Tidal patterns: A brief introduction

TidalCycles (or *Tidal* for short) is a livecoding environment McLean (2014), which approaches algorithmic music in terms of pattern, on multiple scales and levels. It is a kind of programming language, in the same sense as a spreadsheet is a programming language, but many people using it would not self-identify as professional programmers but musicians. Tidal therefore falls within the research field of End-User Programming (Blackwell 2006). It has been created as a system for making music, but may be used to pattern a wide range of media, such as live video, light control, or making fixed images.

Tidal is a Domain Specific Language (DSL), implemented as a library within the *Haskell* programming language. Haskell has a reputation for being difficult to learn, due to its basis in Lambda Calculus and strict type systems, but appears to be a remarkably warm host to musical pattern. We have already explored pattern in a historical context, as a particular approach to creating structure. This is somewhat against normal use of the word in electronic and computer music communities, where *pattern* often simply refers to any discrete sequence. However, by aiming our focus deeper into the structure of pattern we connect with this wider sense, referring to processes inherent in the *making* of sequences. From this perspective, pattern is not about sequences, but computational functions which, just like the weaver at their loom, transform and combine sequences. Transformations of musical patterns may be grouped into categories, for example Laurie Spiegel (1981) provides a list of twelve high-level groupings. In the following we put forward just four high-level categories to help address different levels of pattern transformation, namely *repetition*, *reflection*, *interference* and *deviation*, relating each of these categories to Tidal's representation of music.

Repetition - Patterns generally feature repetitions, often on multiple scales. Accordingly, Tidal's representation of time is based on repetition, with the primary reference point being the metric cycle rather than any fixed beat duration, an approach inspired by the work of Bernard Bel (2001) in formalising Tabla rhythms from Indian Classical Music. Because time is primarily structured by cycles, and not notes or other discrete steps, Tidal breaks from the usual notion of time *signature*, used in much Western music and music software. This results in a metrical fluidity, where cycles can be composed of subcycles with differing temporal structures. Furthermore, time can be stretched, compressed and otherwise manipulated in a variety of ways, meaning that while the cycle is the reference point, metrical structure can vary wildly from one cycle to the next. This allows support for polyrhythm, complex meters, and both flowing (continuous) and grid-based (discrete) patterns, subdividable to arbitrary levels of complexity.

Symmetry - Symmetrical forms are also common to many patterns, for example as reflection, conceived in music in various ways including inversion. With Tidal's cyclic notion of time, there is also the possibility of *rotational* symmetry, where events are moved forward and backwards in (cyclic) time.

Interference - Interference patterns occur in Tidal where two or more patterns can

be combined to create a new one, or a pattern combined with a transformed version of itself, creating complex 'higher-order' patterns out of simple parts. As already discussed above in the context of weaving, resulting interference patterns often bear little or no resemblance to the source material. Another familiar example of a visual interference is the *moire* patterning seen when netting overlaps. Tidal provides a multitude of methods for combining patterns (thanks in large part to Haskell's applicative and monadic constructs), and these methods may in turn be combined (again, thanks to Haskell's functional nature) to create new methods, providing a very large number of possibilities to explore.

Deviation - Where we create recognisable patterns, we set up expectations for what comes next, creating a musical sense of anticipation. Of course, musicians then purposefully break those expectations, to create glitches. Interestingly, the affect of anticipation and deviation from expectation is present even when a listener is very familiar with a piece of music, it is still perceived as musical structure (Huron 2008). With Tidal we can break expectations through chance operations, using 'pseudo-random number' generators – mathematical processes useful for taking arbitrary choices. Such deviation allows a musician to forego making a decision by making a 'random' choice, or add glitches and imperfections as an aesthetic choice in its own right.

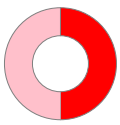
In exploring these different flavours and levels of patternings, it becomes clear that they are rarely independent. For example how many different kinds of symmetry can you add to a pattern before they begin to interfere with each other? How much deviation or interference can you add before the sense of repetition breaks down? How long can a repetition be before you stop perceiving it as such? There is much music to be found on these boundaries.

Tidal does not make sound itself, but as a pattern engine sends messages to synthesisers (hardware or software) to make sound. By default, Tidal is used with *SuperDirt*, a synthesis framework system designed for use with Tidal and implemented in the SuperCollider environment. We will begin however with patterns of colour, in sympathy with the present medium.

Transforming sequences

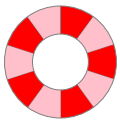
Tidal really comes in two parts, a 'little language' for describing sequences, and a library of functions for transforming them as patterns. The little language for sequences is denoted with double quotes:

```
"red pink"
```



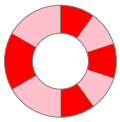
The above pattern is rendered above in a circle, in order to put Tidal's central notion of cyclic pattern in visual form – the end of one cycle is also the beginning of the next. There is much that we can do inside these double quotes to describe complex sequences, but first, lets introduce a simple way to transform this sequence using the `fast` function to 'speed it up':

```
fast 5 "red pink"
```



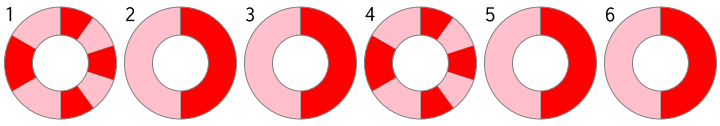
Now we can see five repetitions of red and pink within a single cycle. It is worth noting that the number `5`, as a parameter to `fast` in the above, is itself a pattern. When we give a pattern as a 'bare' number like this, it simply repeats that number, once per cycle. The below gives a sequence of two numbers to `fast` instead, so that the first half of each cycle is 'speeded up' by a factor of five, and the second half by a factor of three.

```
fast "5 3" "red pink"
```



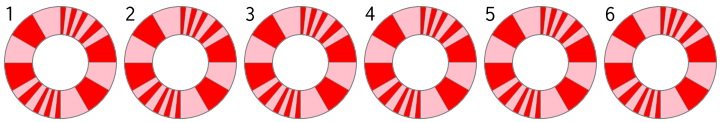
In the above, the `fast` function takes two patterns as input, and combines them to return a new pattern. Pattern transformations tend to operate relative to cycles, but that does not mean that successive cycles are identical. For example in the following, `every 3` is used to apply the function `fast "5 3"` to `"red pink"` as above, but only every third cycle. The first six cycles of the resulting pattern are shown below, so that you can see this change over time.

```
every 3 (fast "5 3") "red pink"
```



We can also squeeze the above six cycles into one, again by 'speeding it up', so that each cycle is identical once more:

```
fast 6 (every 3 (fast "5 3") "red pink")
```



Already we can see a strong part of Tidal's flexibility; it is highly *composable*. Functions like `fast 6` take a pattern as input, and return a new pattern as output, so that it is straightforward to compose multiple functions together into more complex transformations, as we have done above. Furthermore, we did not have to write any code to align `"5 3"` with `"red pink"`, Tidal is *declarative*, in the sense that it takes care of the mechanics of pattern composition for us.[^ TODO - internal link to kairotic coding]

Event time in polyphonic and Polymetric sequencing

We have already learned the standard unit of time in Tidal is the cycle. One impact of this is that if you add additional steps to a sequence, the steps will become shorter in duration, so that they are contained exactly within a single cycle:

```
"red pink orange"
```

```
"red pink orange blue lightblue"
```

In the musical domain, this means that the more events you add to a pattern, the faster they will be played, in order to fit them into a cycle:

```
sound "bd cp sd"
```

```
sound "bd cp sd mt lt"
```

In other words, timing in Tidal is not based around a notion of a fixed 'beat' duration, but on higher level cycles. This lends metrical flexibility, for example, it is possible to break down individual steps into subcycles, using square brackets:

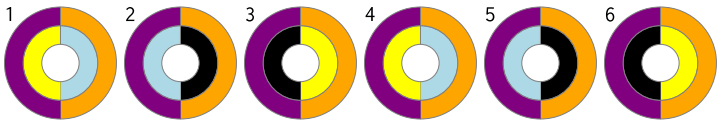
```
"red [orange green] blue [lightblue yellow black]"
```

It is also possible to place more than one subcycle inside a single step:

```
"[orange purple, lightblue yellow black]"
```

In the above we can see that `lightblue yellow black` fills the same span as `orange purple`. It is not significant that one subcycle is placed on the outside of the other; in musical terms this simply indicates that they form a polyphony, happening at the same time. We can change this behaviour by instead using curly brackets to denote the subsequences:

```
"{orange purple, lightblue yellow black}"
```



We can see that the *steps* rather than cycles now align, so that `orange purple` lines up with `lightblue yellow` in the first cycle. We can also see what it means to be a 'subcycle', as the subcycle continues where it left off, over successive cycles. From `lightblue yellow black`, only the first two colours are used in the first cycle to match with the two colours in `orange purple`, so on the second cycle it continues with `black`, cycling back to `lightblue` for its second value. We end up with a structure that repeats every third cycle.

Lets listen to the equivalent in the sound domain, listening to a two-step bass drum (`bd`) - clap (`cp`) sequence against a low (`lt`) - mid (`mt`) - high tom (`ht`) sequence:

```
sound "[bd cp, lt mt ht]"          sound "{bd cp, lt mt ht}"
```

Subcycles can be placed within subcycles. The following contains a subcycle with three steps (with a span of one third of a cycle each), of which the middle step is broken down further into two substeps (one sixth of a cycle each):

```
"red [orange [black green] brown] blue"
```

Rests, gaps and stretches

Silence is of course central to music, and in Tidal sequences you can insert empty gaps with the ~ character.

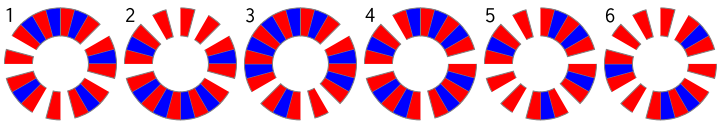
```
"red blue ~ orange [purple ~] green"
```

Alternatively a _ character will stretch the previous step:

```
"red blue _ orange [purple ~] green"
```

You can also use a ? character to replace a step with silence around 50% of the time, varying from one cycle to the next:

```
fast 12 "red blue?"
```



Manipulating time

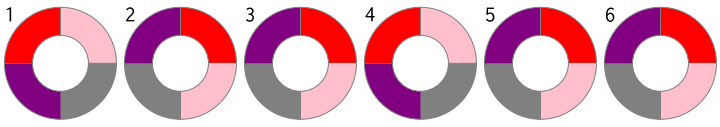
Music is of course a time-based artform, and in Tidal, time is malleable – it both flows in cycles, and develops over time. It can be reversed, shifted forward into the future or back into the past, expanded and contracted, chopped up and rearranged, and subdivided to practically any depth.

We have so far focussed on sequences, but there is a much more to pattern. Let us move on to explore different kinds of patterning, all of which take one or more existing patterns and transform them. The nature of the pattern transformation might be perceivable by the listener, or perhaps only give them a sense of order amongst chaos, but because there is a clear structure in the creation of pattern, the sonic environment that results has the possibility to be an engaging place, explored through the process of listening. Of course, everyone listens differently, and so a pattern is not necessarily a puzzle to be solved, but an environment to be explored.

Many computer music systems represent music as *lists* of events, an approach which certainly has its advantages. However, Tidal instead represents music as a pure, mathematical *function*, with a timespan as input, and returning events active within that timespan as output. Each event has its own timespan, which will intersect with the input. This approach, a form of Functional Reactive Programming (Hudak 2000), allows the temporal structure of Tidal patterns to be efficiently manipulated without being calculated, either as a discrete or continuous signal, and separately from the events which are represented within the signal (McLean 2014). Time itself is represented as a rational number, lending itself to precise subdivision. It is not important to understand Tidal's inner representation of time in detail, but worth noting that much of the flexibility seen in the following stems from Tidal's focus on composing together functions of time, rather than linear procedures over data.

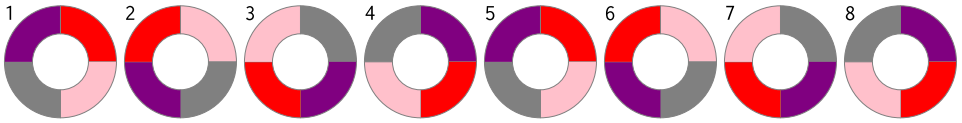
We have already seen the `fast` function for 'speeding up' a pattern. The `<~` and `~>` operators manipulate time a different way, by moving patterns backwards and forwards in time. With Tidal's cyclic notion of time, in practice this results in *rotating* a pattern. The following pattern shows a quarter rotation, every third cycle:

```
every 3 (0.25 <~) $ "red pink grey purple"
```



The `iter` function also shifts time, but keeps shifting it from one cycle to the next, until the cycle returns back to where it started. This takes place over a given number of cycles, which in the following is four:

```
iter 4 "red pink grey purple"
```



Lets hear some sound-based examples of `<~` and `iter`:

```
iter 4 $ sound "lt mt ht cp"
```

```
every 3 (<~ 0.25) $ sound "lt mt ht cp"
```

For a complete introduction to the the pattern transformations available in Tidal, please refer to the website http://tidalcycles.org.

Patterning multiple dimensions of sound

In terms of how it is perceived, sound is multi-dimensional. A modern synthesiser may have a keyboard providing a pitch dimension, but will also have a plethora of knobs and sliders for exploring further timbral dimensions. Accordingly, Tidal allows different aspects of sound to be patterned independently. The following example demonstrates independent patterning of the `legato` (relative duration) and `lpf` parameters, where the structure being defined by the `n` (note) rather than `sound` parameter.

```
off 0.125 (|+| n "-12") $
  jux rev $ n (off "<1%8 1%4>" (+ chord "<major minor major>")
               (palindrome $ "<c(3,8) f(3,8) g(3,8) c6(3,8)>"))
  # sound "superpiano"
  # legato (scale 0.5 2 saw)
  # lpf "<300 [1000 700] 2000>"
  # lpq 0.2
```

Bel, Bernard. 2001. "Rationalizing Musical Time: Syntactic and Symbolic-Numeric Approaches." In *The Ratio Book*, edited by Clarence Barlow, 86–101. Feedback Studio.

Blackwell, Alan F. 2006. "Psychological Issues in End-User Programming." In *End User Development*, edited by Henry Lieberman, Fabio Paternò, and Volker Wulf, 9:9–30. Human-Computer Interaction Series. Dordrecht: Springer Netherlands. http://dx.doi.org/10.1007/1-4020-5386-x_2.

Hudak, Paul. 2000. *The Haskell School of Expression : Learning Functional Programming Through Multimedia*. First. Cambridge University Press. http://www.worldcat.org/isbn/0521644089.

Huron, David. 2008. *Sweet Anticipation: Music and the Psychology of Expectation*. Cambridge, Mass.; London: MIT Press.

McLean, Alex. 2014. "Making Programming Languages to Dance to: Live Coding with Tidal." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*. https://doi.org/10.1145/2633638.2633647.

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