Musical Robot Swarms and Equilibria

Michael Krzyżaniak

University of Oslo, RITMO, P.O. Box 1133 Blindern, 0318 OSLO, Norway

ARTICLE HISTORY
Compiled August 19, 2020

Abstract
This paper studies swarms of autonomous musical robots, and its contributions are twofold. First, I introduce Dr. Squiggles, a simple rhythmic musical robot, which serves as a general platform for studying human-robot and robot-robot musical interaction. Regarding this, I discuss the hardware and software design, with a special focus on how a swarm of autonomous musical robots can stay synchronized predictively and with zero-mean timing error; and how they can use this to accurately transcribe rhythms that other robots play, in order to inform their own playing. Secondly, I use three Dr. Squiggles robots to study what happens when musical robots listen to, learn from, and respond to one another while improvising music together. I show that with each robot using a simple, responsive rhythm generator, the robots almost always eventually reach some sort of equilibrium. Moreover, I show that in some cases, the equilibria demonstrate emergence. The presented analysis scales to larger numbers of robots. This paper has a supplementary video at `https://www.youtube.com/watch?v=yN711HXPfuY` which shows the three robots playing some of the equilibrium rhythms.

KEYWORDS
Robots; musical robots; computer music; algorithmic composition; swarm robotics; swarms; metacreation; metacomposition; robot-robot interaction; equilibrium; beat tracking; Dr. Squiggles; onset strength; machine perception; machine creativity; timing;

CONTACT Michael Krzyżaniak. Email: mkrzyzaniak@protonmail.com

Figure 1. Two happy and very brightly colored Dr. Squiggles rhythm robots.

1. Introduction

Recently, swarms of collaborative robots have been proposed for a wide variety of complex activities (Kolling, Walker, Chakraborty, Sycara, & Lewis, 2015; Mohan & Ponnambalam, 2009). Despite the fact that music-making is a complex activity often undertaken collaboratively by large numbers of people, very little work has been done on swarms of musical robots. In the context of musical robots, I use the word 'swarm' to imply that there are a scalably large number of robots; that each robot is fully autonomous and not governed by any central computer; that each robot has agency to improvise or otherwise make decisions about what music it plays; and that each robot listens to the other robots and adjusts what it plays based on what it hears. There are at least three key reasons why such research should be undertaken. First, since music has very tight and specific requirements, for example precise timing and fast and efficient mechanical control, research in this area might inform other situations where swarm robots could benefit from these properties. Second, an analysis-by-synthesis approach in this area could provide some insight into the inner workings of large ensembles of human musicians, especially regarding the psychology, perception, and methods of communication of individual musicians in the ensemble. Finally, research in this area is interesting in its own right as pure research, even when swarms of musical robots are used in a way that does not model any real human musical ensemble. It is interesting both as an artistic exploration into what types of music can be made by these methods, and as a scientific exploration of open questions that arise naturally from the premise. Some of these open questions are as follows.

(1) As the swarm makes music, how can new musical ideas or motives get introduced and subsequently propagate through the swarm, such that the music stays interesting and apparently structured and coherent from the standpoint of a listener?
(2) How does an individual robot know when it should *not* play anything at all? Given a large swarm of musical robots, in general each robot should probably not play all of the time. It would be nice to have soli, duets, tutti, call-and-response figures, and so forth arise spontaneously over the course of a performance. Despite

a considerable literature on methods that musical robots can use to create music, I have never seen a study on how musical robots can create silence.

(3) Under what conditions will the behaviour of the swarm be chaotic, and how can the music produced by the swarm be made to adapt to the edge of chaos as it evolves over time?

(4) What is the eventual behaviour of the swarm, and in particular, under what conditions will the swarm reach *equilibrium*? By 'equilibrium', I mean that the swarm has fallen into some repetitious behaviour that will continue indefinitely in the absence of any disruption.

In this paper I will focus primarily on the fourth of these questions, and I will touch on the first and second. To that end, I will first discuss the design and operation of a musical robotic platform called Dr. Squiggles, three of whom I built for this purpose. Then I will describe a simple responsive rhythm synthesiser, and analyze the equilibria that arise when a network of Dr. Squiggles robots play music together, each using this synthesiser.

2. Previous Work

In the context of computer music, the term 'swarm' is often synonymous with the famous Craig Reynolds 'boids' algorithm, or some variant thereof. In a typical scenario, several virtual agents move around in Euclidian space, simulating a flock or swarm. The positions or other properties of the agents are used to control some aspects of music. In Tim Blackwell's SWARMUSIC algorithm (Blackwell & Bentley, 2002), the position of each agent in 3D space determines the pitch, loudness, and duration of a single note at each frame of the simulation. Similarly, in the work of Aaron Thomas Albin (Albin, 2011), each agent's speed controls the pitch of a musical sequence, and its degree of proximity to other agents determines the rhythmic pattern of the sequence. He describes the system as being intended to be used on physical mobile robots that produce acoustic sound, although only a simulation is presented. In an interactive artwork by Cameron Browning called Phlock (Browning, 2008), sound is triggered as agents collide with obstacles in the space. In a responsive dance piece entitled 'Separation: Short Range Repulsion' (Krzyzaniak, Akerly, Mosher, & Yildirim, 2014), human dancers attempt to enact the boids algorithm, and the relative speed of each dancer as compared to the centroid is used to manipulate the parameters of a digital sound synthesiser. In all of these, the musical behaviour of the agents is ultimately determined by their positions in space, and sound is added as a secondary layer that sonifies the positions. In none of these systems do the agents respond directly to the sound produced by other agents.

Similarly, in the field of Artificial Life, there have been many agent-based systems that generate music or rhythm. In these systems, software agents interact with music and each other, drawing on metaphors about how humans interact and learn music from each other. There are too many systems of this type to explain in detail, but a reasonable introduction is in (Eigenfeldt & Kapur, 2008). Notable are works such as (Brown, 2005) and (Martins & Miranda, 2007) that specifically identify emergent behaviour that arises in such systems. This field, by its nature, deals primarily with software simulations, but in (Eigenfeldt & Kapur, 2008), the agents correspond to physical percussion robots. This work is especially interesting because they present some analysis on a system in which agents oppose each other rhythmically, and they note that stability is difficult to achieve. I will present further analysis on the stability of a similar situation in Sections

7.3.4 and 7.3.5 of this paper. None of the agents or robots in the cited systems are fully autonomous, as all agents are always simulated on a single central computer, with all of the agents sharing memory, having perfect knowledge, and accessing the same global variables and shared notion of the beat, or flattened representation of time.

There have been several ensembles of musical robots playing together. Bands such as Compressorhead (Machines, 2013), One Love Machine Band (Ciardi, 2015), and Toyota's musical robots (when used in an ensemble), fit this description. However, as far as I can tell, in each case the robots are pre-programmed to play predetermined music via MIDI or similar, and are not in any way responsive. In Pat Metheny's Orchestrion project (Records, 2010), he plays guitar with a large number of musical robots built by the League of Urban Musical Robots (LEMUR) led by Eric Singer (Singer, Feddersen, Redmon, & Bowen, 2004; Singer, Larke, & Bianciardi, 2003). As soon as Metheny plays a note on the guitar, a selected group of robots will immediately play the same note, doubling what he plays. While these robots are responsive, what they play is entirely determined by what he plays, and the robots do not have agency or autonomy to improvise or respond to one another. Several other musical robots with a greater degree of autonomy do exist. Notable among them are the WABOT2 organ robot which can read a printed score and follow a singer's tempo (松島俊明, 金森克洋, & 大照完, 1985); Shimone the marimba robot which can improvise along with a piano player matching the tempo and chords (Hoffman & Weinberg, 2011); and Kiki the djembe robot which can improvise with a human percussionist while learning how to navigate the timbral space of its instrument (Krzyzaniak, 2016). However, as far as I know, robots like Kiki and Shimone have never participated in a swarm of robots with similar capabilities.

There are a few robotic ensembles that are swarms according to the definition above. In the interactive installation 'Sverm Resonans' developed at University of Oslo, visitors carry around augmented guitars (Gonzalez Sanchez et al., 2018). The guitars sense the presence of people and play sound. In a later development, the guitars were made to play plucking sounds, listen through a microphone, and try to synchronise with the other guitars via a firefly-inspired algorithm. This is one of few works that qualifies as a swarm under the given definitions. Likewise, Nicholas Baginsky's legendary rock band the Three Sirens (Baginsky, n.d.) comprises several musical robots whose behaviours are governed by neural networks; the robots listen to and learn from each other as they play. Unfortunately, little technical information is available about how this works, and I don't think that the eventual long-term behaviour of the system has been analyzed. In David Rokeby's installation n-cha(n)t (Rokeby, 2001), several Macintosh computers listen for speech spoken by people or the other computers, and generate spoken responses via speech synthesis. In the absence of human input, the computers will eventually reach equilibrium, where all computers speak the same utterances at the same time. My intention here is to do something similar with rhythmic musical robots, to see what types of equilibria emerge, and under what conditions.

3. Dr. Squiggles

In order to begin studying swarm behaviour in musical robots, I designed and built three Dr. Squiggles robots, two of which are depicted in Figure 1. Dr. Squiggles is the most mechanically simple musical robot that I could conceive. Functionally, it is just a contact microphone and eight solenoids controlled by an embedded computer. It uses its microphone to listen selectively to rhythm in the environment, and its solenoids to tap out rhythms on whatever surface it is on. The overall philosophy behind this design

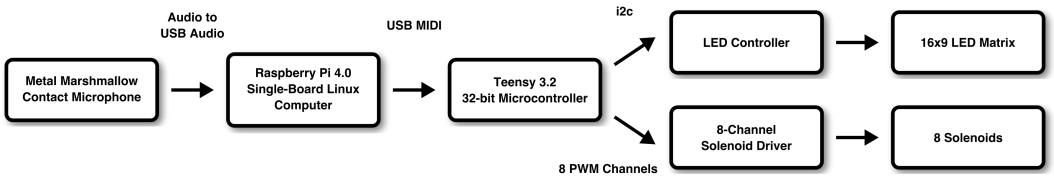

Figure 2. Block diagram illustrating Dr. Squiggles's hardware.

was to build the simplest possible sensor / actuator network and imbue it with the most interesting possible behaviour. Here I will discuss the physical design of the robot, including the hardware and its control via firmware.

3.1. Hardware

A more complete functional block diagram of Dr. Squiggles is shown in Figure 2. It consists of

- Eight solenoids controlled by an array of darlington transistors.
- A 16x9 LED matrix with a dedicated I^2C controller that serves as the the robot's eye.
- A low-noise Metal Marshmallow[1] contact microphone with a builtin powered, buffered preamplifier. The microphone is external to the body of the robot, and Dr. Squiggles uses this to listen selectively to rhythms played by a person, another robot, or itself. It solves the Cocktail Party problem.
- An embedded Raspbery Pi computer, which does the heavier computational tasks of processing the microphone input and deciding when the robot should eject one of its solenoids.
- A Teensy microcontroller that receives USB-MIDI commands from the Raspberry Pi. It applies analog voltage to the solenoids. It also animates the eye; it runs an animation drawing loop and sends I^2C commands to the eye controller. The USB cable that connects the Raspberry Pi to the Teensy is exposed on the bottom of the robot, so that any computer can be substituted for the Raspberry Pi. For example, a user could connect their laptop to the Teensy and use any popular music software to send MIDI messages that cause the solenoids to eject.

The simplicity of the design limits the robots in several key ways. Notably, Dr. Squiggles has limited ability to control the pitch and timbre of its strokes. These are important parts of rhythm generally, and I have studied timbre in musical robots at length previously (Krzyzaniak, 2016). However, the purpose of the present work is to study swarm behaviour in musical robots, and deliberately imposing these limitations has allowed me to focus on this without becoming intractably mired in the complexity that comes with timbre and pitch.

The firmware used by the Dr. Squiggles's Teensy microcontroller is tightly coupled to the function of the hardware, and consequently I think of the firmware as being part of the hardware, so for clarity I will briefly elaborate on its operation here. The firmware has two responsibilities. First, it turns the solenoids on and off, and second it animates the eye, and I will discuss both in turn in the following subsections.

[1] https://michaelkrzyzaniak.com/marshmallow/

3.2. Solenoid Firmware Control

When the firmware recieves a MIDI note-on message from the Raspberry Pi or other computer, it will immediately apply a voltage to a solenoid and continue to apply it for 9.5 ± 0.5 milliseconds. By default, the firmware treats seven of the solenoids as being interchangeable, and uses them in round-robin order. This gives faster repeat rates than could be achieved with one solenoid alone. The solenoids are connected to the microcontroller's PWM channels so that the robot can respond to the velocity part of the incoming MIDI messages. However, varying the PWM duty-cycle has a minimal effect on the perceived strength of a stroke. For this reason, the firmware will use up to three solenoids simultaneously to play a single stroke. As the note velocity increases from minimum to maximum, the robot will first use one solenoid with increasing duty cycle. Once the first solenoid is at 100%, the firmware will fade in a second solenoid with increasing duty cycle, and then a third. This gives an acceptable perceptual range of note-strengths, while guaranteeing that no solenoid will ever be used in two consecutive strokes.

The eighth solenoid is known as the 'beat bell' and responds to a dedicated MIDI message. Prototype Dr. Squiggleses held a metal bell in one of their tentacles, and the corresponding solenoid was mounted specially to strike it. This was to allow the robot to prominently play the beat, or other timekeeping rhythm or bell pattern, against which rhythms can be referenced. I eventually decided for a variety of reasons to remove the physical bell from the design, although this solenoid is still reserved and intended to be used for timekeeping rhythms.

Sometimes it is desirable to put Dr. Squiggles on some object such that each solenoid strikes a separate item. For example, I sometimes set Dr. Squiggles on a bespoke circular glockenspiel in such a way that each solenoid strikes a separate tine. In this case, the automatic selection of solenoids is not always desirable. For this reason, the firmware also responds to dedicated MIDI messages that select the individual solenoid and duty cycle for a given stroke, bypassing the firmware's automatic selection of up to three solenoids per stroke.

3.3. Eye Firmware Control

The firmware draws an eye animation on the 16x9 LED matrix. The eye is drawn parametrically, and the parameters are the size and position of the eye and pupil. It takes about 15 milliseconds to burst-write the LED controller's registers over I^2C, and the eye is redrawn every 33 milliseconds on a low priority thread that can be interrupted by incoming USB messages or anything pertaining to the solenoids. Any subset of the eye's parameters can be animated from within the firmware by specifying sequences of values for the parameters, and interpolation times between the values. Blinking, for example, consists in starting with the current eye size, interpolating to 16 pixels wide and 1 pixel tall over 66 milliseconds, and returning to the original size over another 66 milliseconds. Several animations can be run simultaneously if they adjust different parameters, for example the robot could blink while moving the eye from left to right to indicate 'no'. The firmware defines several short animations such as blinking, indicating 'yes' and 'no', and becoming surprised, inquisitive, or focused. These can be triggered from the Raspberry Pi host computer via special USB MIDI messages. By default, in the absence of such commands, the firmware will make the eye saccade and blink at random intervals, where saccading consists of setting the x or y position of the eye or iris to random values at random times. Whenever the host computer sends a USB MIDI

command to trigger an animation, the firmware will stop automatically blinking and saccading, but it will resume after a few seconds if no other commands are received. The OpenSquiggles software application that runs on the Raspberry Pi, which will be discussed in Section 5 below, normally sends commands that make the robot blink on every downbeat, which provides helpful visual feedback to people that might be interacting with the robot.

As a side note, in Pixar's 2008 animated film WALL-E, the robot EVE's face consists only of two low-resolution monochrome eyes. The eyes are highly expressive and, because EVE is essentially mute, the eyes serve as one her most effective means of communication. This fact did not go unnoticed by robot designers. For instance, in 2016, the (now-bankrupt) company Anki released a toy robot called Cozmo with a very cute and expressive face that is nearly identical to EVE's. At the same time, there is a growing body of research showing how ensemble musicians use eye contact and other non-verbal cues to communicate while playing together (Bishop & Goebl, 2015; Williamon & Davidson, 2002). Nonetheless, information about how human musicians use their eyes while playing still remains largely unapplied to the design of expressive musical robot eyes, and I am leaving this as future work.

4. Onset, Tempo, and Beat Tracking

If a swarm of robots is going to play beat-based music, they are going to need to stay synchronised. Although I have some reservations about the entire project of beat-tracking, especially with regards to percussion robots and human-robot interaction, this turns out to be one of the best ways to keep a decentralised swarm synchronised. My intention was to select a suitable algorithm from the literature and implement it, although I ended up picking and choosing from a few different algorithms and making a few small modifications as necessary. My final implementation is available on the internet[2]. In general, these algorithms operate in three stages: extracting an onset signal from the raw audio, tempo estimation, and beat detection. I will describe each in detail here, and elsewhere I will use the term 'beat tracking' to refer collectively to all three stages.

4.1. Onset Signal

The first stage of the beat tracker extracts an onset signal from the incoming audio. I take an approach that has been described at length in (Mottaghi, Behdin, Esmaeili, Heydari, & Marvasti, 2017) and elsewhere in the literature. Briefly, I take a windowed DFT over the audio. At each window, I first cancel noise by setting to zero any bin whose value is less than -74 dB. Then, I add up the amount of increase in all of the frequency bins that have more energy than they did in the previous window. This is called the spectral flux of the signal. I low-pass filter the spectral flux at 10 Hz, and the result is called the 'onset signal' or 'OSS'. An example OSS extracted from percussion music is shown in Figure 3. In principal the OSS should spike whenever there is a note onset. Observe however that it performs poorly on sustained notes with vibrato or increasing loudness, which have a large spectral flux over the course of a single note. Nonetheless, this method is considered state-of-the-art and works very well for percussive music, such as will be used with the robot.

[2] `https://github.com/michaelkrzyzaniak/Beat-and-Tempo-Tracking`

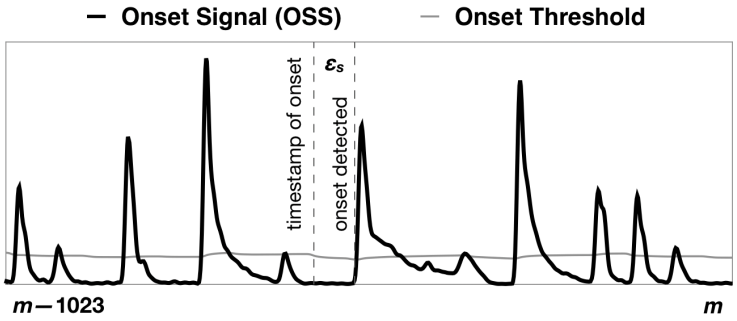

Figure 3. Dr. Squiggles's onset detector, used here to analyze a drum kit beat[4]. The onset signal spikes whenever there is an onset in the music, and an onset is detected whenever the onset signal rises past the adaptive onset threshold. However, due to some latency, the onset actually happened ϵ_s samples previously.

4.1.1. Onset Detection

Later stages of the beat tracker use the OSS directly with no other processing. However, transcription requires knowing exactly when the onsets occur. I achieve this using an adaptive threshold. The threshold is always a constant number of standard deviations above the mean value of the OSS signal, over the past N samples. I low-pass filter the threshold to minimise the effect of large spikes in the OSS. Moreover, the threshold never falls below a defined minimum value, so that it does not fall below the noise floor when the audio signal is silent, which would produce many spurious onsets. In real time, an onset is reported whenever the rising edge of the OSS crosses the threshold. However, because the algorithm introduces a small amount of latency, which I will later refer to as ϵ_s, the onset will have actually occurred in an audio sample that was analyzed some time in the past. The latency is roughly constant for a given set of initialization parameters, therefore, when the onset detector reports an onset, it also reports the timestamp of the onset by subtracting ϵ_s from the current analysis frame. In order for this to work properly, ϵ_s must be determined empirically for a given set of initialization parameters, and this is elaborated in Section 6 below. The operation of the OSS and onset detector is depicted in Figure 3.

This is not a great general-purpose onset detector, and would not work well on, for example, recorded popular music. However, it is very efficient, and works quite well in the present application, where short percussive sounds are recorded through a very low-noise contact microphone.

It is worth noting that from a psychological perspective, it is not clear when note onsets perceptually occur. Is it at the onset of sound, the peak of the amplitude curve, or somewhere else? It depends on a broad variety of conditions (Danielsen et al., 2019). The approach taken here is not based on a perceptual model, but is rather designed to minimise latency by detecting the onset as soon as possible, potentially even before the perceptual center of the stroke. One consequence of this is that the onset detector does not report the strength, loudness, or timbre of the onset, as it reports the onset before enough information is available to compute these. In the past I did implement a timbre-classifying onset detector (Krzyzaniak & Paine, 2015), although that is not the focus of the the present work, which has not necessitated its use. In any event, I am leaving it as future work to locate the onsets using a perceptual model.

[4] `https://www.youtube.com/watch?v=XdK61e4F8hU`

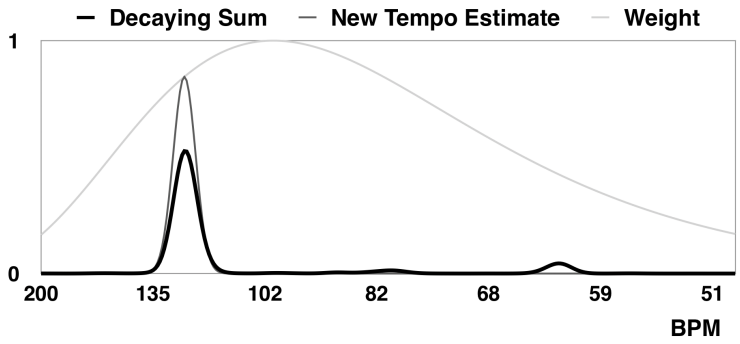

Figure 4. Tempo estimate taken from the middle of the Conquassabit from Handel's Dixit Dominus[6]. At each time step, a new tempo estimate is represented by a Gaussian spike. That is weighted by a log-Gaussian curve and added into a decaying sum of previous tempo estimates. The tempo with the highest value in the decaying sum is taken to be the true tempo of the music at the current moment in time.

4.2. Tempo Tracking

I estimate the tempo of the incoming audio using the stem–base algorithm presented in (Percival & Tzanetakis, 2014). This method involves cross-correlating the OSS with idealised rhythms to obtain tempo candidates, and then scoring the candidates to obtain a tempo estimate. I make only two small modifications to facilitate real time operation. In the op cit paper, they sum together of all the tempo estimates over the course of the song, and at the end the highest peak wins. I additionally decay the sum over time so that it can adapt to changing tempi. Moreover, at each analysis frame, I weight the current tempo estimate by a log-Gaussian window whose peak is centered on 100 bpm. This is so that when the tempo-estimator is confused between two tempi that are separated by a factor of two, it will prefer the more moderate estimate. Figure 4 shows these modifications.

4.3. Beat Detection

My original intention was to implement some combination of existing beat-trackers as described in the literature. However, I ran into some issues and ended up taking an approach that diverges slightly from what is reported elsewhere in the literature.

By default the OSS, described above, is calculated on 44.1 kHz audio, using an STFT with hop-size 128, meaning that the OSS and consequently all other signals discussed in this section operate at approximately 345 Hz. By default, all of these signals are stored in buffers of length 1024, which represents about 3 seconds at this sample rate. Let the current tempo estimate at each analysis frame be called Δt and represent the beat period expressed as an integer number of samples at the OSS sample rate.

I start by calculating a 'Cumulative Beat Strength Signal', or CBSS, as described in (Stark, 2011), page 60. The CBSS is a recursive signal that, at each analysis frame, combines a past value of itself from one beat ago with the current OSS sample. For completeness, the equations are repeated here. The author defines a log-Gaussian weighting function centered on samples one beat in the past:

$$\mathrm{W}(v) = \exp\left(\frac{-(\eta \log(-v/\Delta t))^2}{2}\right) \tag{1}$$

[6] `https://youtu.be/_Ki-LIEwfaM?t=1283`

where the author recommends $\eta = 5$. The weighting function is applied to the CBSS, and the maximum value is picked from the result:

$$\Phi(m) = \max_v(\mathrm{W}(v)\mathrm{CBSS}(m+v)) \tag{2}$$

where m is the current analysis frame, and for efficiency it is only necessary to search for the peak in the range $v \in \{m - 2\Delta t, m - \Delta t/2\}$. This value is then combined with the current OSS sample:

$$\mathrm{CBSS}(m) = (1-\alpha)\mathrm{OSS}(m) + \alpha\Phi(m) \tag{3}$$

where the author recommends $\alpha = 0.9$.

The author then predicts beat locations by projecting the CBSS into the future, windowing it, and picking the peak. This is done once per beat, midway between the beats. The problem with this is that it only checks the tempo estimate once per beat, ignoring many intermediate tempo estimates. It frequently happens that the tempo-estimator is confused between two competing tempi, and a secondary peak will form and occasionally surpass the principal peak in the tempo estimate. So it sometimes happens that the beat tracker ignores many consecutive correct tempo estimates and then predicts the next beat at just the moment when the spurious secondary estimate is winning. In other words, this method is not robust to occasional incorrect tempo estimates. Another paper (Mottaghi et al., 2017) presents a slightly different approach that also makes use of the CBSS. It suffers a slightly amplified version of the the same issue, due to picking two interdependent tempo estimates, one from the beginning and one from the middle of the beat. Moreover, they pick peaks using an algorithm (Scholkmann, Boss, & Wolf, 2012) that, naively, has a time complexity of $\mathcal{O}(n^2)$, poorly defined boundary conditions which need to be addressed for realtime sequential operation, and a bizarre and unnecessary use of random numbers[7]. While the peak-picking can be optimised somewhat, the authors do not discuss the optimizations or the boundary conditions, and it is still by far the most time-consuming part of the algorithm.

To solve these issues, I use a somewhat different method, depicted in Figure 5. At each analysis frame, I define a signal containing four unit impulses spaced one beat apart, starting at sample $-\phi$ and projecting back in time:

$$I(\phi) = \sum_{i=0}^{3} \delta(-(\phi + i\Delta t)); \tag{4}$$

where $\delta(x)$ is a unit impulse at sample x. I then compute the cross-correlation between the impulses and the CBSS, and select the ϕ where the cross-correlation is maximum. I only search in the interval $0 \leq \phi < \Delta t$. Note that because $I(\phi)$ is sparse, the cross-correlation can be done efficiently by direct computation, as

$$\phi_{max}(m) = \operatorname*{argmax}_{0 \leq \phi < \Delta t} \left(\sum_{i=0}^{3} \mathrm{CBSS}(m - \phi - i\Delta t) \right) \tag{5}$$

[7]They could use any nonzero positive numbers, e.g. the value 1, in place of the random numbers $r + \alpha$ in Equation 1, and later in Equation 4, rather than checking that the standard deviation of a column is equal to 0, they could simply check that the sum of the column is equal to 0.

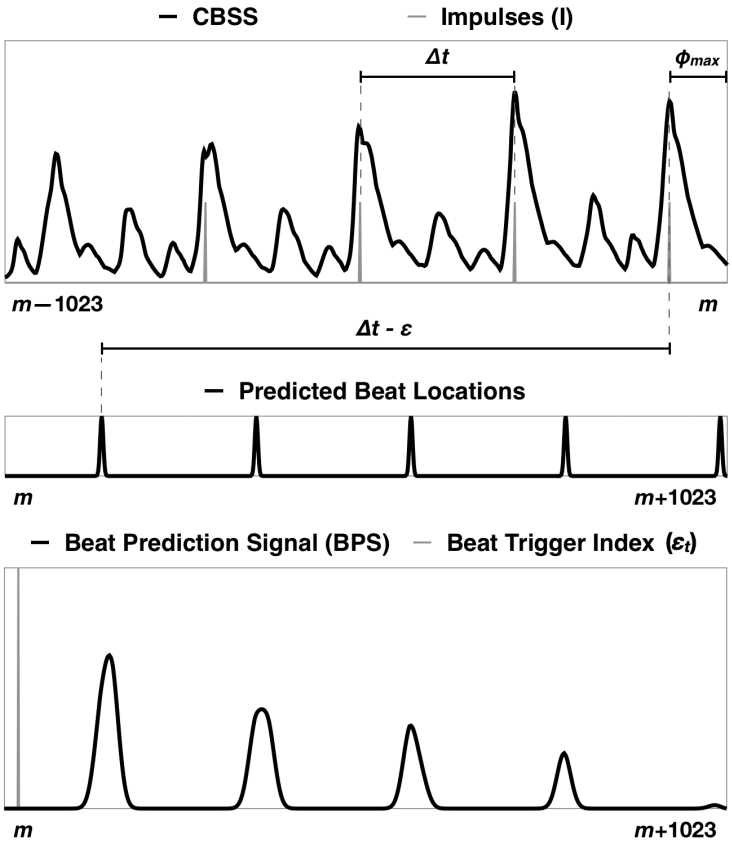

Figure 5. Operation of the beat predictor. At the top, the CBSS signal is computed and cross-correlated with impulses spaced one beat apart at the current tempo. The best fit between the impulses and CBSS is shown. Below that, the impulses are projected into the future, and adjusted slightly earlier in time to compensate for known latencies. Then below that, the BPS is shifted one sample to the left and the forward-projected impulses are added into it. A beat is detected when the maximum value in the BPS is at the beat trigger index. All signals flow from right to left.

where m is the current analysis frame. I then define a signal containing periodic Gaussian spikes that extends $I(\phi_{max}(m) - m)$ indefinitely into the future.

$$P(m,i) = \exp\left(\frac{-[((m+i) \mod \Delta t) - (\Delta t - \phi_{max}(m) - \epsilon)]^2}{w}\right) \tag{6}$$

where m is the current analysis frame, i indexes future frames, and w controls the width of the spikes, by default $w = 10$. I also include the term ϵ which allows the predicted spikes to be moved earlier (or later) in time by any constant amount. This causes the beat-tracker, in real time, to indicate that has detected a beat some amount of time before the beat actually occurs, which is useful, for example, to correct latency. In general, ϵ has three components:

$$\epsilon = \epsilon_o + \epsilon_r - \epsilon_t \tag{7}$$

where ϵ_o and ϵ_r are the offline and real time latency correction factors, discussed in Section 6 below, and ϵ_t is a beat trigger index for which I choose the value 20 by default, whose purpose shall be discussed presently.

$P(m,i)$ represents the best current estimate at analysis frame m of where the future beats will lie. To make use of all such estimates between beats, while minimizing the impact of spurious estimates, I define a 'beat prediction signal', or BPS, which is initialised to 0. At each analysis frame $P(m,i)$ is added into it:

$$\text{BPS}(m,i) = \text{BPS}(m,i) + P(m,i) \tag{8}$$

where again m is the current analysis frame, and i ranges over the buffer that stores BPS, i.e. $i \in [0 \,..\, 1023]$. A beat occurs when the largest value in the signal lies at the beat trigger index:

$$\underset{i}{\text{argmax}}\, \text{BPS}(m,i) = m + \epsilon_t \tag{9}$$

Using $\epsilon_t > 0$ helps prevent repeated beats on the falling edge of the peak as it is shifted out of the left of the buffer. As a variant, the beat detector can ignore beats for a certain period of time after one beat is detected, by default I ignore beats for 40 percent of the estimated beat period at the time a beat is detected.

4.4. Beat Tracking Modes

That characterises the theoretical operation of the onset, beat, and tempo trackers. In practice, I additionally implemented a few operating modes to facilitate their use in the wild. Some of these modes are as follows.

- Regular beat-tracking mode that does everything described thusfar, but does not handle boundary conditions.
- A count-in mode that uses the first two onsets to define the initial tempo estimate before switching back into regular beat-tracking mode.
- Tempo-locked mode, which, after an initial tempo has been established, just plays evenly-spaced beats that are not influenced by audio input. This is accomplished by setting the CBSS α parameter, as described in Equation 3, to unity.

5. Open Squiggles

After implementing the beat-tracking library, I developed a software application called OpenSquiggles[8] which is a command-line utility intended to run on the Raspberry Pi computer inside of Dr. Squiggles, and can also be compiled and run on OSX for development. The purpose of the application is to use the the onset, tempo, and beat trackers to drive rhythm synthesis modules. These rhythm modules generate the music that Dr. Squiggles plays, and in some ways my ongoing research on rhythmic robots consists in the development of new rhythm modules. OpenSquiggles serves as a controller for the modules which can be used interchangeably within the application; it manages one rhythm module at a time, and the user can cycle through the various modules. The overall architecture of OpenSquiggles is a follows:

(1) OpenSquiggles receives small buffers of audio input from the operating system, and passes them to the beat tracker.
(2) The onset tracker notifies OpenSquiggles whenever an onset occurs, including the timestamp of the onset. OpenSquiggles forwards this to the active rhythm module. The module is free to ignore this information or to use it in some way.
(3) The beat tracker notifies OpenSquiggles whenever a beat occurs, and OpenSquiggles forwards this to the active rhythm module. The rhythm module returns an array of onsets that the robot is supposed to play in the upcoming beat. Each onset consists of
    - A timestamp of the onset relative to the upcoming beat, e.g. 0.5 indicates that the onset should occur halfway through the beat.
    - A strength value indicating how hard the robot should strike to produce the onset. Alternatively, the rhythm module may instead use a special flag indicating that OpenSquiggles should apply a default onset strength to the onset. The algorithm that calculates the default onset strength is described in Section 5.2 below.
    - An index indicating which solenoid should be used to play the onset or, alternatively, a special flag that indicates that the default solenoids should be used, according to the algorithm described in Section 3.2 above.
(4) Over the course of the subsequent beat, OpenSquiggles keeps track of time using high precision, high priority timing thread, and when the time comes to play an onset, it sends a MIDI message over USB to the Teensy microcontroller which applies a voltage to the solenoid. This step incurs a small amount of latency, but because the beat-tracker is predictive, the latency can be corrected, as described in Section 6.2 below.

The OpenSquiggles architecture is illustrated in Figure 6.

5.1. Rhythm Modules

Thus, the rhythm modules generate rhythm one beat at a time. For the sake of illustration, some of the rhythm modules I have developed to date are:

- An n-beat delay module that causes the robot to repeat back whatever it hears after n beats (2 by default).
- A quantised n-beat delay module that passes oncoming onsets through the Desain

[8] `https://github.com/michaelkrzyzaniak/OpenSquiggles`

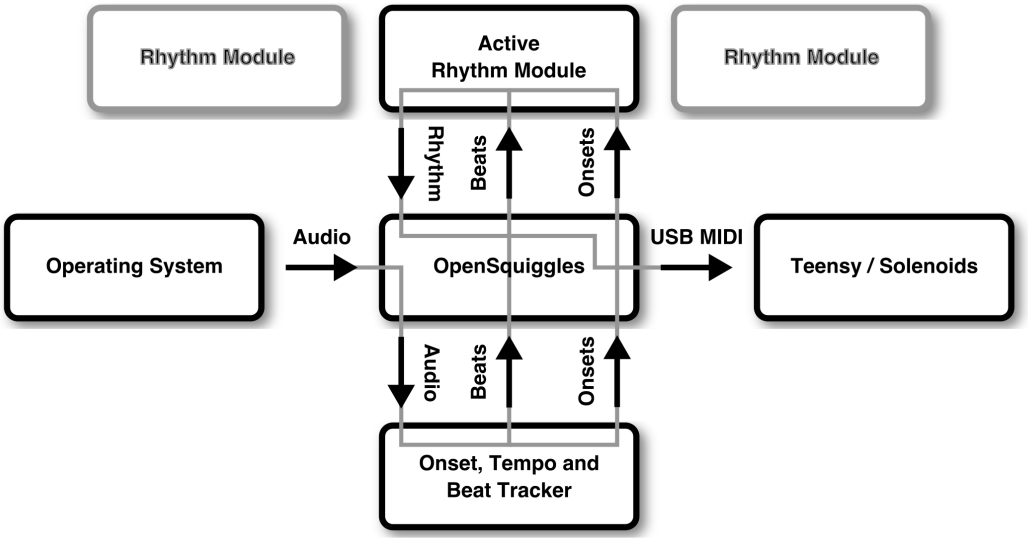

Figure 6. Block diagram of the OpenSquiggles software architecture. OpenSquiggles passes audio samples from the OS to the beat tracker. It then passes information about beats and onsets to one of several interchangeable rhythm modules, which generate rhythms symbolically, one beat at a time. OpenSquiggles then plays the rhythms by sending MIDI messages that eject the solenoids at the correct times.

and Honing quantiser (Desain & Honing, 1989) before repeating them back after n beats.

- A random-from-list module that, at each beat, selects a sequence randomly from a list of precomposed one-beat rhythms.
- An OSC module that broadcasts the timestamps of beats and onsets over OSC. Then, some other software potentially running on another computer can respond over OSC with a list of onsets, which this module will return to OpenSquiggles for playback by the robot. To account for round-trip latency, this module incurs a one-beat delay.
- A histogram module whose operation will be discussed at length in Section 7.1 below.
- Several precomposed repeating rhythms, such as West-African bell patterns, that I enjoy improvising with.

5.2. A note on the strength of note onset strength, and the mystical transformative healing power of number theory

Broadly, when I develop new rhythm modules for Dr. Squiggles, I am interested in rhythm as the division of time. More generally, when people write algorithms that write music, the research often focuses on pitch, rhythm, timbre, melody, harmony. Note-strength[9] is typically not the direct focus of research, and is thus often neglected. However, any rendition of music that does not vary note-strength will be tiresome. Moreover, it is difficult for both humans and robots to synchronise to such music,

[9]I use the term 'note-strength' here to refer generically to a collection of related features like loudness, energy, and note velocity, that contribute to the overall dynamic level of a given note onset. The way that Dr. Squiggles controls note strength is described in Section 3.2 above.

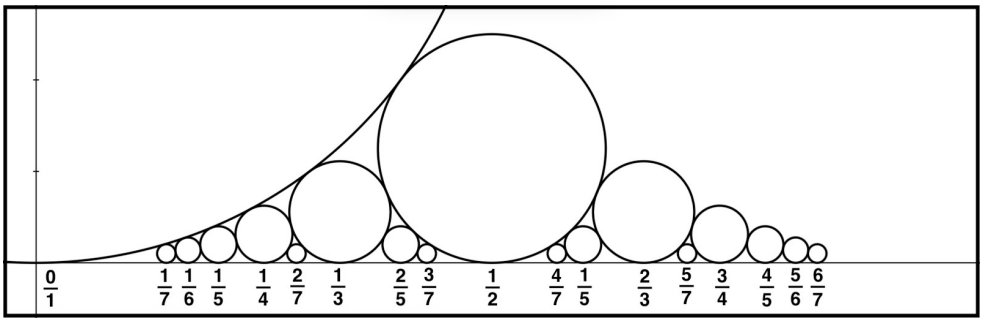

Figure 7. Ford circle packing, shown for denominators not greater than 7. Time is on the X-axis, and one beat is represented here. The size of each circle represents the strength of a note that occurs where the circle touches the X-axis. A note occurring at time t=0 will be the strongest, time t=1/2 will be the next-strongest, etc. In principle, there should be infinitely many smaller circles here, one for each rational number. I note that the circles have diameters $denom(x)^{-2}$, although the absolute size of the circles is not important here.

as note-strength contains important cues about the phase of the underlying beat with respect to the music. Therefore, here I present a quirk of number theory that can be used as a quick and dirty way to assign a default strength to each note as a post-processing step, for music where the beat locations are known.

This method will make the notes that lie on the beat the strongest; those that fall on the half-beat will be the next-strongest, and those on the second and fourth quarter beat will be somewhat weaker than that, etc. Assuming the beat occupies times $0 \leq t < 1$, this implies that the strength is inversely proportional to the denominator of the rational representation[10] of the time where the note onset occurs. This just so happens to be related to the Ford circle packing, depicted in Figure 7, where the circle sizes represent the strengths of notes that occur where the circles touch the X-axis.

If the note onset times are not explicitly quantised, or if a note occurs at an irrational time which cannot be expressed as a fraction with integer coefficients[11], the denominator of the rational representation could be very large, even infinitely so, resulting in very weak onsets. For instance, in this model, a note that occurs at time $t = 0.4999999$ would be imperceptibly weak because its denominator is 10000000. Intuitively, this doesn't seem correct; 0.4999999 should probably count as 1/2 and be relatively strong. Therefore, I compute the nearest rational approximation to the onset time, giving preference to small denominators not greater than n.

The Ford circle packing has a nice property that allows me to do this efficiently for any n. Take any two circles A and B that are touching. By the geometry, there will be exactly one smaller circle C that touches both A and B. Moreover, C will be the largest circle between A and B. The place where C touches the X-axis is the Farey sum of the place where A and B touch the X-axis. The Farey sum is defined to be

$$A \oplus B = C \stackrel{\text{def}}{=} \frac{num(A) + num(B)}{denom(A) + denom(B)} \tag{10}$$

where $\oplus$ indicates the Farey sum, and $num(x)$ and $denom(x)$ extract the numerator and denominator, respectively of the rational representation of x. This, taken together with the fact that that the Ford circle packing is an infinitely deep fractal with circles on all

[10] By 'rational representation', I mean that the number is expressed as a fraction with coprime integer coefficients.

[11] If you throw a dart at a number line, you will, with 100% probability, hit an irrational number.

rational numbers, gives me a means of traversing Figure 7 by successive approximation, to arbitrary precision.

Suppose I want to find a rational approximation of $0 \leq t < 1$, with denominator not grater than n, and return $1/denominator$. In pseudocode this would be done as follows:

Algorithm 1 Note Onset Strength

```
1: procedure GET__NOTE__ONSET__STRENGTH(t, n)
2: init:
3:     A ← 0, B ← 1
4: loop until denom(C) > n:
5:     C ← A ⊕ B
6:     if t < C then
7:         B ← C
8:     else t ≥ C
9:         A ← C
10:    if A = t then
11:        break;
12:    result ← A or B, whichever is closest to t
13:    return 1/denom(result)
```

This is the algorithm that OpenSquiggles uses to apply default onset-strengths to rhythms that are generated by the rhythm modules. Further explanation is on my blog[12] and the production implementation is in the OpenSquiggles source code.

As a brief side note – I previously stated that the onset times do not need to be explicitly quantised, but clearly this algorithm is related to quantization. In fact, it is equivalent to quantizing the onset times to a Farey sequence of order n, which is the ordered list of all rational numbers with denominators not greater than n. Note that this is not equivalent to an isometric grid-based quantiser. An isometric grid with spacing 1/8 would not quantise triplets, whereas a Farey sequence with order 8 would, as well as quintuplets and septuplets. Additionally, despite the increased resolution, the Farey sequence isn't any more likely to mis-quantise notes near the beat than the isometric grid, because Farey only has higher temporal resolution only near finer subdivisions of the beat. So, it may be that Farey would perform better than an isometric grid for quick-and-dirty quantization, although for some reason isometric grids are very commonly used, but I have never noticed a Farey sequence used this way.

6. Timing

The beat tracker presented in Section 4 introduces a many small latencies, both positive (delays) and negative (predictions). If these are not corrected they can lead to a myriad of unexpected timing inconsistencies. For example, when tracking a sequence of isochronous impulses, the robot could report the beats as occurring at slightly different times than the onsets, and both might be incorrect, since those processes are somewhat independent and are subject to separate latencies. The onset and beat tracker identify the location of events in two distinct ways. Firstly, at each beat or onset, they identify the timestamp relative to the start of the audio stream where the beat or onset occurred. I refer to these as offline events, because even when these algorithms are used to process

[12] https://ambisynth.blogspot.com/2019/10/a-note-on-strength-of-note-onset.html

audio files offline (not in real time), they will locate the sample number of each beat or onset in the file. However, because these algorithms use a variety of internal filters, buffers, and probabilistic techniques that add delay and uncertainty, the timestamps could be systematically wrong. The second way these algorithms locate events is by firing event handlers in real time. As soon as they detect a beat or onset, they will call a user-defined function to report the offline timestamp of the event. However, the event handler will not necessarily be called at the same sample as the reported timestamp. In fact, because the onset-tracker is retrospective, the onset event handler will always be called some time after the timestamp of the onset. On the other hand, because the beat tracker in particular is predictive, it can call the corresponding event handler at any time before or after the predicted beat, without changing the reported timestamp of the beat. As will be seen, the event handler must be called at exactly the correct time. Luckily, all of these issues are modelled by the corresponding algorithms, and can thus be corrected. The onset tracker offline latency is modelled by ϵ_s, described in Section 4.1.1 above, and the the beat tracker online and offline latencies by ϵ_o and ϵ_r in Equation 7, respectively. By supplying the appropriate values to each of these terms, these errors can be corrected. Here, I will describe how to find the appropriate values.

6.1. Offline Latency

One might suppose that the time differences between when an onset or beat actually occurs and when it is reported to have occurred (ϵ_s for onsets and ϵ_o for beats) should be able to be calculated exactly, based on the size of the buffers and the order of the filters. However, after considerable effort, I was unable to find a closed-form equation that gave correct results over a range of buffer sizes, filter orders, and other parameters, so I settled on an empirical approach, as follows.

(1) I set ϵ_s and ϵ_o to 0 using the functions declared in the interface to the beat-tracking library.
(2) I generate a 2-minute audio file containing isochronous unit impulses spaced at 90 bpm.
(3) I put the beat tracker into count-in mode so that the first two impulses determine the initial tempo estimate, after which it will revert to actively tracking the beat and tempo.
(4) I sequentially analyze the audio file, keeping separate lists of the timestamps where the beats and onsets reportedly occurred.
(5) For sanity, I check that the beat tracker reported the same number of onsets and beats as there are impulses in the audio file. I then compute the mean error between the unit impulse locations in the audio file and the reported onset timestamps.
(6) I repeat this separately for the reported beat timestamps.
(7) Because there are perverse cases where the mean error depends strongly on the epicycle between the beat duration and the internal buffer sizes, this process is repeated for 100 different tempi between 90 (inclusive) and 180 (exclusive) bpm. I compute the mean of mean errors separately for the beats and onsets. These means are the empirical values of ϵ_s and ϵ_o, respectively.

Using the default buffer sizes, filter orders, and other settings, I find that ϵ_s is approximately 19 milliseconds, so onsets will always be reported 19 milliseconds after they occur, but now with the correct timestamp. I find ϵ_s to be almost 29 milliseconds, and

in the next section I will show how to use ϵ_r to correct this so that beats are reported slightly before they occur without affecting the now correct timestamp of the reported beats.

To validate this outcome, I set I set ϵ_s and ϵ_o to the computed values, and re-run the exact procedure using new tempi that lie between the tempi used the first time. After validation, ϵ_s has a residual error of less than one audio sample, and because beat-tracking is a more uncertain process, ϵ_o has a residual error of about 1 millisecond.

The downside of this empirical approach is that the parameters that affect the latencies cannot easily be changed without disrupting the timing, and are therefore assumed to be constant. If someone wanted, for example, to run the beat tracker with an onset-signal filter order other than the default one, they would first have empirically determine the latency adjustments with this process. Then, in software they could instantiate a new beat tracker with the desired filter order and pre-computed latency adjustments. For convenience, the beat-tracking library comes with a demo program that runs the above procedure for the supplied parameters.

6.2. Online Latency

The above procedure ensures that the beat-tracker will correctly identify the timestamps of the beats (correct at least in the limited sense of timing), but in real time, when should the beat event fire? It should fire a small time before the timestamp of the beat, so there is just enough time for the robot to play a stroke and have it sound exactly on the beat. However, precisely how much time is needed depends on several factors that are external to the algorithms themselves, such how the operating system buffers audio, and, in the case of the robot, how long it takes a solenoid to eject and strike the table after a voltage has been applied to it. Assuming that the offline latencies have already been corrected according to Section 6.1 above, following procedure can be used to correct the online latency.

(1) A robot is set up to listen to itself through its own contact microphone.
(2) The robot is counted in at a reasonable tempo, and then put into tempo-locked beat tracking mode so that it internally fires beat events isochronously and does not attempt to track the tempo or beat phase of its audio input.
(3) At each beat event, the beat-tracker reports the timestamp of the beat to OpenSquiggles. The reporting may happen before or after the timestamp of the beat, and the time of the reporting can be nudged forward and backward in time by adjusting ϵ_r, using the relevant functions declared in the beat-tracking library's public interface.
(4) As soon as it is notified of a beat, OpenSquiggles sends a USB MIDI message to the Teensy microcontroller, which then applies voltage to a solenoid, causing it to eventually strike the table.
(5) The resulting audio is picked up by the robot's microphone, digitised, buffered by the operating system, and eventually passed in to the onset detector. Some time later the onset detector reports the timestamp where the onset occurred. Importantly, due to the prior adjustment of ϵ_s, the timestamp will indicate the correct location of the onset in the audio stream.
(6) ϵ_r is adjusted until there is zero mean difference between the timestamps reported by the onset tracker and beat tracker, at a variety of tempi.

ϵ_r should now be correct, and this is easy to verify using the two-beat delay rhythm

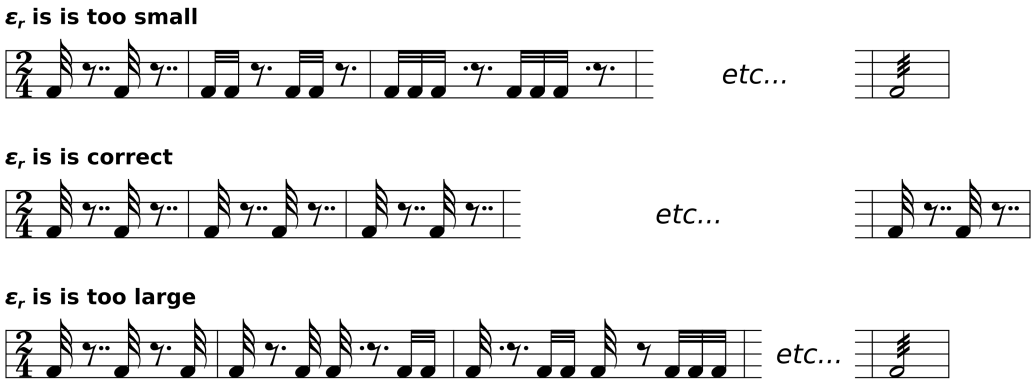

Figure 8. How to adjust Dr. Squiggles's timing. The robot is made to play on the beat while maintaining a constant tempo, listen to itself, and play back what it hears two beats later. If the latency adjustment ϵ_r is too small or too large, the delayed beats will not line up with the new beats, causing several onsets to be played near each beat in a runaway process that eventually fills time with as many onsets as possible. When ϵ_r is correct, the delayed beats will line up with the new beats, and only one onset will be played per beat.

module described in Section 5 above. The robot is kept exactly in the same setup, and additionally the two-beat delay module is activated. The robot will still tap on each beat, and it will additionally tap delayed copies of each previous tap. If ϵ_r is correct, the delayed copies will exactly line up with one another, and the robot will be heard to tap only on the beat. If ϵ_r is not correct, the robot will start tapping several times in rapid succession around each beat, and errors will accumulate in a runaway process that results in the robot filling time with the most rapidly repeated strokes it can produce. This effect is depicted in Figure 8.

Now that the robots can synchronise with each other and detect onsets with correctly adjusted timing, they can accurately transcribe what they hear. Later in this paper, I describe a system in which robots play a delayed copy of what they hear. Some of the results of that are in Figure 13, and some, but very few, transcription errors can be seen there. It is beyond the scope of this paper to benchmark each of the algorithms I have presented against other similar algorithms, although the low number of transcription errors serves as an end-to-end validation of the system as a whole.

7. Equilibrium

Thusfar, I have discussed the design of individual Dr. Squiggles robots, but ultimately I would like to know how *swarms* of musical robots behave. In particular, I would like to know under what conditions a network of robots that are listening and responding to one another will reach *equilibrium*. I use the term 'equilibrium' here to mean that the entire network has become locked-in to a particular rhythmic pattern, repeating it ad infinitum in the absence of any disruption. For clarity, I will point out that I am not claiming that it is necessarily good or desirable from a musical standpoint for the network to reach equilibrium. Nonetheless, equilibrium provides an entry-point for understanding the network behaviour as a whole. In the future, this understanding might be used to design a system that steers a swarm towards or away from equilibrium. To this end, I will first describe a simple responsive rhythm module that I designed for this purpose;

I will analyze the possible topologies for a swarm of Dr. Squiggles robots; and finally I will show some of the equilibria that emerge with the described rhythm module and topologies.

7.1. Histogram Rhythm Module

In order to start exploring the dynamical behaviour of a swarm of Dr. Squiggles robots, I implemented a so-called 'histogram' rhythm module, that is based on something similar used by Guy Hoffman in the marimba robot Shimone (Hoffman & Weinberg, 2011). Let a rhythm be defined to be cyclic, B beats in duration, with b equally-spaced subdivisions per beat. For simplicity, and without loss of generality, in this paper I always use 4 beat rhythms with 4 subdivisions per beat. Each subdivision is either populated by an onset or not, and the strength, timbre, and other features of the onsets are not considered here as they are not part of the present work, although I have treated them in previous work (Krzyzaniak, 2016) and those methods could be incorporated here at a later date. A given beat in the music will be denoted by an integer β, where β increments monotonically over the course of the music. The robot always evaluates rhythms immediately prior to the first subdivision of the current beat in real time, when it has been notified of the beat by the beat tracker.

I define a decaying histogram H which contains $B * b$ values, $H = \{h_0, h_1 \dots h_{Bb-1}\}$. The histogram is accessed cyclically as β increments, and for convenience I will write $H_\beta^\mathscr{b}$ to indicate the histogram value corresponding to subdivision $\mathscr{b}$ in beat β, i.e. $H_\beta^\mathscr{b} = h_{b(\beta \mod B)+\mathscr{b}}$, where $\mathscr{b} \in [0 \,..\, b-1]$. In this experiment, each histogram value is initialised to a uniformly distributed random number in the range $[0, 1)$, which will result in an initial note-density of 50%. Other initial distributions are possible and of interest in musical settings, although this does not affect the overall operation of the algorithm.

I store the incoming timestamps of the beats and onsets as reported by the beat and onset detectors described in Section 4. At each beat I quantise the onsets received since the previous beat using a simple grid quantiser with b gridlines. Using this, I define an onset mask O, where $O_\beta^\mathscr{b} \in \{0, 1\}$ indicates whether the robot heard an onset in subdivision $\mathscr{b}$ of beat β. Then I update the histogram in the previous beat by evaluating:

$$H_\beta^\mathscr{b} = \delta H_{\beta-1}^\mathscr{b} + (1-\delta) O_{\beta-1}^\mathscr{b} \tag{11}$$

for all $\mathscr{b} \in \{0 \,..\, b-1\}$, where the decay coefficient δ is a free parameter in the range $[0, 1)$. I then introduce a nonlinear squashing function, $F(x)$, which will be used to squash the histogram values, defined as

$$F(x) = \begin{cases} \frac{1+\mathscr{i}|2x-1|^k}{2}, & \text{for } x \geq 1/2 \\ \frac{1-\mathscr{i}|2x-1|^k}{2}, & \text{otherwise} \end{cases} \tag{12}$$

where the nonlinear exponent $k \geq 0$ and the inverse flag $\mathscr{i} \in \{-1, 1\}$ are free parameters. Figure 9 shows how the parameters affect this function. The probability P that the robot will play an onset on subdivision $\mathscr{b}$ of beat β is the squashed histogram value,

$$P_\beta^\mathscr{b} = F(H_\beta^\mathscr{b}). \tag{13}$$

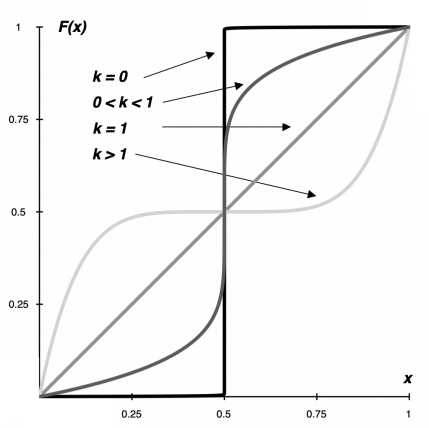

Figure 9. The nonlinear function $F(x)$ for a variety of values k with $i = 1$. To see what happens when $i = -1$, view this graph in a mirror.

This probability is used to generate an output rhythm $R_\beta^\flat$ which the robot will play. This is done one beat at a time, so at each beat in real time, a one-beat rhythmic fragment is generated by sampling P, as

$$R_\beta^\flat = P_\beta^\flat > r \tag{14}$$

for each subdivision $\flat \in \{0 \,..\, b-1\}$, where r is a uniformly distributed random variable in the range $[0, 1)$. When this inequality is true, the robot will play an onset in the corresponding beat and subdivision using the default solenoids and strengths.

The operation of the histogram rhythm module is shown in Figure 10 To recapitulate, the free parameters are

- The inverse flag $i \in \{-1, 1\}$. When $i = 1$, the robot will in some sense imitate what it hears, and when $i = -1$, the robot will play the complement of what it hears, in the sense of playing in the subdivisions where it did not hear anything, similar to hocket. I refer to these as 'forward' and 'inverse' modes, respectively. Inverse mode is interesting because it gives a robot some concept of when it should be silent.
- The decay coefficient $0 \leq \delta < 1$. Smaller values of δ make the histogram decay more rapidly, meaning that it adapts more quickly to new input. When $\delta = 0$ the histogram acts as a quantised B beat delay, repeating back whatever it hears B beats later, regardless of the value of k. I will refer to this as 'delay mode', or 'inverted delay mode', depending on the value of i.
- The nonlinear coefficient $k \geq 0$ which governs how extreme a histogram value has to be in order for the robot to be certain that it will or will not play an onset on the corresponding subdivision. When $k = 0$, F is a threshold function, where an onset will be played wherever the histogram is greater than 0.5.

7.2. Topology

The histogram rhythm generator is intended to operate on each of several robots that are mutually listening and responding to on another. Consequently, one final free parameter is the topology, i.e. the number of robots and their configuration. What topologies

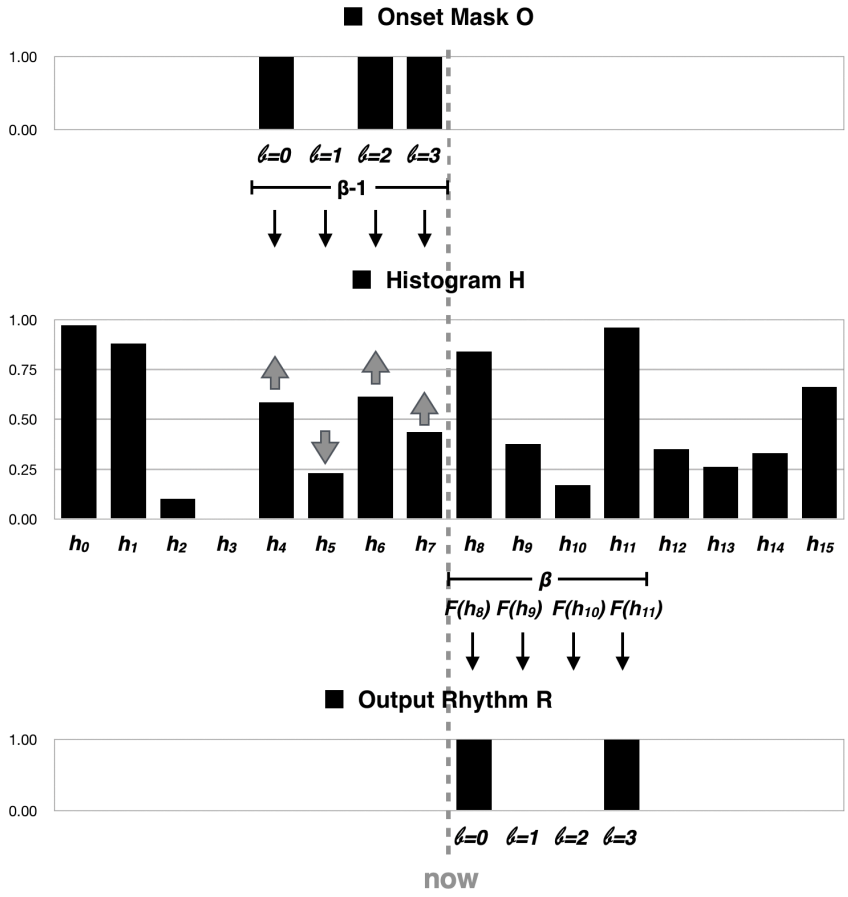

Figure 10. The histogram rhythm generator. At the end of each beat, the robot adjusts the histogram values for that beat up or down according to whether or not it heard an onset in the corresponding subdivision. Then it generates a rhythm for the next beat by passing those histogram values through a nonlinear function F and sampling the result.

Figure 11. Given that the robots listen to each other through contact microphones, the only possible connected topologies contain exactly one loop with zero or more chains attached to it. This figure depicts all possible configurations of three or fewer robots. Each node represents a robot, and the edges indicate where the robots are listening.

are possible? Imagine several robots represented by a directed graph. Each robot is represented by one node in the graph. The robots listen selectively through contact microphones, so an edge extending from node A to node B indicates that A is listening to B. Let us impose the following constraints:

(1) The graph be connected. There should not be two separate components where there is no path from nodes in one component to nodes in the other component. In real life, it might be interesting to put the robots into such a configuration, however each component would be analyzed separately, as several independent connected graphs.
(2) Each robot listens to exactly one robot. Each robot is placed by itself on a surface that is acoustically isolated from the other robots (e.g. a small table on a concrete floor), and a contact mic connects each robot to one such surface, ensuring that the mic picks up only the sound of the robot on that surface and nothing else. In the future it might be interesting to lift this constraint, either by considering mixers, different types of microphone, or by allowing the placement of several robots together on a single surface. However these configurations are not analyzed in the present study.

Additionally, to facilitate analysis, let us temporarily impose one additional constraint that shall be lifted presently.

(3) Each robot is listened-to by at least one other robot. This ensures that each robot contributes to the dynamics of the system as a whole. If there is some interconnected graph plus one robot off to the side listening in, that one robot will respond to what it hears, but if no robot is in turn listening to it, then there is no feedback that carries information about what that robot plays back into the graph.

Then, the only topology that satisfies these constraints is a loop, where A listens to B, B listens to C, etc, and Z listens to A. I will not prove it here, but it is not difficult to see after a few moments of consideration. Moreover, when removing constraint number 3, it is easy to see that the only possible topologies contain exactly one loop with zero or more chains connected to it, where a chain means A listens to B, B listens to C, etc.. and Z listens to any one of the robots in the loop. Figure 11 shows all topologies containing three or fewer robots that satisfy constraints 1 and 2.

Consequently, to understand the behaviour of an arbitrary network satisfying constraints 1 and 2, it is only necessary to analyze loops and chains. However, chains are not interesting, for exactly the reasons stated in constraint number 3, and consequently I will analyze only loops.

7.3. Loops

I will now analyze some of the equilibria that arise in a loop of Dr. Squiggles robots using the histogram rhythm module. I arrived at the analysis using both a purely analytical approach, as well as an empirical approach. In the analytical approach, I assign random initial 4-beat rhythms, $\mathbb{A}, \mathbb{B}, ...etc$ to the robots. For special parameter combinations, it is easy to see how these rhythms then propagate around the loop. For example, a robot in inverted delay mode, upon hearing rhythm $\mathbb{A}$, will output the inverse, $\mathbb{A}^{-1}$. From there, it is often easy to see how this behaviour changes after making small adjustments to the parameters. This type of analysis is depicted by Figure 12, and expounded for each depicted configuration the following subsections. In the empirical approach, I set up three Dr. Squiggles robots in a loop and recorded what they played under a variety of conditions. The experiments proceeded as follows:

- I assigned parameter values to each of the robots.
- I manually counted the first robot in at about 100 bpm and put it into tempo-locked mode, and made it play steady eighth-notes.
- I left the other two robots in regular beat-tracking mode, generating rhythms using the histogram module.
- I waited for these two robots to synchronise with the first and then put them in tempo-locked mode. After this, the network will stay synchronised with no measurable drift over the duration of the experiment.
- I broadcast a message to all the robots that caused them to start the experiment at the same time. Starting the experiment means that they are all using the histogram rhythm generator with freshly initialised histogram values.
- I let the robots run until they reached equilibrium.

During each experiment, I kept a record of what each robot actually played. I encoded each 4-beat rhythm as a 16-bit integer using a breadth-first counting order. This encoding ensures that similar rhythms will be encoded with nearby integers, sparser rhythms have lower encoded values, and for rhythm r, the inverse of r is $|r - (2^{16} - 1)|$ so that the inverse of a rhythm will be its reflection about the graph centerline. More information about this encoding is on my blog[13]. A plot of the rhythms that each robot played, encoded in this way, for a few separate trials, is depicted in Figure 13, and again expounded for each depicted configuration the following subsections. The supplementary video for this paper[14] shows some of the empirical trials. Now I will analyze the equilibria, starting with special cases and ending with general ones.

7.3.1. Static

In the trivial case that a robot has $\delta = 1$, it will ignore input and the underlying histogram will never be updated. When $k = 0$, this means that the robot will play the same rhythm over and over again without variation. A loop of three robots, all of which are in this configuration, is depicted in Figure 12. Each robot plays one B-beat rhythm, and the composite rhythm is B beats in length. If any of these robots has $k > 0$, it will play variations its rhythm, but the variations will all be sampled from the same underlying distribution. With increasing k the variations will be increasingly random. The behaviour is the same for inverse and forward nodes.

[13] `https://ambisynth.blogspot.com/2020/08/breadth-first-binary-numbers-and-their.html`
[14] `https://www.youtube.com/watch?v=yN711HXPfuY`

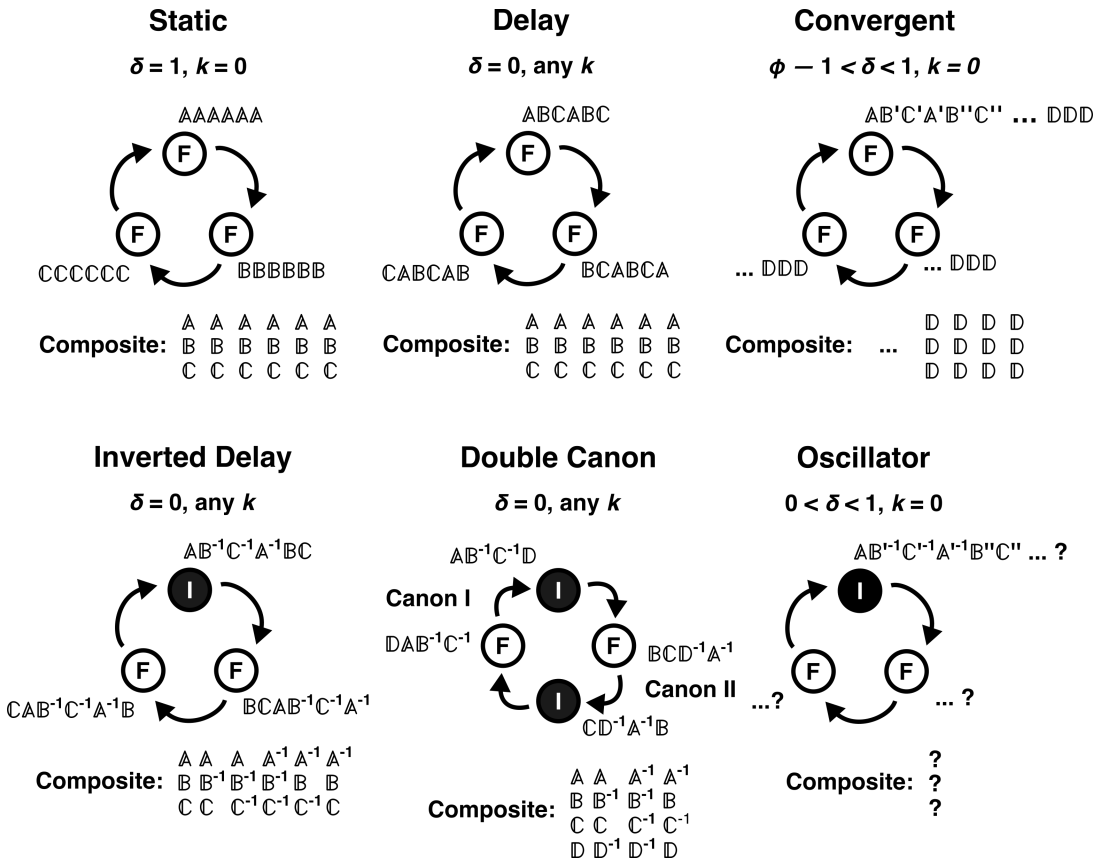

Figure 12. Analysis of several types of equilibria in a network of musical robots using the histogram rhythm module. 'F' indicates a robot in forward mode and 'I' is inverse mode. Arrows indicate the direction of listening. Each robot starts with a random 4-beat rhythm indicated by a single letter in blackboard font. These rhythms then propagate around the loop, with each robot potentially transforming them in some way, depending on the values of the parameters $\delta and k$. Following the rhythms around the loop in this way gives insight into what each robot will play, and into the composite rhythm, which is what a listener hears when the robots are aurally indistinguishable.

7.3.2. Delay

In the special case that a robot in forward mode has $\delta = 0$, it will act as a B beat delay, repeating back whatever it hears B beats later. The histogram values will all be either 0 or 1, and consequently the value of k does not matter. A loop of three robots, all of which are in this configuration, is depicted in Figure 12, and I will refer to this as a 'delay loop'. Each of the robots is initialised to some rhythm which is passed successively to each other robot in the loop. For a delay loop containing N robots, each robot will play N B-beat rhythms, although the composite rhythm will still be only B beats in duration. If any robot makes a transcription error, the network will remain in equilibrium, but the error will be propagated indefinitely and the original rhythm will be lost.

7.3.3. Convergent

If at least one robot in what would otherwise be a delay loop has δ above some critical value, but less than 1, that robot will always play a variation on what it hears, as the input only partially influences the histogram values. For example, if its neighbor plays rhythm $\mathbb{A}$, it will play $\mathbb{A}'$, and when it hears $\mathbb{B}$, it will play $\mathbb{B}'$. Later, once $\mathbb{A}'$ has come back around to it through the delay nodes, it will play $\mathbb{A}''$. Over time its histogram will become the average of these and $\mathbb{A}'''$ will tend towards $\mathbb{B}'''$ such that eventually it only produces one rhythm, $\mathbb{D}$ which is then propagated through the other nodes. A three-robot configuration like this is depicted in Figure 12. Once the rhythms have converged, all robots repeat the same one B-beat rhythm ad infinitum. The overall behaviour is the same when more than one robot has $0 < \delta < 1$. In the specific case shown, the critical value of δ is the inverse of the golden ratio, $1/\phi$. Higher values of δ make the network converge more quickly. If any node has $k > 0$, it converge more slowly and with more randomness along the way, while also lowering the critical value of δ. When there are more robots, the equilibrium rhythm may have one or more bits that are flipped at regular intervals, for instance in a 5-robot loop, one node may play on the downbeat of the first of B beats three times in a row, and then not twice. This can produce an interesting progression in a loop containing a large number for robots, e.g. 100. However, the composite rhythm is still only B beats in duration and is all of the bit-flipped variants or'd together. An empirical trial showing the convergence process is in Figure 13.

7.3.4. Inverted Delay

In the special case of an N-robot delay loop, excepting that at least one robot in inverse mode with $\imath = -1$, there will be $2N$ individual B-beat rhythms that circulate through the network – the N initial rhythms and their inverses. Although this is fairly straightforward, it is interesting to note that the robots in the network only allocate enough memory to store N rhythms, and the other N rhythms are stored, in a sense, in the structure of the network. Note, however, that the additional rhythms have low entropy as they are derived from the original rhythms by the simple process of inversion. A network like this is depicted in Figure 12. It is not necessarily the case that each robot will play each of the $2N$ rhythms. They will if the total number of inverting nodes in the network is odd. Otherwise, each robot will play N rhythms cyclically and the remaining N rhythms will be distributed amongst the other robots. As long as there is at least one forward (and one inverse) node in the network, the composite rhythm will also be of length N or $2N$, when the number of inverting nodes is even or odd, respectively.

However, if all the nodes are inverting, the composite rhythm will be of length 2. These facts can give rise to a number of interesting arrangements. Figure 12 also depicts a Double Canon network, with two forward nodes and two inverse nodes. One pair of robots plays a $4B$ beat simple canon, and the other pair plays a separate $4B$ beat simple canon. An empirical trial showing an inverted-delay loop is in Figure 13.

7.3.5. Oscillator

In the most general case, at least one robot is in inverse mode, and at least one robot has $0 < \delta < 1$. What happens depends on whether there are an even or odd number of inverse nodes in the loop. When even, the network usually behaves as a convergent network, but one or more robots end up repeating the inverse of the converged rhythm. The exception is when all robots have $k = 0$ and δ is less than some critical value, in which case the network eventually behaves as a delay loop, but one or more robots play the inverse rhythms they would play in a regular delay loop.

In the case that the number of inverting nodes is odd, the behaviour is more complex, and cannot easily be analyzed by the same method. The Oscillator diagram in Figure 12 shows two robots in delay mode, and one inverting robot with $0 < \delta < 1$. In this network, suppose that the inverting robot starts by randomly playing a dense rhythm. The next rhythm it plays, and perhaps the one after that, will also probably be dense, since it is only slightly influenced by what it hears, owing the the nonzero value of δ. After this, the delayed copies of the dense rhythms will have travelled around the loop back to the inverting robot. This will cause that robot to increment most of its histogram values multiple times in succession, making that robot more and more likely to play sparse rhythms, since it is inverting. The opposite will happen once the sparse rhythms travel around the network. This ultimately has the effect that this robot will oscillate between dense and sparse rhythms. An empirical trial is showing the time-evolution of this network is in Figure 13. It is surprising that for the depicted network, the period of oscillation is eight 4-beat rhythms. I do not know how to predict the period of the oscillation, although numerical analysis shows that higher δ results in longer periods. For the three-robot network with one inverting robot, periods of length 40 are possible when the inverting node has $\delta = 0.9$. Longer periods are generally the superposition of two shorter periods, for example the period of length 40 appears to be the superposition of 8- and 10-rhythm cycles. Such behaviour is easy to produce numerically, but difficult to produce on the robots, as small transcription errors can easily upset the pattern. If one or more nodes has a small value of $k > 0$, then this eight-rhythm cycle will have variations on each repetition, and with large enough values of k the variations will be so great that the cycle as a whole approaches noise.

Note that long cycles are *emergent* in the sense that there is not enough collective memory in the network to store so many complete rhythms, so the cycle emerges from the structure of the network.

7.3.6. No Equilibrium

The only configurations that I know that do not converge are the trivial Static networks when the nodes have $k > 0$, and some Oscillator networks sufficiently high k. These are the only known situations where noise can be injected without the histograms climbing the gradient of the squashing function F. In other words, if the network can converge, it will.

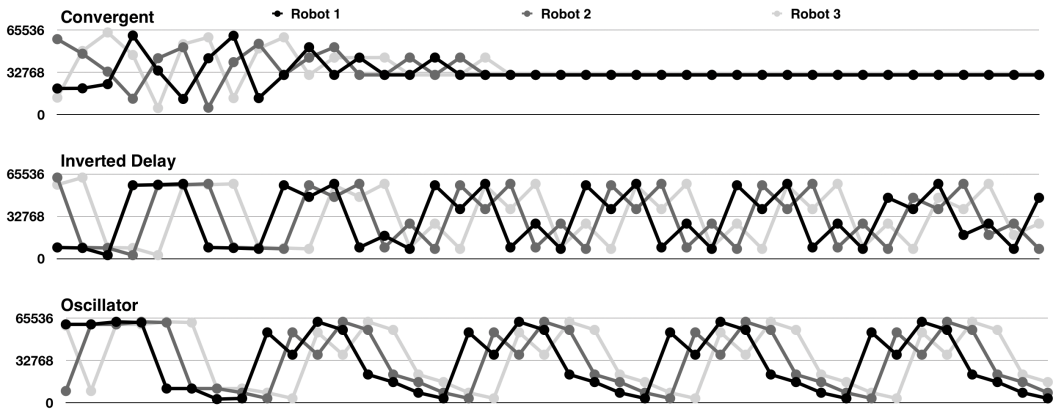

Figure 13. The time evolution of three separate empirical trials. Time is on the x-axis, and each point encodes an entire 4-beat rhythm such that rhythmic density increases up the y-axis, and a rhythm's inverse is its reflection about the graph centerline. On top is a Convergent network where all nodes have $\delta = k = 0.5, i = 1$. In the middle is an Inverted Delay network with all robots having $\delta = k = 0$ (delay mode), and only Robot 1 having $i = -1$ (inverted delay mode). The bottom is an Oscillator network where Robot 1 has $\delta = 0.8, k = 0, i = -1$, and the others have $\delta = k = 0, i = 1$ (delay mode).

8. Discussion

It is surprising that the robots reach equilibrium in so many configurations. In fact, it seems almost *inevitable* that they will reach equilibrium unless the rhythms are deliberately randomised, and even this often helps the robots climb the gradient of F towards equilibrium. Of course the rhythm generator used here is almost trivially simple, and not chaotic. It would be interesting to repeat the experiments using more dynamic, and less constrained models, for instance using neural networks in place of the histogram, or modeling time as a continuous variable. I am currently planning more research in this area. Nonetheless, I would still conjecture that even with such models, if the system can reach equilibrium it will. If this is the case, then equilibria should be a key design consideration for anyone building swarms of musical robots. What are the equilibria, and what is supposed to happen once they reach it?

More generally, physical rigid bodies can exhibit several different types of equilibrium, for example stable, unstable, saddle, and rolling. All of the equilibria discussed in this paper correspond to rolling equilibria, in the sense that if a rhythm is slightly modified, for example due to a transcription error, the network immediately reaches a new equilibrium that incorporates the modified rhythm, similar to how a balanced sphere is in equilibrium, and after rolling a small distance it will still be balanced but on a new point. I do not know whether it is possible to build a system such as described in this paper that exhibits stable, unstable, or saddle equilibrium, or whether any of the laws that govern rigid bodies would be applicable.

9. Squiggles as Interactive Artwork

There is a sense in which Dr. Squiggles is an artistic endeavor, in addition to whatever scientific questions might have arisen during its development. To validate them as such, I have shown them in three separate installations, in a form that is in each case similar

but not identical to what is described in this paper. They are as follows:

(1) I exhibited three of them at the Norsk Teknisk Museum in Oslo, Norway. The robots were using the histogram rhythm module, and the topology was a 'Y' shape, with the first two robots listening to the third, and the third listening to a table where people could tap rhythms[15].
(2) I exhibited them at the 2020 ICLI conference in Trondheim Norway. In this installation, visitors could tap their foot on the floor, and all of the robots synchronised their beat to the foot tapping. Visitors also wore armband muscle sensors so they could influence the rhythms that Dr. Squiggles played with their movements, using a rhythm generator that was not discussed in the present paper (Krzyzaniak, Veenstra, Erdem, Jensinius, & Glette, 2020).
(3) I exhibited them at the NIME 2020 virtual conference, where users could control them telematically using their computer keyboards. The sounds produced by Dr. Squiggles would then excite other musical robots that were part of the same installation[16].

10. Future Work

My colleagues and I are currently in the process of building seven more Dr. Squiggles robots, so that we have a total of ten of them. We plan on using these to investigate a variety of topics related to swarm musical robots. In addition to some of the questions posed elsewhere in this paper, I plan on investigating an intelligent use of pitch; other listening strategies that enable less restrictive topologies; and human-swarm interaction.

11. Acknowledgments

Thanks to Kyrre Glette for continual feedback, advice, and guidance. This work was partially supported by the Research Council of Norway through its Centres of Excellence scheme, project number 262762.

[15] https://www.uio.no/ritmo/english/projects/dr-squiggles/events/2020/light-walk/index.html
[16] https://www.uio.no/ritmo/english/projects/self-playing-guitars/events/2020/index.html

References

Albin, A. T. (2011). *Musical swarm robot simulation strategies* (Unpublished doctoral dissertation). Georgia Institute of Technology.

Baginsky, N. (n.d.). *The three sirens.* `http://www.the-three-sirens.info`. Retrieved from `http://www.the-three-sirens.info` (Accessed: 2020-08-09)

Bishop, L., & Goebl, W. (2015). When they listen and when they watch: Pianists? use of nonverbal audio and visual cues during duet performance. *Musicae Scientiae*, *19*(1), 84–110.

Blackwell, T. M., & Bentley, P. (2002). Improvised music with swarms. In *Proceedings of the 2002 congress on evolutionary computation. cec'02 (cat. no. 02th8600)* (Vol. 2, pp. 1462–1467).

Brown, A. R. (2005). Exploring rhythmic automata. In *Workshops on applications of evolutionary computation* (pp. 551–556).

Browning, C. (2008). *Phlock: Infrared computer vision music sequencer.* `https://www.youtube.com/watch?v=oHdecmpDzqI`. Retrieved from `https://www.youtube.com/watch?v=oHdecmpDzqI` (Accessed: 2020-08-09)

Ciardi, S. (2015). *Interview to Kolja Kugler, maker of "One love machine band".* `https://www.youtube.com/watch?v=qjEDC3XErTE`. Retrieved from `https://www.youtube.com/watch?v=qjEDC3XErTE` (Accessed: 2020-08-08)

Danielsen, A., Nymoen, K., Anderson, E., Câmara, G. S., Langerød, M. T., Thompson, M. R., & London, J. (2019). Where is the beat in that note? effects of attack, duration, and frequency on the perceived timing of musical and quasi-musical sounds. *Journal of Experimental Psychology: Human Perception and Performance*, *45*(3), 402.

Desain, P., & Honing, H. (1989). The quantization of musical time: A connectionist approach. *Computer Music Journal*, *13*(3), 56–66.

Eigenfeldt, A., & Kapur, A. (2008). An agent-based system for robotic musical performance. In *Nime* (pp. 144–149).

Gonzalez Sanchez, V. E., Martin, C. P., Zelechowska, A., Bjerkestrand, K. A. V., Johnson, V., & Jensenius, A. R. (2018). Bela-based augmented acoustic guitars for sonic microinteraction. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 324–327).

Hoffman, G., & Weinberg, G. (2011). Interactive improvisation with a robotic marimba player. *Autonomous Robots*, *31*(2-3), 133–153.

Kolling, A., Walker, P., Chakraborty, N., Sycara, K., & Lewis, M. (2015). Human interaction with robot swarms: A survey. *IEEE Transactions on Human-Machine Systems*, *46*(1), 9–26.

Krzyzaniak, M. (2016). *Timbral learning for musical robots.* Arizona State University.

Krzyzaniak, M., Akerly, J., Mosher, M., & Yildirim, M. (2014). Separation: Short range repulsion. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 303–306).

Krzyzaniak, M., & Paine, G. (2015). Realtime classification of hand-drum strokes. In *Proceedings of the international conference on new interfaces for musical expression* (pp. 400–403).

Krzyzaniak, M., Veenstra, F., Erdem, c., Jensinius, A. R., & Glette, K. (2020). Air-guitar control of interactive rhythmic robots. In *Proceedings of the international conference on live interfaces.*

Machines, R. (2013). *Compressorhead Ace of Spades.* `https://www.youtube.com/watch?v=3RBSkq-_St8`. Retrieved from `https://www.youtube.com/watch?v=3RBSkq-_St8` (Accessed: 2020-08-08)

Martins, J., & Miranda, E. R. (2007). Emergent rhythmic phrases in an a-life environment. In *Proceedings of ecal 2007 workshop on music and artificial life (musical 2007)* (pp. 10–14).

Mohan, Y., & Ponnambalam, S. (2009). An extensive review of research in swarm robotics. In *2009 world congress on nature & biologically inspired computing (nabic)* (pp. 140–145).

Mottaghi, A., Behdin, K., Esmaeili, A., Heydari, M., & Marvasti, F. (2017). Obtain: Real-time beat tracking in audio signals. *arXiv preprint arXiv:1704.02216*.

Percival, G., & Tzanetakis, G. (2014). Streamlined tempo estimation based on autocorrelation

and cross-correlation with pulses. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, *22*(12), 1765–1776.

Records, N. (2010). *Pat Metheny introduces "Orchestrion".* `https://www.youtube.com/watch?v=KsYEOUKS4Yk`. Retrieved from `https://www.youtube.com/watch?v=KsYEOUKS4Yk` (Accessed: 2020-08-08)

Rokeby, D. (2001). *n-cha(n)t.* `https://www.youtube.com/watch?v=FMIjxnNllMA`. Retrieved from `https://www.youtube.com/watch?v=FMIjxnNllMA` (Accessed: 2020-08-08)

Scholkmann, F., Boss, J., & Wolf, M. (2012). An efficient algorithm for automatic peak detection in noisy periodic and quasi-periodic signals. *Algorithms*, *5*(4), 588–603.

Singer, E., Feddersen, J., Redmon, C., & Bowen, B. (2004). LEMUR's musical robots. In *Proceedings of the 2004 conference on new interfaces for musical expression* (pp. 181–184).

Singer, E., Larke, K., & Bianciardi, D. (2003). LEMUR GuitarBot: MIDI robotic string instrument. In *Proceedings of the 2003 conference on new interfaces for musical expression* (pp. 188–191).

Stark, A. (2011). *Musicians and machines: Bridging the semantic gap in live performance* (Unpublished doctoral dissertation). Queen Mary University of London.

Williamon, A., & Davidson, J. W. (2002). Exploring co-performer communication. *Musicae Scientiae*, *6*(1), 53–72.

松島俊明, 金森克洋, & 大照完. (1985). 🄵譜の自動認識システム (wabot-2 の視🄵系). 日本ロボット学会🄵, *3*(4), 354–361.