# Assignment 3:
# An adaptive drum sequencer

B131787

December 2022

## Contents

# 1 Introduction

This project has involved making a midi sequencer in which different active sequences are toggled on/off by varying instrument audio input. During the development of the project, electric guitar input is used and midi outputs are used to trigger drum samples – but it should easily generalize to other set-ups, e.g. using piano input to toggle different synth drone notes on and off.

As well as toggling existing provided sequences on / off, rhythmic events detected in the guitar input (i.e. suddenly loud input) can trigger midi outputs alongside the current active sequences – adding an extra degree of 'playability' to the current drum beat output.

The tempo of the sequencer can also be gradually updated based on the detected rhythmic events – i.e. the user can play a bit faster, and the extent of which events are detected earlier than the nearest beat subdivision nudges the tempo faster.

The aim of the project was to create a sort of 'auto-drummer' which adapts it's drumming in real-time over varying guitar playing. The hope is that it's fun to jam along with and can aid the creative process. In my own experience, programming drums can sometimes be a slow process that restricts what I can compose in a short period of feeling inspired / creative – so it would be great to be able to just load up a plugin I can just instantly play along with.

# 2 Important Notes

## 2.1 Parameter Settings

I wasn't able to integrate all the plugin settings / parameters as a `AudioProcessorValueTreeState` (APVTS), which was essentially because I had to make a custom user interface for editing the drum patterns. For simplicity (and time constraints) I just implemented a sequence editor as a text box (a pattern of ones and zeros define the rhythm), but this doesn't trivially integrate with `AudioParameterFloat` values in an APVTS.

To illustrate a working APVTS, I've created a Reaper preset library with a few presets which affect some of the parameters, specifically the top sliders in the UI, down to (and including) the tempo. These include presets for some of the parameters used for the 2 recordings.

However, the key functionality from this plugin comes from editing the sequence rhythms in the UI. I've included 2 screenshots of the plugin UI in the state that I used it to create the 2 recordings (see the appendix figures 4, and 5).

## 2.2 Recordings

Two recordings are included in the submission: `AP3recording1_B131787.mp3` and `AP3recording2_B131787.mp3`. In my opinion, the second recording is far superior in showcasing the strengths of the plugin, since in hindsight I think I used inferior event-detection parameters in the first recording.

The first has the midi outputs directly sent to a drum sample VST instrument, and the guitar input sound is fed through a standard chain of amplifier / cabinet simulators, with some slight delay and reverb.

The second recording sends the midi outputs through a default Reaper 'midi humanizer' plugin which varies the timings and velocities a bit to sound more like a real drummer. I expect this is a standard way people would use this plugin. A heavier / more distorted guitar sound is used for this recording, with no delay or reverb.

In case it isn't clear, it's worth stating that both recordings are just some live guitar playing (essentially improvised), and recorded in one take after playing around with the parameters.

## 2.3 Future Development

It's safe to say this may have been a slightly over-ambitious project for the given time constraints. As the project progressed, more and more functionality was quickly added, so it's worth noting that some refactoring of the code may be useful before any future development.

It's definitely worth bearing in mind that this project is still a work in progress. The UI certainly needs improvements to make this plugin properly usable.

# 3 Class Structure

## 3.1 Overview

A range of custom classes were developed for this project: `StateHandler`, `Sequence`, `EventDetector`, `TransitionRule` and a range of child classes which inherit from `TransitionRule`. Adding to this some custom UI component classes were created: `SequenceUIBlock`, `TempoUIBlock` and `EventDetectorUIBlock` – see User Interface Design.

## 3.2 `StateHandler`

At the top level, an instance of the `StateHandler` class manages all the sequencer information. The class contains:

- a vector of `Sequence` objects

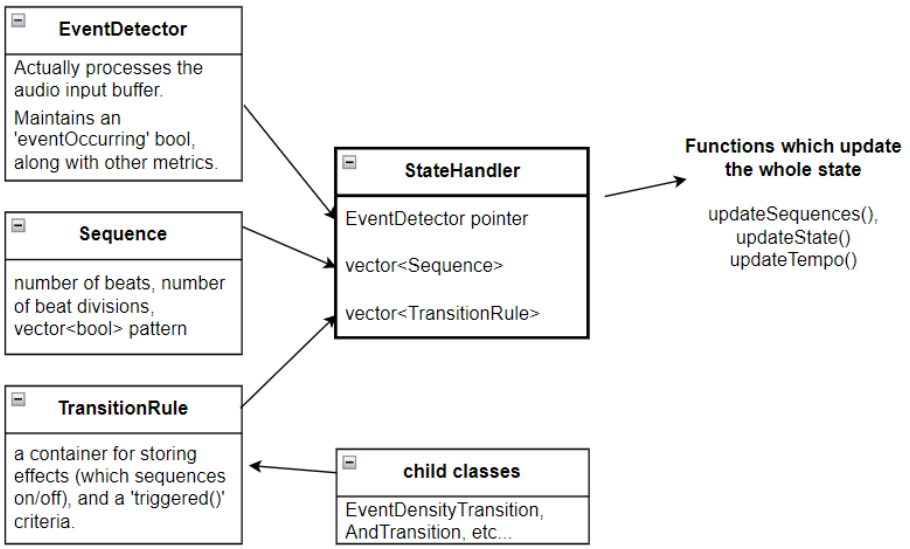


Figure 1: A broad illustration of some key interactions. A `StateHandler` maintains a list of sequences and transition rules, and at every update it updates a global 'beat position' for the sequences, and checks and applies any effects from the list of transition rules. Each `Sequence` object maps a global beat position to a 'is currently outputting' boolean – so this can be easily checked in the main processBlock function, where midi events are added appropriately.

- a vector of `State` values (`State` is an enumeration), representing the current of the whole system (i.e. the `on`, `off`, `turningOn` or `turningOff` state of each sequence).
- a vector of all the `TransitionRule` objects, which determine whether to apply effects (i.e. enumeration `Effect`: `turnOn` or `turnOff`) to different sequences.
- a reference to an `EventDetector` object
- vectors detailing which midi values are outputted by rhythmic events from the EventDetector.
- sample rate, tempo, current beat position (as a float), as well as values for the adapting of tempo.

One adds any number of sequences to the `StateHandler`, and can initialize the state vector to any combination of `on`/`off`. Any number of transition rules can also be initialized and then added to the `StateHandler` – and then the `updateState()` function loops through all the stored `TransitionRule` objects and applies their effects depending on whether criteria are met.

Updating the StateHandler just consists of calling the 3 functions `updateState`, `updateTempo` (if adapting tempo), and `updateSequences` during the process block function.

Turning sequences on and off works with a sort of 'traffic light' system, where for example, if a sequence is turned on it is marked as `turningOn` until the sequence repeats back to the beginning, where it is then updated to fully on. The function `updateSequences()` is responsible for updating a global beat position variable for all the sequences – a float which gradually increases at a speed determined by tempo and sample rate to some 'max number of beats' before resetting to zero. This function then also handles any `turningOff`→`off` or `turningOn`→`on` transitions if a specific sequence has reset back to it's first beat.

### 3.2.1 Adapting Tempo

This works by just comparing every event detected by the event detector to the nearest sub-beat (of 4, hard coded for now). It takes the distance to the nearest sub-beat, scales it by the 'sensitivity' parameter and then adds or takes it away from the StateHandler tempo (depending on if the detected event was early or late).

## 3.3 `Sequence`

The `Sequence` class encapsulates all the relevant information about a rhythmic sequence. It stores:

- the total number of beats in this sequence
- the number of beat subdivisions present
- a vector of boolean values specifying the on/off rhythmic pattern (intuitively, this has a size of number of beats multiplied by number of beat subdivisions).
- a midi note value and velocity to output
- current index of the pattern vector which is occurring.

The key functionality of the `Sequence` class occurs in the `isOutputting()` function. The `StateHandler` updates a global beat position float value for all the sequences, and then each `isOutputting` function calculates the appropriate current index of that `Sequence` object's pattern vector, returning true if it has just changed to a element of the pattern marked as true (i.e. the drum 'beat' moment). Mathematically, the function just does the following:

$$isOutputting(\texttt{beatPosition}) = (\texttt{numBeatDivisions} \times beatNumber) + subBeatNumber$$

where:

$$beatNumber = \lfloor \texttt{beatPosition} \mod \texttt{numBeats} \rfloor$$

$$subBeatNumber = \lfloor (\texttt{beatPosition} \mod 1) \times \texttt{numBeatDivisions} \rfloor$$

Note that $\lfloor \ \rfloor$ is the floor operation (i.e. rounding down / cast to int). Getting this right means one has a handy global beat float to current pattern value mapping.

## 3.4 Transition Rules

A `TransitionRule` object encapsulates a vector of sequence indices to affect, a same-sized vector of effects to apply to each sequence, and the means to check a criteria of whether to apply these effects. Specifically, an instance of this class is initialized with a reference to the StateHandler object, and the `triggered()` function accesses certain values (usually from the `StateHandler`'s `EventDetector`) and determines whether a sequence change should take place.

The class `TransitionRule` itself has an empty virtual `triggered()` function, but a range of child classes are defined which inherit from this class, overriding the `triggered()` function to it's own custom behaviour. These classes are:

- `EventDensityTransition`: triggered if density of detected rhythmic events is greater than some threshold.
- `MeanAmplitudeTransition`: triggered if the average amplitude of input signal within a time-period is above a certain threshold.
- `DecreasingAmplitudeTransition`: triggered if the mean amplitude of the audio input within a certain look-back time is only decreasing.
- `EventOnBeatTransition`: triggered if a rhythmic event occurs within a a specified threshold distance to a specified beat (e.g. beat 2 of 4, subdivision 3 of 4).

### 3.4.1 Composing multiple `TransitionRule` objects

For more custom logic on whether to trigger transitions, some transition rules were defined which are initialized with pointers to existing `TransitionRule` objects. Specifically a `AndTransition`, `OrTransition` and `NotTransition` were created, which have an overridden `triggered()` functions which just return the logical combination (e.g. AND, OR and NOT) of the `triggered()` outputs from the provided `TransitionRule` pointers.

For example, one could define a transition which turns on sequence 2 if an event occurs directly on beat 3 of 4, and the event density is greater than 0.5. As long as these logical transition rules have states affected, effects to apply, and an overridden `triggered()` function, then they are handled fine by the `StateHandler`.

If one adds enough `TransitionRule` objects to the `StateHandler` (including a range logical combinations) to switch the combination of rhythms and midi

output notes present, the hope is that the final outputted drum beat is complex, adaptive to the input sound, and fairly human / 'non-algorithmic' sounding.

### 3.4.2 A rhythmic-event midi transition

One extra transition rule created is `SetTriggerMidiTransition`, whose vectors of "effects" and "sequences affected" are always empty, but instead a vector of indices (0, 1 or 2) and a vector of boolean on/off values are provided. These enable / disable which midi notes are triggered when a rhythmic event is detected by the `EventDetector`. These on or off values are set as a side-effect of the `triggered()` function, and then the function always returns false. The `StateHandler` calls this function along with all the other stored transition rules, but does not set any sequence states.

For example, a vector of {0, 1, 2} and {true, true, true} would turn on the 3 possible midi outputs that the detected event can trigger: a kick sample, open hi-hat sample and a china symbol – but in practice I never configure transition rules that enable all of them at once. For now it is hard coded that an event-on detection can only trigger a kick, open hi-hat or china, and a release-event can only trigger a snare output ({0} and {true}) – but future work could add more possibilities. An important next step for the plugin could be a better class-representation of the possible midi-outputs for the events, instead of hard coding this information in parallel vectors.

# 4 Audio Processing Details (`EventDetector`)

## 4.1 Event Detection

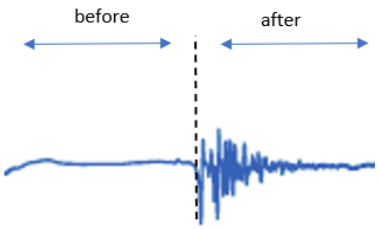


Figure 2: Example audio samples when a rhythmic event is occurring.

The event detection works by considering some buffer (of length 'window size', a variable plugin parameter), and then dividing it in two sections: before and after (see Figure 2). We then calculate the sum of the amplitude (absolute values) of all the samples in the 'before' half, and then calculate the same for the 'after' half. If the ratio of the after-sum divided by the before-sum is above some threshold then an event is detected. If the ratio of the before-sum

to the after-sum is above a threshold, we say that a sudden release-event has occurred.

Intuitively it's just computing whether a measure of average volume suddenly changes from either low to high (as illustrated in Figure 2), or from high to low.

It's worth noting that the `EventDetector` stores a large buffer 8192 samples called `bigBuffer`. This is (generally) much larger than a buffer size of samples processed in one `processBlock` call. The point of this is that it just acts as a first-in first-out (FIFO) queue for us to put audio data into, and hence have a larger current audio buffer to work with. The 'window size' parameter for event detection just varies the amount of `bigBuffer` used for the calculations.

#### 4.1.1 Bias towards events on beats

One extra piece of functionality recently added was biasing the event detection towards moments that are near certain beat subdivisions. The reason for this is to try and account for the fact that we have a higher expectation rhythmic events on actual beat subdivisions.

One provides a specified number of beat divisions when initializing the `EventDetector`, and then `distToNearestSubBeat()` calculates the distance to the nearest sub-beat based on the current `beatPosition` updated by the `StateHandler`.

Then every time when detecting any rhythmic events, the ratio of the 2 sums (the 'before' and 'after', see Figure 2) is also scaled by 1 minus the distance to nearest sub-beat; a value I've termed as the `prior` since it's just 1 if we're currently on a sub-beat position (hence we'd expect rhythmic events here), and 0.5 if exactly between 2 sub-beats. Multiplying the ratio by this `prior` just accounts for our prior knowledge of when these rhythmic events are more likely. Another parameter of the plugin is the `eventOnBeatBias`, which takes values between 0 and 1 and interpolates between whether we actually multiply the ratio by the `prior` or just a uniform value of 1.

### 4.2 Average volume detection

The `EventDetector` also stores a `vector<float>` of 8 average volumes calculated from the samples in `bigBuffer`. This vector also acts as a FIFO queue, where after intervals of 0.2 seconds a new average volume is added to the vector. The mean value of this entire vector is used as a 'longer period of time' mean volume estimate.

#### 4.2.1 `isVolumeDecreasing()`

A function was implemented which just returns true if all the samples in the volume vector are decreasing in value. This is handy because it generally just

returns true if a long sustained guitar chord is played, which we might want to use to trigger some transition in the drum beat generation.

# 5 User Interface Design

Three custom UI components (inheritting from the `juce::Component` base class) were created, which correspond to different UI elements shown in Figure 3.

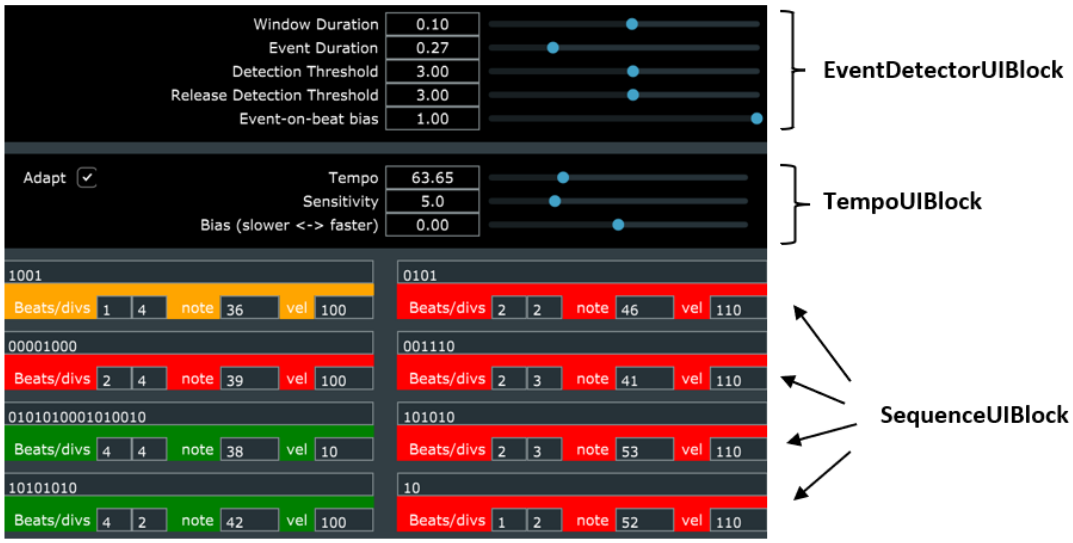


Figure 3: The user interface. The colours represent whether a `Sequence` object is on, off or either turning on/off (like a traffic light), and are updated in real time when using the plugin.

The parameters in the UI (which aren't obvious from the name) are:

- Window Duration: the length of audio used when calculating whether an event occurs.
- Event Duration: the cool-down in which the `EventDetector` has to wait until it can trigger another event.
- Detection Threshold / Release Detection Threshold: how sensitive the `EventDetector` is to the rhythmic on / off events.
- Event-on-beat bias: the extent of which the `EventDetector` will weight the detection prediction by how much we expect events at the current distance to the nearest sub-beat.
- Adapt: the adapt check box enables / disables the adapting of the tempo.

- Sensitivity: how sensitive / how much the tempo will be adapted by after each event detected.
- Bias: a preference to slowing down only (-1.0) to only speeding up (1.0) when adapting tempo.
- Beats/divs: the number of beats and number of beat subdivisions a sequence has.

# 6 Next Steps

- some graphical UI element which displays (and could ideally allow editing) of the `TransitionRule` objects. The main interesting functionality from this plugin comes from setting up the rhythms and the transition rules – but although the `TransitionRule` objects are very flexible and can be logically combined, editing this literally in the code is very tedious.
- a clearer UI, perhaps using time-signatures to specify the size of a Sequence instead of number of beats/subdivisions.
- some refactoring of the code before future development.
- my initial usage of this plugin has been really fun, but more time needs to be spent identifying good combinations of transition rules that lead to diverse and interesting beat behaviour.
- procedural-generation of the sequences (and maybe even transition rules?) could produce some really interesting drum beats.

# A Parameter settings used for recordings


Figure 4: Screenshot of sequencer settings used for recording one (AP3recording1_B131787.mp3).


Figure 5: Screenshot of sequencer settings used for recording two (AP3recording2_B131787.mp3).