Algorithmic Pattern

Alex McLean

0.1 Defining algorithmic pattern

This paper explores the world of algorithmic pattern, and the ways in which it offers an interface to the computer musician. Let's first look at the words pattern and algorithm separately, before putting them together.

Patterns are present everywhere, certainly in textiles, choreography, mathematics, design and music. However, at first glance, the use of the word in music seems comparatively impoverished, at least in the West. At the time of writing, the English language wikipedia page on pattern has no mention of music, and furthermore, when musicians refer to a pattern, they usually mean any sequence that repeats. The word can even take on a pejorative sense in music, for example in a conference paper on transdisciplinary collaboration Hugill recounts how (and I paraphrase) mathematicians spend their careers searching for patterns, while many composers will be seriously offended if you accuse them of making patterns.

By comparison, pattern in textiles is alive with compositional techniques, not only repetitions but reflections and symmetries, and generative structural compositions. Once you scratch the surface however, music is of course also alive with patterns across compositional techniques. For example, the familiar canon structures, where one voice imitates another, played over the top with a delay. So it seems that the difference here is the way words are used; in music, a canon contains patterns, whereas in textiles, such a structure based on glide symmetry could itself be referred to as the pattern.

What *is* a Pattern? Grünbaum and Shephard define patterns as "designs repeating some motif in a more or less systematic manner." They write in the context of geometric tilings, but the same definition largely holds in the music fields; a sequence becomes a pattern, once it is repeated. However, in music we too often focus on the repetition, and not the systematic manner in which it is repeated. For a pattern to be interesting, we need to do more than repeat it; the repetition only provides the metrical ground on which the pattern acts. Looking around the room you are in now, you will likely see patterns of repetition, but also of reflection, rotation, interference/moire, and glitch. The textiles around (or on) you may well have a visual pattern arising not from the colour of threads alone, but from computational interference between colour and structure.

Accordingly, in the present paper, we focus on pattern as a whole family of techniques for working with regularites in the world. Such patterns allow us to perceive repetition, reflection and interference in a material. In other words, the way we percieve pattern is inextricably linked with the structured movements of its making. And, once we are dealing with 'structure movements of making', we are in the world of algorithms.

An *algorithm* is a step-by-step set of instructions. Sometimes it is assumed that by a step-by-step set, we mean a sequence of instructions, but this is not the case; indeed the notion of an algorithm has been formalised as lambda calculus, which may involve recursive steps in to a function declaration, rather than stepping down a stateful sequence of instructions. This clarification becomes important later in this paper, when we address TidalCycles, a live coding environment embedded in the Haskell language, which is itself based on lambda calculus.

So, there is some sense that the words algorithm and pattern are synonyms, and therefore the phrase "algorithmic pattern" is a tautology. This phrase is nonetheless useful for on one hand for clarifying that we address algorithms not just as software engineering tools, but as formalised ways of making that can be perceived in end-results. On the other hand, it clarifies that patterns are not simple sequences, but structural qualities. This builds a perspective that creates a generative and perceptual connection between creation and reception.

0.2 Patterns in NIME

The meaning of *algorithmic pattern* has a particular resonance in interface design, especially in the NIME (New Interfaces for Musical Expression) field. When we write something as an algorithmic pattern, we work at least one step removed from the surface of a 'target domain' such as musical notes. By analogy, we don't hit the drum, we write about hitting the drum. Not even that, we write about relationships between drum hits, the structures that lead to one movement following another. This is a trade off, which creates distance between ourselves and an instrument, losing direct tactile control, but which brings us 'up' to work on a compositional, structural level. Here, we lose physical connection to a drum skin, but instead work in a way where a very small change can create far reaching, even unexpected changes in the music as it unfolds. This generative relationship with algorithmic pattern is what we explore in the following, taking first live coding of music, and then live weaving of textiles as comparative examples.

As things stand, what does the word pattern mean to NIME authors? According to use of the `pdfgrep` utility on a corpus of 1,739 NIME papers, word 'pattern' can be found in 803 papers (46% of the total) of which 315 (18% of the total) contain it more than twice. These were ranked by incidence of the word, and the top twenty sampled and examined (ranging from 26 to 150 occurrences per paper), to gain an impression of how the word is used in the community. Of these twenty papers, I deemed only one (???) explored pattern as an activity, in the context of patterns of interaction emerging

and continually changing in motor feedback loops. Of the remaining papers, nine discussed transformation of patterns in some way (???), although referred to patterns as the end result, rather than as a generation or transformation process or notation. The remaining eleven papers (???@) referred to patterns as fixed sequences. There are no satisfying definitions of pattern, although (???) writes "A pattern can be any part of a score, a MIDI sequence, or a pre-recorded sound".

Wanting to find more examples of papers treating pattern as activity or behaviour rather than sequence, I searched again with the gerund patterning. This returned nine papers, none of which had more than two instances of the word. Most were passing references, but (???) used patterning to refer to the combination of two patterns, (???) described musical form as a process, and patterns in terms of how they relate to one another, (???) as live coded algorithms as patterned behaviour, and (???) quote Meyer in defining a composer's style as connecting patterns in human behaviour with patterns in results. I count all of these as fitting the definition of algorithmic pattern.

0.3 Algorithmic Pattern in Computer Music Interfaces

Algorithmic and generative music systems often come with high-minded claims of infinite variation or artificial intelligence. However on closer examination, these systems often rely upon surprisingly simple systems based on probability (e.g. 'Markov chains'), arbitrary decisions (randomness/chance) and straightforward sequencing, called 'algorithmic' simply because they are expressed as text rather than as a graphical piano roll. In her foundational essay "Manipulations of Musical Patterns", (???) looks beyond such methods, arguing convincingly for greater focus on pattern transformations in computer music interfaces, naming twelve categories of pattern transformation which, Spiegel argues, should be as central to computer music interfaces as copy and paste.

Our argument is not however that algorithmic pattern is complex. Musician and sound artist Mark Fell is one example of someone who fully embraces simplicity of a pattern-based approach to sound. In "Notes on Pattern Synthesis", (???) introduces the Max patches behind his acclaimed album Multistability, which work as studies of interaction within self-enforced guidelines. This minimalist approach results in music has great clarity, but is nonetheless complex in structure. The usual minimalist examples, such as Reich's clapping music fit here too. Simple in its patterned construction, but bringing forth astonishing detail in its outcome. This is core benefit to using a pattern as an interface, embracing the simplest ingredients, but transforming them and composing them together to create complex results. Far from new, this approach grounds discussion of music generation in a rich perspective, able to draw from an expansive variety of cultural practices and artefacts from around the world and across history. Pattern is a universal.

In the following paper, we explore the expansive world of pattern in-depth. One perspective will be from computer music, with focus on the representation and manipulation of pattern in the TidalCycles environment. However, pattern is a universal

concept, and we will also look at examples from textiles, particularly interference patterns in weaves and braids.

0.4 Pattern in TidalCycles

Work on TidalCycles (commonly *Tidal* for short) first began in 2009, and over the past decade has developed into a comprehensive, free/open source environment for algorithmic pattern, mainly in the context of live coding music. At heart, it is a domain specific language (DSL) and environment for patterning Open Sound Control network messages (???), embedded in the pure functional language Haskell. Tidal is most commonly used in tandem with the SuperDirt software, a hybrid framework for sample manipulation, synthesis and MIDI, implemented in SuperCollider. However, Tidal can be applied to any kind of pattern, and has indeed been used to pattern live choreographic scores (???), woven textiles (???), DMX-controlled lighting, and VJing.

While Tidal has been developed alongside creative practice, it upholds strong computer scientific principles. Crucially, a pattern is defined as a pure function, and therefore may be composed with other patterns flexibly and safely. As Tidal has developed, its core representation has grown more succinct, and a recent rewrite resulted in more rigorous understanding of what, in terms of Tidal, a pattern *is*. Tidal types Much, but not all of Tidal's notion of pattern comes from the way it represents pattern within Haskell's type system, as a pure function of time. This folllows work on pure functional reactive programming, where rather than representing data using lists, behaviour is represented, with functions. Accordingly, rather than representing a sequence as a list of events, Tidal represents it as a function, which takes a timespan as input, and returns all the events which are active during that time. In this way, the idea of a pattern being about behaviour rather than a sequence is embedded in Tidal's core.

Lets have a look at the type declarations themselves, describing each for those unfamiliar with the Haskell language.

```
data Arc = Arc {start :: Rational, stop :: Rational}
```

A timespan is expressed as an arc of time, consisting of a start and end time. Timespans are referred to as arcs, in sympathy with a cyclic notion of time. Importantly, time is represented at a rational (rather than floating point or integral) number, therefore allowing time to be arbitrarily subdivided with 100% accuracy.

```
data Event a = Event {
    whole :: Maybe Arc,
    part :: Arc,
    value :: a
}
```

An event contains a value of some type `a`, and two arcs. The `part` arc represents the timespan during which the event is active. An event may be a fraction of an event, in

which case the 'whole' arc gives the timespan for the original. If a whole is not set, this indicates that the event is analogue; that is, rather than having a discrete beginning and end, the value continuously changes, and the one that is returned is sampled at the midpoint of the query arc. This approach allows both discrete and continuously varying events to co-exist in the same pattern.

```
type Query a = (Arc -> [Event a])
data Pattern a = Pattern {query :: Query a,
                          controls :: StateMap
}
```

A query represents the pattern's behaviour, as a function from time to events. In particular, a query takes an arc as input, and returns a set of events which are active during that arc as output. The values of the events must all be of the same type, and the parts of the events will be contrained to the query arc. If an event's 'whole' extends beyond the query, it is returned as-is, but its 'part' is curtailed. In other words, you may be returned a fragment of an event for a given query, but will still have access to the arc of the whole, of which the event is part.

0.4.1 Tidal composition

The above strict definition of pattern does not say much about Tidal as an interface, but what follows from it is a rich approach to composition, supported by a large library of pattern combinators. Composition is meant here in both a musical and computer scientific sense, in terms of composing together musical behaviours into new, generally more complex behaviours. Tidal supports a multitude of ways to combine patterns together, most ride on how Tidal allows you to simply treat patterns as values, allowing you to take any function that combines two values, and use it to combine two patterns of values.

As a trivial example, lets combine two tidal patterns `fastcat [1,2,3]` and `fastcat [4,5]`. The first thing to note is that `fastcat` combines a list of patterns by concatenating them in a contiguous sequence, of equal durations over a cycle. The notion of a metrical cycle is important here, as it is most often the reference point in a composition, rather than a beat or step. So, we are combining two patterns with different structures - one has three events per cycle, and the other has two. We can visualise them like this:

```
fastcat [1,2,3]
```

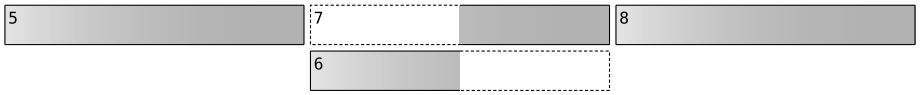

```
fastcat [4,5]
```

The default way to combine them is with +, producing the following:

```
fastcat [1,2,3] + fastcat [4,5]
```

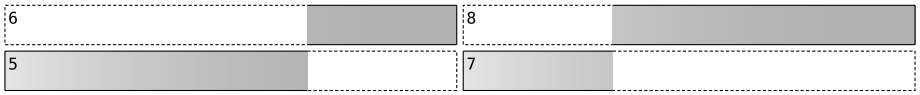
You can see that in the above, 1 gets added to part of 4, 2 gets split between 4 and 5, and 3 gets added to part of 5. An alternative operator, |+, privileges structure on the left. The same events are matched up, but the resulting events maintain the 'wholes' from the pattern on the left hand side of the operator:

```
fastcat [1,2,3] |+ fastcat [4,5]
```



Conversely, the +| operator privileges structure from the right hand pattern:

```
fastcat [1,2,3] +| fastcat [4,5]
```



Note that when such a pattern structure reaches the scheduler, only the events that have their onsets intact will result in an event actually being triggered. That is, the start of an event's 'part' must be the same as an event's 'whole' to result in sound, otherwise it represents a fragment of an event's tail, only useful for combining with other events. However, a model that works beyond trigger messages, allowing continual varying of a sound's parameters after it has been triggered, is at working prototype stage.

There is a library of such operators for basic arithmetic, but any function can be used to combine patterns together in this way by using Haskell's standard syntax for applicative functors, with Tidal's additional nonstandard operators <* and *> for privileging structure on the left or right. For example, to merge two colour patterns:

```
blend <$> (slow 4 sine) *> "red" <*> "blue orange"
```

Instead of using 'fastcat', the above

It is worth reiterating at this point that patterns are functions, and not data structures. By combining them in this way we are not computing anything, only creating a new pattern function composed of other pattern functions. No calculation actually take place until the resulting pattern is queried.

0.4.2 Patterned parameters

TidalCycles has a large library of combinators, but for the purpose of this paper we will focus on just one, the `fast` function, which simply speeds up (or for factors < 1 slows down) a pattern. It s definition is minimal, multiplying query time and dividing result time by a given factor:

```
fast' timepat pat =
  innerJoin ((\time -> withResultTime (/ time) $ withQueryTime (* time) pat)
              <$> timepat
            )
```

What is interesting about above is that the 'time' factor is itself a pattern. Using the `<$>` operator, time values inside the time pattern are applied to the manipulation of the pattern that is being manipulated. The result is a pattern of pattern of values, which are then joined back into a pattern of values using the `innerJoin` function. This higher order magic uses the much same procedure for combining two patterns described above, and turns a simple function for changing the speed of a pattern, to a highly flexible function for patterning the speed of a pattern.

```
fast (fastcat [2, 3, 0.5]) ("red pink")
```