TidalVortex Zero

Alex McLean
Then Try This
alex@slab.org

Raphaël Forment
Université Jean Monnet (UJM)
raphael.forment@gmail.com

Sylvain Le Beux
LANDR Audio
artheist@gmail.com

Damián Silvani
University of Buenos Aires (UBA)
munshkr@gmail.com

ABSTRACT

In this paper we introduce 'version zero' of TidalVortex, an alternative implementation of the TidalCycles live coding system, using the Python programming language. This is open-ended work, exploring what happens when we try to extract the 'essence' of a system like TidalCycles and translate it into another programming language, while taking advantage of the affordance of its new host.

First, we review the substantial prior art in porting TidalCycles, and in representing musical patterns in Python. We then compare equivalent patterns written in Haskell (TidalCycles) and Python (TidalVortex), and relate implementation details of how functional reactive paradigms have translated from the pure functional, strongly typed Haskell to the more multi-paradigm, dynamically typed Python. Finally, we conclude with reflections and generalisable outcomes.

1. INTRODUCTION

TidalCycles is a live coding environment developed since 2009 [1], for improvising music with algorithmic patterns [2]. It is a domain specific language (DSL) embedded in Haskell, a pure functional programming language. It makes use of Haskell's type system to represent patterns as functions of time, based on pure functional reactive programming (FRP) techniques [3], extended to unify continuous signals and discrete sequences within a single representation of patterns. That is, in TidalCycles, patterns are not stored as data structures, but as functional *behaviour*.

Part of what makes TidalCycles expressive is the ability to flexibly compose patterns together into new behaviours, thanks to Haskell's foundational support for applicative functors and monads. Explaining these type-level structures is outside the scope of this paper, so instead we explain what they do in TidalCycles. The applicative functor definition for patterns means that a live coder can, for example, add together two patterns of numbers to create a new pattern, even if the patterns have complex structures that are very different from one another. Similarly, the pattern monad definition supports the ability for pattern manipulations to themselves be patterned, for example by patterning the factor by which another pattern is being speeded up or down. The result is a terse language that even groups of eight year olds are able to grasp sufficiently to make and perform music together within an hour [4]. So while these type-level concepts have a reputation for being difficult to understand, we argue that any difficulty is not exposed to the TidalCycles end-user programmer.

It is not only applicative functors and monads which TidalCycles depends on for its expressive pattern DSL. Haskell's type inference, partial application, string overloading, and *parsec* parser combinator library [5] are all made extensive use of in creating a language that is terse and expressive enough to support fast-paced, from-scratch live coding.[1] This leads to the question explored in this paper - what happens when you try to extract the 'essence' of TidalCycles, and translate it into another programming language?

2. FROM HASKELL TO PYTHON

To answer the above question, we re-implemented the TidalCycles pattern representation in Python, a multi-paradigm programming language in widespread use including across industry, scientific and creative contexts. At the time of writing, the port to Python is around one month old, and is codenamed *TidalVortex*. Thanks to a team of collaborators (the present authors) it is already usable, with a custom editor, using the SuperDirt engine[2] created by Julian Rohrhuber for TidalCycles, and with networked time synchrony using the Link protocol [5].

Before outlining the challenges of creating TidalVortex, we will first review related Python live coding systems.

2.1 Prior art in Python live coding systems

Existing Python-based live-coding environments include *FoxDot*[3] by Ryan Kirkbride, *Bespoke*[4] by Ryan Challinor and *Isobar*[5] by Daniel Jones. These are each unique, highly capable systems, built on different foundations and designed with particular aims. FoxDot

[1] For full details on the TidalCycles syntax and functionality, see https://tidalcycles.org/
[2] https://github.com/musikinformatik/SuperDirt
[3] https://foxdot.org/
[4] https://www.bespokesynth.com/
[5] http://ideoforms.github.io/isobar/

cites some influence from TidalCycles for its notation, but is based on an object-oriented paradigm, privileging stateful interaction between musical voices [6]. Bespoke integrates live coding within an expansive, visual patcher-style audio environment, where the end-user live coder is given free reign to define their own musical constructs within a callback-based system. Isobar and TidalCycles emerged at the same time and from the same lab (the Intelligent Sound and Music Systems group in Goldsmiths, University of London), and Isobar offers an extensive library of functions for composing and manipulating patterns. However again the underlying representations differ, with Isobar closely influenced by SuperCollider's own pattern DSL [7], supporting stateful operations on sequences rather than functions of time. Without getting too deeply into the design trade-offs at play, the pure FRP approach of TidalCycles results in patterns which are very easy to compose, combine and transform, but due to the lack of state, have relatively poor support for some 'classic' algorithmic composition techniques such as L-systems, cellular automata and Markov chains. There's no particular reason why Tidal (-Cycles or -Vortex) couldn't become multi-paradigm and better support these techniques, but in any case, readers interested in them are encouraged to take a look at Isobar.

2.2 Prior art in porting Tidal features outside Haskell

Several non-Haskell projects have already focused on porting some TidalCycles features for integration into existing live coding environments. However these projects have not tended to port Tidal's internal pattern representation. Rather, they have focussed on the TidalCycles mini-notation for sequences, which is itself inspired by Bernard Bel's *Bol Processor* [8]. Examples of projects mapping the mini-notation to their own representations include the JavaScript-based tidal.pegjs project[6] implemented for Gibber [9], and Bacalao[7] for SuperCollider. David Ogborn's *Estuary* platform and its built-in MiniTidal language deserves special mention [10]. This project allows multiple users to use a large subset of TidalCycles referred to as "MiniTidal" from a web browser. Strictly speaking, this is not a port, behind the scenes it runs the entirety of TidalCycles via the ghcjs Haskell-to-Javascript compiler, with a custom parser.

TidalVortex Zero differs from these examples in porting the 'innards' of TidalCycles, by re-implementing its core FRP approach in Python.

2.3 Challenges of porting code from Haskell to Python

On the face of it, Haskell and Python are very different languages. Haskell is pure functional and strictly typed, offering a conceptual environment which can feel inflexible and unforgiving, but which therefore demands clear thinking and very well-defined representations. By contrast, Python is 'multi-paradigm' in terms of supporting object-oriented, functional and procedural approaches to structuring a program. *Functional* is quite an ambiguous term, but in this case means that Python allows functions to be treated as values, and therefore passed to and returned from other functions. Unlike Haskell, it is not however *pure* functional, so there is no guarantee that a function will not, for example, change the state of the program in an unpredictable way. Relatedly, Python does not support features like inbuilt memoisation or tail-recursion optimisation, which Haskell programmers routinely rely upon when making efficient programs in a functional style. Nonetheless, Python *does* in general have well developed support for functional programming constructs, including the `lambda` call for constructing anonymous functions, list comprehensions and maps, and its *functools* library for e.g. supporting partial application[8].

Our feeling then is that it would have been difficult to create something close to TidalCycles in Python in the first place. The conceptual development of its representation of pattern required deep thinking, with clarifying moments heavily supported by Haskell's type system. However with the work done in developing TidalCycles in Haskell, we have found that to a large extent it is possible to translate it to Python.

2.4 What parts of TidalCycles have made it to TidalVortex?

The TidalVortex port from Haskell to Python was primarily based not on the current TidalCycles codebase, but on an ongoing experimental rewrite of it [12]. The core representation of this rewrite aims to be functionally identical to mainline TidalCycles, but has been refactored, and is therefore easier to translate.

TidalCycles really consists of *two* languages, one embedded in the other. The embedded language is known as its *mini-notation*, for quickly describing sequences, including syntax for defining subsequences, polymeters, polyrhythms, random selection and so on. Mini-notation sequences are denoted with double-quotes but are not otherwise treated as strings — they are immediately parsed into patterns, i.e. functions of time, for further manipulation and combination with other patterns. This is done with the extensive combinator library that forms the core of TidalCycles, and allows the end-user live coder to make complex wholes from simple parts. So we can say that TidalCycles consists of a library of pattern functions for transforming and combining patterns, with a mini-notation embedded into it as a quick way of expressing sequences into patterns.

The mini-notation is really a short-hand, and although the vast majority of TidalCycles users make heavy use of it, it is not essential, and so far we have not prioritised

[6] https://github.com/gibber-cc/tidal.pegjs

[7] https://github.com/totalgee/bacalao

[8] To explore further, see the dedicated page: https://docs.python.org/3/howto/functional.html

implementing it in TidalVortex. This is because the lack of a mini-notation allows positive design focus on making the base DSL as expressive as possible.

As well as the lack of mini-notation, a large portion of the TidalCycles library of functions are not yet implemented. Focus has instead been on the low-level representation of patterns, and the end-user live coding interface in terms of how patterns are manipulated and combined.

3. COMPARING PATTERNS

In this section we will share examples of equivalent patterns written in TidalVortex and TidalCycles. These are not intended to demonstrate the full range of possibilities in either, or showcase them as musical interfaces. Rather, we aim only to compare and contrast their practicalities, in order to informally evaluate the success of the TidalVortex port so far. To help avoid confusion between them, we show TidalCycles examples with a grey background, before equivalent TidalVortex examples in white. We refer to properties common to both TidalCycles and TidalVortex implementations simply as *Tidal*. Some basic familiarity of TidalCycles will help understand what follows, so please refer to the documentation for a primer if you are new to all this.[9]

As we mentioned earlier, TidalVortex does not have a mini-notation, so is therefore more verbose[10]:

```
sound "bd ~ [sd cp]"
```

```
sound("bd", silence, ["sd", "cp"])
```

The first example in TidalCycles (and therefore Haskell) shows the mini-notation in double quotes. The second example in TidalVortex expresses the same structure only using lists. Nonetheless they have similar structures, because functions that take patterns in TidalVortex automatically turn values into patterns, treat multiple parameters as a sequence that are 'concatenated' into a single pattern, and treat lists as subpatterns that are treated as a single 'step' in the sequence. This mirrors the behaviour of subsequences in the mini-notation.

It's also possible to define the same pattern without mini-notation in TidalCycles:

```
sound $ fastcat
   [pure "bd", silence,
    fastcat(pure "sd", pure "cp")]
```

Without mini-notation, the TidalCycles example ends up being more verbose, as due to Haskell's strict typing, everything must be expressed as patterns.

[9] See https://tidalcycles.org/ for documentation.

[10] The re-introduction of a mini-notation is planned, potentially using Python's *parsy* library, which is similar to Haskell's *parsec*.

The mini-notation angle brackets `<>` are equivalent to using the `slowcat` function in Tidal, so that the following have identical results:

```
sound "bd ~ <sd cp>"
```

```
sound("bd", silence, slowcat("sd", "cp"))
```

In the mini-notation, braces `{}` denote polymeters, which are made in TidalVortex using the `polymeter` function, or its shorthand `pm`:

```
sound "bd {cp sd, lt mt ht}"
```

```
sound("bd", pm(["cp", "sd"],
               ["lt", "mt", "ht"]))
```

Similarly, polyrhythms denoted with mini-notation square brackets `[]` can be specified with `polyrhythm` or it's shorthand `pr`, and may be embedded:

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

```
sound("bd", pm(["cp", "sd"],
               [pr(["lt", "mt"],
                   ["bd", "bd", "bd"]
                  ),
                "ht"
               ]))
```

Note that `polyrhythm`/`pr` is equivalent to `stack` in TidalCycles, and indeed TidalVortex supports that as an alias too.

Combining patterns is similar, with TidalVortex using the `>>` operator:

```
sound "bd sd cp" # speed "1 2"
```

```
sound("bd", "sd", "cp") >> speed (1, 2)
```

Unlike in Haskell, it's not possible to add new operators to Python, but it *is* possible to repurpose existing operators, in this case the `>>` operator usually used for bit-shifting. This means the full range of TidalCycles inline operators aren't implementable in TidalVortex, which will have to use prefix functions and methods instead.

Applying functions is also similar:

```
rev $ sound "bd sd"
```

```
rev(sound("bd", "sd"))
```

Functions are also defined as pattern methods, so we can swap things around, avoiding embedded parenthesis:

```
sound("bd", "sd").rev()
```

In this way, Python's `.` operator for object methods[11] is used similarly to the `$` operator in Haskell, in that they both separate a function from its input[12]. The following three examples are equivalent, but the final TidalVortex example avoids some of the potentially confusing embedded parenthesis. It reads similarly to the TidalCycles example, only in the opposite direction, being applied from left to right rather than from right to left.

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

```
jux(rev, every(3, fast(2),
                sound("bd", "sd")))
```

```
sound("bd","sd").every(3, fast(2)).jux(rev)
```

The above three examples demonstrate the use of partial application, which is built-in to Haskell, but requires a little work in Python, with the support of the *functools* library. In particular, the `fast` function requires two parameters as input, namely the factor by which a pattern is sped up by, and the pattern to be sped up. When it is given only the first parameter, rather than treating this as an error, it simply returns a function that accepts the second argument. Partial application may be a little difficult to understand in theory, but as we think the above examples show, is very easy to use in practice. Without partial application, instead of `fast(2)`, we would have to write `lambda pat: fast(2, pat)`, which is much more cumbersome.

As with TidalCycles, arithmetic operators may also be used with patterns of numbers. For example:

```
n ("1 2 3" + "4 5") # sound "drum"
```

```
n (sequence(1,2,3) + sequence(4,5))
   >> sound "drum"
```

In both implementations, the values are matched up across patterns in order to perform the addition. The resulting pattern is identical to:

```
n "5 [6 7] 8" # sound "drum"
```

In the previous example, it would be nice if `sequence(1,2,3) + sequence(4,5)` could instead be expressed as `[1,2,3] + [4,5]`. However, that would involve overriding the standard, expected behaviour of adding lists together in Python, i.e. concatenation, and going further with treating lists as patterns will likely lead to confusion. We are wary of going too far down this route, but may explore making this optional in the future. For now, we have to explicitly turn sequences into patterns before performing such actions on them.

TidalVortex also supports continuous patterns, for example sinewave signals:

```
speed("1 2 3" + sine)
```

```
speed(sequence(1,2,3) + sine)
```

In the above, the sinewave is continuous, but would be sampled from in order to combine values with the discrete pattern on the left. This works out to be roughly equivalent to:

```
speed "1.93301 2.5 3.06698"
```

In summary, as a proof of concept, TidalVortex demonstrates that representational concepts and code UI features translate fairly well from Haskell to TidalVortex. Indeed the flexibility of Python's type system means that there is perhaps a lesser need for a stripped-down mini-notation for sequences there. If we do without a mini-notation, everything is expressed within the host language, which may turn out to be easier to work with and understand. We do still intend to implement the mini-notation in TidalVortex, but look towards a broader aim of unifying Tidal patterns and sequences, as outlined in a previous paper [12].

4. IMPLEMENTATION

The examples shown in the previous section rely upon a few concepts that are standard in Haskell but less common in Python. We will expand a little on some of these in this section, in particular the functor, applicative and monadic definitions. Before that though, we go some way to explaining the basic model for representing patterns in Python. For full details, see the source code.[13] As before, where we make statements true of both TidalVortex and TidalCycles, we refer to them collectively as Tidal.

In Tidal, time is rational, allowing subdivisions to be properly represented. As such there is no fixed 'tatum' or indivisible unit of time, any 'step' can be arbitrarily subdivided.

4.1 Object classes

In TidalVortex, patterns are represented using object classes. The objects are treated as immutable, in that object methods should never change the object, but return

[11] Note that the dot (.) operator for calling methods in Python is very different from the dot operator in Haskell for function composition.

[12] In Python an object (conventionally named `self`), is passed as the first input to its methods.

[13] The TidalVortex source code is available at github.com/tidalcycles/vortex, under the GPLv3 license.

a new object with any changes made. In other words, all the object methods are pure functions.

A timespan (also known as an *arc*) is one of three such object classes defined in TidalVortex, consisting of a pair of time values specifying the beginning and end of the timespan, and methods to perform operations such as returning the intersection between two timespans.

The second object class represents events, with data consisting of a value, and a timespan indicating when that value is 'active'. An event might be a fragment of another one, which is represented by a second timespan.

The final object class is the pattern itself, which represents patterned behaviour as a function, with a timespan as input, and a list of events active during that timespan as output. This object class already has a large number of methods for combining and transforming patterns, although again none of them change the object that the method is called upon, but rather return a new, transformed pattern. This reflects the pure functional underpinnings of Tidal.

This use of object classes allows the end-user to transform a pattern with methods, but all these methods have aliases as top-level functions, where a pattern to be transformed is given as a final parameter. So, to transform a pattern named `pat`, we can either call `pat.fast(3)`, or `fast(3, pat)`. The top-level aliases support partial application (via the afore-mentioned *functools* library), and are only intended for passing to other functions such as with `every` in the earlier examples.

4.2 Functor, applicative and monadic operations

As mentioned in the introduction, much of Tidal's functionality comes from three constructs; a) the definition of patterns as functors, so that patterns of values can be manipulated as values, b) the definition of patterns as applicative functors, so that more than one pattern of values can be treated as values and therefore combined using a function with multiple values as input, and c) the definition of values as monads, so that e.g. patterns of patterns can be flattened into patterns.

The above paragraph is a bit of a riddle, and we do not have space to go into implementation details. However, the practicalities can be explained with simple examples.

```
sequence(1,2,3).with_value(lambda x: x+1)
```

The `with_value` method, called on the pattern created by the `sequence` function, simply applies the provided function to every value in the pattern, in this case resulting in a pattern equivalent to `sequence(2,3,4)`. This is known as the `fmap` or *functor map* in Haskell, and indeed has the same `fmap` alias in TidalVortex. As mentioned earlier, the addition operator is overridden for patterns, so the above can be expressed more simply as:

```
sequence(1,2,3) + 1
```

However, we've also seen patterns added to patterns, like this:

```
sequence(1,2,3) + sequence(4,5)
```

How is that possible? This is where *applicative* comes in. With two patterns that you want to add together (`pat1` and `pat2`), the first step is to make a new pattern (`patf`), that is a pattern of functions that add the numbers in the first pattern:

```
pat1 = sequence(1,2,3)
pat2 = sequence(4,5)
patf =
 pat1.with_value(lambda x: lambda y: x + y)
```

To resolve the pattern of functions back to a pattern of values, you call the `app` method, which will match up the values in pat2 with the functions in patf:

```
pat3 = patf.app(pat2)
```

An end-user live coder would rarely call `app` directly; this is what happens internally when you use `+` to add two patterns together. But being able to apply a pattern of values to a pattern of functions, as though they were simple values being passed to functions, is of great use, especially as these operations can be chained to work with functions with more than two inputs. The hard work in matching up events from the different patterns at play is done by the `app` function, so that we can combine patterns together freely. The `app` method is also behind the `>>` operator in the earlier examples, performing a union of control patterns.

What about the fabled monadic operations? Consider:

```
sequence(1,2,3).fast(2,4)
```

We start with a sequence, and then 'speed it up' with the `fast` method. But! The parameter to `fast` is itself the patterned sequence `2,4`[14]. How can we pattern with patterns? First, let's define the pattern of factors separately from the pattern we want to speed up:

```
pat = sequence(1,2,3)
factor_pat = sequence(2,4)
```

The first step is to use the `with_value` method, or here for brevity its alias `fmap`:

```
pat_pat = factor_pat.fmap(lambda factor:
                          pat.fast(factor)
                         )
```

[14] Remember that if you pass more than one value to a method, it will treat it as a sequence.

This speeds up our pattern, but *inside* our pattern of factors. So, we end up with a *pattern of patterns of values*. To flatten this to a pattern of values, we call the `bind` method:

```
new_pat = pat_pat.bind()
```

Similar to app, bind does the job of matching up the events in the 'outer' pattern with the events in the 'inner' pattern. This is difficult to conceptualise, especially when we remember that patterns aren't data structures, but functions of time. However this (monadic) bind is very easy to use in practice, we just use simple functions like `fast` without thinking about what's going on under the hood.

5. CONCLUSION

As a proof-of-concept, TidalVortex succeeds in demonstrating that it is possible to port the 'essence' of Tidal to a multi-paradigm language, and has already seen successful use in live performance. This work has mainly been driven by curiosity, but having achieved a working system, we now have cause to look for motivations to continue work on it. One motivation is the popularity of Python, both in terms of the larger pool of developers able to work on the project, and Python's reserve of freely available libraries for e.g. graphical programming, creative coding and machine learning that could open up new possibilities including new mini-notation parsers, GUI and user-experience alternatives.

TidalVortex also offers a starting point for further ports to similar programming languages, representing the theoretical foundations of Tidal in a multi-paradigm language. Indeed experimental Tidal ports based on TidalVortex Zero have already started to appear.[15]

We have resisted the temptation to map out next steps in a 'future work' section, but nonetheless work remains to be done, including documenting the implementation and way of operation of this experimental software. However due to its shared inner-workings, the existing TidalCycles documentation can be adapted to TidalVortex.

Perhaps more than anything, through this project we are happy to have better understood what Tidal is, independent of a particular implementation. This not only has technical but cultural value, where there is often lack of understanding between communities that form around particular programming languages. With TidalVortex, we are happy to work against these artificial barriers.

Acknowledgments

Work by Alex McLean on this paper was supported by a UKRI Future Leaders Fellowship [grant number MR/V02526].

[15] For example Kidal, a port of Tidal for the Kotlin language https://gitlab.com/ndr_brt/kidal, and Strudel, for Javascript https://strudel.tidalcycles.org/

6. REFERENCES

[1] McLean, Alex. ‘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*. Gothenburg, 2014. https://doi.org/10/gfvbzc.

[2] Mclean, Alex. ‘Algorithmic Pattern’. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 265–70. Birmingham, UK, 2020. https://doi.org/10/gn3zd5.

[3] C. Elliott and P. Hudak, ‘Functional reactive animation’, in Proceedings of the second ACM SIGPLAN international conference on Functional programming, New York, NY, USA, Aug. 1997, pp. 263–273. doi: 10.1145/258948.258973.

[4] McLean, Alex, and Renick Bell. ‘Pattern, Code and Algorithmic Drumming Circles’. Proceedings of the Fourth International Conference on Live Coding. Presented at the International Conference on Live Coding 2019 (ICLC 2019), Madrid, 16 January 2019. https://doi.org/10.5281/zenodo.3946174.

[5] D. Leijen and E. Meijer, ‘Parsec: Direct Style Monadic Parser Combinators for the Real World’, presented at the 11th International Conference on User Modeling, 2007.

[6] F. Goltz, ‘Ableton Link – A technology to synchronize music software’, presented at the Linux Audio Conference 2018, Berlin, Aug. 2018.

[7] Kirkbride, Ryan. "Foxdot: Live coding with python and supercollider." In *Proc. of the International Conference on Live Interfaces*, pp. 194-198. 2016.

[8] S. Wilson, N. Collins, et D. Cottle, Éd., The SuperCollider book. MIT Press, 2011, pp. 179–207.

[9] B. Bel, "Migrating musical concepts: An overview of the Bol processor", *Computer Music Journal*, vol. 22, n° 2, p. 56-64, 1998.

[10] C. Roberts, M. Pachon-Puentes. "Bringing the tidalcycles mini-notation to the browser" *in Proceedings of the Web Audio Conference*, 2019.

[11] D. Ogborn, J. Beverley, L. Navarro del Angel, E. Tsabary, et A. McLean, "Estuary: Browser-based Collaborative Projectional Live Coding of Musical Patterns", *in Proce. of the International Conference on Live Coding (ICLC)*, 2017.

[12] McLean, Alex. ‘Alternate Timelines for TidalCycles’. Presented at the International Conference on Live Coding (ICLC2021), Valdivia, Chile, 15 December 2021. https://doi.org/10.5281/zenodo.5788732.