Making programming languages to dance to: Live Coding with Tidal

Alex McLean
Interdisciplinary Centre for Scientific Research in Music, University of Leeds
a.mclean@leeds.ac.uk

Abstract

Live coding of music has grown into a vibrant international community of research and practice over the past decade, providing a new research domain where computer science blends with the arts. In this paper the domain of live coding is described, with focus on the programming language design challenges involved, and the ways in which a functional approach can meet those challenges. This leads to the introduction of Tidal 0.4, a Domain Specific Language embedded in Haskell. This is a substantial restructuring of Tidal, which now represents musical pattern as functions from time to events, inspired by Functional Reactive Programming.

Categories and Subject Descriptors J.5 [*Performing Arts*]; J.5 [*Music*]; D.3.2 [*Applicative (functional) languages*]

Keywords domain specific languages, live coding, music

1. Introduction - Live programming languages for music

Live coding is where source code is edited and interpreted in order to modify and control a running process. Over the past decade, this technique has been increasingly used as a means of creating live, improvised music (Collins et al. 2003), with new programming languages and environments developed as end-user music interfaces (e.g. Wang and Cook 2004; Sorensen 2005; Aaron et al. 2011; McLean et al. 2010). Live coding of music and video is now a vibrant area of research, a core topic in major Computer Music conferences, the subject of journal special issues, and the focus of international seminars. This research runs alongside emerging communities of live coding practitioners, with international live coding music festivals held in the UK, Germany and Mexico. Speculative, isolated experiments by both researchers and practitioners have expanded, developing into active communities of practice.

Live coding has predominantly emerged from digital performing arts and related research contexts, but connects also with activities in Software Engineering and Computer Science, under the developing umbrella of live programming language research (see for example the proceedings of the LIVE workshop, ICSE 2013). These intertwined strands are revitalising ideas around liveness first developed decades ago, explored in now well-established systems such as Self, SmallTalk, Lisp, command line shells and indeed spreadsheets. Continuing this tradition, and making programming languages "more live" is of interest in terms of making programming easier to teach and learn, making programs easier to debug, and allowing programmers to more easily achieve creative flow (Blackwell et al. 2014). How these different strands weave together is not always clear, but cross-disciplinary engagement is certainly warranted.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Live coding as a design challenge

Live coding of music brings particular pressures and opportunities to programming language design. To reiterate, this is where a programmer writes code to generate music, where a running process continually takes on changes to its code, without break in the musical output. The archetypal situation has the programmer on stage, with their screen projected so that the audience may see them work. This might be late at night with a dancing nightclub audience (e.g. at an algorave Collins and McLean 2014), or during the day to a seated concert hall audience. The performer may be joined by other live coders, or perhaps instrumental musicians, but in any case the programmer will want to enter a state of focused, creative flow and work beyond the pressures at hand.

There are different approaches to live coding music, but one common approach is based on an improvised Jazz model. The music is not composed in advance, instead the music is developed through live interaction, with live coders "playing off" each other, or shaping the music in sympathy with audience response. The improvisers might add extra constraints, for example the Mexico City live coding community are known to celebrate the challenge of live coding a performance from scratch, lasting precious few minutes. "Slow coding" is at the other end of the scale, exploring a more conversational, meditative ethos (Hall 2007).

At this point it should be clear that live coding looks rather different from mainstream software engineering. There is no time for test driven development, little time to develop new abstractions, and where the code is deleted at the end of a performance, there are no long-term maintenance issues. However, live programming languages for music do have strong design pressures. They need to be highly expressive, both in terms of tersity, and also in terms of requiring close domain mapping between the code and the music that is being expressed. As music is a time-based art-form, representation of time structures is key. Familiar mechanisms such as revision control may be employed in unusual ways, supporting repeating structures such as chorus and verse, where code is branched and merged within short time frames, creating cyclic paths of development.

2.1 Liveness and feedback

It is worth considering what we mean by the word *live*. In practice, the speed of communications is never instantaneous, and so in a sense nothing is completely live. Instead, lets consider liveness in terms of *live feedback loops*, where two agents (human or computational) continually influence one another. We can then identify different forms of liveness in terms of different arrangements of feedback loops.

In a live coded performance, there are at least three main feedback loops. One is between the programmer and their code; making a change, and reading it in context alongside any syntactical errors or warnings. This loop is known as *manipulation feedback* (Nash and Blackwell 2011), and may possibly include process and/or data visualisation through debugging and other programmer tools. A second feedback loop, known as *performance feedback* (Nash and Blackwell 2011), connects the programmer and the program output, in this case music carried by sound. In live coding of music, the feedback cycle of software development is shared with that of musical development. The third loop is between the programmer and their audience and/or co-performers. We can call this feedback loop *social feedback*, which is foregrounded at algorave events, where the audience is dancing.

2.2 Programming Language Paradigms for Music

A large number of programming languages have been designed for music and digital sound processing (DSP) over the past few decades, for example ChucK, SuperCollider, Max/MSP, HMSL, Common Music and the MusicN languages. As processor frequencies have increased, the promise of realtime processing has put new design pressures on languages. Live coding has emerged over the past decade, as the promise of realtime as an exploratory activity.

There are a range of programming language paradigms in computer music. Perhaps the most dominant paradigm is dataflow programming; declarative functions which do not return any values, but take streams of data and inputs, and send streams of output to other functions as a continual side effect. These languages, such as Max/MSP, PureData and VVVV, usually have a graphical "Patcher" interface, where words are contained within "boxes", connected with "wires" to form the dataflow network. The accessibility of these systems may be attributed to their similarity to the analogue synthesisers which preceded and inspired them (Puckette 1988).

However the most common paradigm in live coding performance seems to be functional programming; many live coding environments such as Overtone, Fluxus and Extempore are Lisp dialects, and the pure functional Language Haskell is the basis of a number of live music EDSLs (embedded domain specific languages); namely Conductive (Bell 2011), Live-Sequencer (Thielemann 2012) and Tidal. The Tidal language is introduced in the following section, with emphasis on its approach to the representation of time.

3. Introducing Tidal

Tidal represents many years of development, and the present paper supersedes earlier work (McLean and Wiggins 2010), with several major rewrites since. At its essence it is a domain specific language for musical pattern, of the kind called for by Spiegel (1981), and present within many other systems including HMSL, SuperCollider (McCartney 2002) and ixilang (Magnusson 2011). Tidal has been developed through use, informed by many dozens of high profile performances to diverse audiences, and within diverse collaborations. The present author has predominantly used it within algorithmic dance music (Collins and McLean 2014, algorave;) and improvised free Jazz performances (Hession and McLean 2014),

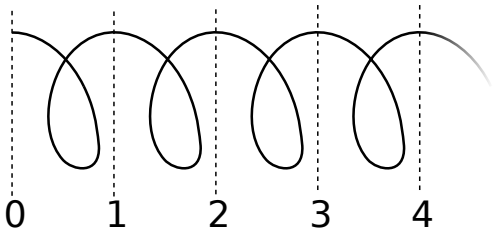


Figure 1. The Tidal timeline as an infinite spiral, with each cycle represented as a natural number, which may be subdivided at any point as a rational number.

as well as in live art (McLean and Reeve 2012) and choreographic (Sicchio 2014) collaborations. The software is available under a free/open source license, and it now has a growing community of users.

Tidal is embedded in the Haskell language, taking advantage of its rich type system. Patterns are represented using the below datatype, which we will explain in the following.

```
type Time = Rational
type Arc = (Time, Time)
type Event a = (Arc, Arc, a)
data Pattern a = Pattern (Arc → [Event a])
```

3.1 Representing Time

In Tidal, time is rational, so that musical subdivisions may be stored accurately as simple fractions, avoiding rounding errors associated with floating point numbers. Underlying this is the assumption that time is structured in terms of rhythmic (or more correctly, metric) *cycles*, a perceptual phenomena that lies at the basis of a great many musical traditions including Indian classical (Clayton 2008), and electronic dance musics. The first beat of each cycle, known as the *sam*, is significant both for resolving the previous cycle and for starting the next. The number line of whole numbers represents successive sam beats.

The Tidal timeline can be conceptualised as a spiral, as figure 1 illustrates. This raises the point that there is no expectation that cycles do not change from one cycle to the next. Indeed, polyrhythms are well supported in Tidal, but this simple cycle structure acts as the metric anchor point for Tidal's pattern operations.

In practice, when it comes to turning a pattern into music, how cycles relate to physical time depends on how fast the musician wants the music to go. This is managed externally by a scheduler, and multiple live coders can share a tempo clock over a network connection, so that the playback of their patterns is in time.

In sympathy with the focus on cycles, as opposed to the linear progression of time, a time range is called an `Arc`, specified as a start and stop time. When an arc represents the occurrence of a musical event, the start and stop are known as the event *onset* and *offset*, which are standard terms borrowed from music informatics.

An `Event` associates a value with two time arcs; the first arc gives the onset and offset of the event, and the second gives the 'active' portion. The need for the second arc will be explained later, for now we will just say that if an event is cut into pieces, it is important for each piece to store its original arc as context.

Finally, a `Pattern` represents an infinite series of Events as a function, from an `Arc` to a list of events. To retrieve events from the pattern, it is queried with an `Arc`, and all the events active during

the given time are returned. The arcs of these events may overlap, in other words supporting musical *polyphony*.

All Tidal patterns are notionally infinite in length; they cycle indefinitely, and can be queried for events at any point. Long-term structure is certainly possible to represent, although Tidal's development has been focused on live coding situations where such structure is already provided by the live coder, who is continually changing the pattern.

This use of functions to represent time-varying values borrows ideas from Functional Reactive Programming (Elliott 2009). However, the particular use of time arcs appears to be novel, and allows both continuous and discrete patterns to be represented within the same datatype. For discrete patterns, events active during the given time arc are returned. For continuous structures, an event value is sampled with a granularity given by the duration of the `Arc`. In practice, this allows discrete and continuous patterns to be straightforwardly combined, allowing expressive composition of music through composition of functions.

3.2 Building and combining patterns

We will now look into how patterns are built and combined in Tidal. Our focus in this section will be on implementation rather than use, but this will hopefully provide some important insights into how Tidal may be used.

Perhaps the simplest pattern is `silence`, which returns no events for any time:

```
silence :: Pattern a
silence = Pattern $ const []
```

The 'purest' discrete pattern is defined as one which contains a single event with the given value, for the duration of each cycle. Such a pattern may be constructed from a single value with the `pure` function, which Tidal defines as follows:

```
pure x =
  Pattern $ λ(s, e) →
    map (λt → ((t%1, (t+1)%1),
               (t%1, (t+1)%1),
                x
               )
              )
              [floor s .. ((ceiling e) - 1)]
```

This is an internal function which is not often used directly; we will show alternative ways of constructing patterns later.

Having constructed some patterns, we can combine them in different ways. For example, the `cat` function returns a pattern which cycles through the given list of patterns over time. The patterns are interlaced, i.e. taking the first cycle from each pattern, then the second, and so on. To make this possible, the resulting pattern needs to manipulate time values that are passed to it, forward those values on to the patterns it encapsulates, and then manipulate the time values of the events which are returned.

Although Tidal is designed for musical pattern, our example patterns will be of colour, in sympathy with the current medium. The x axis represents time travelling from left to right, and the y axis is used to 'stack up' events which co-occur. Here we visualise the first cycle of a pattern, which interlaces pure blue, red and orange patterns:

```
cat [pure blue, pure red, pure orange]
```

We can use the `density` combinator to squash, or 'speed up' the pattern so we can see more cycles within it:

```
density 4 $ cat [pure blue, pure red, pure ↙
    ↳ orange]
```

Like `cat`, density works by manipulating time both in terms of the query and the resulting events. Here is its full definition, along with its 'antonym' `slow`:

```
density :: Time → Pattern a → Pattern a
density 0 p = p
density 1 p = p
density r p =
  mapResultTime (/ r) (mapQueryTime (* r) p)

slow :: Time → Pattern a → Pattern a
slow 0 = id
slow t = density (1/t)
```

The combinator `slowcat` can be defined in terms of `cat` and `slow`, so that the resulting pattern steps through the patterns, cycle by cycle:

```
slowcat :: [Pattern a] → Pattern a
slowcat ps =
  slow (fromIntegral $ length ps) $ cat ps
```

Now when we try to visualise the previous pattern using `slowcat` instead of `cat`, we only see blue:

```
slowcat [pure blue, pure red, pure orange]
```

This is because we are only visualising the first cycle, the others are still there.

The definition for combining patterns so that their events co-occur is straightforward:

```
overlay :: Pattern a → Pattern a → Pattern a
overlay p p' = Pattern $ λa → (arc p a) ++ ↙
    ↳ (arc p' a)

stack :: [Pattern a] → Pattern a
stack ps = foldr overlay silence ps
```

```
stack [pure blue, pure red, pure orange]
```

The vertical order of the events as visualised above is not meaningful; that the events co-occur simply allow us to make 'polyphonic' music, where multiple events may sound at the same time.

By combining the functions we have seen so far, we may already begin to compose some interesting patterns:

```
density 16 $ stack [pure blue,
                    cat [silence,
                         cat [pure green,
                              pure yellow]
                        ],
                    pure orange]
```

3.3 Parsing strings

The functions we have defined so far for constructing patterns are quite verbose, and therefore impractical. Considering that Tidal is designed for live musical performance, the less typing the better. So, a simple parser `p` is provided by Tidal, for turning terse strings into patterns. The previous example may be specified like this:

```
p "[blue, ~ [green yellow], orange]*16"
```

So, values within square brackets are combined over time with `cat`, and `stacked` if they are separated by commas. A pattern can have its density increased with `*`. Silence is specified by `~`, analogous to a musical *rest*.

For additional tersity, the GHC string overloading feature is used, so that the `p` function does not need to be specified.

So far we have only shown the core representation of Tidal, but this already allows us to specify fairly complex patterns with some tersity:

```
"[[black white]*32, [[yellow ~ pink]*3 ↙
    ↳ purple]*5, [white black]*16]]*16"
```

If curly brackets rather than square brackets are used, subpatterns are combined in a different way, timewise. The first subpattern still takes up a single cycle, but other subpatterns on that level are stretched or shrunk so that each element within them are the same length. For example compare the following two patterns:

```
density 6 $ "[red black, blue orange green]"
```

```
density 6 $ "{red black, blue orange green}"
```

In musical terms, the first example would be described as a triplet, and the latter a polyrhythm.

3.4 Patterns as functors

It is useful to be able to operate upon all event values within a pattern irrespective of their temporal position and duration. Within the functional paradigm, this translates to the requirement to define the pattern datatype as a functor. Because Haskell already defines functions and lists as functors, defining `Pattern` as a `Functor` instance is straightforward:

```
instance Functor Pattern where
  fmap f (Pattern a) =
    Pattern $ fmap (fmap (mapThd f)) a
  where mapThd f (x,y,z) = (x,y,f z)
```

This already makes certain pattern transformations straightforward. For example, musical transposition (increasing or decreasing all musical note values) may be defined in terms of addition:

```
transpose :: (Num a) ⇒ a → Pattern a
    → Pattern a
transpose n pattern = fmap (+n) pattern
```

The `Applicative` functor is a little more complex, but allows a pattern of values to be mapped over a pattern of functions. A minimal definition of Applicative requires `pure`, which we have already seen, along with the `<*>` operator:

```
(Pattern fs) <*> (Pattern xs) =
  Pattern $ λa → concatMap applyX (fs a)
  where applyX ((s,e), (s', e'), f) =
     map (λ(_, _, x) → ((s,e), (s', e'), f x))
         (filter
          (λ(_, a', _) → isIn a' s)
          (xs (s',e'))
         )
```

In combination with `<$>` (which is simply `fmap` but in operator form), this operator allows us to turn a function that operates on values, into a combinator which operates on patterns of values. For example, we can use the library function `blend`, which operates on two colours, into a combinator which operates on two colour patterns:

```
blend 0.5
  <$> "[blue orange, yellow grey]*16"
  <*> "white blue black red"
```

In the above, the `blend` function is only able to operate on pairs of colours, but the applicative definition allows it to operate on pairs of colours taken from 'inside' the two patterns. It does this by matching co-occurring events within the first pattern, with those in the second one, in particular the events in the second pattern with arcs which contain the onset of those in the first pattern. For example in the following `red` matches with the onsets of `black` and `grey`, and `green` matches with the onset of `white`, so we end up with a pattern resulting from blends of the colour pairs (red, black), (red, grey) and (green, white).

```
(blend 0.5 <$> "[black grey white]"
           <*> "red green")
```

Notice that the resulting pattern will always maintain the 'structure' of the first pattern over time. However where an event in the left hand pattern matches with multiple events in the right hand pattern, the number of events within this structure will be multiplied. For example:

```
(blend 0.5 <$> "[black grey white]" <*>
       "[red green, magenta yellow]")
```

4. Transformations

From this point, we will focus less on the implementation of Tidal, and more on its use. Please refer to the source code for any implementation details.

Reversal Reversal operations, and the resulting symmetries, are fundamental to pattern. Because Tidal represents a notionally infinite timeline, reversing a whole pattern is not possible. However, the notion of a cycle is core to Tidal, and reversing each cycle within a pattern is relatively straightforward.

```
rev "blue grey orange"
```

every Reversing a pattern is not very interesting unless you contrast it with the original, to create symmetries. To do this, we can use `every`, a higher order transformation which applies a given pattern transformation every given number of cycles. The following reverses every third cycle:

```
density 16 $ every 3 rev "blue grey orange"
```

whenmod is similar to every, but applies the transformation when the remainder of the first parameter divided by cycle number is less than the second parameter.

```
density 16 $ whenmod 6 3 rev "blue grey orange"
```

Shifting/turning patterns The `<~` transformation shifts a pattern to the left, or in cyclic terms, turns it anticlockwise. The `~>` does the opposite, shifting it to the left/clockwise. For example, to shift it one third to the left every fourth repetition, we could do this:

```
density 16 $ every 4 ((1/3) <~ ) "blue grey ↙
    ↳ purple"
```

The above shows every fourth cycle (starting with the first one) being shifted to the left, by a third of a cycle.

iter The `iter` transformation is related to `<~`, but the shift is compounded until the cycle gets back to its starting position. The number of steps that this takes place over is given as a parameter. The shift amount is therefore one divided by the given number of steps, which in the below example is ¼.

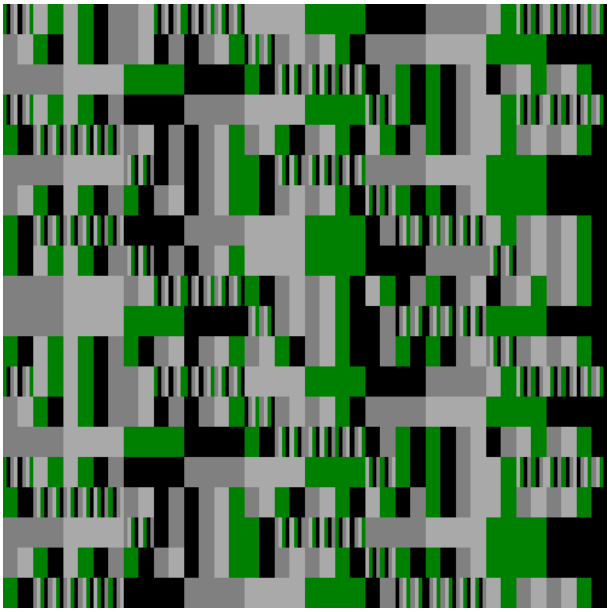
```
density 4 $ iter 4 $ "blue green purple orange"
```

superimpose is another higher order transformation, which combines the given pattern with the result of the given transformation. For example, we can use this with the transformation in the above example:

```
density 4 $ superimpose (iter 4) $ "blue ↙
    ↳ green purple orange"
```

Combining transformations All of these pattern transformations simply return another pattern, and so we can compose transformations together to quickly create complex patterns. Because these transforms operate on patterns as functions, and not simply lists, this can be done to arbitrary depth without worrying about storage; no actual events get calculated and manipulated until they are needed. Here is a simple example:

```
whenmod 8 4 (slow 4) $ every 2 ((1/2) <~ ) $
  every 3 (density 4) $ iter 4 "grey ↙
      ↳ darkgrey green black"
```



To visualise some of the repeating structure, the above image shows a ten-by-twenty grid of cycles, scanning across and down.

5. Working with sound

The visual examples only work up to a point, and the multidimensional nature of timbre is difficult to get across with colour alone. Tidal allows many aspects of sound, such formant filters, spatialisation, pitch, onset and offset to be patterned separately, and then composed into patterns of synthesiser control messages. Pattern transforms can then manipulate multiple aspects of sound at once; for example the `jux` transform works similarly to `superimpose`, but the original pattern is panned to the left speaker, and the transformed pattern to the right. The `striate` pattern effectively cuts a sample into multiple 'sound grains', so that those patterns of grains can then be manipulated with further transforms. For details, please refer to the Tidal documentation, and also to the numerous video examples linked to from the homepage `http://yaxu.org/tidal`.

6. Live coding with Tidal

7. The Tidal community

Over the past year, a community of Tidal users has started to grow. This followed a residency in Hangar Barcelona, during which the Tidal installation procedure was improved and documented. This community was surveyed, by invitation via the Tidal on-line forum, encouraged to give honest answers, and fifteen responded. Two demographic questions were asked. Given an optional free text question "What is your gender?", 10 identified as male, and the remainder chose not to answer. Given an optional question "What is your age?", 7 chose "17-25", 4 chose "26-40", and the remainder chose not to answer.

Respondents were asked to estimate the number of hours they had used Tidal. Answers ranged from 2 to 300, with a mean of 44.2 and a standard deviation of 80.8. We can say that all had at least played around with it for over an hour, and that many had invested significant time in learning it; the mode was 8 hours.

A surprising finding was that respondents generally had little or no experience of functional programming languages before trying Tidal. When asked the question "How much experience of functional programming languages (e.g. Haskell, Lisp, etc) did you have when you started with Tidal? ", 6 selected "No experience at all", 6 selected "Some understanding, but no real practical experience" and 3 selected "Had written programs using functional programming techniques". *No* respondents said that they had "In depth, practical knowledge of functional programming".

Despite the general lack of experience with functional languages, respondents generally reported that they could make music with Tidal (14/15), that it was not difficult to learn (11/15), and that it had the potential to help them be more creative (13/15). Furthermore, most said they could learn Tidal just by playing with it (10/15), and that they didn't need theoretical understanding in order to use it for music (8/15). These answers were all captured as Likert responses, see Figure 2. From this we conclude that despite Haskell's reputation for difficulty, these users did not seem to have problems learning a DSL embedded within it, that uses some advanced features.

Perhaps the loudest note of warning that we should ring before leaving this section is that this is very much a self-selecting group, attracted by what may be seen as a niche, technological way to make music.

8. Conclusion

We have given some context to live coding Tidal, described some implementation details and a selection of the functionality it provides. Much of this functionality comes from Haskell itself, which has proved a suitable language for describing pattern. This is borne out from a survey of 15 Tidal users, who generally reported positive learning experiences despite not being experienced functional programmers.

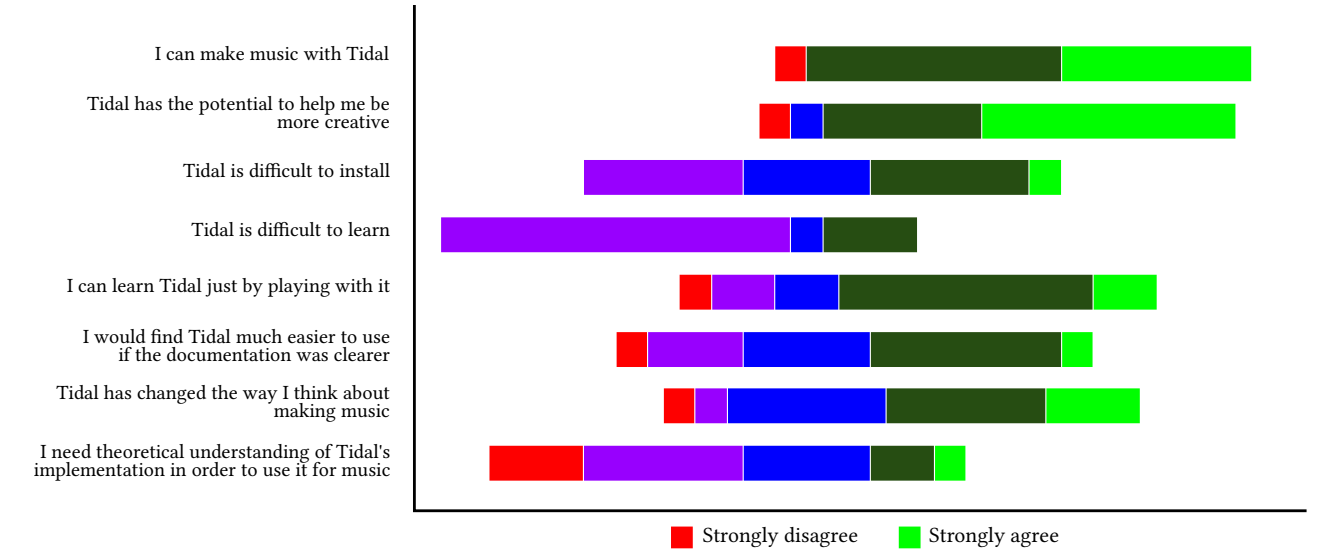


Figure 2. Likert scale questions from survey of Tidal users

References

S. Aaron, A. F. Blackwell, R. Hoadley, and T. Regan. A principled approach to developing new languages for live coding. In *Proceedings of New Interfaces for Musical Expression 2011*, pages 381–386, 2011.

R. Bell. An Interface for Realtime Music Using Interpreted Haskell. In *Proceedings of LAC 2011*, 2011.

A. Blackwell, A. McLean, J. Noble, and J. Rohrhuber. Collaboration and learning through live coding (Dagstuhl Seminar 13382). *Dagstuhl Reports*, 3(9):130–168, 2014. . URL `http://drops.dagstuhl.de/opus/volltexte/2014/4420`.

M. Clayton. *Time in Indian Music: Rhythm, Metre, and Form in North Indian Rag Performance (Oxford Monographs on Music)*. Oxford University Press, USA, Aug. 2008. ISBN 0195339681. URL `http://www.worldcat.org/isbn/0195339681`.

N. Collins and A. McLean. Algorave: A survey of the history, aesthetics and technology of live performance of algorithmic electronic dance music. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2014.

N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003. . URL `http://dx.doi.org/10.1017/s135577180300030x`.

C. Elliott. Push-pull functional reactive programming. In *Proceedings of 2nd ACM SIGPLAN symposium on Haskell 2009*, 2009.

T. Hall. Towards a Slow Code Manifesto. Published online; http://www.ludions.com/slowcode/, Apr. 2007.

P. Hession and A. McLean. Extending Instruments with Live Algorithms in a Percussion / Code Duo. In *Proceedings of the 50th Anniversary Convention of the AISB: Live Algorithms*, 2014.

T. Magnusson. ixi lang: a SuperCollider parasite for live coding. In *Proceedings of International Computer Music Conference 2011*, 2011.

J. McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002. URL `http://www.mitpressjournals.org/doi/abs/10.1162/014892602320991383`.

A. McLean and H. Reeve. Live Notation: Acoustic Resonance? In *Proceedings of International Computer Music Conference*, pages 70–75, 2012.

A. McLean and G. Wiggins. Tidal - Pattern Language for the Live Coding of Music. In *Proceedings of the 7th Sound and Music Computing conference 2010*, pages 331–334, 2010.

A. McLean, D. Griffiths, N. Collins, and G. Wiggins. Visualisation of Live Code. In *Proceedings of Electronic Visualisation and the Arts London 2010*, pages 26–30, 2010.

C. Nash and A. F. Blackwell. Tracking virtuosity and flow in computer music. In *Proceedings of International Computer Music Conference 2011*, 2011.

M. Puckette. The Patcher. In *Proceedings of International Computer Music Conference 1988*, pages 420–429, 1988.

K. Sicchio. Hacking Choreography: Dance and Live Coding. *Computer Music Journal*, 38(1):31–39, Mar. 2014. . URL `http://dx.doi.org/10.1162/comj\_a\_00218`.

A. Sorensen. Impromptu: An interactive programming environment for composition and performance. In *Proceedings of the Australasian Computer Music Conference 2005*, pages 149–153, 2005.

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

H. Thielemann. Live-Musikprogrammierung in Haskell. *CoRR*, abs/1202.4269, 2012.

G. Wang and P. R. Cook. On-the-fly programming: using code as an expressive musical instrument. In *Proceedings of New interfaces for musical expression 2004*, pages 138–143. National University of Singapore, 2004.