Learning TidalCycles

Patterns I Have Known and Loved

BY

ALEX McLEAN

SLAB LABORATORY
SHEFFIELD
ROYAUME UNI
MMXIX

Chapter 1

Introduction

This is the beginning of a work-in-progress book, *Learning TidalCycles*. It is currently being developed alongside the *Tidal Club* online course, also with financial help from ko-fi supporters.

Please note that this is a work in progress, and you might find it a difficult journey, with beginner material mixed with advanced topics. Please skip bits that don't yet make sense - sections will be re-ordered as things develop..

1.1 What is TidalCycles?

TidalCycles, or just *Tidal* for short, is a language environment for exploring algorithmic pattern. It is 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 with algorithms.

The concept of *algorithmic pattern* is central to Tidal, so let's have a go at defining what it means. In music, any sequence is often called a pattern, if it gets *repeated*. But there is a lot more to pattern than repetition. Think about patterns in textiles - you get a lot of repeating patterns, but also patterns of *reflection* or *rotation* creating symmetry, *fractal* patterns playing with similarity at different scales, *interference* patterns between different elements. You also get *glitches* in patterns, where expectations set up by other kinds of pattern are confounded through 'errors', often introduced on purpose. All these kinds of patternings can be explored with Tidal, and more. With "algorithmic pattern" I focus on the way a pattern is *perceived*, in particular how the movement and processes transformation can be perceived in the end results.

In summary, an algorithmic pattern is where you see, hear or otherwise perceive ways of making.

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, alongside a resurgence of 'creative coding' as a means to make art. We asked ourselves, 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 worldwide 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 does not 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 is ready. They might then perform with it by manipulating and tweaking the code, rather than writing it from scratch, or just use live coding tools as an expressive way to produce recorded music. All of this is of course completely fine, it is totally up to you how you fit Tidal into your creative workflow and life in general.

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 are working with pattern, you might have three or four layers of manipulations going on, resulting in strange interactions that can make it difficult to predict exactly what will happen when you make a change on a particular layer. However if you are happy to embrace a little uncertainty, you will get a tacit understanding for Tidal patterns, and get a feel for high level control of its 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/Installation.

1.4 Architecture of a Tidal environment

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

The diagram below shows 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*.

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

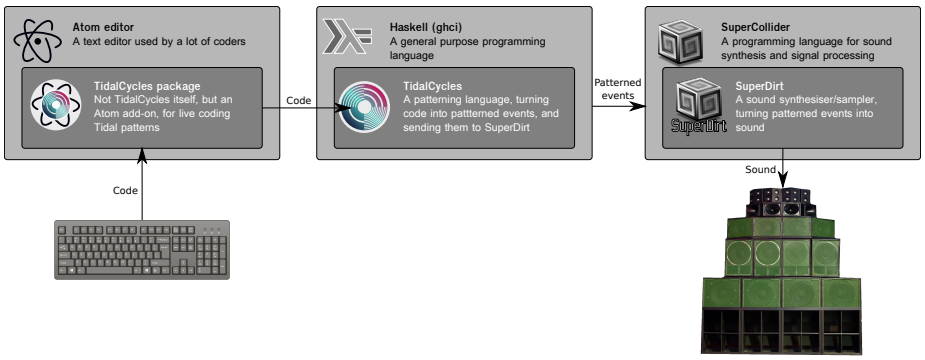


Figure 1.1: Diagram of a Tidal environment

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 in Haskell. In turn, there is a "mini-notation" for describing sequences 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 Haskell 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 put in 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 your editor, 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. `mylovelypatterns.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.

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.

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

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, 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 patterns together, in this case the `sound` and the `crush` patterns. Super simple to use, but underneath there are some complexities about how values inside the patterns are matched up and 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 controls as input (in this case `sound` and `crush` control patterns combined), 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 calculated before being passed to the function on the left. Again, we'll get more familiar with the usefulness of `$` later on. Looking at the whole pattern again, you can see there's actually two `$`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 the mini-notation.

Chapter 2

Mini notation

We've already seen that Tidal can be broken down into two parts: a mini-notation for quickly describing sequences, and a library of functions for transforming pattern. In this chapter, we focus on the mini-notation. 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, far-out polyrhythmic minimalism, or a musical genre entirely of your own invention.

2.1 Sequences and sub-sequences

The mini-notation is all about *sequencing*, describing how one event follows another, in repeating, looping structures. Whenever you see something in speech marks (""), that will 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 mini-notation sequence, the `sound` specifies that it's a pattern of sounds, and the `d1 $` sends the pattern to be turned into sound.

Note 2

A note for experienced programmers - in Tidal, mini-notation sequences are immediately parsed into patterns, so although they *look* like strings, you can't treat them as such.

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 most music software, musical time is based on the *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 mini-notation sequence, the faster it is played. 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.

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 the 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.

So in Tidal, the *cycle* is the reference point for patterning, and not the event. That doesn't mean that things *have* to fit inside a cycle, or that one cycle has to be the same as the next.

You can change the current tempo with the `setcps` function, for example, to play at a rate of 0.45 cycles per second:

```
setcps 0.45
```

If you had four events in a cycle, that would feel like (0.45 x 4 =) 1.8 beats per second, or (0.45 x 4 x 60 =) 108 beats per minute.

2.1.2 Rests (gaps) with ~

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

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

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, though. The following still has four steps, but the second 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 second 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"
```

From the following visual representation, we can see the cycle divided into four steps, with the second step 'sped up':

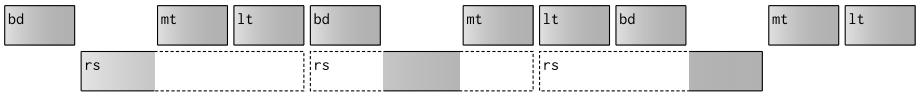
```
"bd rs*3 mt lt"
```

Just as * speeds up a step, 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.

Note 3

When events get cut into parts like this, the *whole* sound is triggered when (and only when) the *first* part of the event plays. This is a little counter-intuitive, but will start to make more sense when we look at combining patterns together in chapter xxx. We'll also look at fun ways of properly chopping up sounds into bits in chapter xxx.

These modifiers can be applied to a subsequence too. If you slow down a subsequence with three elements in it, by a factor of three, 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. Lets have a look at three cycles worth of that (this time making use of a function, `fast`):

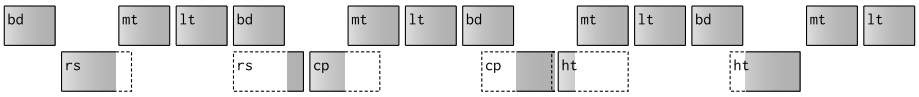
```
fast 3 "bd [rs cp ht]/3 mt lt"
```

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 four cycles worth of that:

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



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 around 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 only 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 sounds can happen at the same time. 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 different ways to match up events in the different subsequences, 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]]"
```

You can see that the two subsequences are squashed to fit the cycle.

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 start with a simple visual example:

```
"[a b c, d e]"
```

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:

```
"{a b c, d e}"
```

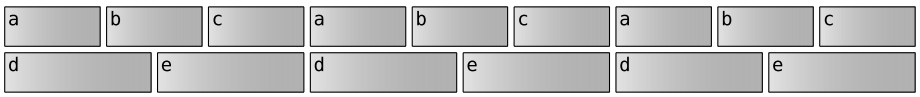
The first subsequence has remained the same, but the steps in the second subsequences now line up with the steps in the first. Because there aren't enough steps in the second sequence, it loops round. It is clearer what is going on if we speed up the whole thing by a factor of three, in order to see three cycles of the mini-notation sequence:

```
fast 3 "{a b c, d e}"
```

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

Here's what happens if we change that pattern from curly to square brackets:

```
fast 3 "[a b c, d e]"
```



So to recap, square brackets allow you to create *polyrhythms* where subsequences repeat at the same rate, but can have different rhythmic structures. Curly brackets allow *polymetre*, where different parts have 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 all subsequenc, because there are three steps in the first subsequence:

```
"{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 from the subsequences:

```
"{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 subsequence, . is placed *between* successive sequences. This is sometimes nice to use, as a way of 'marking out' the rhythmic pulse in a cycle. However, you can't 'nest' subpatterns inside subpatterns with . alone. You can mix and match . and [].haskell, 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: <>, also known as angle brackets. These simply slow a subsequence down to one step per cycle.

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

The angle brackets slow 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"
```

Here's six cycles of a mininotation pattern, where the third step cycles through a subpattern, returning one event each time around:

```
fast 6 $ "a b <c d e> f"
```

Again, you can mix-and-match this with other forms of subpatterns:

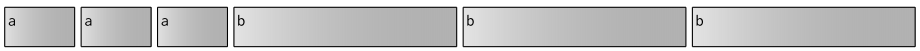
```
fast 6 $ "a b <c [d e] [f g h]> i"
```

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 three examples all produce exactly the same result:

```
"[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"
```

In the above, the first subsequence is stretched to take up the space of two steps, and the second the space of three steps. That makes five in total, so the two subsequences take up two fifths and three fifths of a cycle respectively.

2.6 Random choices with ? and |

Randomness provides a quick way to introduce variety into a sequence. We'll cover randomness in detail in chapter xxx, but lets have a quick look at making random choices within the mini-notation, right now.

A way to randomly skip playing a step is by using the question mark (`?`). By default, there will 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]?"
```

It also works with 'sped up' events, for example the eight repetitions of `bd` in the second step here will be silenced at random:

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

Lets see what randomness *looks* like:

```
"orange*24? [[black blue grey]?]*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 `orange` segments in the following will be removed at random, around 90% of the time:

```
"orange*100?0.9"
```

The `|` character is used in a similar way to the comma (`,`) in that it separates subsequences. However, 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]`. Here's a visual example:

```
"[white blue|yellow orange red]*16"
```

Chapter 3

Effecting sound with control patterns

So far, we've seen a lot of sound patterns. We learned that the word `sound` is the name of a function that turns a pattern of *words* (like `"bd sd"`) into a pattern of *controls* (like `sound "bd sd"`). A *sound* control is one that defines what kind of sound to play, in particular which set of samples or synthesiser notes to choose from. There are many more functions allowing you to pattern other aspects of sound, such as loudness, pitch, distortion, panning, filtering. This chapter will introduce them, and how to combine them together.

3.1 Combining control patterns

Lets start by looking at `crush` control patterns. Here's an example:

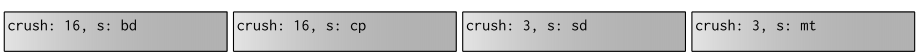
```
d1 $ crush "16 3"
```

The `crush` function creates a control pattern, just like `sound`, however its input is a pattern of *numbers* rather than words. Also, if you run this code, it doesn't actually make any sound! `crush` is for applying a 'bitcrushing' distortion effect, but you won't hear anything until you also give a sound to be crushed:

```
d1 $ sound "bd cp sd mt" |+ crush "16 3"
```

Now we hear something! Here the `sound` and `crush` controls have been combined together with the `|+` operator. Lets have a look at what we end up with:

```
d1 $ sound "bd cp sd mt" |+ crush "16 3"
```

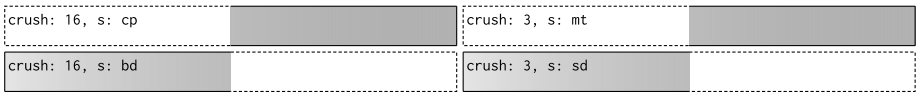


There are four events in the result, even though we only gave two values to the `crush` control. This is because when combine patterns with the `|+` operator, Tidal will start with events on the left hand side, and match them up with values on the right hand

side. Note that the first two events have a `crush` value of 16, and the second two have a value of 3, in line with the values that are active .

There's also a `+|` operator, which instead starts with events from the right-hand side, and matches them up with values on the left. Lets change our previous example to use that:

```
d1 $ sound "bd cp sd mt" +| crush "16 3"
```



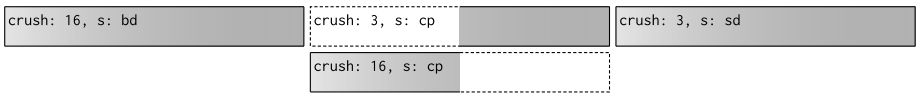
This will take a bit more explaining! Comparing both examples above, the shaded parts are the same, but the second example has 'remembered' the the original events on the right. Whereas the `bd` in the first example was a simple event taking up the first quarter of the cycle, in the second example, it's a *fragment* of an event. It still takes up a quarter of a cycle, but it remembers that it is originally from an event that took up half a cycle.

In the second example above, the `crush` value of `16` has been split in half, the first half matching with `bd`, and the second matching with `cp`. Again, the shaded part shows the half you are left with, in both cases. The `3` event has also been cut in half, between the `sd` and `mt`.

When it comes to listening to the above pattern though, you only hear the first (`bd`) and third (`sd`) sounds, and *not* the second (`cp`) and fourth (`mt`). This is because sounds are *only triggered at the start of events*. The first half of both `cp` and `mt` have been cut off, and so they are never triggered.

This combining of events works even when they don't line up. Here's a pattern where three events are combined with two:

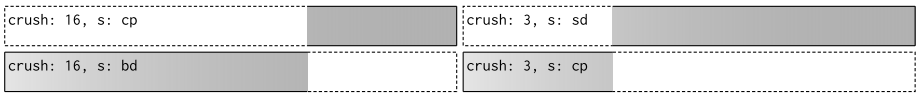
```
d1 $ sound "bd cp sd" |+ crush "16 3"
```



With the `|+` operator, we keep the structure of the three events from the left hand side. The middle one gets split in half, between the two events on the right hand side. In the end, three events are triggered; the `cp` has lost its start, so doesn't play.

The below shows that if we instead use the `+|` operator, the event structure comes the pattern on the right, and is split between events on the left. This time there are only two events with their 'starts' intact (`bd` and `cp`, and that therefore result in a sound being triggered.

```
d1 $ sound "bd cp sd" +| crush "16 3"
```



There is one more operator in this family, `|+|`. Whereas `|+` starts with events on the left, and `+|` starts with events on the right, `|+|` results in events which 'forget' where

they came from. Here it is in practice:

```
d1 $ sound "bd cp sd" |+| crush "16 3"
```

In this final example, all four are 'whole' events, and so you hear all four.

3.2 Summary

You might be wondering why events are kept at all, if they have lost their trigger point, and so can't be heard. The answer is that these event fragments become useful when it comes to using patterns in different ways, such as combining with yet another control pattern, or feeding it into a function that transforms the resulting pattern so fragmentary event onsets once more come into play.

Chapter 4

Haskell syntax

Tidal is a language for making patterns, but it is embedded in another, general purpose programming language called *Haskell*. You don't need to learn Haskell to learn TidalCycles, so you are encouraged to skip any parts of this chapter that don't bring you joy right now. Later on though, you might want to clear up some confusion and come back with a heart full of questions.

4.1 Organising functions with $ (or ())

Tidal is all about functions. A function is anything which takes one or more inputs, and gives you an output. For example the `rev` function takes a pattern as input, and returns a reversed version of it as output. That's basically it, but things get a little more complicated; a function takes multiple inputs, and one (or sometimes more!) of those might be another function. We'll have a look at some of those a little later.

You see the dollar sign `$` a *lot* in Tidal patterns, and it can take a while to get your head around exactly what they're doing. In fact, besides helping structure code, they don't really do anything! However once you're used to them, they make tidal code quick to write and work with.

So if they don't do anything, what are they for? Basically, for helping organise inputs to functions. Lets have a think about why `$` is used in this simple pattern:

```
d1 $ sound "bd sd"
```

The `$` sits between `d1`, which is a function, and `sound "bd sd"`, a pattern of sounds. Using the `:t` instruction, we can ask haskell what kind of function `d1` is. If we run `:t d1` from a tidal session, this (or something very similar) will appear in the output buffer:

```
d1 :: ControlPattern -> IO ()
```

This means that `d1` takes a single `ControlPattern` (i.e. a control pattern) as input, and then does some `IO ()` (which stands for input/output but really means any action in the outside world, in this case outputting some sound) as output. What happens if you try to do without the `$`?

```
d1 sound "bd sd"
```

If you try to run the above, you'll get an error saying something like "The function 'd1' is applied to two arguments, but its type 'ControlPattern -> IO ()' has only one". Well, that error could be clearer, but what it's trying to say is that you're trying to give both `sound` and `"bd sd"` to `d1`, which is two things, whereas `d1` only wants a single thing. What the `$` or `()` does is work out everything to the right of it first, in this case giving the `"bd sd"` pattern to `sound`. The single result of *that* (a control pattern) is then given to `d1`.

To confirm things, we can also ask for what *type* of thing `sound` is with `:t sound`, which says:

```
sound :: Pattern String -> ControlPattern
```

There we see that `"bd sn"` is read as a `Pattern String`, which `sound` turns into `ControlPattern`, which is exactly what `d1` wants.

As an alternative, it's possible to get things happening in the right order by using `()` instead of `$`, for example the following will work fine:

```
d1 (sound "bd sd")
```

The disadvantage of the above is that you have to match opening and closing brackets, which can get difficult in more complicated patterns. The advantage is that it works in more situations. In particular the `$` only works for the *last* input to a function, as it tries to eat up *everything* on its right. So often, you'll mix and match `()` and `$`, for example:

```
d1 $ every 3 (fast 2) $ sound "bd sn"
```

In the above, the `every` function wants three inputs, and in this example gets the number `3`, the function `fast 2`, and the pattern `sound "bd sn"`. The second input has to be wrapped in parenthesis, but for the final input we can use the dollar instead. If we want to make that whole pattern extra-fast, we can do that quickly by passing it all to `fast 4`, with a dollar:

```
d1 $ fast 4 $ every 3 (fast 2) $ sound "bd sn"
```

The `every` function takes *three* inputs - a count, a function, and a pattern. The function gets applied to the pattern every time the cycle count is up. Lets have a close look at its type, with `:t every`:

```
every :: Pattern Int -> (Pattern a -> Pattern a) -> Pattern a -> Pattern a
```

This is again trying to tell us that `every` takes three inputs, and gives one output. First there's a `Pattern Int`, the count (*int* stands for integer, i.e. a whole number). Next there's `(Pattern a -> Pattern a)` the thing that is done, in particular a function that simply takes a pattern as input, and gives a pattern as output. Examples of such functions would be `rev` (reverse the pattern), or `fast 2` (make the pattern run twice as fast). Then there's `Pattern a`, the input pattern that is being transformed, and finally the ouput `Pattern a`, which is the output – the transformed pattern. It says `Pattern a`, rather than `Pattern String` or whatever, because `every` will work on *any* kind of pattern, no matter what the contents of it is.

Ok, that got a little technical. If all this still isn't clear, don't worry – you'll get more of a feel for it through practice.

4.1.1 $ vs

As an aside, it's common for Tidal beginners to mix up $ and #, because they both seem to sort of glue things together in a pattern. They have quite different roles, though. $ sits between a function its (last) input, and # sits between two patterns, and combines them together. That's all!

4.2 Passing effects as 'sections'

4.3 Composing functions with .

Sometimes you'll have a function (lets stick with `every`) that lets you do something to a pattern, but you want to do *two* things, or maybe more.

```
d1 $ every 3 rev $ sound "bd sd"
```

Lets say you wanted to do the above, but as well as reversing the pattern, you also wanted to speed it up. This is where the . operator comes in handy – it lets you join two functions together into a single function. Here we go:

```
d1 $ every 3 (fast 2 . rev) $ sound "bd sd"
```

Or with a section as well:

```
d1 $ every 3 (fast 2 . rev . (# speed 2)) $ sound "bd sd"
```

The . operator is kind of *chaining together* the functions. It creates a new function that passes its input to `(# speed 2)`, passes the result of *that* to `rev`, and then another . passes the result of *that* to `fast 2`. The . function is kind of similar to $, but the $ separates an input from a function, and the . joins a function to another function, to create a new function. There!

This all becomes super useful when you start getting more confident with Tidal, but again if it's unclear, don't worry about it for now!

4.4 More Haskell resources

Tidal is embedded in Haskell, and all the syntax explored in this chapter has been about Haskell in general rather that Tidal in particular. If your ears are pricked and you'd like to know more, there are a lot of books and online resources for you. For example, some really enjoy *Learn You A Haskell For Great Good*, available for free at on learnyouahaskell.com. Others are less happy with the nature of its humourous tone, and find the Haskell Book suits them better, which you can find via haskellbook.com.

However, you really don't need to learn any more Haskell than we've covered in this chapter. Writing Tidal is otherwise generally all about using its functions and mini-notation. Furthermore, general Haskell texts tend to focus a lot on things like lists and monads, which Tidal makes very little use of.