morecycles

Quint Guvernator, Debdutta Guha Roy,
Kaustav Kumar Choudhury, Jay Warnock,
Gideon Pol, Milan La Rivière

Sunday 30th March, 2025

Abstract

This paper describes the design and implementation of six new musically relevant pattern manipulation functions for the music live-coding environment *TidalCycles*.

Contents

1 How to use this? 2

1.1 Compiling the report . . . 2

1.2 Compiling and testing the library . . . 2

1.3 Using the functions in TidalCycles compositions . . . 2

2 About this project 3

3 Functions 3

3.1 jumble . . . 3

3.1.1 Tests . . . 4

3.2 gracenotes . . . 5

3.3 jitter . . . 7

3.3.1 Jitter Test Cases . . . 11

3.4 PolyShift . . . 13

3.4.1 Tests . . . 14

3.5 RhythmMask . . . 14

3.5.1 Tests . . . 16

3.6 SwingTime . . . 18

4 Conclusion **20**

Bibliography **20**

1 How to use this?

The source code of this project simultaneously defines a report as a LaTeXdocument; as well as a Haskell library with an included test suite. The latter can itself be used in two ways: as a Haskell library purely for pattern manipulation, or as a set of function definitions for the TidalCycles environment.

1.1 Compiling the report

To generate the PDF: clone the repository, open `report.tex` in your favorite LaTeXeditor, and compile. You can also generate the report in a terminal by running `pdflatex report; bibtex report; pdflatex report; pdflatex report`

If you don't have a LaTeXenvironment installed locally, you can also upload the *entire* repository to a service like Overleaf, set the main file to `report.tex`, and compile it there.

1.2 Compiling and testing the library

You should have stack installed (see `https://haskellstack.org/`) and open a terminal in the same folder.

- To compile everything: `stack build`.
- To open ghci and play with the code: `stack ghci`
- To run the tests: `stack clean && stack test --coverage`

1.3 Using the functions in TidalCycles compositions

Getting the code to work in a live TidalCycles environment can be tricky. Doing it properly involves following the following steps, which are quite general because the particulars will depend on which platform you are using:

- Create a new stack project that pulls in both this library and TidalCycles as dependencies
- Install SuperCollider and SuperDirt locally
- Follow the rest of the installation instructions on the TidalCycles website, but use the local stack project instead of installing the package globally
- Create a `BootTidal.hs` file somewhere which imports the `morecycles` functions you want to use and binds them to the appropriate names
- Configure you reditor to read the `BootTidal.hs` file

However, experience in our group has shown that this can be a time-consuming and error-prone process on some platforms (particularly: the commercial operating systems). If the above steps are not working or take too long, we offer the following as a quick-and-dirty alternative:

- Install TidalCycles, SuperCollider, and SuperDirt exactly as described on the TidalCycles website
- Select the Haskell source of the function(s) you want to use
- Remove all newlines and comments
- Paste these into the editor you configured for TidalCycles in the first step
- Use the keyboard shortcut you typically use for executing a line, to execute this code, adding the name binding to the current interpreter session

2 About this project

TODO

3 Functions

3.1 jumble

In Western music, beats are often arranged hierarchically into a metrical structure. Depending on the rhythm of a pattern, some beats are more important to the pattern's structure than others which may just be present to add variety.

It would be nice if musicians could specify which beats are important, allowing the rest to vary automatically. We call this function `jumble`.

```
module Jumble where

import Sound.Tidal.Pattern
import Sound.Tidal.UI
import Sound.Tidal.Core
import Data.List
```

`jumble` takes two patterns as input: a content pattern (‘Pattern a‘) and a mask pattern (‘Pattern Bool‘). This is similar to the signature of the ‘mask‘ function.

The function produces as output a pattern where any event which overlaps with a 1 in the mask pattern is passed through undisturbed, and the values at the other events are rotated. The smallest "empty space" becomes the granularity at which the unmasked events are chopped.

To implement this non-deterministic function, we first create a deterministic version where the permutation index is given explicitly as a parameter:

```
-- deterministic version of jumble, which takes a permutation index
jumble' :: Int -> Pattern Bool -> Pattern a -> Pattern a
jumble' 0 _ p = p
jumble' i mp p = stack [static, variable] where
  static = mask mp p
  variable = rotateValues $ mask (inv mp) p

  -- rotate a list (head to last) a certain number of times
  rot8 :: Int -> [a] -> [a]
  rot8 _ [] = []
  rot8 n' (x:xs) = if n == 0 then x:xs else rot8 (n-1) (xs ++ [x]) where
    n = n' ‘mod‘ length (x:xs)
```

```
  -- extract values from an event
  getValue :: Event a -> a
  getValue (Event _ _ _ v) = v

  -- rotate the values of an event
  rotateValues = cyclewise f where
    f events = inject (rot8 i values) events where
      values = getValue <$> events

  -- swap out the value in each event with values from a list
  inject :: [a] -> [Event b] -> [Event a]
  inject (v:vs) (e:es) = e{value=v} : inject vs es
  inject _ _ = []

  cyclewise f Pattern{query=oldQuery} = splitQueries Pattern{query=newQuery} where
    newQuery (State (Arc t0 t1) c) = (narrow . f) expandedEvents where
      expandedStart = fromInteger $ floor t0
      expandedEnd = fromInteger $ ceiling t1
      expandedArc = Arc expandedStart $ expandedEnd + (if expandedStart == expandedEnd then
           1 else 0)
      expandedState = State expandedArc c
      expandedEvents = sortOn (\Event{part=Arc{start=t}} -> t) (oldQuery expandedState)
      narrow es = [e{part=Arc (max t0 p0) (min t1 p1)} | e@Event{part=Arc p0 p1} <- es,
          isIn (Arc t0 t1) (wholeStart e)]
```

Using this definition, we can create a non-deterministic version by choosing a permutation index randomly:

```
-- non-deterministic version of jumble, which randomly chooses a new permutation each cycle
jumble :: Pattern Bool -> Pattern a -> Pattern a
jumble _mp _p = Pattern{query=newQuery} where
  newQuery _state = undefined -- choose [jumble' i mp p | i <- [0..length (query p state)
      -1]] -- TODO
```

3.1.1 Tests

```
{-# OPTIONS_GHC -Wno-orphans #-}
module Main where

import Jumble
import TestUtils

import Sound.Tidal.Pattern
import Sound.Tidal.Core
import Sound.Tidal.ParseBP

import Test.Hspec
import Test.QuickCheck
```

First, we need to describe how to create arbitrary `Pattern` instances:

```
instance (Arbitrary a) => Arbitrary (Pattern a) where
  arbitrary = sized m where
    m n | n < 4 = listToPat . (:[]) <$> arbitrary
    m n = fastCat <$> oneof [ sequence [resize (n `div` 2) arbitrary, resize (n `div` 2)
        arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary, resize (n `div` 2) arbitrary] ]

instance (Fractional a, Arbitrary a, Eq a) => Arbitrary (ArcF a) where
  arbitrary = sized m where
    m i = Arc 0 . notZero <$> x where
      x = resize (i `div` 2) arbitrary
      notZero n = if n == 0 then 1 else n
```

We can now define our tests:

```
main :: IO ()
main = hspec $ do
  describe "Jumble" $ do

    it "should change the pattern if nothing is masked" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[0]") (parseBP_E "[a b]")) (
          parseBP_E "[b a]" :: Pattern String)

    it "should change the pattern with a complex mask 1" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[1 0 1 0]") (parseBP_E "[a b c d]"
          )) (parseBP_E "[a d c b]" :: Pattern String)

    it "should change the pattern with a complex mask 2" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[1 0]") (parseBP_E "[a b c d]")) (
          parseBP_E "[a b d c]" :: Pattern String)

    it "should change the pattern with a complex mask 3" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[0 [1 0]]") (parseBP_E "[a b c d]"
          )) (parseBP_E "[b d c a]" :: Pattern String)

    it "should change the pattern with a complex mask 4" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[1 0 1 0]") (parseBP_E "[bd [hh cp
          ] sd cp]")) (parseBP_E "[bd [cp cp] sd hh]" :: Pattern String)

    it "shouldn't change the pattern when the permutation index is zero" $
      property $ \a mp p -> compareP a (jumble' 0 mp p) (p :: Pattern Int)

    it "shouldn't change the pattern when the whole pattern is masked" $
      property $ \a i p -> compareP a (jumble' i (parseBP_E "[1]") p) (p :: Pattern Int)

    it "shouldn't change the pattern when the permutation index loops around" $
      property $ \a -> compareP a (jumble' 2 (listToPat [True, False, True, False]) (
          listToPat [1, 2, 3, 4])) (listToPat [1, 2, 3, 4 :: Int])
```

3.2 gracenotes

In Western music, grace notes are quick, ornamental notes that precede a main note, adding expressiveness and variation. These notes do not affect the rhythm but add embellishment to a melody.

It would be nice if musicians could specify where grace notes should be added, allowing for dynamic and varied performances. We call this function `gracenotes`.

`gracenotes` takes two patterns as input: a content pattern (Pattern a) and a mask pattern (Pattern Bool). The function produces an output pattern where any event that overlaps with a 'True' in the mask pattern is duplicated with an additional grace note occurring just before it.

```
module GraceNotes where

import Sound.Tidal.Pattern
import Sound.Tidal.UI
import Sound.Tidal.Core
import Data.List
```

the function gracenotes takes as input a Time variable, a Pattern Bool, and an original Pattern. The output is the resulting pattern with grace notes added. The grace notes are added to the notes that match the mask pattern. The Double parameter specifies how early the grace note starts, relative to the main note. So a grace note with offset 0.125 starts 1/8th of a cycle before the main note, and ends when the main note starts. The pitch of the grace note is always the same as the next note in the original pattern. This, will wrap around to the next note in the

pattern if the end of the pattern is reached.

```
gracenotes' :: Time -> Pattern Bool -> Pattern a -> Pattern a
gracenotes' offset mp p = stack [original, graceNotes] where
  original = p
  -- Get all events from the original pattern for a given state
  getOriginalEvents state = query p state
  -- Create grace notes for events that match the mask
  graceNotes = Pattern{query=newQuery}
  newQuery state =
    let
      -- Get events that match the mask
      maskedEvents = query (mask mp p) state
      -- Get all events from the original pattern
      allEvents = getOriginalEvents state
      -- If there are no events, return empty list
      result = if null maskedEvents || null allEvents
               then []
               else
                 let
                   -- Sort all events by start time to establish sequence
                   sortedEvents = sortBy (\e1 e2 -> compare (start $ part e1) (start $ part
                       e2)) allEvents
                   -- Create a circular list of values from the original pattern
                   originalValues = cycle $ map value sortedEvents
                   -- For each masked event, find its position in the original sequence
                   -- and get the next value from the original sequence
                   createGraceNotes e =
                     let
                       eTime = start $ part e
                       -- Find the index of this event in the sorted sequence
                       -- (or the closest one if exact match not found)
                       findNextIndex [] _ = 0
                       findNextIndex [_] _ = 0
                       findNextIndex (x:xs) t =
                         if abs (start (part x) - t) < 0.0001
                         then 0  -- Found the event
                         else 1 + findNextIndex xs t
                       eventIndex = findNextIndex sortedEvents eTime
                       -- Get the next value in the original sequence (wrapping if needed)
                       nextValue = originalValues !! (eventIndex + 1)
                       -- Create the grace note
                       t0 = start $ part e
                       --  graceStart = if t0 - offset < 0 then 1 + (t0 - offset) else t0 -
                           offset
                       --  graceEnd = graceStart + offset
                       graceStart = t0 - offset
                       graceEnd = t0
                       gracePart = Arc graceStart graceEnd
                       graceEvent = e {part = gracePart, whole=Just gracePart, value =
                           nextValue}
                     in graceEvent
                 in map createGraceNotes maskedEvents
    in result


-- | A simpler version of gracenotes that automatically generates a random Boolean pattern
-- and uses 0.125 as the grace note duration.
gracenotes :: Pattern a -> Pattern a
gracenotes p = Pattern{query=newQuery} where
  -- Default grace note duration (1/8 of a cycle)
  defaultDuration = 0.125
  -- Use a non-deterministic approach similar to jumble
  newQuery state =
    let randomMask = fastcat [pure True, pure False]
    in query (gracenotes' defaultDuration randomMask p) state
```

To test this functionality manually, you can use the following commands; In the ghci terminal when purely working with patterns:

```
p2e $ gracenotes' 0.125 (s2p "[1 0 1 0]" :: Pattern Bool) (s2p "[a b c d]" :: Pattern
    String)
```

```
p2e $ gracenotes (s2p "[a b c d]" :: Pattern String)
```

When working with tidal, producing sounds:

```
d1 $ gracenotes' 0.125 ("1 0 1 0" :: Pattern Bool) (n "c a f e" # sound "supermandolin")
```

3.3 jitter

In live-coded music, perfect quantization can sometimes sound mechanical and rigid. Human musicians naturally introduce slight variations in timing, creating a sense of groove, swing, or expressiveness. In genres like jazz, funk, and experimental electronic music, these subtle shifts are an essential part of musical feel.

It would be useful if musicians could introduce controlled randomness into their patterns, allowing each event to slightly vary in timing while still maintaining the overall rhythmic structure. This function, which we call `jitter`, enables such organic fluctuations by introducing small, randomized shifts to event start times.

```
module Jitter where

import Sound.Tidal.Pattern
import Sound.Tidal.Context -- Import rand from Sound.Tidal.Context
import System.Random
import System.IO.Unsafe (unsafePerformIO) -- Import unsafePerformIO to extract a random
    value from an IO action.
```

The function `myModifyTime` enables precise timing modifications by applying a transformation function to the start time of each event in a pattern. It works by querying all events within a given time span and then updating each event's timing arc. Specifically, it:

- Retrieves the event's start time (stored as a Rational) from its timing arc.
- Converts the start time to a Double so that the transformation function can operate on it.
- Applies the provided function to compute a new start time.
- Converts the modified time back to Rational and updates the event's arc—while preserving the original stop time.

This mechanism allows for controlled alterations in rhythmic feel, making patterns more flexible and human-like. It serves as a core building block for higher-level functions, such as jitter effects, which introduce randomness or other time-based modifications into a performance.

```
myModifyTime :: Pattern a -> (Double -> Double) -> Pattern a
myModifyTime pat f = Pattern $ \timeSpan ->
  map updateEvent (query pat timeSpan)
  where
    -- updateEvent takes an event e and modifies its 'part' field.
    updateEvent e =
      let eventArc = part e          -- eventArc :: ArcF Time, where Time is Rational
          currentStart = start eventArc -- get the start time (a Rational)
          -- Convert the start time to Double, apply f, and convert back to Rational.
          newStart = toRational (f (realToFrac currentStart))
          -- Build a new arc with the updated start, keeping the original stop.
          newArc = eventArc { start = newStart }
      in e { part = newArc }
```

The function `jitterWith` introduces controlled timing variations by applying an offset function to the start time of each event in a pattern. It works by simultaneously querying two patterns: the input pattern and a continuously generated random pattern, using the built-in `rand` function. For each event, it takes a random value from the `rand` pattern, applies the provided offset function to that value, and then adds the resulting offset to the event's start time. This process creates systematic deviations from strict timing, enabling effects like swing, groove, or subtle rhythmic fluctuations—all while preserving the overall structure of the pattern.

Example: `d1 $ jitterWith (*0.05) (sound "bd sn cp hh")` In this example, each event's start time is shifted by an amount that is 0.05 times a random value, adding a controlled, random variation to the rhythm.

```
jitterWith :: (Double -> Double) -> Pattern a -> Pattern a
jitterWith offsetFunc pat = Pattern $ \timeSpan ->
  let events = query pat timeSpan
      randomOffsets = query (rand :: Pattern Double) timeSpan
      updateEvent (e, r) =
        let eventArc = part e
            currentStart = start eventArc
            timeOffset = offsetFunc r -- Apply the random value to the offset function
            newStart = toRational (realToFrac currentStart + timeOffset)
            newArc = eventArc { start = newStart }
        in e { part = newArc }
  in zipWith (curry updateEvent) events (map value randomOffsets)
```

The function `jitter` introduces natural-sounding randomness by applying a small, unpredictable time shift to the start time of each event in a pattern. It creates a more dynamic, human-like feel in rhythmic sequences by varying event timing within a controlled range. When the provided maximum jitter is zero, the function simply returns the original pattern unaltered. Otherwise, it leverages `jitterWith` along with a helper function (`randomOffset`) that generates a random value between $-$`maxJitter` and `maxJitter`. This random offset is added to each event's start time, ensuring that every execution produces slightly different timing variations.

```
jitter :: Pattern a -> Double -> Pattern a
jitter pat maxJitter
  | maxJitter == 0 = pat -- No jitter when max jitter is 0
  | otherwise = jitterWith randomOffset pat
  where
    -- Generate a random offset between -maxJitter and maxJitter
    randomOffset :: Double -> Double
    randomOffset _ = unsafePerformIO (randomRIO (-maxJitter, maxJitter))
```

The function `jitterP` introduces random timing variations to a pattern, but with a twist: the maximum jitter applied to each event is determined dynamically by a separate pattern (`maxJitterPat`). For each event in the input pattern, `jitterP` identifies the corresponding event in `maxJitterPat`—based on overlapping time cycles—and uses its value as the upper bound for a random offset. A random value is then generated uniformly in the range $[-m, m]$, where m is the extracted jitter value, and this offset is added to the event's start time. As a result, the rhythm is varied in a dynamic and expressive way, enabling more nuanced and human-like timing fluctuations while preserving the overall structure of the pattern.

```
jitterP :: Pattern a -> Pattern Double -> Pattern a
jitterP pat maxJitterPat = Pattern $ \timespan ->
    let contentEvents = query pat timespan
        maxEvents     = query maxJitterPat timespan
        getMaxForEvent e =
            let t = start (part e)  -- current event start time (Rational)
                matching = filter (\e' ->
                        let p = part e'
```

```
                        pStart = start p
                        pStop  = stop p
                    in t >= pStart && t < pStop)
                maxEvents
        in case matching of
            (m:_) -> value m
            []    -> 0
    updateEvent e =
        let currentStart = start (part e)
            m   = getMaxForEvent e
            timeoffset       = unsafePerformIO (randomRIO (-m, m))
            newStart     = toRational (realToFrac currentStart + timeoffset)
            newArc       = (part e) { start = newStart }
        in e { part = newArc }
in map updateEvent contentEvents
```

Implementation

To test and see how `jitter` and `jitterP` work, one can follow these steps:

- **Load the Module:** In your TidalCycles session, ensure that your module is in the search path and load it by running: `:set -i"/morecycles/lib"`
  `:m + Jitter`
- **Test the Fixed Jitter Function:** Apply the `jitter` function to a pattern with a fixed maximum jitter value. For example: `d1 $ jitter (sound "bd sn cp hh") 0.02`
  This command will randomly shift the start time of each event by up to ± 0.02 cycles.
- **Test the Variable Jitter Function:** Apply the `jitterP` function to a pattern using a dynamic maximum jitter value derived from a pattern. For example: `d1 $ jitterP (sound "bd sn cp hh") (range 0.01 0.05 sine)`
  Here, the maximum jitter value varies between 0.01 and 0.05 cycles following a sine wave, so the random shift applied to each event is determined by the corresponding value in this pattern.
- **Observing the Effects:** Listen carefully to the output of each command. The first test should produce a consistent, fixed range of timing variations, while the second test will yield dynamically changing variations in timing, resulting in a more expressive and humanized rhythmic feel.

Deterministic vs. Non-Deterministic Functions

A function is said to be **deterministic** if it always produces the same output given the same input. In other words, its behavior is completely predictable. For example, a function that adds 0.1 to a value, such as

$$f(x) = x + 0.1,$$

is deterministic because $f(1) = 1.1$ every time it is called.

A **non-deterministic** function, on the other hand, may produce different outputs for the same input. This typically happens when the function involves randomness, external state, or time-dependent behavior. For example, a function that returns a random number between -0.02 and 0.02 will produce different results on each call, even when provided with the same input.

In our implementation, the behavior of the functions falls into two categories:

- **Deterministic Functions:** A deterministic function always produces the same output given the same input and transformation. In our code, the function `myModifyTime` is deterministic. It applies a user-supplied transformation function to the start time of every event in a pattern. For example, if you apply

  ```
  myModifyTime pat (+0.1)
  ```

  every event's start time will be shifted exactly by 0.1 cycles every time. There is no external randomness, so the output pattern is predictable.

- **Non-Deterministic Functions:** Non-deterministic functions, on the other hand, can produce different outputs on each execution even with the same input. In our code, the functions `jitterWith`, `jitter`, and `jitterP` are non-deterministic because they incorporate randomness in their calculations.

  - `jitterWith` is non-deterministic when its offset function involves randomness. In our implementation, we use a function that retrieves random values (from the built-in `rand` pattern or via `unsafePerformIO (randomRIO ...)`), so each time the function is applied, different random offsets may be added to the event start times.
  - `jitter` uses `jitterWith` with a helper function that generates a random offset in the range $[-\texttt{maxJitter}, \texttt{maxJitter}]$ via `randomRIO`. For example, executing

    ```
    d1 $ jitter (sound "bd sn cp hh") 0.02
    ```

    will randomly shift each event's start time by a value between -0.02 and +0.02 cycles. Even if you run the same command repeatedly, the exact timing of the events will vary because of the random values.
  - `jitterP` further extends the concept by determining the maximum jitter for each event from a provided pattern (e.g., `(range 0.01 0.05 sine)`). Although the maximum allowed jitter for each event is derived from a deterministic pattern, the actual offset applied is still chosen randomly within the calculated bounds. Thus, executing

    ```
    d1 $ jitterP (sound "bd sn cp hh") (range 0.01 0.05 sine)
    ```

    yields event timings that change from cycle to cycle in a non-predictable manner.

  Summary Examples:

  1. With `myModifyTime`, if the transformation is `(+0.1)`, every event is delayed by exactly 0.1 cycles:

     `myModifyTime pat (+0.1)` $\Rightarrow$ Output is the same every time.

  2. With `jitter`, using a fixed maximum jitter:

     ```
     d1 $ jitter (sound "bd sn cp hh") 0.02
     ```

     Each execution produces different offsets for the events (each between -0.02 and +0.02 cycles), so the rhythm is varied each time.

3. With `jitterP`, the jitter range for each event comes from a pattern. For example:

```
d1 $ jitterP (sound "bd sn cp hh") (range 0.01 0.05 sine)
```

   Here, the maximum jitter value changes according to a sine wave, and the random offset applied is within the current bound. Although the `range` pattern is deterministic, the resulting event shifts are non-deterministic due to the randomness in offset selection.

In conclusion, while `myModifyTime` deterministically applies a specified transformation to event timings, the higher-level functions `jitterWith`, `jitter`, and `jitterP` incorporate randomness to produce non-deterministic, dynamic variations in timing. This non-determinism is essential for creating a more humanized, less mechanical rhythmic feel.

3.3.1 Jitter Test Cases

To verify the functionality of the `jitter` function, we will create test cases that ensure the jitter function doesnt completely disrupt the rhythm and that the overall structure of the pattern remains intact.

```
module Main where
import Jitter (jitter, jitterWith, jitterP)
import Sound.Tidal.Pattern
import Sound.Tidal.Core

import Test.Hspec -- Import Hspec for writing our test suite.

-- | A simple test pattern created using 'fromList'.
-- This pattern has three events with values "a", "b", and "c".
testPattern :: Pattern String
testPattern = fromList ["a", "b", "c"]
```

This test verifies that jitterWith, when given a constant offset function (e.g., always adding 0.1 cycles), properly increases each event's start time by 0.1.

```
debugEvent :: Show a => EventF (ArcF Time) a -> String
debugEvent e =
  "Event { start = " ++ show (start (part e)) ++
  ", stop = " ++ show (stop (part e)) ++
  ", value = " ++ show (value e) ++ " }"

debugEvents :: Show a => [EventF (ArcF Time) a] -> String
debugEvents events = "[" ++ unlines (map debugEvent events) ++ "]"

testJitterWith :: Spec
testJitterWith = describe "jitterWith" $ do
  it "shifts each event's start time within the expected range" $ do
    let maxJitter = 0.1
        jitteredPattern = jitterWith (const maxJitter) testPattern
        state = State (Arc 0 3) 0
        originalEvents = query testPattern state
        jitteredEvents = query jitteredPattern state
        differences = zipWith (\e1 e2 -> abs ((start . part) e2 - (start . part) e1))
                              originalEvents jitteredEvents
        maxJitterRational = toRational maxJitter
    differences `shouldSatisfy` all (<= maxJitterRational)
```

This test checks that applying the jitter function with a maximum jitter of 0 results in an unchanged pattern (i.e. no timing shifts occur).

```
testJitterZero :: Spec
testJitterZero = describe "jitter (fixed max jitter)" $ do
  it "leaves event timings unchanged when max jitter is 0" $ do
    let jitteredPattern = jitter testPattern 0
        state = State (Arc 0 3) 0
        originalEvents = query testPattern state
        jitteredEvents = query jitteredPattern state
        originalStarts = map (start . part) originalEvents
        jitteredStarts = map (start . part) jitteredEvents
        originalVals   = map value originalEvents
        jitteredVals   = map value jitteredEvents
    jitteredStarts `shouldBe` originalStarts
    jitteredVals   `shouldBe` originalVals
```

Tests for jitterP:

- When the maximum jitter pattern is pure 0, the output should equal the input.
- When a constant max jitter is provided (e.g., 0.1), each event's timing shift should be within the bound [-0.1, 0.1].

```
testJitterPNoJitter :: Spec
testJitterPNoJitter = describe "jitterP (zero max jitter pattern)" $ do
  it "does not change event timings when max jitter pattern is 0" $ do
    let maxJitterPat = pure 0 :: Pattern Double
        jitteredPattern = jitterP testPattern maxJitterPat
        state = State (Arc 0 3) 0
        originalEvents = query testPattern state
        jitteredEvents = query jitteredPattern state
        originalStarts = map (start . part) originalEvents
        jitteredStarts = map (start . part) jitteredEvents
    jitteredStarts `shouldBe` originalStarts

testJitterPWithinBounds :: Spec
testJitterPWithinBounds = describe "jitterP (constant max jitter pattern)" $ do
  it "ensures each event's timing shift is within the specified bound" $ do
    let maxJitter = 0.1
        maxJitterPat = pure maxJitter :: Pattern Double
        jitteredPattern = jitterP testPattern maxJitterPat
        state = State (Arc 0 3) 0
        originalEvents = query testPattern state
        jitteredEvents = query jitteredPattern state
        differences = zipWith (\e1 e2 -> abs ((start . part) e2 - (start . part) e1))
                              originalEvents jitteredEvents
        maxJitterRational = toRational maxJitter
    differences `shouldSatisfy` all (<= maxJitterRational)
```

The main function runs the test suite.

```
main :: IO ()
main = hspec $ do
  testJitterWith
  testJitterZero
  testJitterPNoJitter
  testJitterPWithinBounds
```

The following test cases have been designed to verify that the jitter functions behave as expected. Below is a summary of each test case, its relevance, and why it was chosen.

- **testJitterWith:** This test verifies that `jitterWith`, when provided with a constant offset function, correctly shifts the start time of every event by a fixed amount.
  - **What it does:** It applies `jitterWith` with an offset function that always returns the same value (0.1). The test then compares the start times of the original events with those of the modified events, expecting each to be increased by exactly 0.1 cycles.

- **Relevance:** This ensures that the underlying mechanism for modifying event timing via `myModifyTime` works correctly in a controlled, deterministic scenario.
- **Example:** If an event originally starts at 0.5 cycles, after applying `jitterWith (_ -> 0.1)` it should start at 0.6 cycles.

- **testJitterZero:** This test checks that applying the `jitter` function with a maximum jitter of 0 leaves the pattern completely unchanged.
  - **What it does:** It runs `jitter testPattern 0` and verifies that both the start times and the event values remain identical to the original pattern.
  - **Relevance:** This test confirms that the function correctly handles a boundary condition (i.e., no jitter should be applied when the maximum jitter is 0).
  - **Example:** Running `jitter pat 0` should yield the exact same event timings as `pat`.

- **testJitterPNoJitter:** This test verifies that when the maximum jitter pattern is constant at 0 (using `pure 0`), `jitterP` produces an output that is identical to the input pattern.
  - **What it does:** It creates a max jitter pattern that always returns 0 and applies `jitterP`. The test then compares the start times of events in the original and jittered patterns.
  - **Relevance:** This ensures that `jitterP` behaves deterministically when the maximum allowed jitter is 0.
  - **Example:** Every event's timing remains unchanged because no jitter is allowed.

- **testJitterPWithinBounds:** This test ensures that when a constant maximum jitter is provided via a pattern (e.g., `pure 0.1`), the shift applied to each event is always within the expected bound of $[-0.1, 0.1]$ cycles.
  - **What it does:** It applies `jitterP` with a max jitter pattern that is constant at 0.1 and then computes the absolute difference between the original and modified event start times. The test asserts that all these differences do not exceed 0.1 (converted to a Rational).
  - **Relevance:** This test confirms that the random offset generated for each event never exceeds the specified maximum jitter bound, ensuring controlled randomness.
  - **Example:** If an event's original start time is 1.0 and the max jitter is 0.1, the jittered start time should be between 0.9 and 1.1 cycles.

The main function aggregates all test cases and runs them using Hspec. This ensures that any change in the jitter functions can be quickly validated against our expectations for both deterministic and non-deterministic behavior.

3.4 PolyShift

```
module PolyShift where

-- plays a pattern only on the kth cycle

playOnly n k p = every' n k (<> p) "~"
```

```
{-
The goal is that the below functions will begin with the pattern playing normally during
    the first cycle. Then it
should delay the start time the next cycle, starting the delay over at the beginning of the
     cycle if it exceeds one.
This will terminate when it reaches exactly one, i.e., when in n steps, where n is the
    denominator of the delay time,
since time is represented as fraction of cycles.
-}

helper 0 _ _ _ _ _ = "~"
helper j k l m n p
    | l > 1  = playOnly k m $ (~> (l-1)) p <> helper (j-1) k (l+n) m n p
    | l <= 1 = playOnly k (m+1) $ (~> (l-1)) p <> helper (j-1) k ((l-1)+n) (m+1) n p

polyShift n p = helper (denominator n) (denominator n) 0 0 n p
```

3.4.1 Tests

```
{-# OPTIONS_GHC -Wno-orphans #-}
module Main where

import Jumble
import TestUtils

import Sound.Tidal.Pattern
import Sound.Tidal.Core
import Sound.Tidal.ParseBP

import Test.Hspec
import Test.QuickCheck
```

First, we need to describe how to create arbitrary `Pattern` instances:

```
instance (Arbitrary a) => Arbitrary (Pattern a) where
  arbitrary = sized m where
    m n | n < 4 = listToPat . (:[]) <$> arbitrary
    m n = fastCat <$> oneof [ sequence [resize (n `div` 2) arbitrary, resize (n `div` 2)
        arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary, resize (n `div` 2) arbitrary] ]

instance (Fractional a, Arbitrary a, Eq a) => Arbitrary (ArcF a) where
  arbitrary = sized m where
    m i = Arc 0 . notZero <$> x where
      x = resize (i `div` 2) arbitrary
      notZero n = if n == 0 then 1 else n
```

We can now define our tests:

```

main :: IO ()
main = hspec $ do
  describe "Jumble" $ do

    it "" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[0]") (parseBP_E "[a b]")) (
          parseBP_E "[b a]" :: Pattern String)
```

3.5 RhythmMask

```
{-# OPTIONS_GHC -Wno-name-shadowing #-}
{-# OPTIONS_GHC -Wno-type-defaults #-}
```

"RhythMask": Probability-Based Masking The idea behind RhythMask is to create a rhythmic effect where some beats are probabilistically dropped or kept each cycle, rather than being strictly fixed by a binary mask.

Theoretical implementation idea Input: -> A content pattern (Pattern a): The original rhythmic sequence. -> A probability pattern (Pattern Double): A probability value (between 0.0 and 1.0) that determines the likelihood of each beat being played.

Output: A modified version of the input pattern with beats dropped probabilistically

Steps of implementation: ->Extract Events: The function first queries the input pattern to get a list of its events. ->Apply Probability Filtering: Each event is evaluated against the corresponding probability value. ->Random Decision Making: A random number is generated for each event, and if it is below the probability threshold, the event passes on to the final output, otherwise it is removed. ->Reconstruction of the Pattern: The remaining beats are reconstructed into a new TidalCycles pattern.

Additional Functionality: -> Instead of passing a set of probability values, I am passing a binary mask (e.g. [1 0 1 0]), where 1 is True and 0 is False. This determines which beats will be kept in a sequence and which ones will be dropped. However, there is a helper function at the beginning, that generates a random mask.

```
module RhythMask (
  probMaskPattern,
  rhythmaskProb,
  rhythmaskProbWith,
  randomMaskString,
  parseMask,
  myFilterEvents,
  rhythmask,
  rhythmaskWith
) where

import Sound.Tidal.Context

-- | Probabilistic mask: generates a Pattern Bool from a list of Doubles (0.0 to 1.0)
probMaskPattern :: [Double] -> Pattern Bool
probMaskPattern probs =
  fastcat $ map (\p -> fmap (< p) (rand :: Pattern Double)) probs

-- | Applies a probabilistic mask to a pattern
rhythmaskProb :: Pattern a -> [Double] -> Pattern a
rhythmaskProb pat probs = myFilterEvents pat (probMaskPattern probs)

-- | Applies a transformation to masked-out events using a probabilistic mask
rhythmaskProbWith :: Pattern a -> [Double] -> (Pattern a -> Pattern a) -> Pattern a
rhythmaskProbWith pat probs transform =
  let maskP    = probMaskPattern probs
      notMaskP = fmap not maskP
      kept          = myFilterEvents pat maskP
      transformed   = myFilterEvents (transform pat) notMaskP
  in stack [kept, transformed]

-- | generates a random set of True/False (0/1) values to drop or keep beats.
randomMaskString :: Int -> String
randomMaskString n = unwords $ take n $ map (\x -> if even (x * 37 `mod` 7) then "1" else "
    0") [1..]

{-|
  parseMask converts a binary mask string (e.g. "1 0 1 0") into a Pattern Bool.
  It splits the string into words, maps "1" to True and any other word to False,
```

```
  cycles the resulting list, and then limits it to one cycle using take.
-}
parseMask :: String -> Pattern Bool
parseMask s =
  let bits = map (== "1") (words s)
  in fastcat (map pure bits)
{-|
  myFilterEvents takes a pattern 'pat' and a boolean mask pattern 'maskPat'
  and produces a new pattern that only keeps events where the mask is True.
  (It extracts the events from both patterns at a given time, zips them, and
  keeps an event if its corresponding mask event (extracted via value) is True.)
-}
myFilterEvents :: Pattern a -> Pattern Bool -> Pattern a
myFilterEvents pat maskPat = Pattern $ \s ->
  let es = query pat s
      bs = query maskPat s
      filtered = [ e | (e, b) <- zip es bs, value b ]
  in filtered

{-|
  rhythmask applies a mask (given as a string like "1 0 1 0") to a pattern.
  Only events corresponding to a True (or "1") in the mask will be kept.
-}
rhythmask :: Pattern a -> String -> Pattern a
rhythmask pat maskStr = myFilterEvents pat (parseMask maskStr)

{-|
  rhythmaskWith applies a transformation to the events that are masked out.
  It splits the pattern into two parts:
    1. 'kept' events where the mask is True,
    2. 'transformed' events (obtained by applying the given transformation)
       where the mask is False.
  These two layers are then combined using 'stack'.
-}
rhythmaskWith :: Pattern a -> String -> (Pattern a -> Pattern a) -> Pattern a
rhythmaskWith pat maskStr transform =
  let maskP    = parseMask maskStr
      notMaskP = fmap not maskP
      kept         = myFilterEvents pat maskP
      transformed  = myFilterEvents (transform pat) notMaskP
  in stack [kept, transformed]
```

3.5.1 Tests

This is the test suite for the `Rhythmask` module. It uses `hspec` to test the following functions:

- **testParseMask**: Verifies that a mask string (e.g. `"1 0 1 0"`) is parsed into exactly one cycle of boolean values.
- **testMyFilterEvents**: Ensures that filtering a pattern with a boolean mask returns the expected events.
- **testRhythmask**: Tests that applying a mask string retains only the events marked as `"1"`.
- **testRhythmaskWith**: Checks that the function applies a transformation to masked-out events.
- **testProbMaskPattern**: Verifies that a probabilistic mask produces one cycle of the expected boolean values when given all zeros or all ones.
- **testRhythmaskProb**: Ensures that a probabilistic mask with zero probability filters out all events.

- **testRhythmaskProbWith**: Checks that the transformation is applied correctly for events with probability 0, with kept events output first.
- **testRandomMaskString**: Confirms that the generated mask string contains the specified number of bits.

```
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Test.Hspec
import RhythMask
import Sound.Tidal.Context

-------------------------------------------------------------------
-- Helpers

-- | Create a State covering one cycle for n events.
stateFor :: Int -> State
stateFor n = State (Arc 0 (fromIntegral n)) 0

-- | A sample test pattern of 4 events.
testPattern :: Pattern String
testPattern = fromList ["a", "b", "c", "d"]

-------------------------------------------------------------------
-- Test: parseMask
testParseMask :: Spec
testParseMask = describe "parseMask" $ do
  it "parses '1 0 1 0' into [True, False, True, False] (one cycle)" $ do
    let maskPat = parseMask "1 0 1 0"
        -- Using take to capture only one cycle.
        events  = take 4 (query maskPat (stateFor 4))
        values  = map value events
    values `shouldBe` [True, False, True, False]

-------------------------------------------------------------------
-- Test: myFilterEvents
testMyFilterEvents :: Spec
testMyFilterEvents = describe "myFilterEvents" $ do
  it "filters a pattern using a boolean mask" $ do
    let maskPat  = parseMask "1 0 1 0"  -- keep events (positions) 1 and 3
        filtered = myFilterEvents testPattern maskPat
        events   = take 4 (query filtered (stateFor 4))
        values   = map value events
    values `shouldBe` ["a", "c"]

-------------------------------------------------------------------
-- Test: rhythmask
testRhythmask :: Spec
testRhythmask = describe "rhythmask" $ do
  it "applies a mask string to keep only events with mask '1' (1 - True, 0 - False)" $ do
    let result = rhythmask testPattern "0 1 0 1"  -- keep events (positions) 2 and 4
        events = take 4 (query result (stateFor 4))
        values = map value events
    values `shouldBe` ["b", "d"]

-------------------------------------------------------------------
-- Test: rhythmaskWith
testRhythmaskWith :: Spec
testRhythmaskWith = describe "rhythmaskWith" $ do
  it "applies a transformation to masked-out events and outputs kept events first" $ do
    -- With mask "1 0 1 0":
    --   Kept events: events (positions) 1 and 3 ("a" and "c")
    --   Transformed events: events (positions) 2 and 4 ("b!" and "d!")
    let transform = fmap (\x -> x ++ "!")
        result    = rhythmaskWith testPattern "1 0 1 0" transform
        events    = take 4 (query result (stateFor 4))
        values    = map value events
    values `shouldBe` ["a", "c", "b!", "d!"]

-------------------------------------------------------------------
-- Test: probMaskPattern
```

```
testProbMaskPattern :: Spec
testProbMaskPattern = describe "probMaskPattern" $ do
  it "produces all False when probability is 0" $ do
    let pat    = probMaskPattern [0.0, 0.0, 0.0]
        events = take 3 (query pat (stateFor 3))
        values = map value events
    values `shouldBe` [False, False, False]
  it "produces all True when probability is 1" $ do
    let pat    = probMaskPattern [1.0, 1.0, 1.0]
        events = take 3 (query pat (stateFor 3))
        values = map value events
    values `shouldBe` [True, True, True]

-------------------------------------------------------------------
-- Test: rhythmaskProb
testRhythmaskProb :: Spec
testRhythmaskProb = describe "rhythmaskProb" $ do
  it "filters out all events when probability is 0" $ do
    let pat    = fromList ["a", "b", "c"]
        result = rhythmaskProb pat [0.0, 0.0, 0.0]
        events = take 3 (query result (stateFor 3))
        values = map value events
    values `shouldBe` []

-------------------------------------------------------------------
-- Test: rhythmaskProbWith
testRhythmaskProbWith :: Spec
testRhythmaskProbWith = describe "rhythmaskProbWith" $ do
  it "applies transformation to events with probability 0 and outputs kept events first" $
      do
    -- For probabilities [1.0, 0.0, 1.0]:
    --   Kept events: events (positions) 1 and 3 ("a" and "c")
    --   Transformed event: event (position) 2 ("b!")
    let pat       = fromList ["a", "b", "c"]
        transform = fmap (\x -> x ++ "!")
        result    = rhythmaskProbWith pat [1.0, 0.0, 1.0] transform
        events    = take 3 (query result (stateFor 3))
        values    = map value events
    values `shouldBe` ["a", "c", "b!"]

-------------------------------------------------------------------
-- Test: randomMaskString
testRandomMaskString :: Spec
testRandomMaskString = describe "randomMaskString" $ do
  it "generates a mask string with the specified number of bits" $ do
    let n    = 5
        s    = randomMaskString n
        bits = words s
    length bits `shouldBe` n
    all (\b -> b == "0" || b == "1") bits `shouldBe` True

-------------------------------------------------------------------
-- Main: Run all tests
main :: IO ()
main = hspec $ do
  testParseMask
  testMyFilterEvents
  testRhythmask
  testRhythmaskWith
  testProbMaskPattern
  testRhythmaskProb
  testRhythmaskProbWith
  testRandomMaskString
```

3.6 SwingTime

In Western music, particularly in jazz and blues, swing refers to a rhythmic feel where alternate beats are slightly delayed, creating a "long-short" pattern instead of an evenly spaced beat

structure. This gives the rhythm a more dynamic, groovy feel.

The `swing` function allows musicians to apply this effect by defining which beats should be shifted and by how much. This is controlled using a mask pattern, which determines which beats remain steady and which are swung.

Deterministic Swing Uses a fixed swing amount, consistently altering the rhythm in the same way each time. This is useful when a specific groove is needed for synchronization.

Non-Deterministic Swing Randomly selects a swing amount for each cycle, introducing variations in the groove. Can create a more organic, expressive feel. Whereas the goal of `jitter` is to make the music more human-like, the aim of non-deterministic swing is to create a dynamic groove that retains musical cohesion while introducing subtle rhythmic variation

```
module SwingTime where

import Sound.Tidal.Pattern

-- deterministic swing function with a given swing amount
swing' :: Rational -> Pattern Bool -> Pattern a -> Pattern a
swing' amt mp p = p {query = \st -> concatMap (applySwing mp st) (query p st)}
  where
    applySwing :: Pattern Bool -> State -> Event a -> [Event a]
    applySwing maskPattern st ev =
      case whole ev of
        Nothing -> [ev]  -- If there's no whole, return the event unchanged
        Just a ->
          let startTime = start a
              endTime = stop a
              swingAtSt = query maskPattern (st {arc = Arc startTime endTime})
              shouldSwing = not (null swingAtSt) && any (isTrue . value) swingAtSt
              isTrue True = True
              isTrue _ = False
              swingShift = if shouldSwing then amt else 0
              newArc = Arc (startTime + swingShift) (endTime + swingShift)
          in [ev {part = newArc, whole = Just newArc}]

-- non-deterministic swing function that randomly selects a swing amount for each cycle
-- it also randomizes the mask pattern
randomswing :: Pattern a -> Pattern a
randomswing p = p {query = \st -> concatMap (applyRandomSwing st) (query p st)}
  where
    applyRandomSwing :: State -> Event a -> [Event a]
    applyRandomSwing st ev =
      case whole ev of
        Nothing -> [ev]  -- If there's no whole, return the event unchanged
        Just a ->
          let startTime = start a
              endTime = stop a
              -- Determine if this is an off-beat based on position in cycle
              cyclePos = startTime - (fromIntegral $ floor startTime)
              -- Simple pseudo-random function based on cycle position
              isOffBeat = (floor (cyclePos * 4) `mod` 2) == 1
              -- Generate a pseudo-random swing amount based on cycle number
              cycleNum = floor startTime
              swingAmount = 0.05 + (fromIntegral (cycleNum `mod` 11) / 100)
              -- Apply swing if it's an off-beat
              swingShift = if isOffBeat then swingAmount else 0
              newArc = Arc (startTime + swingShift) (endTime + swingShift)
          in [ev {whole = Just newArc}]
```

Test in ghci:

```
p2e $ swing' 0.125 (s2p "[1 0 1 0]" :: Pattern Bool) (s2p "[a b c d]" :: Pattern String)
p2e $ randomswing (s2p "[a b c d]" :: Pattern String)
```

Test in tidal:

```
d1 $ swing' 0.125 ("1 0 1 0" :: Pattern Bool) (n "c a f e" # sound "supermandolin")
```

4 Conclusion

TODO