2012, pp. 3–4), are:

> On a wall surface, any
> continuous stretch of wall,
> using a hard pencil, place
> fifty points at random.
> The points should be evenly
> distributed over the area
> of the wall. All of the
> points should be connected
> by straight lines.

Below is an implementation of these instructions using the Processing programming language. See Figure 3.10 in Chapter 3 for an example of the output of this code.

```
// implementation of Sol LeWitt's Wall Drawing #118

import processing.pdf.*;


int n_points = 50;
float[][] list_of_points = new float[n_points][2];
float[][] random_direction = new float[n_points][2];


void new_points() {
    for (int i = 0; i < n_points; i++) {
        list_of_points[i][0] = random(50, width - 50);
        list_of_points[i][1] = random(50, height - 50);
    }
}


void setup() {
    size(1280, 960, PDF, "sol.pdf");
    background(248);
    strokeWeight(0.5);
    smooth();
    new_points();
}


void draw() {
    stroke(0, 32);
    for (int i = 0; i < n_points - 1; i++) {
        for (int j = i + 1; j < n_points; j++) {
            line(list_of_points[i][0],
                 list_of_points[i][1],
                 list_of_points[j][0],
                 list_of_points[j][1]);
        }
    }
    stroke(0);
    fill(0);
    for (int i = 0; i < n_points - 1; i++) {
        circle(list_of_points[i][0],
               list_of_points[i][1],
               4);
    }
    exit();
}
```