Chapter 14: Performing with Patterns of Time

Thor Magnusson and Alex McLean

Abstract

Music is a time-based art form often characterised by patternings; manipulations of sequences over time. Composers and performers may think in terms of patterns, although the structure of patterned sequences are often not made explicit in musical notation. This chapter explores how musical sequences can be created and transformed in real-time performance through patterning functions. Topics related to the use of algorithms for pattern-making are discussed, and two systems are introduced - ixi lang and TidalCycles, as high level and expressive mini-languages for musical pattern.

Keywords: musical pattern, mini-languages, tidalcycles, ixi lang, musical notation

1 Introduction

"The process of creating music involves ... a working knowledge of all the processes of transformation which can aesthetically be applied to [patterns of sound]. Beyond these there needs to be a practised awareness of how such materials and operations, and the specific characteristics of each, relate to and influence each others' potentials" (Spiegel 1981)

Varèse famously got so tired of people's comment that his work was "interesting," immediately followed by the question "but is it music?," that he decided to call the outcome of his practice "organised sound" (Varese 1966, p. 18). A related definition explored here, is of music as patterns of sound. These patterns are about relationships between sonic events (whether pitched or not) taking place in time, a strongly mathematical domain which has been extensively explored (for example du Sautoy (2003); Burack (2005); Fauvel and Wilson (2006)). It is also a cognitive domain: in music psychology, pattern recognition is seen as a principal feature of human cognition, perhaps explaining the fascination many humans have in the repetitive nature of music, where the *re-cognition* of patterns form units or words that build up a larger musical meaning (Sloboda 2005, 18). It is fitting that in one of the key texts on computational music, Taube's *Notes from the Metalevel*, we find a reference to the philosopher Alfred North Whitehead, who says: "Art is the imposing of a pattern on experience, and our aesthetic enjoyment is recognition of the pattern" (Taube 2004, 233).[1]

Musical patterns are not simply the sounds heard: they also refer to the embodied actions performed when playing music. Musicians describe how entrained repetitive practice of performing patterns becomes embodied, tacit knowledge of scales, chords or arpeggios; motor movement patterns based on (sometimes unarticulated) musical theory and their incorporation into motor memory (Merleau-Ponty 2002 p.168; Sudnow 1993; Hayles 1999 p.199; Parente 2015). In musical performance, the instrumentalist relies on past practice and this relates equally to written and improvised music. These embodied patterns resulting in musical performance are often mirrored - albeit not isomorphically - in how people respond to music. Diverse dance forms, from ballet to pogo, with other gestural types such as headbanging, luftguitar, head nodding, foot tapping, orchestra-conducting, etc. are all embodied interpretations of musical patterns. We can look to neuropsychology for a biological perspective, for instance in the work

[1]Although note that going to the original reference, we find that this is in the context of Whitehead arguing against quoting soundbites.

of Patel and Iversen (2014), who find neuroimaging evidence to support the hypothesis that human beat perception is supported by two-way interaction between motor-planning and auditory areas of the brain. This suggests that patterns we hear are strongly informed by simulated patterns of movement.

When talking about patterns in this chapter, we are primarily thinking of the temporal patterns of musical performance. As mentioned, these patterns can be understood through the embodied mode of dancing as well as represented algorithmically through a formula. We celebrate the fact that a sequence of events can be represented with different patterns in different languages or systems. Indeed, two people might describe, perceive or understand the same sequence as having very different pattern structures. Issues of translation, transduction and transmission become interesting in this context.

If pattern links sound with movement in perception, the present chapter is concerned with how a third element may enter this relation: symbolic notation - and how this can be written in live performance. Notation systems allow music to be expressed in a format suitable for preservation and sharing, where musical patterns may manifest as visual patterns through diverse systems of scoring. For example, in staff notation pitch is described vertically on a horizontal timeline, with secondary notation for dynamics, articulation and accents. In this case, ornamentation, timbre, and many other articulations are often left for interpretation, and abstract relationships structuring the composition are not made explicit. The profound challenge that we hold in hand for algorithmic music is to notate structural and multidimensional aspects of musical pattern, by using algorithmic representations that go beyond the usual dimensions of music notation, while still allowing expressive use by a composer. Furthermore, notation allows us to explore the rich interferences which emerge from combining pattern transformations, where notation becomes a process of live exploration, rather than description.

2 Background to Musical Patterns

We approach the expansive topic of pattern in algorithmic music with a clear question: how can we directly express musical pattern with computer code? And how can this be achieved, for example within the constraints of a live performance? For our purposes, the *patterning* of music is where a composer represents *and* transforms source material using a set of strategies. This is an inclusive definition, but the context of algorithmic music gives us a special focus on where strategies are expressed in a programming language. By notating sequences and their transformation through code, the abstract structures of pattern can be made explicit, analysable and shareable. Code also allows us to generalise aspects of pattern-making, on one hand allowing a transformation to be made on multiple scales and timbral dimensions, and on the other allowing transformations to be combined in diverse ways, creating an explosion of possibilities to explore.

This chapter can be read as a response to a call to arms made by Laurie Spiegel 35 years ago in her paper "Manipulations of Musical Patterns" (Spiegel 1981). Spiegel listed twelve classes of pattern transformation, such as transposition, reversal, rotation and repetition. What is striking about Spiegel's short description of each is how open to interpretation and implementation they are. Each stands for a huge range of possibilities and interpretations, as we will see later in considering the apparently straightforward concept of *reversal*.

Many programming languages designed for computer music include libraries for pattern representation and transformation, such as HMSL, SuperCollider and Common Music. However, these libraries do not often live up to the promise of a comprehensive library of pattern transformations proposed by Spiegel, perhaps relying on an underlying general purpose language for much of the functionality. Indeed, much of the operation of code can be considered in terms of pattern manipulation. For instance, we can look at basic bit-level operations, loop constructions, data flow rules and functional mapping and define those as elements in pattern building. This challenges our ability to compare or even standardize pattern libraries; the programming paradigm at play (e.g., functional, logical, object-oriented, mixed) impacts on how patterns are represented, and therefore on the musical constraints and affordances that the composer works with.

The reason for the diverse pattern libraries comes down to affordance: by defining a function of pattern generation or transformation - such as Fibonacci or inversion - the system creator designs an addition to the vocabulary of their compositional language, thus expanding the musical search space immediately available to the end-user. The pattern function becomes an abstraction of a computational process that may be trivial or complex, but by naming it and including it in a set of other functions, a coherent vocabulary is built up: one that will influence the music or style made with the system.

3 Patterns in Music

Organising sound, planning events in time, arranging pitched patterns: most definitions of music outline some kind of description of rules that define its temporal nature and the melodic, harmonic and rhythmic elements therein. In the following we explore a range of musical forms where algorithmic rules of pattern have been defined as compositional heuristics. We will jump over much musical history and geography, but mention a few traditions that serve as historical underpinning to the way algorithmic music is now composed and produced with computers.

3.1 Fugue / counterpoint

In the form of 17th century contrapuntal fugues, as perfected by J. S. Bach, we find a musical subject which is introduced as the first voice. The second voice appears shortly afterwards, responding to the first, and this voice is called 'countersubject'. A third voice appears. Each voice states a subject - a melody that references the first voice, but the art of the counterpoint is to use transformations such as *stretto* (overlapping melodies, often where one starts after the other), *inversion* (turning the pattern upside down), *augmentation* (lengthening the note duration) and *diminution* (shortening the notes), *retrograde* (play the melody backwards), *retrograde inversion*, and further combinations of the above. Counterpoint composition is a highly mathematical task, and it is no wonder that musical systems in computational creativity have been highly successful in the production of new fugues. A good example is Kemal Ebciolu's system *CHORAL* (Ebciolu 1988), which is perhaps even *too* Bach-like to be convincing to Bach experts.

3.2 Serialism

Serialist music is also characterised by a strongly mathematical approach to pattern. This is a music that lacks tonal and, at times, metric centers; it is composed through rather rigid rule sets of fixed permutation of tone rows. The music of Schoenberg and Webern are good examples of the serialist technique, where a musical row is created with all 12 tones in the octave (thus the descriptive label of the *12-tone technique*) and no note is repeated in a single voice before all twelve notes have been played. A row may then undergo four different permutations: *prime*, with the "original" ordering of tones and intervals of non-repeating notes; *retrograde*, which is a reversion of the intervallic structure reversed; *inversion*, where the intervallic structure is inverted (up instead of down and vice versa); and *retrograde-inversion*, where the intervallic structure is simultaneously reversed and inverted. This results in 12 transpositional levels and thus 48 possible forms.

3.3 Minimalism

Terry Riley's 1964 piece *In C* is a good example of the minimalist approach to melodic and rhythmic patterning of sounds. Riley's piece consists of 53 phrases between half and 32 beats. The piece can be performed by an infinite amount of players, but Riley suggests at least 35. The 53 phrases are performed in order (although phrases can be skipped). The performers can choose a phrase that they repeat until they decide to move on to the next phrase. This brings aleatoric and improvisational elements to the music (something Cage had of course explored earlier).

Many of the minimalists would reject the label, but prominent composers working with sonic materials in an approach that might be defined as being minimalist would include La Monte Young, Terry Riley, Philip Glass, Phill Niblock, Tony Conrad, Louis Andriessen, Henryk Górecki, Arvo Pärt, John Tavener and perhaps the archetype of the genre, Steve Reich. Reich, influenced by African drumming and polymetric structures, is known for exploring 'phasing' musical material, for example by playing two tape loops in sync but slowing the speed of one; or repeating patterns in a musical score where one part is subsequently delayed at regular intervals. His pieces *It's Gonna Rain* and *Clapping Music* are examples of the respective approaches.

The minimalist approach has not resulted in defined methods such as those we find in contrapuntal or serialist music, but there is a clearly identifiable compositional method that can be characterised by steady pulse, repetition, gradual transformation of sequences, and a harmony that is built up of fast melodic progressions, often gradually evolving rather than drastically changing in chord or key.

3.4 Electronic Music

The compositional approaches taken in twentieth century pre-computer electronic music owe much to the musical affordances of the hardware available at the time. Early equipment consisted of synthesisers and tape (often mapped with a sweeping generalisation onto the factions of German *Elektronische Musik* and French *musique concrète*) where the focus was on sonic materials, typically arranged using the primitive methods of cutting and pasting slices of tape. The physical materiality of this work process resulted in music where patterns were largely absent: they could be created with tape loops, but this could only be achieved with some difficulty.

Repeating sequences were introduced into electronic music with electronic sequencers that would output currents to voltage controllable oscillators. Jumping quickly over to computer-based hardware, we find mass-manufactured sequencers, whose design has been aimed at the many. Although the *RCA Mark II* used punch cards to control voltages, cheaper and more popular sequencers, such as the *Moog 960*, used potentiometers to set the voltage values. Eight or sixteen-step sequencers were the most common devices and this tradition continued into the design of software sequencers.

In modern musical software we typically find systems that derive their design metaphors - both in terms of interface and interaction design - from past traditions, such as the musical score, piano rolls, and the hardware sequencer. Musical patterning in such software is therefore still often subject to the hardwired mechanisms of historical physical equipment.

In the light of the constraints imposed by hardware, and the software simulating it, there is no wonder that musical and audio programming languages have opened up multiple doors for creative musical exploration and expression (Stowell and McLean 2012). What attracts composers and performers here are the open possibilities of defining their own patterns, synths, and hardware instruments; users can write any pattern generating or manipulating algorithm conceivable without being bound to the musical constraints of software or physical hardware.

‘Underground’ electronic dance music has fostered a range of experimentation along the hardcore continuum and beyond, including some in manipulations of pattern. Contemporary software applications include a range of means for arranging patterns, for example arpeggiators with parameters that are often pushed beyond their normal limits to create ‘hyperreal’ effects in trance, and software for algorithmic slicing and rearranging breakbeats such as the methods available in SuperCollider (Collins 2006), and the commercial *iZotope Breaktweaker* software created by the trance producer BT.

3.5 Modular synthesis

We should also mention in passing the current resurgence of modular hardware synthesisers, particularly the huge range of ‘Eurorack’ modules now available from many manufacturers, all designed in standard sizes and voltages to be used together. Aside from the great focus on analogue synthesis, there are a great many pattern generation modules available. One example is the *Stoicheia* module from Rebel Technology, which generates rhythmic sequences where a given number of events are distributed over a given number of steps, in a manner that resembles the operation of Euclid’s algorithm. As a module, it produces impulses intended as trigger signal, which could be plugged into a separate synthesiser or indeed a second rhythm generator to add further complexity. Such configuring and reconfiguring of pattern generation modules is a very tangible form of live coding (Hutchins 2015).

4 Patterns in Computer Music

Returning to the 1981 text mentioned above, Laurie Spiegel (1981) offers a library of techniques of the most elementary transformations of musical patterns as they appeared to her at the time. Her aim was to present computer musicians with patterns that are “tried-and-true” from the musical tradition. Spiegel describes the twelve pattern operations, many of whom come from the domain of traditional musical composition, and with names that are already evocative: 1. Transposition, 2. Reversal, 3. Rotation, 4. Phase Offset, 5. Rescaling, 6. Interpolation, 7. Extrapolation, 8. Fragmentation, 9. Substitution, 10. Combination, 11. Sequencing, 12. Repetition. Under a thirteenth title, “The Great Unknown”, Spiegel discusses the possibility of discovering further patternings in the future.

Any attempt to review pattern languages should focus both on general classes of pattern representation and transformations, as well as the detail of implementation, where even small differences can have fundamental results on the music. In computer music it becomes clearer that any musical data is of a numerical nature, so any algorithmic procedure can be applied to a row of numbers. A case study in the difficulty of representing a general agreement in the meaning of pattern-like words could be described by a patterning function such as reversal.

4.1 A Case Study: Reversal

Reversal is an example of a patterning function with an operation that seems straightforward to implement, but looking deeper we find a large scope for variety in both implementation and use. This underlines a central point: that each of Spiegel’s classes of pattern is not a constraint, but a heuristic for guiding us, whose operation varies wildly depending on such things as our conception of time, of an event, the scale(s) at which we working and the other elements at play.

Let us begin with reversing a trivial sequence, such as `a-b-c-d-`. Perhaps the most obvious reversal would be `d-c-b-a-`, but this already carries a number of assumptions. Firstly, that we are reversing a whole sequence, rather than subsections of it; if the sequence represented two bars, we might instead decide to reverse every bar, rather than a whole sequence. This would make particular sense if we were trying to reverse a sequence of unknown or infinite length, as can often be the case in algorithmic music. In this case, we might end up with `b-a-d-c-`.

Reversal gets more complex than this. Looking closer at our sequence, we see dashes, which might represent continuations of the previous symbol, or rests. If we reverse the whole sequence, by assuming that a dash is a rest, we would end up with `-d-c-b-a`. In our original reversal of `d-c-b-a-`, we quietly assumed that a dash was a continuation, which in turn hides all manner of detail about the nature of an event and its representation. Operationally, what do we think a reversal is doing? One answer might be subtracting the onset of each event from the total duration of its pattern. However, if we are reversing an event, shouldn't the event onsets and offsets be swapped? This really depends on the nature of the pattern you are working with; in music, event durations often have an expressive quality quite separate from those of onsets, and so if we swap onsets with offsets, the result will be incoherent. There are still further issues at play with reversal, for instance the user might expect each sound sample played should be reversed in time, or its expressive envelope played backwards.

The above example illustrates the problem of signification and interpretation of patterning function names, such as those listed in Spiegel's article. Furthermore, it points to the hermeneutic problem of translation, transmission and interpretation when pattern functions from one language are written into another. Each programming language creates its own vocabulary that defines a particular way of thinking, which means we can only go so far in generalising concepts of pattern making across them. This lack of standards contributes to the beauty and diversity in the ecosystem of expressive languages and it should be celebrated.

5 Pattern Libraries in Computer Music Systems

Patterns often begin with sequences, and there are many nuances to the question of how to represent musical sequences in a programming language. How can they be notated such that it becomes intuitive and natural for the composer to write music of any musical genre? How should the traditional notions of pitch and duration be represented, for example? How can the design solve the problem of an event duration with a note length, or sustain, that exceeds the time of the duration? And moreover, since we are writing for synthesized sound, how can do we notate timbre, envelopes, and other synthesis parameters that, clearly, should be controllable from any pattern system?

The SuperCollider language includes an advanced pattern library, but one could argue that it is difficult to read the musical sequences written in it. Having created the following SuperCollider SynthDef called "piano"

```
SynthDef(\piano, { arg out=0, freq=440, amp=0.1, gate=1, decay=0.8, sustain=0,
                   mix=0.4, room=0.5, damp=0.5;
        var signal, reverb;
        signal = MdaPiano.ar(freq, gate, release: 0.9, stereo: 0.3, decay:decay,
        sustain: sustain);
        reverb = FreeVerb.ar(signal, mix, room, damp);
    DetectSilence.ar(reverb, 0.01, doneAction:2);
    Out.ar(out, reverb * amp);
}).add;
```

one could write a sequence like this, transcribing Mozart's Piano Sonata No 16 in C major:

```
Pbind(
  \instrument, \piano,
  \midinote,Pseq([72,76,79,71,72,74,72,81,79,84,79,77,76,77,76],1),
  \dur,Pseq([4,2,2,3,0.5,0.5,4,4,2,2,2,1,0.5,0.5,4]/4,1)
 ).play
```

Here the Pbind "binds" values to keys, where the key `\instrument` is given a symbol (the name of the instrument or synth definition to be used), and other keys, such as `\midinote` or `\dur` are given another pattern - Pseq - which is a specifies a sequence as a list, with the number of times the sequence is repeated following the value array. Pseq could then be swapped out with other pattern types, such as: Prand, Pser, Pshuf, Ptuple, Place, Pslide, Pwalk, and other list patterns whose names try to express their functionality: a pattern that will be randomised, a pattern series, a pattern to be shuffled, interlaced, etc.

One problem with the above - particularly when compositions become more complex - is that the music is difficult for a human to read, for example the list of note lengths is not visually aligned with the list of pitches. This could be lined up with spaces, but this quickly becomes too time consuming to be practical. One solution could be grouping the pitch and duration into a sub array, as follows:

```
Pbind(
  \instrument, \piano,
  [\midinote, \dur], Pseq([[72,1], [76, 0.5], [79, 0.5], [71, 0.75], [72, 0.125],
                           [74, 0.125], [72, 1], [81, 1], [79,0.5], [84, 0.5],
                           [79, 0.5], [77, 0.25], [76, 0.125], [77, 0.125],
                           [76,1]], 1)
 ).play
```

Some might find this representation more logical, as musical events are here spatially grouped. Of course any parameter in the synth can be controlled, and we could add features such as note sustain and the damp of the reverb:

```
Pbind(
        \instrument, \piano,
         \midinote,Pseq([72,76,79,71,72,74,72,81,79,84,79,77,76,77,76],1),
         \dur,Pseq([4,2,2,3,0.5,0.5,4,4,2,2,2,1,0.5,0.5,4]/4,1),
         \sustain,Pseq([1,0.2,0.2,0.5,0.25,0.25,0.5,1,0.5,0.5,0.5,0.5,0.25,
                      0.25,2]/4,1),
         \damp,Pseq([0.5,0.4,0.2,0.5,0.5,0.45,0.5,0.3,0.5,0.5,0.5,0.5,0.25,
                   0.45,0.5]/4,1),
 ).play
```

This could also be represented with a data collection called "Event". Here each event contains all the information, such as note/frequency, duration, amplitude, or any other parameter that the user might want to control in the synth. In the example below, it makes sense to use the event system for the right hand, but since the left hand plays notes of the same duration (quarter note) throughout, it can be simply represented with a Pseq:

```
Ppar([
// right hand - using the Event-style notation
Pseq([
        (\instrument: \piano, \midinote: 72, \dur: 1),
        (\instrument: \piano, \midinote: 76, \dur: 0.5),
        (\instrument: \piano, \midinote: 79, \dur: 0.5),
        (\instrument: \piano, \midinote: 71, \dur: 0.75),
        (\instrument: \piano, \midinote: 72, \dur: 0.125),
        (\instrument: \piano, \midinote: 74, \dur: 0.125),
        (\instrument: \piano, \midinote: 72, \dur: 1),
        (\instrument: \piano, \midinote: 81, \dur: 1),
        (\instrument: \piano, \midinote: 79, \dur: 0.5),
        (\instrument: \piano, \midinote: 84, \dur: 0.5),
        (\instrument: \piano, \midinote: 79, \dur: 0.5),
        (\instrument: \piano, \midinote: 77, \dur: 0.25),
        (\instrument: \piano, \midinote: 76, \dur: 0.125),
        (\instrument: \piano, \midinote: 77, \dur: 0.125),
        (\instrument: \piano, \midinote: 76, \dur: 1)
], 1),

// left hand - array notation
Pbind(\instrument, \piano,
        \midinote, Pseq([60,67,64,67,60,67,64,67,62,67,65,67,60,67,64,67,
                        60,69,65,69,60,67,64,67,59,67,62,67,60,67,64,67 ],1),
        \dur, 0.25
        )], 1).play
)
```

This is all good - we are notating music very much as we do in the traditional Western notation, but here in the language of the computer. The pattern system in SuperCollider does well what it is designed for, but in this chapter we are wanting to explore pattern transformation, and live manipulation of pattern. Such live manipulation of running patterns is a little harder to do in SuperCollider, unless you redefine patterns whilst they are running (using Pdefs, for example) or use PatternProxy's like below:

```
~note = PatternProxy(Pseq([72,76,79,71,72,74,72,81,79,84,79,77,76,77,76], inf));
(
Pbind(
        \instrument, \piano,
         \midinote, ~note,
         \dur, Pseq([4,2,2,3,0.5,0.5,4,4,2,2,2,1,0.5,0.5,4]/4, inf)
 ).play
)

~note.source = Pshuf([72,76,79,71,72,74,72,81,79,84,79,77,76,77,76], inf);
```

In the example above, a pattern sequence (Pseq) has been placed in a proxy whose source can be redefined in runtime. However, the syntax for this is less suitable for live coding or real-time experimentation/composition.

As a domain specific language for music, SuperCollider provides a comprehensive set of methods of array manipulation which are often of musical nature. These are not explicitly part of the pattern system, but methods that work on lists. The examples below show how list transformations (a form of what we call *patterning* in this chapter) are reintroduced into the source of a playing pattern:

```
a = [72, 76, 79, 71, 72, 74, 72, 81, 79, 84, 79, 77, 76, 77, 76];
b = a+7; // up a fifth
```

```
~note.source = Pseq(b,inf);

b = a.reverse; // reverse
~note.source = Pseq(b,inf);

b = a.scramble; // randomize the pattern
~note.source = Pseq(b,inf);

b = a.pyramid; // create a pyramid structure of the pattern
~note.source = Pseq(b,inf);

// or custom made algorithms
b = a.collect({arg note; if(note.even, {note+7}, {note-5})})
~note.source = Pseq(b,inf);
```

Another approach could be to create Pattern definitions which contain keys (such as Pseq or Prand) standing for sub patterns, which can be hot swapped in real-time:

```
Pdefn(\notes, Pseq([72,76,79,71,72,74,72,81,79,84,79,77,76,77,76], inf));

Pbindef(\x,
    \midinote, Pdefn(\notes),
    \dur, 0.125
).play;

Pdefn(\notes, Prand([72,76,79,71,72,74,72,81,79,84,79,77,76,77,76], inf));
```

As seen above, the SuperCollider pattern system is highly flexible and productive system to work with. It is ideal for the writing of complex algorithmic pieces and it has inspired other musical languages for almost two decades. People equally write sequences by hand or write algorithmic pattern generators. However, the system is not straightforward to compose with, and certainly not in a live performance context such as the one we find in the practice of live coding.[2]

For this reason, this chapter will explore the representation of patterns in two systems designed to be written, understood and manipulated easily in real-time, in particular during improvised live coding performances. We present *ixi lang* and *TidalCycles*, explaining how the design of these systems are aimed at live performance, live coding and real-time sketching in a compositional process. The systems are both high-level, constraining the user possibilities to a higher degree than SuperCollider, but what is gained is the speed of writing music, and the 'pleasure' of the constrained system (Magnusson and Hurtado Mendieta 2007).

5.1 ixi lang

ixi lang is a high level mini-language written on top of SuperCollider. It establishes a language that translates high level notation into the SuperCollider pattern system. The system aims at fast composition, readability, and high tolerance for syntactical mistakes. ixi lang allows for a convenient way of exploring musical patterns and reverts to prior states effortlessly, either through undoing, or saving the state of

[2]Although some third party libraries do extend the Pattern functionality of SuperCollider, noteably JITlib and the PLx quark.

the code and the running patterns. This is achieved through creating system agents that are assigned performance scores. Agent behavior can be changed through calling methods (or verbs), which in turn update the code for the agent, turning the code into a form of visual (yet still textual) score.

The key elements of the language relate to melody and rhythm, and these are controlled through the use of agents that are assigned a score:

```
lucy -> |q   q   q c q   |
```

Here the agent 'lucy' has been given a rhythmic score (specified by using the pipe "|" symbol) where the characters 'q' and 'c' stand for sampled sounds, a range of which have been mapped to the letters of the roman alphabet. The spaces are silences, so the use of monospaced font is essential in ixi lang. It would be easy to create a polymeter by adding another agent:

```
yoko -> |q   q   q c q   |
john -> |z    z x z   |
```

A fundamental feature here is to represent features of time and event as close together as possible, to ease cognitive load and speed up the compositional process. In ixi lang the elements of time and sound/note are the most important features and they gain primary representation in the notational language. Secondary parameters, such as amplitude, panning, note length, etc. can then be written behind the score as post-fix sequences, which can add further descriptive transformations of the events:

```
john -> |z    z x z   | <1928> // panning from left (1) to right (9)

john -> |z    z x z   | (1442) // note sustain (awhole note, two quarter notes and
                               // a half note)

john -> |z    z x z   | ^4419^ // amplitude

john -> |z    z x z   | !16 // wait sixteen steps before repeating the pattern
```

and all can be combined, of course

```
john -> |z    z x z   | ^4419^ (1992) <195528> ! 12
```

Note how there are six values in the panning argument. Here it wraps around, such that the second time the pattern plays, the first 'z' will be panned 2 to the left. This creates a polyrhythmic effect.

Another mode of ixi lang is the melodic mode. Here the items in the score represent pitches and are therefore numerical:

```
scale minor
paul -> obo[1 2 4   1   2   ]
```

Above we have basic rhythmic and melodic sequences. There are 'actions' that can be applied to the agents, for example

```
swap paul

paul -> obo[4 1 1   2   2   ]

shake paul
```

Other methods include revert, expand, >shift, transpose, etc, much in the spirit of what we find in Spiegel's text referenced above. These pattern transformations can be set to take place in time, automatically, for example using the future function:

```
future 4b:20 >> shake yoko
```

Here the score of agent yoko will be shaken (scrambled) every 4 bars, twenty times. An important feature of the ixi lang is that the score is updated in the document when it is transformed through code: the code in the document rewrites itself. The way this happens is that the text is highlighted for half a second in a different color and then the score is replaced with a new and running score.

Agents can also receive effects, such as

```
yoko >> distort >> reverb
```

Here the output of agent yoko is routed through a distortion effect and then through a reverb effect unit. << removes all effects. Here we find another example of ixi lang's graphical design, where the idea is that the ">>" operator is a visual reference to jack cables used with electric guitars.

In terms of tempo, it is clear how easy it is to create polymeter in ixi lang:

```
yoko -> |q   q   q c q   |
koyo -> |a   a s a   |
```

However, for polyrhythm, the calculations would have to be slightly more advanced:

```
yoko -> |q   q   q c q   |
koyo -> |a   a s a   |*1.333333
```

where agent koyo has now been stretched to the length of yoko.

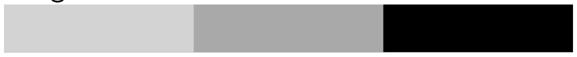
ixi lang is a notational live coding language. On its own it is not fully fledged programming language, but it harnesses the power of SuperCollider for more complex coding. The focus here is on speedy input, redesign, reevaluation, manipulation of agents' scores and the routing of agents' output through effects.

6 TidalCycles

TidalCycles, known as 'Tidal' for short, is a mini-language for pattern embedded in Haskell, a pure functional programming language. This functional basis allows Tidal to define patterns within generalised type structures, which in practical terms means Tidal has a very strict, formal model of what constitutes a pattern, yet is highly flexible in how those patterns are expressed and combined together.

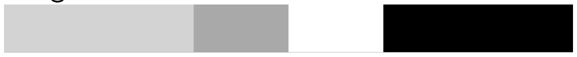
Figure 1: The below demonstrates some features from Tidal's terse, yet expressive mini-language for specifying sequences. It is also possible to represent different kinds of polyrhythm, spread sequences over multiple cycles, and add rests and random variation; see the Tidal documentation for full details.

`"light dark black"`



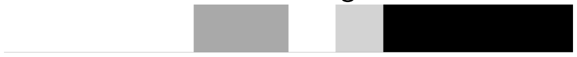
(a) Sequences are specified in double quotes, with steps separated by spaces.

`"light [dark white] black"`



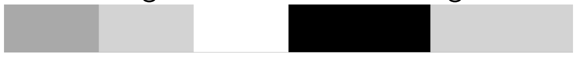
(b) Steps can be broken down into subsequences using square brackets.

`"white [dark [white light]] black"`



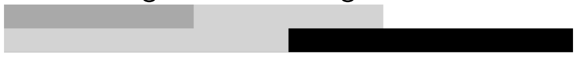
(c) Steps within subsequences can be further broken down.

`"[dark light white] [black light]"`



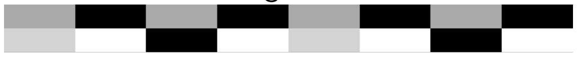
(d) Steps may be broken down into irregular parts.

`"[dark light white, light black]"`



(e) A step may be broken down into more than one subsequence, creating polymeter.

`"[dark black, [light black]/2 white]*4"`



(f) A step may be slowed down with the symbol '/', and sped up with '*'.

Tidal is a domain specific language, in that it provides an alternative model of computation designed for its domain of patterning. In conventional (i.e. general-purpose, imperative) languages, statements within an algorithm describe a list of steps to be evaluated one after another, over time. In Tidal, a first step produces a sequence of events over time, and then each successive step may transform that pattern. In other words, time in Tidal is represented not by control flow, but by a functional relationship between time and events. A pattern may therefore be transformed in terms of time (e.g. reversing, slowing down or stuttering), in terms of events (e.g. transposing, inverting), or combined with another pattern through the combination or juxtaposition of events over time. This flexibility results in set of simple operators and functions which offer an explosion of possibilities in how they may be combined together.

Tidal is really two languages in one, a language for sequencing events, and another for combining and transforming those sequences into a pattern. Some of the sequencing aspects of Tidal are illustrated in Figure 1, demonstrating how polyphonic sequences with compound and polyrhythmic time structures can be specified using a terse syntax. The remainder of this section is focussed on combining and transforming these sequences as patterns.

In Tidal, a `Pattern` type is defined as instance of Haskell's applicative functor type, which simply means that the end-user live coder can treat whole patterns of things as if they were single things. This needs an example:

```
"1 2 3" + "4 5 6"
```

The above uses the addition function `+` to add together the two sequences `"1 2 3"` and `"4 5 6"` to create `"5 7 9"`. Tidal defines which pairs of numbers are given to the `+` operator in order to construct a new pattern, so that the end-user need only think about what combination they want, rather than how it should be achieved. The advantage of this declarative approach to combining patterns becomes clearer in the below slightly more complex example, which combines two patterns with different structures:

```
"1 [2 3] 4"  + "1 2"
```

The above results in the pattern `"2 [3 5] 6"`, demonstrating that the structure of the first pattern is maintained, with the first half having `1` added, and the second half having `2` added. This split extends into the two steps of the middle subsequence.[3]

6.1 Timbral dimensions

So far we have been discussing Tidal in the abstract, even using patterns of colour rather than sound to illustrate its output. The abstract nature of Tidal's approach is also its strength, in that many of its functions are polymorphic, operating on patterns containing values of any particular type. However, lets ground our discussion with a more musical example. The following is a Tidal pattern composed of simple parts, with complex results:

```
jux (iter 4) $ (every 3 (density 1.5) $
             sound (pick <$> "bd can*2 [sn cp] can" <*> (slow 8 $ scan 8)))
             # speed (slow 4 $ sine1 + 1)
             # delay "1"
             # delayfeedback "0.7"
             # delaytime "[0.02 0.01 0.03 0.02]/3"
             # vowel "[e x a, x i x i]/4"
```

The first two lines of the above centre around the `sound` pattern, specifying some sound samples (`bd`, `can`, `sn` and `cp`), which are combined with a pattern of numbers generated by `scan` using the `pick` function, which together steps through variants of those samples in a way that gradually increases complexity. In addition, every 3rd cycle of the pattern has its `density` increased by 50%.[4] The remaining lines combine the sound pattern with effect patterns; the `speed` of sample playback (i.e. changing its pitch) follows a sinewave over four cycles, a comb filter-style delay effect cycles through four values over three cycles, and a `vowel` filter has a polyphonic pattern of `e`, `a` and `i` formants applied over four cycles. Finally, the use of `jux` and `iter` causes the whole pattern to shift in steps of a quarter cycle every cycle, but only in the right hand channel, creating a stereo panning effect.

Although the above example is relatively straightforward, it contains patterns of different types being composed together into patterns of synthesiser control messages, their time structure being manipulated, and functions being selectively applied in terms of time (in this case with `every`) and space (in this case with `jux`). The majority of functions in Tidal take one or more patterns as input, and produce another pattern as output, and so it is easy to chain simple transformations together and achieve rich results.

One particularly interesting Tidal function is `weave`, which in the following case combines three `sound` patterns using a fourth `pan` pattern:

```
weave 16 (pan sine1)
  [sound "bd sn cp",
   sound "casio casio:1",
   sound "[jvbass*2 jvbass:2]/2",
   sound "hc*4"
  ]
```

By design, `weave` offsets each of the `sound` patterns in time, after applying the `pan` pattern, which is itself stretched over the given number of 16 cycles. The end result is that the three patterns are spatialised,

[3] If you are having trouble understanding the structure of `"2 [3 5] 6"`, see Figure 1b, which shows a colour example with the same structure.

[4] For the purposes of this chapter, we only need a general understanding of this code; for a closer understanding, refer to the tidal tutorial on the Tidal website (tidalcycles.org).

each moving between the two (or potentially, multichannel) speakers following a sine wave pattern, but phase-shifted, so that when two of the patterns are hard left and right, the other two are meeting each other in the centre.

The `weave` function was designed with this spatialisation technique in mind, but can be applied to any effect, for instance it works well to have distortion effects rising and falling across different patterns in different phases. There are, however, surprising affordances which fall out of this generalisation. We may use 'weave' in a different way:

```
jux rev $ weave 16 (sound (samples "arpy*8" (run 8)))
  [vowel "a e i",
   vowel "i [i o] o u",
   vowel "[e o]/3 [i o u]/2"
  ]
```

Instead of applying different phases of an effect pattern to a set of sound patterns, the above does the opposite, applying different phases of a sound pattern to a set of different effects. In musical terms, the result is a canon; the `run` function in the above gives us a rising scale of `arpy` notes, and the overlaying of different phases results in a canon that seems to continuously rise. Furthermore, swapping the sound and effect patterns in this way causes the rhythmic structure of each part to come not from the sound pattern, but from the effect patterns, so that the combined result is a rich polymetry.

The latter usage of `weave` to create canons was discovered by chance. The structural correspondence between patterns arranged over time in pitch and in space are not surprising, but it is enjoyable to see this functionality appear almost by magic, through the process of generalising patterning functions.

6.2 Levels of patterning

Tidal's functional approach leads to a multilayered view of pattern making, where each layer builds upon those beneath. On the base layer, we find a view of patterns as *sequences*, which are potentially polyphonic, or polymetric, but are described in a linear, imperative fashion, as we have seen in the SuperCollider examples above.

Also familiar to our common conception of pattern is the concept of *symmetry* in pattern, which in musical terms can be understood in terms of time, for example reversal or rhythmic rotation, or in terms of note values, for example inversion.

Pattern can also be understood in terms of *deviation* from a structure: imperfections, glitches and confounded expectations, which can be explored through the introduction of random number generators. For example in Tidal, the `sometimes` combinator can be used to apply a given combinator, but only sometimes (`rarely` and `often` are also available).

We should also not forget the key importance of *composition* to pattern making; of joining together different patterns, and the method of joining itself becoming a core part of the pattern. A special form of composition is *interference*, where constituent patterns combine to form a new pattern, with features not present in the originals. Such interference patterns can be well understood from patterns in weaving, where colour sequences in warp and weft threads, combined with the weave structure at play, form surprising images (Harlizius-Klück, 2008). Such interference is explained in our above description of Tidal's aptly named 'weave' function.

As we climb up these layers of sequencing, symmetry, glitch, composition and interference, we find that code becomes increasing important to the creative process, in generating surprising results that are otherwise beyond the imagination of the programmer. At this point code becomes more like physical material, with results emerging through continual reaction to sensory feedback, rather than transcription of a pure idea. In practice then, Tidal's focus on higher order patterns and interference affords an improvisatory approach to music, where language becomes an exploratory environment. Perhaps it is not too controversial to wonder whether interference patterns, which can also be found in ixi lang, SuperCollider and any other programming language, lie at the very heart of algorithmic music.

7 Conclusion

An algorithm is a description of a pattern. By defining an algorithm, the author engages in a process of generalisation and normalisation, where certain features are notated into patterns that can be abstracted into a standardised system of notation. With the advent of automated machines the ideal sequence repeating engine appeared, where diverse input mechanisms, such as the punch card, can be used to represent the patterns. However, with computer code this becomes infinitely more powerful as we have language constructions that make this very flexible, for example with the use of for-loops and recursion. With a meta-machine controlled by the programming language, we can automatically generate, transform, and analyse patterns from new data, and represent them across different media domains.

High level pattern languages are useful as they are mini-languages 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 might or might not be easy to write in a standard language. There are no standards for pattern languages - so the implementation of Spiegel's methods would hardly ever be unified across different systems. The system's method names thus become semantic entities in the compositional thinking of the composer/performer. They outline the scope of the possible.

The creation of patterns in computer music languages is therefore a two-edged sword: on the one hand it provides a language, a vocabulary, a 'technology for thinking' about music, which enables the composer to build structures through the scaffolding of the system. On the other hand, the patterns are compositional structures on their own, thus influencing, perhaps limiting or directing the compositional thoughts in a way that would not have been the case if the composer had to write their own pattern systems. For us, this is a question about time, affordances, 'ready-at-handness' of a musical system that should be capable of an engaging performance in the practice of coding in front of a live audience.

We will probably never have high level, universal definitions of patterns, because interpretations of what the linguistic signifiers stand for can differ greatly. Every system presents its own approach for working with patterns and some might reject the idea of writing patterns completely, expecting that such work is of a compositional nature and should be done by the user.

The two systems discussed in this chapter - ixi lang and Tidal - are constrained, purpose built live coding systems that are used by people all over the world. The authors have given workshops, presentations and performances with the systems and they have become relatively well known in the world of computer music performance. Although they have proven to be good tools for musical performance, perhaps the greatest contribution with this research has been to rethink the computer language design and purpose, where performance and the conception of the code as something that be sculpted in real-time is given a high priority.

This chapter has focussed on the computer music language as something we use in sketching and performing real-time music. Although the focus has been on performance, we are not excluding the possibility, well covered elsewhere in this book, that we might start to perform with agents of computational creativity, where creative processes are delegated to AI agents. Indeed, such features are already taking shape in ixi lang's "autocode" function where the language begins coding on its own, as well as the application of evolutionary algorithms to Tidal (Hickenbotham and Stepney, 2016), both already resulting in some fine music.

8 Acknowledgements

Thanks to Geraint Wiggins and James Harkins for discussing some topics in this text with us during the writing. Also, many thanks to the editor of the text, Roger Dean, for superb comments and pointers.

This is a draft of a chapter/article that has been accepted for publication by Oxford University Press in the forthcoming book Oxford Handbook on Algorithmic Music, edited by McLean and Dean, due for publication in 2018.

9 Bibliography

Burack, J. 2005. “Uniting Mind and Music: Shaw’s Vision Continues.” *American Music Teacher* 55 (1): 84–87.

Collins, Nick. 2006. “Towards Autonomous Agents for Live Computer Music: Realtime Machine Listening and Interactive Music Systems.” PhD thesis, Centre for Science; Music, Faculty of Music, University of Cambridge.

du Sautoy, Marcus. 2003. *The Music of the Primes.* Fourth Estate.

Ebciolu, Kemal. 1988. “An Expert System for Harmonizing Four-Part Chorales.” *Computer Music Journal* 12 (3): 43–51.

Fauvel, John; Raymond Flood, and Robin Wilson. 2006. *Music and Mathematics: from Pythagoras to Fractals.* Oxford University Press.

Harlizius-Klück, Ellen. 2008. “Arithmetics and Weaving. From Penelope’s Loom to Computing (poster).” In *8. Münchner Wissenschaftstage.*

Hayles, Katherine. 1999. *How We Became Posthuman: Virtual Bodies in Cybernetics, Literature, and Informatics.* University of Chicago Press.

Hutchins, Charles. 2015. “Live Patch / Live Code.” In *Proceedings of the 1st International Conference on Live Coding.*

Magnusson, Thor, and Enrike Hurtado Mendieta. 2007. “The Acoustic, the Digital and the Body: a Survey on Musical Instruments.” *New Interfaces for Musical Expression (NIME07),*

Merleau-Ponty, Maurice. 2002. *Phenomenology of Perception: an Introduction. 2 Edition.* Routledge.

Parente, Thomas J. 2015. *The Positive Pianist: How Flow Can Bring Passion to Practice and Performance.* Oxford University Press, London ; New York :

Patel, Aniruddh D., and John R. Iversen. 2014. “The Evolutionary Neuroscience of Musical Beat Perception: the Action Simulation for Auditory Prediction (ASAP) Hypothesis.” *Frontiers in Systems Neuroscience* 8.

Sloboda, John. 2005. *Exploring the Musical Mind: Cognition, Emotion, Ability, Function.* Oxford University Press.

Hickinbotham, Simon, and Susan Stepney. 2016. “Augmenting live coding with evolved patterns.” In *Proceedings of the 5th International Conference on Evolutionary and Biologically Inspired Music, Sound, Art and Design (EvoMUSART).*

Spiegel, Laurie. 1981. "Manipulations of Musical Patterns." In *Proceedings of the Symposium on Small Computers and the Arts*, 19–22.

Stowell, Dan, and Alex McLean. 2012. "Live Music-Making: a Rich Open Task Requires a Rich Open Interface." In *Music and Human-Computer Interaction*, edited by S. Holland, K. Wilkie, P. Mulholland, and A. Seago, 139–152. Springer.

Sudnow, David. 1993. *Ways of the Hand: Organization of Improvised Conduct.* MIT Press, Cambridge, MA.

Taube, Heinrich K. 2004. *Notes from the Metalevel: Introduction to Algorithmic Music Composition.* Lisse, The Netherlands: Swets & Zeitlinger.