Learning TidalCycles

Patterns I Have Known and Loved

BY

ALEX M^c^LEAN

SLAB LABORATORY
SHEFFIELD
ROYAUME UNI
MMXIX

Chapter 1

Introduction

This is the beginning of a work-in-progress book, *Learning TidalCycles*. Feedback is very welcome!

This is something I've wanted to write for a long time - a complete guide to TidalCycles. I can't promise that I'll ever finish it, but you can make that much more likely by sending coffee! You can do that here: https://ko-fi.com/yaxulive/

1.1 What is TidalCycles?

TidalCycles, or just *Tidal* for short, is a language environment for exploring pattern. It's known as a live coding environment (see below), and often used in dance music contexts, but really can be applied to anything pattern-related, whether repetitive dance music, abstract minimalist music, weird electroacoustic music, or outside the world of music completely - people have used Tidal to make video art, choreograph dance, and even pattern textiles.

Pattern is central to Tidal, so lets have a go at defining it. In music, any sequence is often called a 'pattern', if it gets *repeated*. But there's a lot more to explore pattern than repetition. Think about patterns in textiles - you get a lot of repeating patterns, but also *reflecting* patterns creating symmetry, *fractal* patterns playing with similarity at different scales, *interference* patterns between different elements. And then you get *glitched* patterns, where expectations set up by a pattern are confounded through 'errors', often introduced on purpose. All these kinds of patternings can be explored with Tidal, and more.

1.2 Live coding and embracing error

Live coding, as a community and movement, has been around since around the year 2000, giving itself the name TOPLAP at the first international meeting in Hamburg in 2004. As a practice, live coding arose from a few different places at once, perhaps in reaction to the increasingly slick 'seamlessness' of music software, as well as a ressurgence of 'creative coding' as a means to make art. "Instead of using prewritten software in the studio, why not write software from scratch on stage?" A pretty strange idea, but it worked out pretty well, live coding has turned into a world-wide movement, with a wide range of live coding environments and technologies for the live performing arts.[1]

Tidal was originally made for live coding, but that doesn't have to involve being in front of a live audience. Many prefer to write code alone in the studio, and only share it if/when they feel it's ready. They might then perform with it by manipulating and tweaking the code, rather than writing it from scratch. That is of course completely fine, it's totally up to you how you fit Tidal into your workflow and life.

Still, Tidal's focus on patternings does encourage a certain kind of approach. If you come to Tidal with fixed ideas, you might get frustrated. When you're working with pattern, you might have three or four layers of manipulations going on, resulting in strange interactions going on that can make it difficult to predict exactly what will happen when you make a change on a particular layer. However if you're happy to embrace a little uncertainty, you'll get a tacit understanding for Tidal patterns, and get a feel for high level control of it's expansive possibilities.

1.3 Installation

To install TidalCycles, you will need a laptop or desktop computer, running Linux, MacOS X or Windows. Your computer doesn't need to be particularly powerful, but you might well need full admin rights to it. All components of the TidalCycles system are free/open source. For the latest installation information, please refer to the instructions on https://tidalcycles.org/

1.4 Architecture of a Tidal environment

While your installation script is running, lets pause to reflect on the different parts of a full Tidal environment installed - the Tidal library, the Haskell language, the editor, SuperCollider, SuperDirt, and how all they fit together. This will later help with imagining what is going on behind the scenes when you're typing in your Tidal patterns.

Figure xxx shows a schematic with all the different bits, and how they fit together and communicate. *Haskell* is a general purpose programming language, which *Tidal* is written in. The program that runs Haskell is called *ghci*.

When you're writing Tidal patterns, you're writing Haskell code, using the Tidal library. Tidal is a bit more than an add-on for Haskell though, it provides its own operators and a computational model for dealing with Patterns, so is really a language in its own right. In computer science terms, it's a domain specific language, embedded

[1] Check the long list of technologies in the TOPLAP "All things Live Coding" page here: https://github.com/toplap/awesome-livecoding

in Haskell. In turn, the mini-notation for describing sequences is embedded in Tidal. Tidal does all the pattern generation itself - it turns the code you write into messages that are sent to a sound synthesiser, most often *SuperDirt*.

Just as Tidal is written in Haskell, SuperDirt is written in SuperCollider. SuperCollider is a programming environment for audio synthesis and digital signal processing (DSP) in general. SuperCollider is amazing - you'll find SuperCollider under the hood of a lot of audio live coding environments. In fact, many people use it as a great live coding system in its own right. If you like, and are a superhero, you can live code synthesisers and effects in SuperCollider while live coding patterns to trigger them in Tidal.

With Tidal making patterns, and SuperDirt making sound, the only thing left is a text editor to work in. There are a few editors that have plugins for talking with Tidal - atom, vscode, emacs or vim. Whichever you choose, the plugin will take care of starting ghci for you, loading the Tidal library, and setting up the connections with SuperDirt.

1.5 Starting up a Tidal environment

Once everything is installed, it's time to start things up.

Normally, you'd start by starting SuperDirt inside SuperCollider, and then starting tidal inside your text editor.

1.5.1 Starting SuperDirt

Here's a good way to configure SuperCollider to start SuperDirt:

1. Start the SuperCollider application (the system that SuperDirt runs in)
2. Open the 'File' menu then click on 'Open startup file'
3. In that file, paste in `SuperDirt.start`, and then save (File -> Save in the menus)

You've now configured SuperDirt to start whenever you open SuperCollider. So, if you close the supercollider application and start it again, SuperDirt should automatically open for you.

SuperDirt has a lot of configuration options, that you can add to the same startup file. We'll cover those in detail in chapter xxx.

1.5.2 Starting Tidal, running your first pattern.

Starting tidal should just be a case of typing some code into a tidal file, and running it. A default Tidal installation will be configured to use the *atom* editor, but it's much the same deal whatever editor you're using.

1. Start atom
2. Open (or create and save) a file with the `.tidal` extension (e.g. `mylovelycode.tidal`).
3. Type or paste in some code (e.g. `d1 $ sound "bd sn"`)

4. Running the code, by making sure the cursor is on it, and pressing *shift-enter* or *control-enter*

shift-enter runs a single line of code, and *ctrl-* (or on a mac, *cmd-*) *enter* will run a pattern that runs over multiple lines.

When you want to stop the sound, you can replace the pattern with silence by running this: `d1 $ silence`, or just `hush` by itself.

Note 1

If you're running multiple lines of code (with 'ctrl-enter'), you can still only run one pattern at a time. Make sure there's a blank line above and below the pattern you want to run.

1.6 Structure of a Tidal pattern

Now you know how to start and stop a pattern, lets jump ahead and look at a more complicated example. The aim here isn't to understand everything, but to start to get an idea about what a pattern looks like, and what tidal is capable of. Here we go:

```
d1 $ chunk 4 (hurry 2) $ sound "bd [~ rs] mt [lt ht]" # crush 5
```

Running the above, you should start hearing a shifting drum pattern. Again, to stop it, run this:

```
d1 $ silence
```

or this:

```
hush
```

What just happened? There's already quite a lot to take in here, but lets have a look at the different bits, working from right to left.

```
crush 5
```

This is a *control pattern*. Here it sets SuperDirt's *bit crusher* audio effect on, using the constant value 5. This adds some fairly subtle distortion to the sound output (try lower values for more distortion). You can pattern these effects too, we'll come to that in chapter xxx.

Reading back some more, we find another control pattern, this time setting the *sound* that's played.

```
sound "bd [~ rs] mt [lt ht]"
```

This time the value is in speech marks, which means that it's specified using Tidal's flexible *mini-notation* for sequences. The words inside - `bd`, `rs`, `mt` etc, are all names of sample banks (`bd` is short for bass drum, `rs` for rimshot, and `lt`, `mt` and `ht` for low, mid and high toms). We'll start looking at mini-notation syntax including `[]` and `~` in the next chapter, but for now lets just say that it's all about rhythm.

Lets think about what the `sound` function actually does. It takes `"bd [~ rs] mt [lt ht]"`, which is a *pattern of words*, and turns it into a *pattern of sounds*. That is, it takes one kind of pattern as input, and returns another kind of pattern as output. In Tidal, everything either tends to be a pattern, or a function for working on patterns. **It's patterns all the way down.**

You might have noticed that between the `sound` and `crush` control patterns, there's a `#` character:

```
sound "bd [~ rs] mt [lt ht]" # crush 5
```

The job of `#` is to join the two control patterns together. Super simple to use, but underneath there are some complexities about how patterns are combined. We'll look into those in chapter xxx.

Reading further back, we see this construction:

```
chunk 4 (hurry 2)
```

This is a funky bit of code, which adds a lot of rhythmic variety by shifting along, progressively 'speeding up' a quarter of a pattern per cycle. Again, we'll look at these patterning functions in detail later, but for now think of this as a machine that takes a pattern as input, and returns a mangled version of that pattern as output.

```
d1 $
```

Reading right back to the start, we get to `d1`. `d1` is another function, which takes a pattern of sound controls as input, and sends it to the synthesiser to be turned into the actual sounds you can hear. The `$` operator is there to divide up the line; whatever is on the right of the `$` is passed to the function on the left. Looking at the whole pattern again, you can see there's actually two `$`s in it. One makes sure the `sound` and `crush` controls are combined before being mangled by the `chunk` function, and the other makes sure everything gets worked out before finally being passed to `d1`.

```
d1 $ chunk 4 (hurry 2) $ sound "bd [~ rs] mt [lt ht]" # crush 5
```

That completes our tour of this particular pattern. It'll take a while to really get your head around all of this, but don't worry, we'll cover it all again properly later. Next, we go back to basics to have a proper look at that mini-notation.

Chapter 2

Mini notation

We've already seen that Tidal can be broken down into two parts: a library of functions for transforming pattern, and our focus in this chapter – a mini-notation for quickly describing sequences. Built on Tidal's flexible approach to musical time, the mini-notation is a quick way to express rhythms, whether you're making canonical techno or far-out polyrhythmic minimalism.

2.1 Sequences and sub-sequences

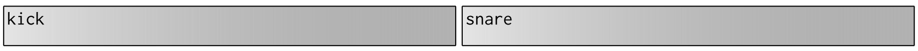
Whenever you see something in speech marks (`""`), that'll almost always be a mini-notation sequence. Here's a simple example:

```
d1 $ sound "kick snare"
```

The above plays kick after snare after kick, one after the other, forever. The `"kick snare"` represents the repeating sequence, the `sound` specifies that it's a pattern of sounds, and the `d1 $` sends the pattern to be turned into sound.

Most examples in this chapter will be visual, rather than musical, so you can have a good look at the pattern next to its code. Some examples will visualise patterns of words from left to right, like this:

```
"kick snare"
```



Others will show patterns of words as colours:

```
"orange purple green"
```

Sometimes, I'll show colour patterns as a circle, clockwise from the top:

```
"orange purple green"
```

Patterns will most often be visualised as words, as they're unambiguous, and accessible to colourblind people. I will use colour patternings from time to time though, and give at least one solid musical example for each concept.

2.1.1 Cycle-centric time

This will depend on your cultural background, but in electronic music and Western classical music musical time is generally based on the musical *beat*. Whether you're using a software sequencer or writing sheet music, you'll generally express things relative to a tempo (musical speed) measured in *beats per minute*. In Tidal, things tend to be measured in *cycles*, not beats. In musical terms, a cycle is equivalent to a *measure* or *bar*. What does this mean in practice?

First of all, you'll notice that the more events you add to a pattern, the faster it goes. Compare these two:

```
d1 $ sound "kick snare clap clap"
```

```
d1 $ sound "kick snare clap clap bd bd"
```

The latter goes 1.5 times faster than the former, to fit all the events into a single cycle.

This doesn't mean that things *have* to fit inside a cycle, or that one cycle has to be the same as the next. That isn't true at all. What this means is that in Tidal, the cycle is the reference point for patterning, and not the event.

In other software, you might define a number of beats per bar, and set the tempo in beats per minute (BPM). In Tidal though, you set cycles (bars) per second, and temporal structure within a cycle is fluid - beats can fall all over the place, with structure coming from complex and compound ratios rather than a strict metrical grid.

2.1.2 Rests (gaps) with ~

The 'tilde' token ~ leaves a step empty, creating a musical rest (gap):

```
sound "a b ~ c"
```

WontCompile [GhcError {errMsg = "<interactive>:3:28: error:\n \8226 Couldn't match type \8216Map String Value\8217 with \8216[Char]\8217\n Expected type: Pattern [Char]\n Actual type: ControlPattern\n \8226 In the expression: e_1\n In the expression:\n (let e_1 = sound \"a b ~ c\" in e_1) :: Pattern [Char]\n In an equation for \8216_compileParsedExpr\8217:\n _compileParsedExpr\n = (let e_1 = sound \"a b ~ c\" in e_1) :: Pattern [Char]"}]

The above pattern still has four 'steps' of equal length, but the third step is left empty. Here's an audio equivalent:

```
sound "kick snare ~ clap"
```

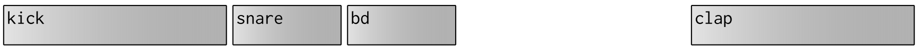
2.1.3 Subsequences with []

Events don't have to be of equal length, though. The following still has four steps, but the third step contains a *subsequence*, denoted with square brackets:

```
d1 $ sound "kick [snare bd] ~ clap"
```

So now `kick` and `clap` each take up a quarter of a cycle, and `snare` and `bd` each take up an eigth of a cycle. If we draw out the cycle from left to right, the structure looks like this:

```
"kick [snare bd] ~ clap"
```



The following illustrates what this structure looks like as a colour cycle, clockwise from the top:

```
"darkblue [lightblue grey] ~ black"
```

The subsequences can be subdivided however you like. The following has two steps of half a cycle each, the first one having a subsequence of three steps, and the second of four steps:

```
"[darkblue blue lightblue] [purple red orange yellow]"
```

You can also have subsequences inside subsequences, to any level of depth.

```
"[red [blue green] orange] [[red [pink grey] yellow] purple]"
```

2.1.4 Speeding up and slowing down

If you want a step within a sequence to play faster, you can use `*` followed by a speed factor. For example:

```
d1 $ sound "bd rs*3 mt lt"
```

The above is still a four step sequence, but the third one is played three times as fast, so that the rimshot sound is heard three times in the space of one. The following sounds exactly the same:

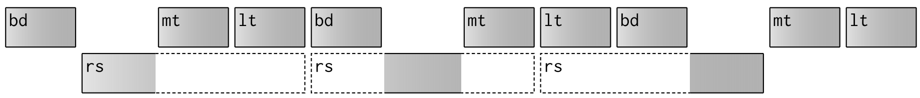
```
d1 $ sound "bd [rs rs rs] mt lt"
```

Similarly, the symbol for divide, /, slows a step down:

```
d1 $ sound "bd rs/3 mt lt"
```

As a result, you now only hear the rimshot every third step. Lets have a look at a diagram of this pattern, but sped up by a factor of three with []*3, so that we see three cycles' worth of the pattern as a subsequence:

```
"[bd rs/3 mt lt]*3"
```



You can see that we get a different third of the `rs` event each time around; the shaded part of each event is the 'active' part. We only hear a sound when the first third of it plays, because a sound is only triggered at the start of an event.

These modifiers can be applied to a subsequence too. If you slow down a subsequence with three elements in it, you will hear one of them per cycle:

```
d1 $ sound "bd [rs cp ht]/3 mt lt"
```

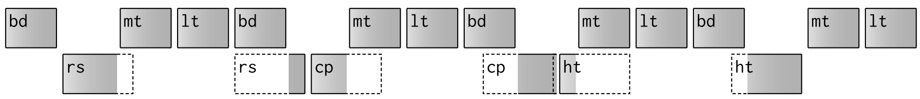
In other words, you hear one third of the subsequence each time, and the next time around, it carries on where it left off.

Spreading three events over three cycles is straightforward, but what if the numbers aren't so easily divisible? The answer is, things start sounding funky. Here's an example with those three events spread over four cycles:

```
d1 $ sound "bd [rs cp ht]/4 mt lt"
```

Lets have a look at that:

```
"[bd [rs cp ht]/4 mt lt]*4"
```



You can see that Tidal does a good job of splitting the sequence in four, so that you end up with fragments of events. Remember that a sound is only triggered by the *start* of an event, so the first time we hear a rimshot at the start of the second step in the subsequence, the second time a clap one third of the way into the step, the third time a high tom two thirds into the step, and the fourth time we don't hear anything during that step - we get the tail end of the high tom, which doesn't trigger anything.

2.1.5 Polyphony

In music, *polyphony* simply means that two or more notes or other events can sound at once. There are a lot of ways to layer things up in Tidal, but in the mini-notation there is really just one way - separating sequences with commas. There are a few variants for how the events in the different subsequences get matched up, though.

If we stick with the square brackets used above, then the sequences get layered, so that their cycles match up perfectly.

So if we have a simple pattern of tom patterns ...

```
d1 $ sound "lt ht mt"
```

... and a pattern of rimshots ...

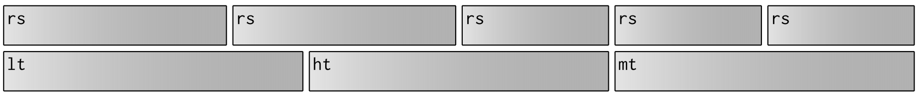
```
d1 $ sound "[rs rs] [rs rs rs]"
```

... we can play them at the same time by putting a comma between them, and wrapping the lot in square brackets:

```
d1 $ sound "[lt ht mt, [rs rs] [rs rs rs]]"
```

Here's how that looks in diagram form:

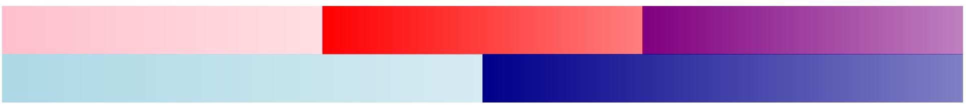
```
"[lt ht mt, [rs rs] [rs rs rs]]"
```



2.1.6 Layering [] polyrhythm vs [] polymetre

So far we have seen and heard that when there are multiple subsequences inside square brackets, they are layered on top of each other, with cycles aligned. Lets see a colour example of that:

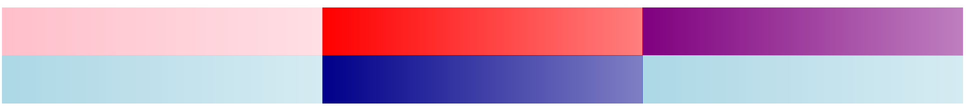
```
"[pink red purple, lightblue darkblue]"
```



When you have two rhythms on top of each other, such as three against two above, it's known as a *polyrhythm*.

If we replace the square brackets with curly brackets {}, then instead the *steps* align:

```
"{pink red purple, lightblue darkblue}"
```



The first subsequence has remained the same, with the two steps in the second subsequences lining up with the three steps in the first. Because there aren't enough steps in the second sequence, it loops round. It is clearer what is going on if we speed up the whole thing by a factor of three:

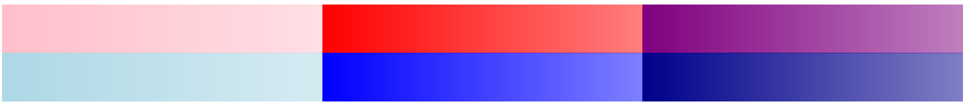
```
"[{pink red purple, lightblue darkblue}]*3"
```

This kind of construction, where you layer up sequences with the same step duration but with differing number of steps, is known as a *polymetre*.

So to recap, square brackets allow you to create *polyrhythms* where subsequences repeat at the same rate but have different rhythmic structures, and curly brackets allow *polymetre*, with the same rhythmic structure but different periods of repetition.

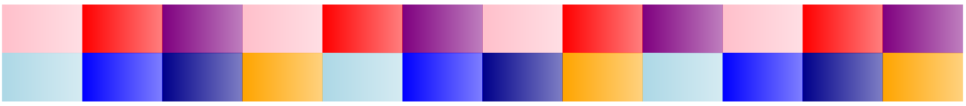
There's one more thing to note about polymetre. We have seen that with `{}`, steps align, and that the number of steps per cycle is given by the first subsequence. for example the following will take three steps per cycle, from both subsequences:

```
"{pink red purple, lightblue blue darkblue orange}"
```



However you can manually set the number of steps per cycle, by adding `%` and a number after the closing curly bracket. For example to take twelve steps per cycle:

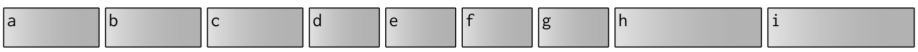
```
"{pink red purple, lightblue blue darkblue orange}%12"
```



2.2 Rhythmic 'feet' with .

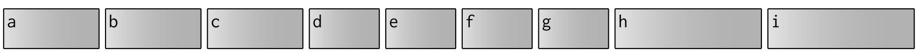
The `.` (full stop/period) character provides an alternative to grouping with `[]`. For example this ...

```
"[a b c] [d e f g] [h i]"
```



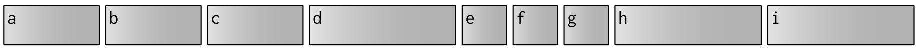
... does the same as this ...

```
"a b c . d e f g . h i"
```



Whereas `[]` is placed around each part, `.` is placed between successive parts. This is sometimes nice to use, but you can't 'nest' subpatterns inside patterns with `.` alone. You can mix and match them, though:

```
"a b c . d [e f g] . h i"
```



2.3 One step per cycle with <>

There is one more pair of symbols for denoting subsequences: `<>`, known as angle brackets. These simply slow a subsequence down to one step per cycle.

```
d1 $ sound "<lt ht mt>"
```

It's the same as slowing down a subsequence by the number of steps in it, for example the following does the same as the above.

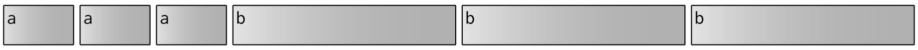
```
d1 $ sound "[lt ht mt]/3"
```

2.4 Repeating steps with !

We've seen that `*` speeds up time *inside* a step, effectively causing a step to repeat itself, but squashed in the same space. `!` instead *duplicates* steps.

You can see the difference here:

```
"a*3 b!3"
```



`"a*3"` repeats `a` within the step, and `"b!3"` repeats `b` as additional steps.

If you write a `!` without a number, it'll simply repeat the previous step. So, these are the same:

```
"[a b]!2 c!3"
```

```
"[a b] ! c ! !"
```

```
"[a b] [a b] c c c"
```

2.5 Elongating steps with @

The `@` symbol is similar to `!`, but instead of repeating a step, is stretches it out over the given number of steps.

```
"a b@2"
```

This gets particularly interesting when applied to subpatterns:

```
"[a b c]@2 [d e]@3"
```

2.6 Random choices with ? and |

Random choice is a quick way to introduce variety into a sequence. We'll be looking at taming randomness in Tidal as a whole in detail in chapter xxx, but lets have a quick look at making random within choices the mini-notation now.

A way to make a step optional is using the question mark. By default, there'll be a 50% chance of an event playing or not. In the following, the second and fourth steps will be silent roughly half the time:

```
d1 $ sound "bd sd? bd cp?"
```

If a step contains a subsequence, then the randomness will be applied individually to the steps within:

```
d1 $ sound "bd [mt ht lt ht]?"
```

This is true of 'sped up' events, the eight repetitions of `bd` in the second step here will be played at random:

```
d1 $ sound "cp bd*8?"
```

You can make an event more, or less likely to play by adding a decimal number between 0 (never play) and 1 (always play). For example, the `bd` in the following will play 80% of the time:

```
d1 $ sound "bd?0.8"
```

The `|` character works in a similar way to `,` in that is separates subsequences, but instead of layering them up, it picks one of them to play at random, each cycle.

```
d1 $ sound "bd [mt|ht lt ht]"
```

Sometimes the above will play the equivalent of `bd mt`, and others it will play `bd [ht lt ht]`.